Diseño, desarrollo y simulación de modelo comportamental de múltiples unidades de ejecución con memoria jerárquica como Dispositivo Bajo Verificación para el diseño de estrategia con cobertura mayor al 90% en su verificación pre-silicio

Por:

Janeth Monica Salas Alcantara

Tesis sometida como requisito parcial
para obtener el grado de

MAESTRO EN CIENCIAS EN LA ESPECIALIDAD DE ELECTRÓNICA

en el

Instituto Nacional de Astrofísica, Óptica y Electrónica

Noviembre 2023
Tonantzintla, Puebla

Supervisada por:

Dr. Jorge Francisco Martínez Carballido

Departamento de Electrónica
INAOE

©INAOE 2023
Derechos Reservados
El autor otorga al INAOE el permiso de reproducir y distribuir copias de esta tesis en su totalidad o en partes mencionando la fuente.

Resumen

Con el paso de los años la complejidad en los procesos de diseño, fabricación y verificación de circuitos tiende a aumentar de manera significativa debido a la cantidad de transistores que los integran, esto no solo ha conducido a la constante mejora de procedimientos, técnicas o estrategias que se implementan, sino también al desarrollo de muchas otras herramientas que provean solucionen eficaces para los distintos retos con los que las personas involucradas en dichos procesos se encuentran.

La verificación pre – silicio es fundamental en el desarrollo de un proyecto, ya que es la fase de pruebas realizadas una vez que un diseño ha sido completado, el objetivo de la verificación es proporcionar la certeza de que el diseño es una representación correcta de las especificaciones dadas al diseñador, una vez que la etapa de pruebas se concreta, se puede pasar a la siguiente etapa que es la validación.

En la actualidad, el proceso de verificación consume tantos recursos como el desarrollo mismo del diseño, en estudios de proyectos ASIC/IP realizados en 2022 se calcula una media entre 50% y 60% del tiempo total de un proyecto dedicado a la tarea de verificación, lo cual se refleja en la necesidad de crecimiento del área de personas dedicadas a dicha tarea y por supuesto en costos.

El presente trabajo de tesis está dividido en dos partes importantes, la primera consta del diseño, desarrollo, síntesis y simulación del modelo comportamental de un procesador con múltiples unidades de ejecución y jerarquía de memoria. La elección de usar un modelo comportamental nace de la necesidad de encontrar un diseño con la suficiente complejidad para implementar estrategias más cercanas a los procesos reales de verificación, ya que en la gran mayoría de los trabajos académicos publicados se utilizan diseños sencillos donde estos procesos suelen ser limitados y la alta confidencialidad de propiedad intelectual de las empresas dedicadas al diseño no permite disponer del código fuente de un dispositivo con facilidad. El modelo comportamental está diseñado de acuerdo a distintos manuales de información acerca de las características de procesadores con más de una unidad de ejecución.

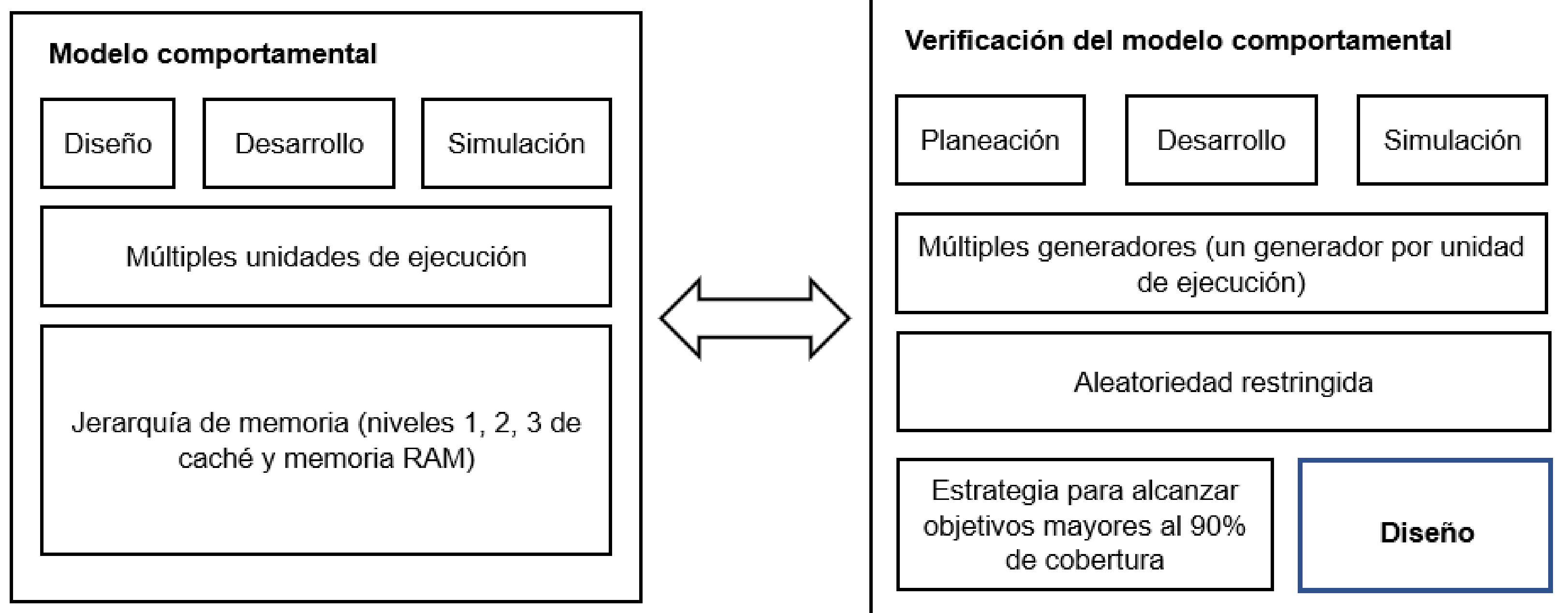
La segunda parte consta de la planeación, desarrollo y puesta en práctica de una estrategia de verificación del modelo comportamental para alcanzar porcentajes de cobertura mayores a 90%. Esta etapa de verificación está basada en el análisis de las características del modelo, identificando aquellas de especial interés. Ya que el modelo comportamental del procesador cuenta con múltiples unidades de ejecución se plantea el uso de un generador de instrucciones por unidad, lo que permite realizar pruebas específicas en cada una de ellas según se requiere. Los casos que se detallan en este documento son la ejecución de todo el conjunto de instrucciones implementado en el diseño, la ejecución de instrucciones que escriben a registros y por último la ejecución de instrucciones que escriben en memoria, incluyendo pruebas que logren coberturas del 100% en cada uno de estos.

La estrategia de verificación que se presenta se basa en la aleatoriedad restringida, que al ser dirigida de acuerdo al análisis de características del diseño proporciona los porcentajes esperados en la cobertura, es decir,

porcentajes más allá del 90%. El trabajo muestra que la estrategia planteada logra la verificación con cobertura del 100%.

Se contribuye con un modelo de múltiples unidades de ejecución con una jerarquía de memoria de 3 niveles de caché y RAM; además de la estrategia de verificación para coberturas altas.

Resumen Gráfico



Resumen gráfico del trabajo de tesis.

Abstract

Over the years, the complexity in the design, manufacturing, and verification processes of circuits has significantly increased due to the number of transistors they integrate. This has led to continuous improvement of procedures, techniques, and strategies, as well as the development of several tools to provide effective solutions for the different challenges faced by individuals involved in these processes.

Pre-silicon verification is fundamental in a project's development, as it involves testing performed once a design is completed. The objective of verification is to ensure that the design accurately represents the given specifications to the designer. After the testing phase is completed, the project moves on to the validation stage.

Currently, the verification process consumes as many resources as the design development itself. In ASIC/IP project studies conducted in 2022, an average of 50% to 60% of the total project time is estimated to be dedicated to the verification task. This highlights the need for an increase in the workforce dedicated to this task and, consequently, the associated costs.

This thesis is divided into two important parts. The first involves the design, development, synthesis, and simulation of the behavioral model of a processor with multiple execution units and memory hierarchy. The choice of using a behavioral model stem from the need to find a design complex enough

to implement strategies closer to real verification processes. In academic works, simple designs are often used, limiting these processes. The high intellectual property confidentiality of design companies also makes it difficult to access the source code of a device. The behavioral model is designed based on several manuals providing information about the characteristics of processors with more than one execution unit.

The second part involves planning, development, and implementation of a verification strategy for the behavioral model to achieve coverage percentages greater than 90%. This verification stage is based on analyzing the model's characteristics and identifying those of special interest. Due the processor's behavioral model has multiple execution units, the use of an instruction generator per unit is proposed, allowing specific tests to be performed on each unit as needed. The cases detailed in this document include the execution of the entire set of instructions implemented in the design, execution of instructions that write to registers, and execution of instructions that write to memory. The work shows that the proposed strategy achieves verification with 100% coverage.

The verification strategy presented is based on constrained randomization, which, when directed according to the design characteristics' analysis, provides the expected coverage percentages, surpassing 90%. The work demonstrates that the proposed strategy achieves verification with 100% coverage. It contributes a model with multiple execution units and a 3-level cache and RAM memory hierarchy, as well as a verification strategy for achieving high coverage.

Agradecimientos

A mi asesor, el Dr. Jorge Francisco Martínez Carballido, por compartir conmigo sus conocimientos y ayudarme a construir nuevos, por su apoyo dedicación y paciencia al dirigir este trabajo.

Al Instituto Nacional de Astrofísica, Óptica y electrónica (INAOE) por abrirme sus puertas y brindarme los recursos necesarios en esta etapa de mi desarrollo profesional.

Al consejo Nacional de Humanidades Ciencias y Tecnología (CONAHCYT), por el apoyo económico con número de identificación CVU 1148579 durante la realización de mi maestría.

Se agradece el valioso tiempo dedicado en múltiples reuniones al M.C. Joaquín García Ramírez y al M.C. Román Bernal, ya que con los conocimientos adquiridos a lo largo de más de 36 años de experiencia acumulada en la industria en el área de verificación contribuyeron en el entendimiento y aplicación de aspectos relevantes que acercan este trabajo a los procesos reales de verificación.

Al Dr. Juan Manuel Ramírez Cortés, Dr. José de Jesús Rangel Magdaleno y Dr. Rogerio Adrián Enríquez Caldera, por el tiempo dedicado a la mejora de este trabajo.

Página en blanco intencionalmente.

Dedicatoria

A mis padres, por enseñarme a ser una persona responsable y dedicada.

A mis hermanas, Georgina, Karla y Maura, por cuidar de mí y ser mi ejemplo a seguir en los diferentes aspectos de mi vida.

A Alexi, Bruno y Nagini, por darme apoyo incondicional, ánimo y, sobre todo, por esperar pacientemente cada una de mis visitas.

Página en blanco intencionalmente.

Tabla de contenido

Resumen ... i

Resumen Gráfico ... iv

Abstract ... v

Agradecimientos ... vii

Dedicatoria ... ix

Tabla de contenido ... xi

Capítulo 1 Introducción ... 1

1.1 Introducción ... 1

1.2 Justificación ... 2

1.3 Objetivos ... 4

1.3.1 Objetivo general ... 4

1.3.2 Objetivos específicos ... 5

Capítulo 2 Marco Teórico ... 7

2.1 Verificación ... 7

2.2 Lenguajes de Verificación ... 8

2.2.1 SystemVerilog ... 8

2.2.2 Metodología Universal de Verificación (UVM) ... 9

2.3 EdaPlayground ... 9

2.4 Cobertura 10
2.5 Cobertura Funcional 12
2.5.1 Grupos, puntos de cobertura y ‘bins’ 12
2.5.2 Opciones de cobertura 13
2.5.3 Verificación Aleatoria restringida 15
2.6 Ambiente de Verificación 17
Capítulo 3 Metodología 25
3.1 Elección del diseño a Verificar 25
3.2 Características del Modelo Comportamental 26
3.2.1 Unidades de ejecución 27
3.2.2 Registros de propósito general 27
3.2.3 Módulo de Memoria 27
3.2.4 Conjunto de instrucciones 28
3.2.5 Formato de instrucción 29
3.2.6 Módulo de generación aleatoria de instrucciones 30
3.2.7 Ejecución de una instrucción 32
3.3 Ambiente de Verificación 34
3.3.1 Modificaciones al Ambiente de Verificación 35
3.4 Estrategia de Verificación 37
3.4.1 Generación aleatoria de restricciones 39
Capítulo 4 Resultados 43
4.1.1 Modelo comportamental propuesto 43
4.2 Verificación funcional del modelo comportamental 50
4.2.1 Restricciones propuestas 52

4.2.2 Plan de verificación 52

4.2.3 Caso de instrucciones generales 53

4.2.4 Caso de instrucciones que escriben a registros 58

4.2.5 Caso de instrucciones que escriben en memoria 62

4.2.6 Pruebas extendidas 67

4.2.6.1 Instrucciones con escritura a registros. 68

4.2.6.2 Instrucciones de escritura a memoria. 73

Capítulo 5 Conclusiones 79

5.1 Trabajo futuro 81

Lista de figuras 83

Lista de tablas 85

Lista de listados de código 87

Bibliografía 89

Capítulo 1

Introducción

1.1 Introducción

El crecimiento en la tecnología de semiconductores ha llevado a la integración de decenas de miles de millones de transistores en un espacio considerablemente reducido, esto ha ocasionado que los procesos de diseño, fabricación, verificación y validación de los dispositivos sea cada vez más complejo.

Durante el proceso de desarrollo de circuitos hay dos etapas muy importantes, ambas separados por un evento crítico que es la fabricación del chip, la primera es la etapa pre – silicio, que es el diseño antes de fabricar el chip y la segunda es la etapa post – silicio, que consta de la validación y análisis después de fabricar el mismo. Estas dos etapas se pueden dividir según las fases de prueba en dos clases, la verificación, que son todas las pruebas realizadas durante el pre – silicio; y la validación, que son todas las pruebas realizadas durante el post – silicio.

La verificación se considera una actividad fundamental diferente del diseño, esto ha conducido al desarrollo de un lenguaje enfocado para la verificación. SystemVerilog permite crear entornos de verificación confiables

que pueden usarse repetidamente en múltiples proyectos, sin perder de vista que el objetivo no es únicamente la detección de errores, sino asegurarse de que el dispositivo bajo verificación sea una representación precisa y correcta de la especificación dada [1].

Las pruebas de esta tesis se basan en la metodología de pruebas aleatorias restringidas para alcanzar los objetivos de cobertura funcional en la verificación pre – silicio.

El contenido está organizado en cinco capítulos, en el Capítulo 1, Introducción, se detalla la justificación del presente trabajo de tesis y los objetivos a alcanzar. El Capítulo 2, Marco teórico, contiene la información necesaria para comprender los conceptos de verificación tratados en este documento, en el Capítulo 3, Metodología, se plantea una estrategia de verificación utilizando la aleatorización restringida con el fin de alcanzar los objetivos de verificación de una manera eficaz. En el Capítulo 4, Resultados, se presenta la aplicación de la estrategia propuesta para la verificación, por último, en el Capítulo 5 se encuentran las conclusiones y el trabajo futuro.

1.2 Justificación

La metodología que se desarrolla en esta tesis está enfocada en el diseño de una estrategia de verificación que permita realizar este proceso de forma eficiente.

Los lenguajes de Descripción y Verificación de Hardware (HDVL) como SystemVerilog son una herramienta que permite la creación de pruebas robustas que llevan a la exitosa ejercitación del diseño, sin embargo, como se observa en la figura 1.1 alcanzar los objetivos de cobertura en la verificación es cada vez más complejo, en los diseños de gran tamaño el progreso en la

exploración del diseño se detiene en algún momento del proceso y el tiempo que se emplea en tratar de sobrepasar la barrera del espacio que no ha sido probado no conduce a mejoras significativas [2], por lo que es fundamental el desarrollo de estrategias que faciliten esta tarea.

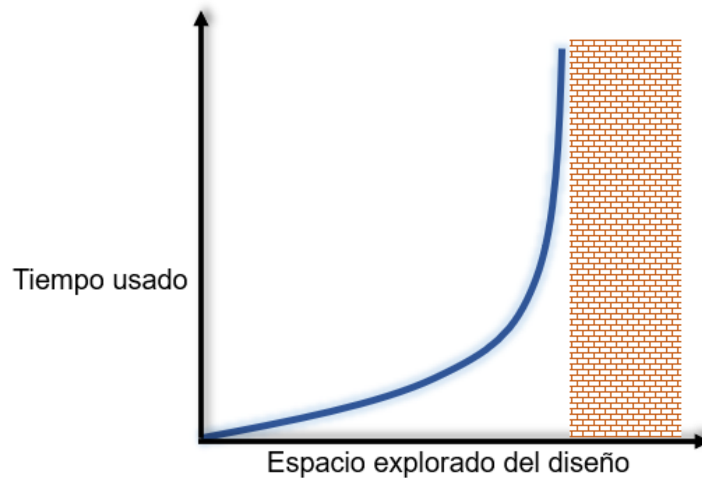


Figura 1.1 Progreso del proceso de Verificación, derivado de [2]

De acuerdo con el Estudio de verificación funcional del Wilson Research Group de 2022 [3] se dedica una media de 50% a 60% del tiempo total del proyecto a la tarea de verificación, en los proyectos donde se dedica muy poco tiempo a verificar se utilizan grandes cantidades de diseños integrados que ya han sido verificados previamente, mientras que en diseños de reciente creación todo debe ser verificado.

En la actualidad, la verificación supone un fuerte reto para las personas involucradas en esta área, ya que los diseños son cada vez más complejos y por tanto más difíciles de verificar, uno de los grandes desafíos es identificar soluciones de verificación y diseño orientado a verificación que se reflejen en el aumento de la productividad, por lo que la plantilla de ingenieros dedicados al diseño y a la verificación en un proyecto también ha sufrido algunos cambios significativos como se aprecia en la figura 1.2

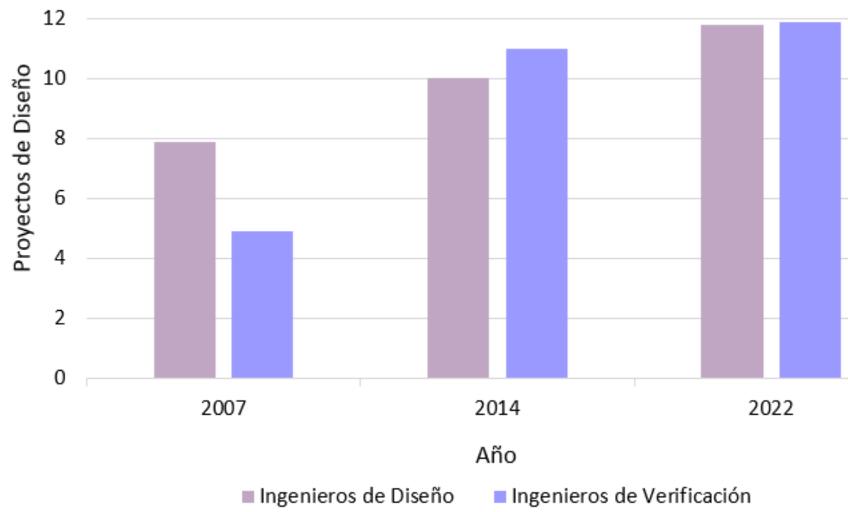


Figura 1.2 Número medio de ingenieros máximos por proyecto, derivado de [3].

Otro de los retos significativos es la planificación de los esfuerzos dirigidos hacia los objetivos de verificación que se desea alcanzar, ya que se considera que el proceso de depuración en el proceso de verificación utiliza un 47% del tiempo total dedicado a esta tarea [4].

La verificación Aleatoria Restringida es una de las metodologías más utilizadas para la escritura de pruebas en la verificación pre – silicio. En la presente tesis se aborda el desarrollo de una estrategia de verificación usando dicha metodología.

1.3 Objetivos

1.3.1 Objetivo general

El objetivo de la presente tesis es diseñar, desarrollar y simular el modelo comportamental de un procesador con múltiples unidades de ejecución, así como planear, desarrollar y simular el proceso de verificación

del modelo utilizando estrategias específicas que permitan mejorar la cobertura alcanzada durante dicho proceso.

1.3.2 Objetivos específicos

- Diseñar, desarrollar y simular el modelo comportamental de un procesador con múltiples unidades de ejecución.
- Diseñar y desarrollar un ambiente de verificación que permita el manejo del modelo comportamental.
- Desarrollar una estrategia funcional para mejorar el porcentaje de cobertura funcional alcanzado durante el proceso de verificación.
- Realizar pruebas con distintos casos restrictivos de acuerdo al conjunto de instrucciones que el modelo comportamental puede ejecutar.

Página en blanco intencionalmente.

Capítulo 2

Marco teórico

2.1 Verificación

El continuo crecimiento en la capacidad de integración de transistores en áreas de menor tamaño ha generado la necesidad de desarrollar herramientas y métodos que sean suficientemente robustas para probar el funcionamiento de los circuitos diseñados. La verificación de un diseño es el proceso mediante el cual se realizan pruebas para corroborar que este cumpla con las especificaciones bajo las que fue diseñado antes de pasar al proceso de generar los elementos necesarios para su puesta en silicio y su posterior fabricación. Existen distintos tipos de verificación, la verificación temporal, verificación funcional y la verificación formal.

En la verificación temporal, se debe revisar que el diseño satisface los tiempos de reloj y respuesta especificados. La verificación funcional busca comprobar que el diseño funciona según las indicaciones del diseñador, es decir produce el resultado esperado. En la verificación formal se utiliza el modelado de las características como expresiones matemáticas para confirmar que el diseño es correcto.

2.2 Lenguajes de Verificación

2.2.1 SystemVerilog

SystemVerilog es un lenguaje de verificación y diseño de Hardware (HDVL por sus siglas en inglés *Hardware Description and Verification Language*) que permite modelar diseños de cualquier tamaño y complejidad y a la vez verificar que estos sean funcionalmente correctos. Es una extensión de Verilog, un lenguaje de Descripción de Hardware (HDL por sus siglas en inglés *Hardware Description Language*), y está compuesto por una combinación de propiedades de otros lenguajes, tales como C, C++ y OpenVera, un lenguaje de verificación de alto nivel.

En 2005 SystemVerilog fue adoptado como estándar IEEE 1800-2005, la versión actual es el estándar 1800-2017 [5]. Es ampliamente utilizado en aplicaciones prácticas de codificación y síntesis en el diseño de circuitos y en la verificación y creación de entornos de verificación.

Entre sus características más significativas para el diseño y verificación de Hardware están:

- El elemento base ‘interface’ que permite encapsular las señales de comunicación facilitando la interacción con el diseño.
- Tipos de datos de lenguaje C, por ejemplo, los enteros ‘int’.
- Tipos definidos por el usuario usando la palabra reservada ‘typedef’.
- Tipos enumerados, arreglos simples y dinámicos.
- Control de procesos con estructuras ‘fork’, ‘wait’, ‘disable’.
- Estructuras ‘for, ‘foreach’, ‘while’, ‘repeat’, entre otras.

- Funciones ‘function’, tareas ‘task’, módulos ‘module’, clases ‘class’, paquetes ‘package’.
- Operandos de asignación como ++, --, +=.

Al igual que Verilog, SystemVerilog permite el modelado de comportamiento a un nivel abstracto, lo que proporciona herramientas para el modelado de la funcionalidad de un sistema en las etapas de análisis y verificación.

2.2.2 Metodología Universal de Verificación (UVM)

UVM por sus siglas en inglés *Universal Verification Methodology*, está conformado por bibliotecas de clases definidas que permiten desarrollar un entorno de verificación compuesto por componentes reutilizables, es un estándar empleado en la verificación de hardware. UVM está orientado a simulación, abordando la verificación y la recopilación de datos de una manera eficiente.

2.3 EdaPlayground

‘EdaPlayground’ es un entorno de desarrollo que ofrece una opción práctica para simular SystemVerilog, Verilog, C++/SystemC y otros lenguajes de Descripción de Hardware en un navegador web. Permite escribir en una pestaña un diseño que puede ser sintetizable o únicamente simulado mediante un ‘test’ que se escribe en otra pestaña separada, también integra un espacio similar a una consola para observar los indicadores de compilación, posibles errores o advertencias durante los procesos, y una ventana donde se pueden ver las formas de onda de las señales o variables que conforman el diseño.

Es una herramienta gratuita y contiene una gran variedad de simuladores comerciales, tales como ‘Aldec Riviera Pro 2022.04’, ‘Cadence Xcelium 20.09’, ‘Mentor Questa 2021.3’ y ‘Synopsis VCS 2021.09’. También incluye simuladores libres como ‘Icarus Verilog’ en diferentes versiones. Para usar los simuladores ‘Synopsis VCS’ y ‘Cadence Xcelium’ se debe tener una cuenta registrada y validada, ‘Aldec Riviera Pro’ no necesita la validación.

Al ser una herramienta sin costo puede tener algunas restricciones en cuanto a tiempos de simulación o algunas características de los lenguajes según el simulador elegido, sin embargo, estas restricciones no son significativas cuando se trata de proyectos que no necesitan una gran cantidad de recursos; además, provee un amplio margen de visualización de lo que ocurre en el tiempo en que permite simular.

EdaPlayground está en un constante proceso de mejora y proporciona a sus usuarios el soporte de las herramientas de diseño y descripción de hardware que tiene disponibles. Puede usarse en casi cualquier navegador web, pero los más recomendados son Firefox, Chrome e Internet Explorer 9 o uno superior, se puede acceder a EdaPlayground en la página de internet edaplayground.com.

2.4 Cobertura

La cobertura se define como los objetivos que se alcanzan con los métodos de verificación utilizados, en la figura 2.1 se observan las principales características de la cobertura, esta puede ser de dos tipos, código y funcional.

La primera se deriva directamente del código del diseño, lo cual significa que no está especificado por el usuario que realiza las pruebas, por lo que es

altamente probable no detectar las fallas de funcionamiento, esto lleva a crear un número muy grande de pruebas aun cuando el diseño no sea complejo.

En la segunda, la llamada cobertura funcional, el verificador puede especificar las características del diseño que desea comprobar. Para las pruebas de datos se usan grupos de cobertura, que, a su vez, contienen puntos de cobertura con *'bins'* o contenedores.

También suele utilizarse la cobertura cruzada para las variables o puntos de cobertura que están relacionados entre sí. El control en el flujo del diseño se especifica mediante secuencias de comportamiento.

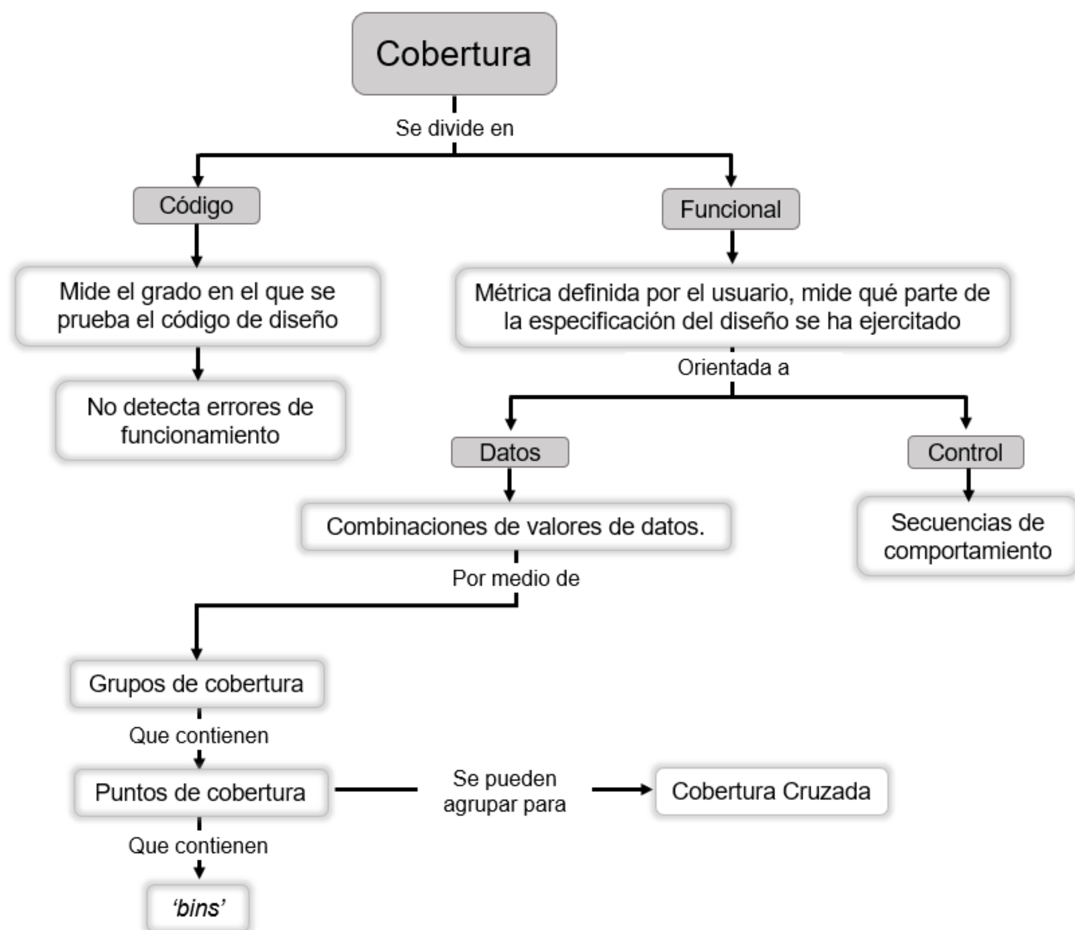


Figura 2.1 Conceptos de Cobertura.

2.5 Cobertura Funcional

La cobertura funcional permite especificar aquellas funciones del diseño que se desean cubrir con los métodos de verificación usando los llamados grupos y puntos de cobertura.

2.5.1 Grupos, puntos de cobertura y 'bins'

En *SystemVerilog* un grupo de cobertura *'Covergroup'* es un tipo predefinido que permite muestrear en grupo las transiciones y/o cruces de las variables que ocurren en un tiempo determinado, estos grupos se pueden definir dentro de un módulo *'Module'*, paquete *'Package'*, interfaz *'interface'* o una clase *'class'*. Un grupo de cobertura puede tener uno o más puntos de cobertura que pueden ser variables o expresiones.

Los contenedores o *'bins'* se usan para declarar el contenido que se quiere muestrear de una variable, esto quiere decir que se pueden detallar por separado cada una de las características de interés, lo que conduce a un mejor proceso de verificación.

En el listado de código 2.1 se muestra un *'Covergroup'* llamado *'Memory'* que muestrea en cada flanco positivo de la variable *'en'* y tiene un punto de cobertura *'addr'* con *'bins'* denominados *'low'*, *'med'*, y *'high'* para rangos bajo, medio y alto de las direcciones de memoria, un punto llamado *'par'* con *'bins' 'even'* para paridad par y *'odd'* para impar y un punto designado *'rw'* con *'bins' 'read'* para lectura y *'write'* para la escritura.

Listado de código 2.1 Declaración de un grupo de cobertura con puntos de cobertura y *'bins'*.

```
covergroup memory @ (posedge en);
      address: coverpoint addr {
```

```
            bins low    = {0,50};
            bins med    = {51,150};
            bins high   = {151,255};
}
      parity: coverpoint par {
           bins even  = {0};
           bins odd   = {1};
}
      read_write: coverpoint rw {
            bins  read  = {0};
            bins  write = {1};
endgroup
```

Para ver el ejemplo completo puede acceder a la página de internet ‘EdaPlayground’ en esta referencia [4].

2.5.2 Opciones de cobertura

El objetivo de la verificación es garantizar que el diseño se comporte correctamente en su entorno real de acuerdo con las especificaciones. Definir un modelo de cobertura es muy importante en el desarrollo de un banco de pruebas, ya que mientras mejor planificado esté, mejores serán los resultados de la verificación. En SystemVerilog existen opciones de cobertura con las que se puede especificar información en los grupos de cobertura, a continuación, se presentan algunas de ellas.

- **Option.name**: Con esta opción se le puede dar nombre a un grupo de cobertura para facilitar su reconocimiento.
- **Option.per_instance**: Es posible instanciar un grupo de cobertura en un mismo módulo más de una vez, por lo que es útil cuando se tiene un flujo de datos distinto para cada una de estas instancias, ya que permite ver los reportes por separado de cada una de ellas.

- **Option.at_least**: Esta opción posibilita el definir la cantidad de veces que una condición o evento debe ocurrir como número mínimo en el proceso de verificación y es particularmente conveniente cuando no se tiene suficiente información de lo que puede ocurrir en el diseño durante dicho proceso.
- **Option.goal**: En SystemVerilog el objetivo de cobertura para un punto o grupo de cobertura es el nivel en el que el grupo o punto se considera cubierto, esta opción admite fijar el objetivo porcentual de cobertura a cubrir, por defecto está configurada al 90%.
- **Option.weight**: Cada punto de cobertura puede tener un peso específico, es decir, es posible definir cuanto aportará dicho punto a la cobertura total del grupo al que pertenece.

En el listado de código 2.2 se muestra una propuesta de un grupo de cobertura para una memoria de 4 bits de domicilio. El grupo está identificado como *‘mem’* y se muestrea en cada flanco positivo de la variable *‘we’*, el nombre con el que aparecerá en el reporte de cobertura es *‘address_cov’* debido a la configuración de la opción *‘option.name’*, ya que es la única instancia del grupo la opción *‘option.per_instance’* está definida como 1 y se quiere cubrir al 100% como se indica con *‘option.goal’*.

El grupo de cobertura tiene dos puntos de cobertura, el primero es el punto *‘ar’* y está dedicado a cubrir rangos de domicilios en los 4 bits, por lo que tiene los *‘bins’* llamados *‘ar1’*, *‘ar2’* y *‘ar3’* cada uno para un rango diferente, cada uno de estos *‘bins’* debe cubrirse al menos tres veces, lo cual está establecido con la opción *‘option.at_least’* y el porcentaje que aporta a la cobertura total del grupo es de 40% según la opción *‘option.weight’*.

El segundo punto de cobertura se llama ‘*dat’* y se encarga de verificar algunas características de la variable *‘di’*. Es una buena práctica de verificación

corroborar que ocurran ciertas combinaciones de datos en una variable para asegurarse de que al fabricar un diseño todos los pines que involucra estén correctamente conectados, algunas de estas combinaciones son colocar todos los bits de la variable en ‘1’, todos en ‘0’ y alternar entre ‘1’ y ‘0’. Los *‘bins’* del punto de cobertura *‘dat’* están escritos de manera que estas combinaciones sean cubiertas. Cada una debe ocurrir al menos 2 veces y aporta un peso de 10% a la cobertura total.

Listado de código 2.2 Grupo de cobertura propuesto para una memoria de 4 bits de domicilio.

```
covergroup mem () @(posedge i_intf.we);
option.name = "address_cov";
option.per_instance = 1;
option.goal  = 100;
ar: coverpoint i_intf.addr
    {
     option.weight = 40 ;
option.at_least = 3;
          bins ar1 = {[0:2]};
          bins ar2 = {[7:8]};
          bins ar3 = {[13:15]};
    }
dat: coverpoint i_intf.di
    {
     option.weight = 10;
     option.at_least = 2;
bins di0 = {240};
          bins di1 = {170};
          bins di2 = {85};
          bins di3 = {15};
    }
  Endgroup
```

2.5.3 Verificación Aleatoria restringida

La verificación Aleatoria restringida es una metodología efectiva para alcanzar los objetivos de cobertura más rápido. Una de sus grandes ventajas

es que con las restricciones correctas se puede llegar a cubrir los casos extremos, es decir, aquellos que es poco probable que ocurran en un escenario normal, en la verificación estos casos son de especial interés.

En la aleatoriedad restringida se utilizan marcadores para verificar que los datos lleguen a su destino correctamente y se monitorean las conexiones para conseguir la información de cobertura, las restricciones centran la verificación en los escenarios que dicha información proporciona para así cumplir con los objetivos de cobertura. En este enfoque de verificación es posible tener menos casos de prueba y alcanzar las metas que se fijan durante la verificación.

En el ejemplo de la memoria de 4 bits se han generado “bins” específicos para cubrir los domicilios en rangos de datos alto, medio y bajo, los datos a escribir se cubrirán como cadenas de ‘1’ y ‘0’ alternados o consecutivos. Para que esta cobertura sea más efectiva se utiliza la generación aleatoria restringida de las variables de interés, en este caso los domicilios y los datos de entrada se restringen como se muestra en el listado de código 2.3.

Listado de código 2.3 Restricciones en las variables ‘addr’ y ‘di’ (domicilio y dato respectivamente)

```
constraint addr_c {addr [3:0] dist {[0:2]: =3, [7:8]: =5,
[13:15]: =6};}
constraint data_c {di [7:0] inside {170,85,240,15};}
```

La etiqueta ‘addr_c’ contiene las restricciones para los domicilios, en donde se utiliza una distribución con diferentes pesos para generar cada una de estas en los rangos especificados, mientras que para los datos en ‘data_c’ se generan solo los valores en cadenas de unos y ceros alternados (170, 85) y consecutivos (240,15). Con estas restricciones, el objetivo especificado de

100 % en el grupo de cobertura se cubre totalmente como se aprecia en la figura 2.2

```
COVERGROUP COVERAGE
================================================================
|          Covergroup          |   Hits   |  Goal /  | Status  |
|                              |          | At Least |         |
================================================================
| TYPE /tbench_top/mem         | 100.000% | 100.000% | Covered |
================================================================
| INSTANCE address_cov         | 100.000% | 100.000% | Covered |
|------------------------------|----------|----------|---------|
| COVERPOINT address_cov::ar   | 100.000% | 100.000% | Covered |
|------------------------------|----------|----------|---------|
| bin ar1                      |       17 |        3 | Covered |
| bin ar2                      |       13 |        3 | Covered |
| bin ar3                      |       20 |        3 | Covered |
|------------------------------|----------|----------|---------|
| COVERPOINT address_cov::dat  | 100.000% | 100.000% | Covered |
|------------------------------|----------|----------|---------|
| bin di0                      |       13 |        2 | Covered |
| bin di1                      |       15 |        2 | Covered |
| bin di2                      |       10 |        2 | Covered |
| bin di3                      |       12 |        2 | Covered |
================================================================
```

Figura 2.2 Reporte de cobertura del listado de código 2.2.

Puede acceder al ejemplo descrito en la página de internet de EdaPlayground en esta referencia [7].

2.6 Ambiente de Verificación

Un ambiente de verificación es una arquitectura de banco de pruebas que nos permite interactuar con el dispositivo bajo prueba para comprobar su funcionalidad. Típicamente, en SystemVerilog, el ambiente de verificación se compone como se aprecia en la figura 2.3.

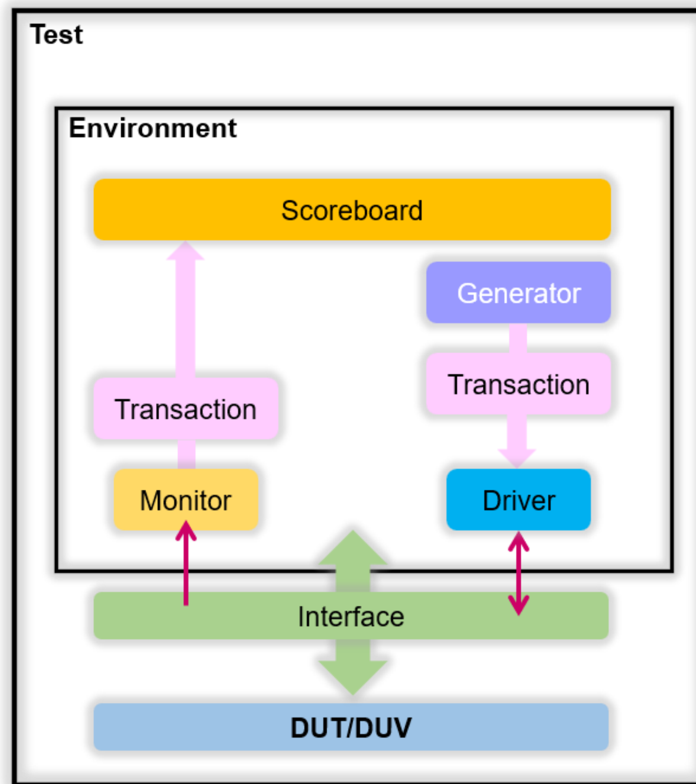


Figura 2.3 Estructura de un ambiente de verificación

El *Device Under Test* DUT o *Device Under Verification* DUV por sus siglas en inglés, es el diseño sobre el que se requiere realizar el proceso de verificación. El DUV está conectado al ambiente de verificación '*Environment*' a través de una interfaz *'Interface'*, esta comunicación es bidireccional.

El ambiente de Verificación tiene un generador *'Generator'* donde se producen las transacciones '*Transaction*', estas son recopiladas por el '*Driver*' para ser enviadas al DUV por medio de la interfaz. El monitor *'Monitor'*, muestrea las señales y las convierte de nuevo en transacciones, las transacciones del monitor se envían al '*Scoreboard*', donde se comparan con el resultado que se espera tener. Las líneas de conexión entre los elementos son *'Mailbox'*.

Todos los bloques están controlados por el *Test*, que es donde se integran el ambiente de verificación, la interfaz y el diseño a verificar. Esto posibilita la adición o eliminación eficiente de elementos para prueba, así como una forma más sencilla de escribir las condiciones y las pruebas que se quiere realizar. A continuación, se describen algunas características importantes de un ambiente de verificación tomado de la página de internet Verification Guide que puede consultar completo en esta referencia [8].

En la figura 2.4 se observa la estructura de un sumador ‘adder’ de 4 bits, tiene una entrada ‘valid’ que indica cuando hay valores validos de los operandos de entrada ‘a’ y ‘b’ que esta sincronizada con una señal de reloj llamada ‘clk’, el resultado de la suma estará presente en ‘c’.

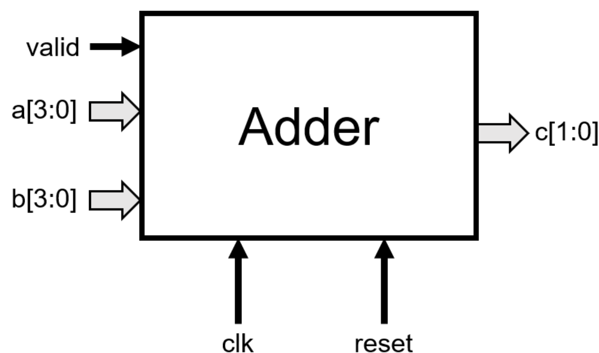


Figura 2.4 Estructura de un sumador de 4 bits.

De acuerdo a la figura 2.3 el ambiente de verificación del sumador, que para este ejemplo es el dispositivo bajo verificación DUV debe contar con una interfaz *‘Interface’*, un generador *‘Generator’*, las transacciones ‘*Transaction’*, un monitor *‘Monitor’* y un ‘*Scoreboard’*, que después se agrupan en el ‘environment’.

La conexión entre el DUV y el ambiente de verificación debe ser eficaz y sencilla, agrupar señales en una interfaz tiene el objetivo de que la

comunicación entre estos los componentes sea precisa y fácil de manejar. En SystemVerilog existe la palabra reservada ‘interface’ para el encapsulado de las señales, para el sumador interfaz se puede escribir de la siguiente forma, donde el nombre de la interfaz es ‘intf’ y se puede especificar en los paréntesis que tiene como entradas las señales ‘clk’ y ‘reset’ tal y como se presenta en el listado de código 2.4.

Listado de código 2.4 Declaración de la interfaz del sumador de la figura 2.4

```
interface intf(input logic clk,reset);
  logic    valid;
  logic [3:0] a;
  logic [3:0] b;
  logic [6:0] c;
endinterface
```

En el listado de código 2.5 se encuentran las transacciones que es donde se especifican las características de los estímulos que se van a generar.

Listado de código 2.5 Transacciones de los estímulos para el sumador de la figura 2.4.

```
class transaction;
  rand bit [3:0] a;
  rand bit [3:0] b;
       bit [6:0] c;
  function void display(string name);
    $display("-------------------------");
    $display("- %s ",name);
    $display("-------------------------");
    $display("- a = %0d, b = %0d",a,b);
    $display("- c = %0d",c);
    $display("-------------------------");
  endfunction
endclass
```

Para las transacciones se utiliza una clase, que es también una propiedad del lenguaje, definir clases es muy útil porque pueden ser

reutilizadas. Para este ejemplo se requiere que las entradas ‘a’ y ‘b’ sean aleatorias para tener un amplio margen de los datos a generar.

Las transacciones se crean aleatoriamente en el generador ‘*Generator*’ que se muestra en el listado de código 2.6 con el nombre ‘trans’ y se envían al ‘*Driver*’ por medio de un ‘*Mailbox*’ llamado ‘gen2driv’, el número de veces que esto ocurre está controlado por la variable ‘repeat_count’.

Listado de código 2.6 Generador de transacciones del sumador de la figura 2.4.

```
class generator;
  rand transaction trans;
  int  repeat_count;
  mailbox gen2driv;
  event ended;
  function new(mailbox gen2driv);
    this.gen2driv = gen2driv;
  endfunction

  task main();
    repeat(repeat_count) begin
    trans = new();
    if( !trans.randomize() ) $fatal("Gen:: trans randomization
failed");
      trans.display("[ Generator ]");
      gen2driv.put(trans);
    end
    -> ended;
  endtask
endclass
```

El ‘Driver’ recibe las transacciones y las envía una a una al DUV por la interfaz. En este ejemplo está compuesto de dos tareas que pueden verse en el listado de código 2.7, una donde reestablece los valores de ‘a’, ‘b’ y ‘valid’ en ‘0’, en la otra tarea envía los estímulos al DUV de forma sincronizada con la señal de reloj ‘clk’ e indicando que hay datos validos estableciendo en ‘1’ la señal ‘valid’, para el caso contrario la establece en ‘0’.

Listado de código 2.7 Extracto de código de ‘Driver’ para el sumador de la figura 2.4.

```
class driver;
//extracto de código para el ‘Driver’
//puede consultar el código completo en la referencia indicada

  task reset;
    wait(vif.reset);
    $display("[ DRIVER ] ----- Reset Started -----");
    vif.a <= 0;
    vif.b <= 0;
    vif.valid <= 0;
    wait(!vif.reset);
    $display("[ DRIVER ] ----- Reset Ended   -----");
  endtask

  task main;
    forever begin
      transaction trans;
      gen2driv.get(trans);
      @(posedge vif.clk);
      vif.valid <= 1;
      vif.a     <= trans.a;
      vif.b     <= trans.b;
      @(posedge vif.clk);
      vif.valid <= 0;
      trans.c   = vif.c;
      @(posedge vif.clk);
      trans.display("[ Driver ]");
      no_transactions++;
    end
  endtask
endclass
```

El monitor *‘Monitor’* recibe las transacciones del DUV por la interfaz y las envía una a la vez al ‘*Scoreboard*’ mediante otro ‘*Mailbox*’ llamado ‘*mon2scb*’ en el ejemplo. Está compuesto por la creación del ‘*Mailbox*’ y una tarea que recibe las transacciones de manera ordenada y sincronizada de acuerdo al ‘Driver’. Las dos tareas se encuentran en el listado de código 2.8.

Listado de código 2.8 Extracto de código de ‘Monitor’ para el sumador de la figura 2.4.

```
class monitor;
//extracto de código para el ‘Monitor’
//puede consultar el código completo en la referencia indicada

  task main;
    forever begin
      transaction trans;
      trans = new();
      @(posedge vif.clk);
      wait(vif.valid);
      trans.a   = vif.a;
      trans.b   = vif.b;
      @(posedge vif.clk);
      trans.c   = vif.c;
      @(posedge vif.clk);
      mon2scb.put(trans);
      trans.display("[ Monitor ]");
    end
  endtask
endclass
```

Una vez la transacción es recibida por el ‘Scoreboard’, se verifica que el resultado de la suma sea correcto y se imprimen los valores de la transacción, así como el mensaje correspondiente en caso de que sea equivocado, en el listado de código 2.9 se aprecian las dos tareas que componen la clase ‘Scoreboard’.

Listado de código 2.9 Extracto de código de ‘Scoreboard’ para el sumador de la figura 2.4.

```
class scoreboard;
//extracto de código para el ‘scoreboard’
//puede consultar el código completo en la referencia indicada

  task main;
    transaction trans;
    forever begin
      mon2scb.get(trans);
        if((trans.a+trans.b) == trans.c)
          $display("Result is as Expected");
        else
```

```
          $error("Wrong   Result.\n\tExpeced:   %0d   Actual:
%0d",(trans.a+trans.b),trans.c);
        no_transactions++;
      trans.display("[ Scoreboard ]");
    end
  endtask
endclass
```

En el ‘environment’ se instancian las clases que corresponden a cada componente con sus respectivos mailbox, y es también donde ocurre la sincronización de cada uno de ellos de acuerdo al número de transacciones que se generan.

Capítulo 3

Metodología

3.1 Elección del diseño a Verificar

La decisión de utilizar un procesador con múltiples unidades de ejecución surge de la necesidad de encontrar un diseño o dispositivo bajo prueba lo suficientemente complejo que permita implementar estrategias de verificación más cercanas a las que se emplean en procesos reales. La mayoría de los trabajos académicos publicados relacionados a verificación abarcan diseños más simples, por lo que los métodos de verificación y cobertura suelen ser más limitados.

Durante el proceso de búsqueda de un dispositivo es usual encontrar solo algunas partes disponibles de un diseño, algunos otros contienen errores en su código lo que los hace no funcionales, poco confiables o difíciles de reparar. Por otra parte, el acceso al código fuente con cierta complejidad de algún producto es sumamente complicado debido a los términos y condiciones de confidencialidad aunados a otras restricciones de las empresas que los producen. Utilizar un modelo comportamental resuelve esta situación, ya que con base en la documentación existente se pueden conocer las principales características que el modelo debe tener.

3.2 Características del Modelo Comportamental

La figura 3.1 representa la estructura a seguir para el modelo comportamental de un procesador con múltiples unidades de ejecución. Este modelo contiene los componentes básicos necesarios para la ejecución de instrucciones. Para conocer información detallada acerca de las partes que componen un procesador puede consultar los manuales *'Intel® 64 and IA-32 Architectures Software Developer Manuals'* [9] y la documentación de '*Software Oprimization Resources'* [10], los cuales contienen la información de estudio para la elaboración del modelo que se utiliza en este trabajo.

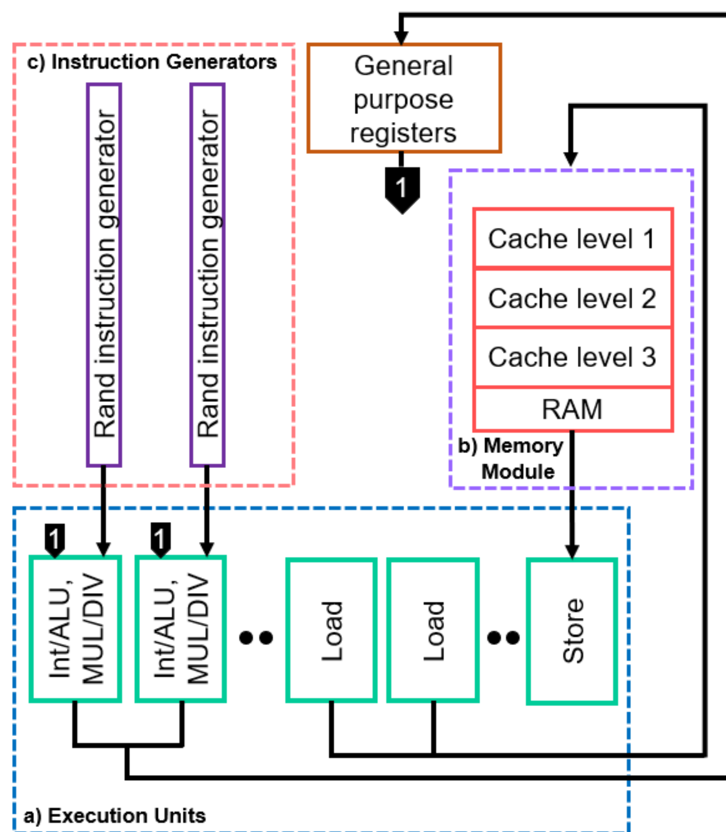


Figura 3.1 Estructura del modelo comportamental, a) Unidades de ejecución '*Execution units*', b) Módulo de memoria '*Memory Module'*, c) Generadores de instrucciones '*Instruction Generators*'.

3.2.1 Unidades de ejecución

En la figura 3.1 a) se aprecia que el modelo consta de dos unidades que realizan operaciones aritméticas y lógicas sobre datos con formato entero, además pueden multiplicar y dividir. También tiene dos unidades *'Load'* para cargar los datos correspondientes en las unidades de ejecución y una unidad *'Store'* para almacenarlos en memoria. El modelo permite agregar tantas unidades como se desee.

3.2.2 Registros de propósito general

Son registros de 32 bits que se utilizan para ejecutar instrucciones de propósito general, estas instrucciones son aquellas que realizan operaciones aritméticas y lógicas básicas sobre los datos. El modelo comportamental considera ocho registros de propósito general con los nombres EAX, EBX, ECX, EDX, ESI, EDI, EBP, y ESP.

3.2.3 Módulo de Memoria

Para el comportamiento del módulo de memoria se consideran niveles de memoria caché 1, 2, 3 y RAM como se observa en la figura 3.1 b) del modelo comportamental, estos se emulan por medio de retrasos en tiempo (ciclos de reloj) y con un arreglo dinámico donde se insertan localidades de 32 bits según se requieran durante el proceso de verificación. En la tabla 3.1 se presentan los ciclos de reloj considerados para los retrasos de los niveles de memoria, según la información consultada el cache de nivel tres es compartido cuando existe más de un núcleo en el procesador, por lo que es el nivel más lento después de la memoria RAM [11].

Tabla 3.1 Ciclos de reloj para niveles de memoria 1, 2, 3 y RAM.

Nivel de Memoria	Número de Ciclos de retraso	Rango de ciclos de reloj
Caché 1	4	
Caché 2	14	
Caché 3	24	
RAM	34	34 - 85

3.2.4 Conjunto de instrucciones

El conjunto de instrucciones o *Instruction Set Architecture* (ISA por sus siglas en inglés) es el repertorio de instrucciones que un procesador puede ejecutar. Los conjuntos de instrucciones pueden ser diferentes de un procesador a otro, ya que pueden ser diferentes en los tamaños de instrucción, el tipo de operación que permiten, los tipos de operandos sobre los que pueden operar y los resultados que pueden entregar [12].

En un conjunto de instrucciones las operaciones básicas suelen ser de registro a registro, registro a memoria, memoria a registro, memoria a memoria o involucrar datos inmediatos. Para considerar que es un conjunto eficiente las instrucciones deben contener la información necesaria para ejecutarse y no depender de la ejecución de alguna otra. En general, una instrucción se compone de un código de operación, operandos fuente y operandos de destino como se aprecia en la figura 3.2, sin embargo, los campos de información y su extensión pueden variar según la arquitectura del procesador para la que fueron diseñados.

Figura 3.2 Formato general de Instrucción.

3.2.5 Formato de instrucción

En la figura 3.3 se observa el formato de instrucción utilizado en el modelo a verificar. El código de operación es de 8 bits y con este se selecciona la operación que realizan las unidades de ejecución.

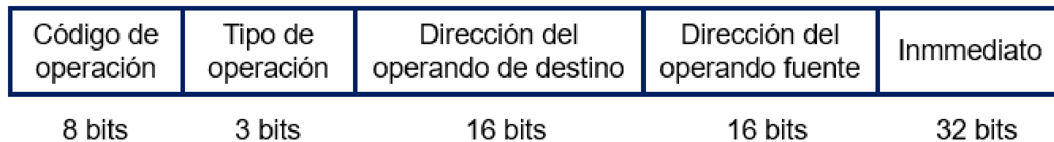


Figura 3.3 Formato de Instrucción.

En la tabla 3.2 se encuentra la información correspondiente a la identificación del tipo de operación con extensión de 3 bits, este campo del formato de instrucción define si la operación tiene operandos de memoria, registro o inmediatos. La dirección del operando de destino y la dirección del operando fuente pueden ser de 16 bits mientras que el inmediato es de 32 bits.

Tabla 3.2 Código de identificación del tipo de operación.

Tipo de operación (según los operandos)	Código de identificación en binario
Registro a registro	000
Registro a memoria	001
Memoria a registro	010
Inmediato a registro	011
Inmediato a memoria	100

Registro (Un solo operando)	101
Memoria (Un solo operando)	110

3.2.6 Módulo de generación aleatoria de instrucciones

Cada unidad de ejecución aritmético - lógica cuenta con un módulo de generación aleatoria de instrucciones *'Rand Instruction Generator'* los cuales están representados en la figura 3.1 c), en estos generadores es donde se originan las instrucciones que serán ejecutadas durante la verificación. El conjunto de instrucciones que se pueden generar aleatoriamente se muestra en la tabla 3.3.

Tabla 3.3 Conjunto de instrucciones disponibles en el generador aleatorio.

#	Código de Operación (HEX)	Descripción de operación	Tipo de operación (Binario)	Operando de destino	Operando fuente	Operando inmediato
1	05	ADD EAX, imm32	011	EAX		imm32
2	01	ADD r/m32, r32	000	r32	r32	
			001	m32	r32	
3	03	ADD r32, r/m32	000	r32	r32	
			010	r32	m32	
4	25	AND EAX, imm32	011	EAX		imm32
5	21	AND r/m32, r32	000	r32	r32	
			001	m32	r32	
6	FF	INC r/m32	101	r32		
			110	m32		
7	F7	NOT r/m32	101	r32		

			110	m32		
8	0D	OR EAX, imm32	011	EAX		imm32
9	09	OR r/m32, r32	000	r32	r32	
			001	m32	r32	
10	0B	OR r32, r/m32	000	r32	r32	
			010	r32	m32	
11	2D	SUB EAX, imm32	011	EAX		imm32
12	81	SUB r/m32, imm32	011	r32		imm32
			100	m32		imm32
13	29	SUB r/m32, r32	000	r32	r32	
			001	m32	r32	
14	2B	SUB r32, r/m32	000	r32	r32	
			010	r32	m32	
15	35	XOR EAX, imm32	011	EAX		imm32
16	31	XOR r/m32, r32	000	r32	r32	
			001	m32	r32	
17	33	XOR r32, r/m32	000	r32	r32	
			010	r32	m32	

El código de operación en la primera columna es el identificador en hexadecimal de cada una de las operaciones del conjunto de instrucciones y tiene una extensión de 8 bits. En la segunda columna se describe brevemente la operación aritmética o lógica que la unidad de ejecución realiza según el código correspondiente.

Tanto en el código de operación como en los operandos de destino y fuente r32 significa que el operando es cualquier registro de 32 bits, m32 es cualquier domicilio de memoria de 32 bits, r/m32 puede ser un operando de registro o uno de memoria según el tipo de operación, imm32 corresponde a un operando inmediato de 32 bits.

En la tercera columna se especifica con un número en binario de 3 bits el tipo de operación según los operandos que involucra, el tipo de operación detallado se encuentra en el apartado 3.2.4 en la tabla 3.2. En el modelo comportamental se considera que no pueden realizarse operaciones donde se involucren dos operandos de memoria a la vez.

3.2.7 Ejecución de una instrucción

Un procesador pasa por cuatro etapas de manera repetitiva como se puede ver en la figura 3.4, la primera es la obtención de una instrucción (*Instruction Fetch* en inglés), una vez que esta se carga en el registro de instrucción se pasa a la segunda etapa llamada decodificación (*Instruction Decode* en inglés) donde se transforma por la unidad de control (*control unit*) en micro – operaciones, que son las operaciones elementales que hacen posible que los datos estén disponibles para la unidad de ejecución correspondiente en el procesador. La tercera etapa es la ejecución (*Execute* en inglés) y es donde se lleva a cabo la operación en la unidad aritmética lógica (ALU) indicada en la instrucción, la última etapa es el almacenamiento (*Store* en inglés) que significa escribir el resultado de la operación en el registro o domicilio de memoria que corresponda.

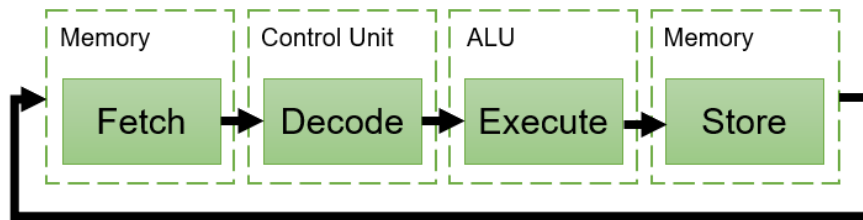


Figura 3.4 Ciclo de Instrucción.

En el módulo de generación aleatoria de la estructura de la figura 3.1 se producen el código de operación de la instrucción, la dirección de los registros que contienen los operandos, la dirección de memoria si la instrucción lo requiere y un parámetro denominado micro – operaciones (µops), este es un indicador del tiempo que la unidad de ejecución usa para tener disponible un resultado, la ejecución de una instrucción que opera únicamente sobre registros será más rápido que una instrucción que opera sobre registros y memoria debido a la localización del dato en memoria. Para comprender mejor el funcionamiento del modelo propuesto para verificación se muestra en la figura 3.5 el flujo de ejecución de una instrucción de registro a registro.

En la figura se indica con un número 1 el primer paso, que corresponde a la generación de los parámetros de la instrucción, el código de operación se envía a la unidad de ejecución (ALU) para seleccionar que operación se debe realizar. Al mismo tiempo se leen las direcciones de registro (indicado con un número 2) que contienen los datos para la operación que se va a ejecutar en la ALU, el parámetro de µops se envía hacia un bloque que controla dos multiplexores que dependiendo del tipo de operación habilitan las entradas de registro, memoria o inmediatos para cada operando, en una instrucción de registro a registro se permite el paso de los dos datos que vienen de los registros de propósito general (RPG).

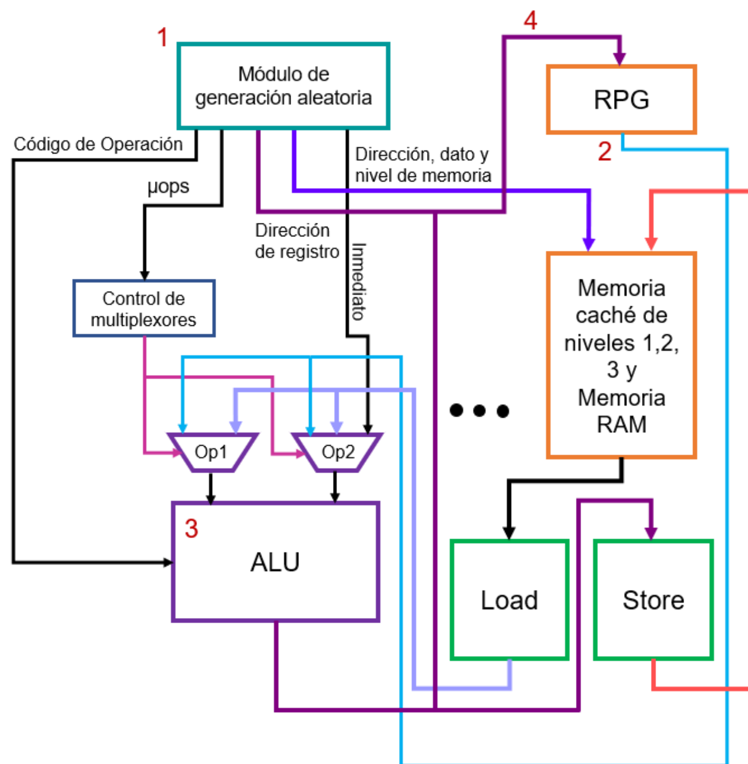


Figura 3.5 Flujo de ejecución de una instrucción de registro a registro.

Una vez disponibles los datos la ALU (número 3) realiza la operación con el tiempo de µops correspondiente, cuando el resultado está listo se almacena en los registros de propósito general indicado en la figura con un número 4. La figura contempla solo una unidad de ejecución aritmética – lógica para ilustrar mejor las interacciones con las unidades *'Load' y 'Store'.*

3.3 Ambiente de Verificación

El ambiente de verificación para el modelo comportamental propuesto contiene los componentes vistos en la figura 2.2 de la sección 2.4, un generador *'Generator'* para producir las transacciones '*Transaction'* con sus respectivas restricciones, un '*Driver'* para enviar dichas transacciones al dispositivo bajo verificación por medio de la interfaz, un *'Monitor'*, y un '*Scoreboard'* donde se reciben y comprueban los resultados.

Sin embargo, al tener más de una unidad de ejecución y con base en el diagrama de la figura 3.1 donde se muestran los componentes del modelo comportamental, es necesario hacer una serie de modificaciones que permitan realizar la articulación y verificación de todos los elementos del sistema. El ambiente de verificación está basado en el ejemplo de la página de internet *'Verification Guide'* usado también en la sección 2.6 en el que se realiza la verificación de un sumador [13].

3.3.1 Modificaciones al Ambiente de Verificación

A continuación, se describen las modificaciones al ambiente de verificación para el modelo comportamental a verificar.

- **Generador** *'Generator'*: Debido a que el modelo cuenta con dos unidades de ejecución aritmético - lógicas, se requiere tener un generador de instrucciones independiente para cada unidad, de esta forma cada una puede ejecutar instrucciones de distinto tipo y con diferente tiempo de ejecución sin depender de la otra.
- **Transacciones** *'Transactions'*: En las transacciones es donde se encuentran las restricciones para cada unidad de ejecución, por lo que al igual que el generador debe existir la posibilidad de utilizar distintas restricciones para cada unidad aritmético - lógica, esto permite controlar parámetros como el rango en las direcciones de memoria, datos, prioridad del tipo de instrucción entre otros.
- **Driver**: El *'driver'* es el encargado de conducir las transacciones hacia el dispositivo bajo prueba, por lo que ambas unidades de ejecución aritmético - lógicas deben tener un driver propio.
- **Monitor** *'Monitor'*: Cada una de las unidades aritmético - lógicas tiene un monitor que recibe los resultados obtenidos.

- **Scoreboard**: Existe un *'scoreboard'* por cada unidad de ejecución aritmético - lógica, estos reciben los resultados de los monitores y los comparan con los resultados esperados.

Aun cuando forman parte de un mismo dispositivo bajo prueba, cada unidad de ejecución aritmético – lógica tiene una interfaz que le permite comunicarse con los elementos que le corresponden en el ambiente de verificación, las unidades 'Load' y 'Store realizan µops que dependen de las instrucciones que se generan aleatoriamente para cada unidad aritmético – lógica, a su vez, todo se agrupa en la prueba '*test*' general como se aprecia en la figura 3.6.

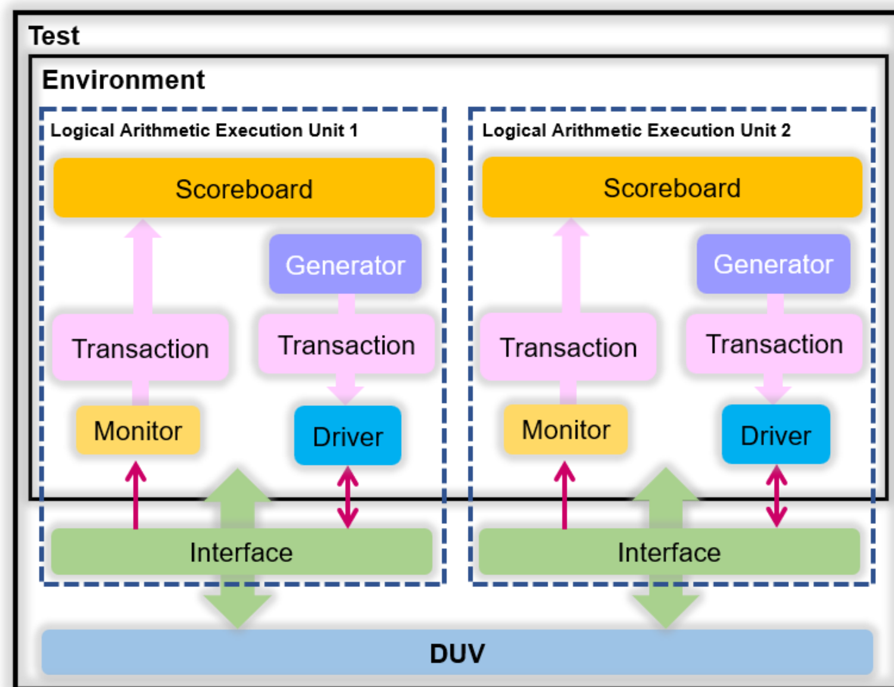


Figura 3.6 Ambiente de verificación modificado.

Al igual que en el dispositivo bajo prueba, es posible agregar en el ambiente de verificación elementos para nuevas unidades de ejecución si así se requiere.

3.4 Estrategia de Verificación

El capítulo 2 sección 2.5.3 brinda una breve explicación de la metodología de Verificación Aleatoria Restringida, la cual permite enfocar mediante restricciones en las variables los estímulos necesarios para lograr los objetivos de verificación más rápido.

Generar una estrategia de verificación tiene la finalidad de establecer un proceso a seguir con las funciones de un diseño que se desean probar, en la figura 3.7 se muestra el método planteado en este trabajo.

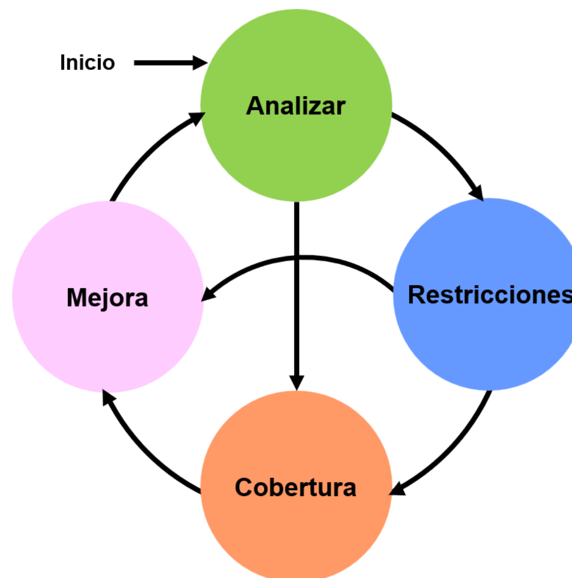


Figura 3.7 Estrategia de Verificación.

El proceso de verificación es cíclico y comienza con el análisis de lo que se desea comprobar en el diseño. Identificar las variables de interés y sus respectivas funciones es de especial importancia, ya que se pueden definir por orden de trascendencia o por las implicaciones que existen entre ellas, por ello es necesario tener un amplio conocimiento del diseño y de su operación para

analizar y especificar correctamente las características especiales de las funcionalidades a probar, el verificador debe pensar de acuerdo a su experiencia en aquellas condiciones en las que el diseño puede fallar.

El siguiente paso es encontrar las restricciones que permitan generar eficientemente los estímulos en las variables de acuerdo a las particularidades a probar. Este paso no es peculiarmente complicado, sin embargo, el tener conocimiento del lenguaje utilizado es fundamental para realizarlo, ya que cuanto más se conozca del lenguaje, mucho mayores son las herramientas que se pueden usar, por ejemplo, la distribución con que se generan los distintos valores de una variable, la ponderación de algunos de estos valores sobre otros, definir el orden de generación e incluso el poder anular alguno que no sea de importancia relevante.

Una vez que se escriben las restricciones de acuerdo al análisis realizado se procede a especificar la cobertura. Escribir adecuadamente las condiciones a cubrir es esencial para que los objetivos sean cubiertos satisfactoriamente. Los grupos y puntos de cobertura van a recopilar los datos del comportamiento del diseño de acuerdo a los estímulos generados con las restricciones.

El proceso de mejora se da una vez que se realiza la primera ejecución de la verificación y con ayuda de la información recabada mediante los grupos, puntos de cobertura y sus respectivos ‘bins’. El reporte de cobertura provee un informe de cuanto se han logrado los objetivos especificados. Si estos no se cumplen según los requerimientos se debe realizar un nuevo análisis para encontrar donde y porque las condiciones han podido fallar.

Esto puede llevar a una modificación en las restricciones o en las especificaciones de la cobertura. Si se da en las restricciones a veces es

necesario ajustar también la cobertura, de forma que los estímulos agregados o eliminados sean considerados. Si la modificación ocurre en la cobertura puede cambiar la forma en la que la información es recopilada, dando mayor importancia a algunos puntos a cubrir o reemplazando el valor de peso de un grupo o punto a la cobertura total.

El proceso continúa hasta que la verificación alcance los objetivos deseados por el verificador, quien a su vez debe siempre tener en cuenta los objetivos del diseñador.

En general, la verificación de un diseño involucra grandes grupos de personas que buscan que este procedimiento sea eficaz, logrando así reducir el tiempo en que se puede saber si un diseño o no es factible para su producción en silicio y los recursos computacionales que se utilizan a lo largo del trabajo de verificación.

3.4.1 Generación aleatoria de restricciones

En el apartado anterior se menciona la importancia de generar restricciones con las que se puedan alcanzar de manera más eficiente los objetivos de verificación, para detallar mejor este tema suponga un sumador con operandos de entrada de 32 bits como el de la figura 3.8.

Figura 3.8 Sumador de 32 Bits.

En un sumador de 32 bits es posible realizar 2^{32} combinaciones por operando ‘a’ y ‘b’, lo cual da como resultado un total de 2^{64} sumas que son aproximadamente $16x10^{18}$ combinaciones de entradas a sumar que son reflejadas en ‘c’. Si este proceso se realiza a la velocidad de una suma por nano - segundo tardaría aproximadamente $16x10^{9}$ segundos, lo que equivale a aproximadamente 507 años.

En la actualidad hay computadoras con altas velocidades que podrían reducir este tiempo considerablemente, sin embargo, nótese que un sumador es un diseño con escasa complejidad y probarlo exhaustivamente resulta impráctico, de ahí la importancia de una estrategia de verificación que sea eficaz, es decir, que pueda comprobar las características relevantes del diseño y aquellos casos que pueden ser difíciles de ocurrir. Restringir las condiciones permite ser específico en los casos de importancia y encontrar soluciones a las fallas en menor tiempo debido a que el verificador puede asegurarse de pasar al menos una vez por todos los casos o pasar únicamente por casos determinados con determinada frecuencia.

También es importante recordar que cuando se habla de un número aleatorio se están utilizando realmente los pseudoaleatorios que se pueden generar con ayuda de los algoritmos o herramientas del lenguaje que se emplea, lo cual facilita definir las características o frecuencia de los casos a probar.

En la figura 3.1 de la sección 3.2 se observa que cada unidad de ejecución aritmético – lógica posee su propio generador aleatorio de instrucciones, lo que da libertad de probar las unidades de forma independiente.

De acuerdo al diagrama de la figura 3.7 donde se habla sobre la estrategia de verificación, se plantean las restricciones con base al análisis del modelo comportamental. El módulo de generación aleatoria es donde se especifican las restricciones con que se van a generar los parámetros de una instrucción.

En la figura 3.9 se muestra un diagrama de orden de generación de estos parámetros, el primero para la unidad de generación aleatoria es el código de operación, una vez que está definido se puede generar el tipo de operación teniendo en cuenta la tabla 3.3 donde se encuentra el conjunto de instrucciones disponibles. Según el tipo de operación se generan las localidades de memoria o registros y el dato y nivel de memoria o inmediato si se requieren.

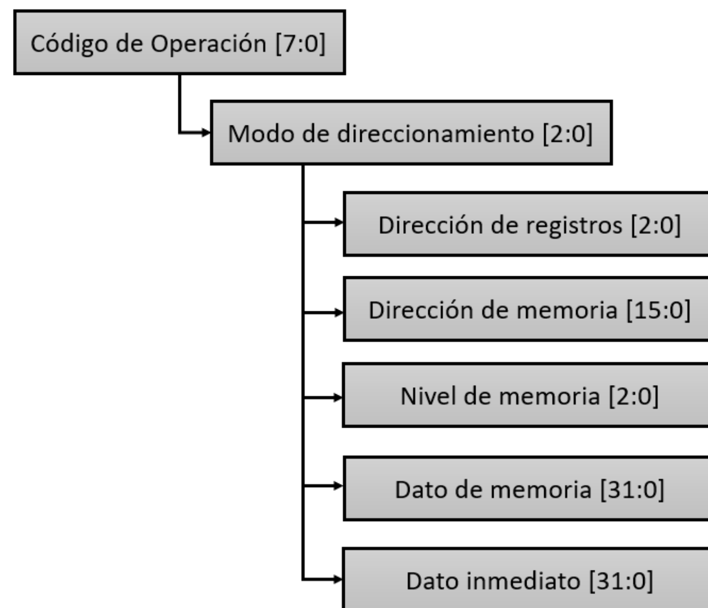


Figura 3.9 Prioridad en la generación aleatoria.

La generación de un parámetro dependiente de otro es posible mediante las herramientas de SystemVerilog que es el lenguaje de verificación utilizado. Por ejemplo, en el listado de código 3.1 se especifica con las palabras

reservadas *‘solve’* y *‘before’* la prioridad de generar el código de operación de la unidad de ejecución 1 sobre el tipo de operación en la misma unidad.

Listado de código 3.1 Ejemplo del uso de las palabras reservadas ‘solve’ y ‘before’.

```
constraint Cod_Eu1 {solve opcode_Eu1 before type_op_Eu1;}
```

Capítulo 4

Resultados

4.1.1 Modelo comportamental propuesto

Como se menciona en el capítulo 3 apartado 3.1 el diseño a verificar es el modelo comportamental de un procesador con múltiples unidades de ejecución. La estructura final se aprecia en la figura 4.1 en un diagrama RTL sintetizado en Vivado de Xilinx.

Los componentes Eu1 y Eu2 son iguales, en la figura 4.2 se presenta el componente EU1 de la figura 4.1 a detalle, como se puede apreciar contiene una ALU que recibe los operandos fuente y destino según un multiplexor de control, lo que corresponde con la figura 3.5 del apartado 3.2.7 donde se ejemplifica la ejecución de una instrucción.

Cada uno de los componentes EU recibe los datos correspondientes desde el formato de instrucción, cuyos campos detallados en el apartado 3.2.5 en la figura 3.3 se producen en el módulo de generación aleatoria con ayuda del ambiente de verificación donde se crean las transacciones. Estos campos corresponden al código de operación, tipo de operación, los datos de los operandos y el nivel de memoria del que proceden, así como el parámetro de

µops. Una vez disponibles los datos se ejecuta la instrucción según las µops que se indiquen.

En la figura 4.2 se observa también el elemento LU que corresponde a la unidad de carga de datos *'Load'* y el elemento denominado 'cont_level_uops' que es el encargado de proporcionar los retardos en los niveles de memoria y ejecución de µops de acuerdo a los parámetros del módulo de generación aleatoria.

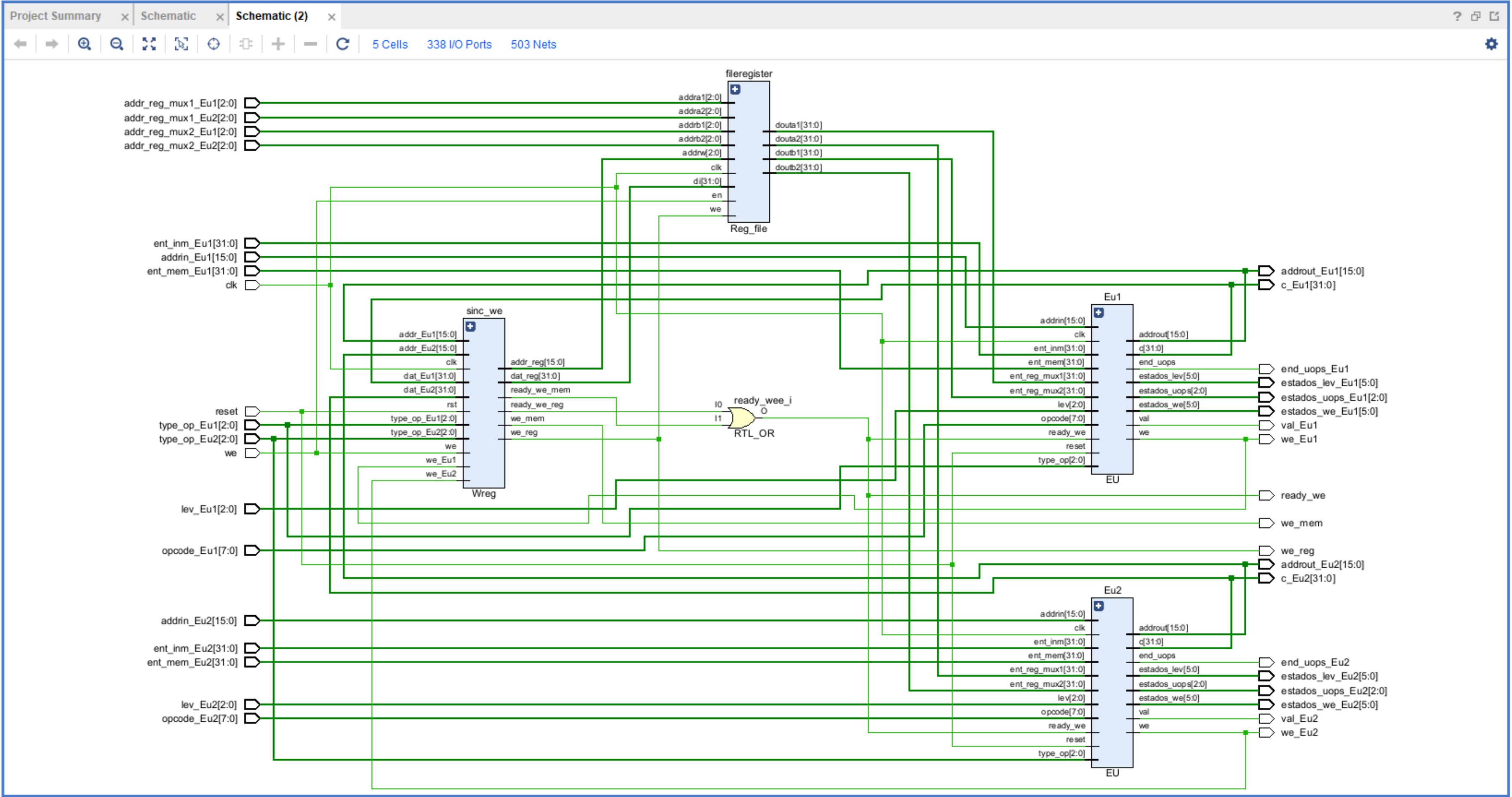


Figura 4.1 Diagrama RTL del modelo comportamental derivado del código.

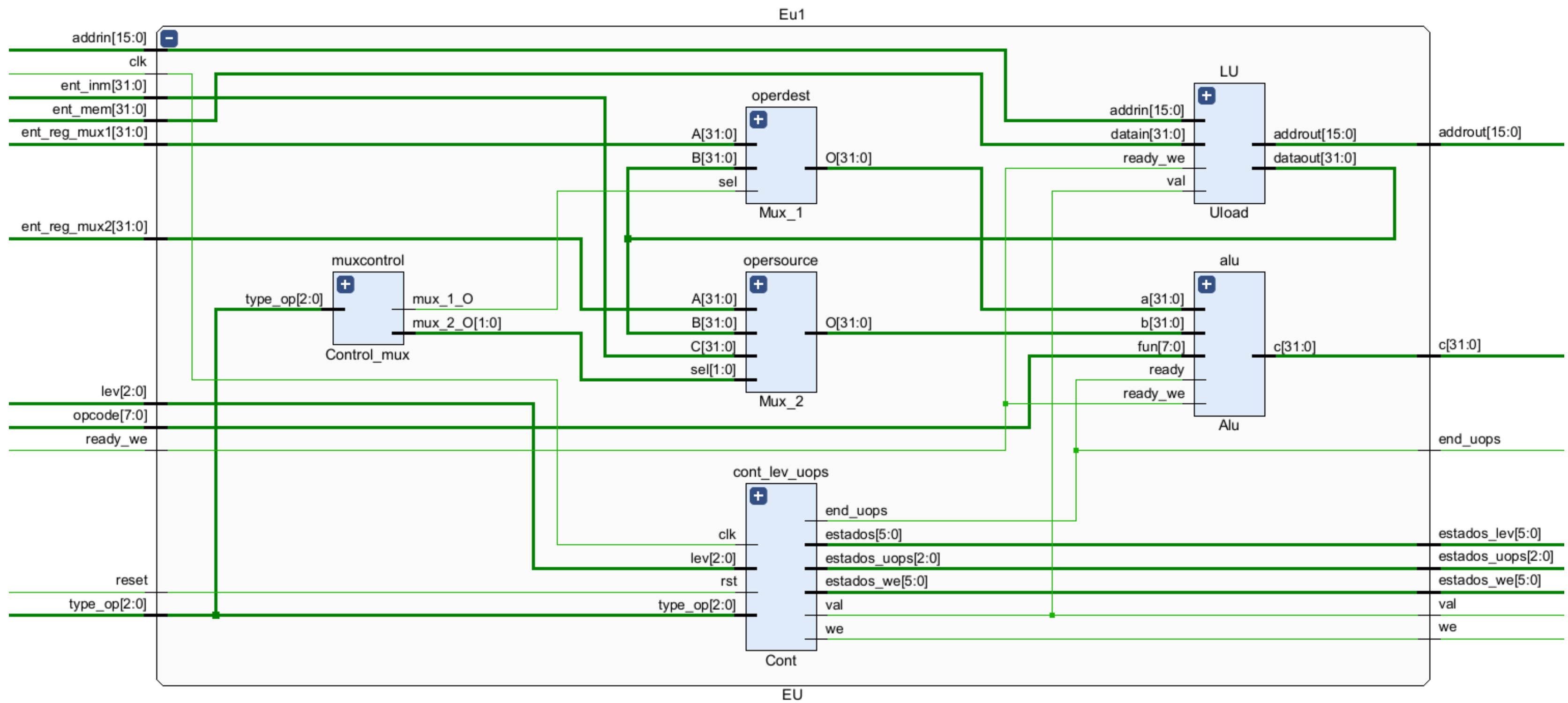


Figura 4.2 Diagrama detallado del módulo ‘EU1’.

En la figura 4.3 se observa el ‘file register’ que corresponde a los registros de propósito general que como se menciona en el apartado 3.2.2 son 8 registros de 32 bits cada uno. Este módulo tiene la capacidad de realizar 4 lecturas a la vez y solo una escritura que se habilita con las variables de entrada ‘en’ y ‘we’.

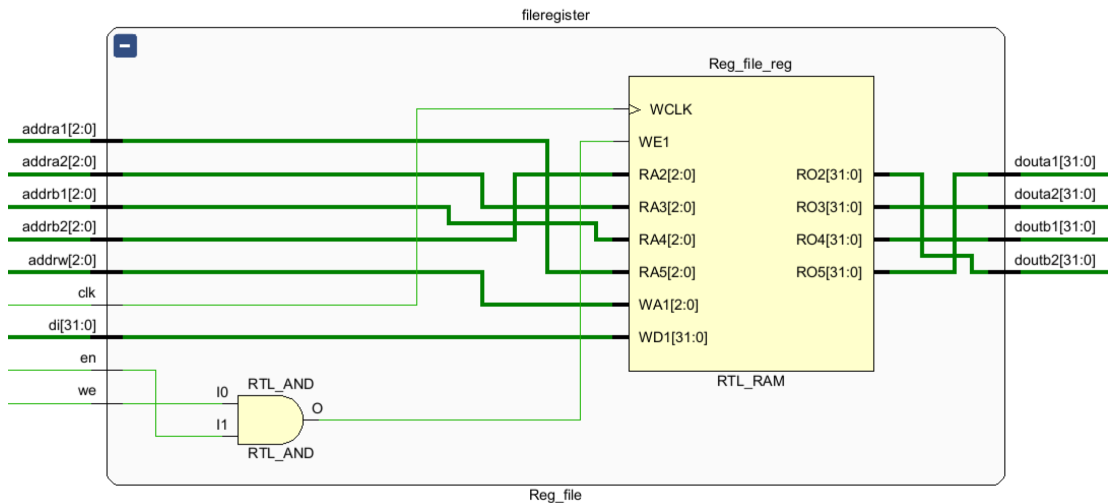


Figura 4.3 Detalle del módulo ‘File_Register’.

El módulo ‘sinc_we’ de la figura 4.4 es el encargado de sincronizar el almacenamiento. Para cada una de las unidades de ejecución ‘Eu1’ y ‘Eu2’ existe un componente ‘WEu’ que administra las solicitudes de escritura, también se puede observar un componente ‘we_memoria’ y un componente ‘we_registros’ que atienden dichas solicitudes según la instrucción que se ejecuta. Una de las características importantes de este módulo es que cuando existen dos solicitudes de memoria al mismo tiempo se realiza la escritura solicitada por una unidad y pasado un ciclo de reloj se efectúa la escritura de la otra siempre teniendo en cuenta el nivel de memoria al que corresponden. En la figura 4.5 se aprecia el diagrama de señales de la solicitud de escritura doble.

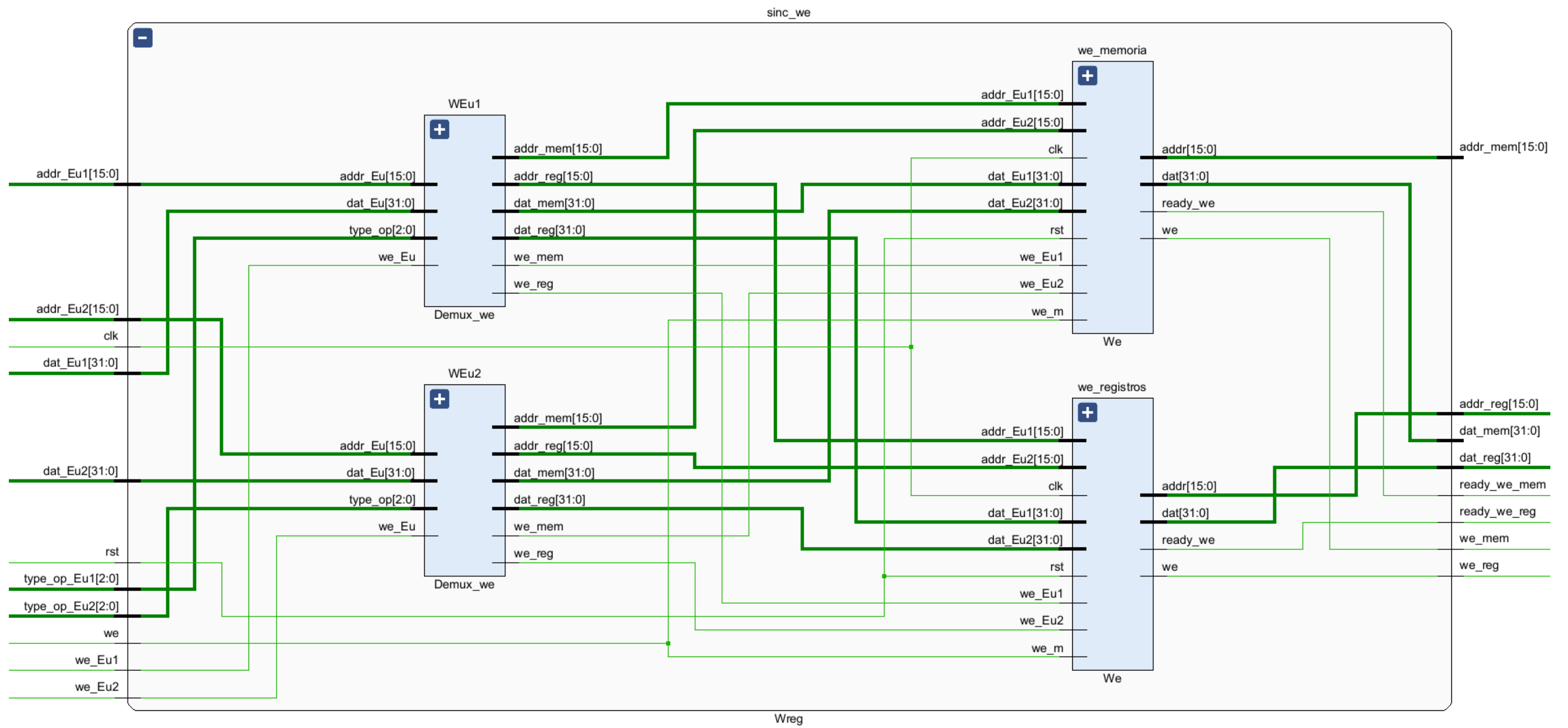


Figura 4.4 Detalle del módulo 'Sinc_we'.

La primera unidad de ejecución desea escribir el dato 0hCCCCCCCC en la localidad 0b1111 de la memoria, mientras que la unidad 2 requiere escribir el dato 0heeeeeeef en la localidad 0b2222, en el primer ciclo de reloj se atiende la solicitud de la unidad 1 y pasado un ciclo la de la unidad 2.

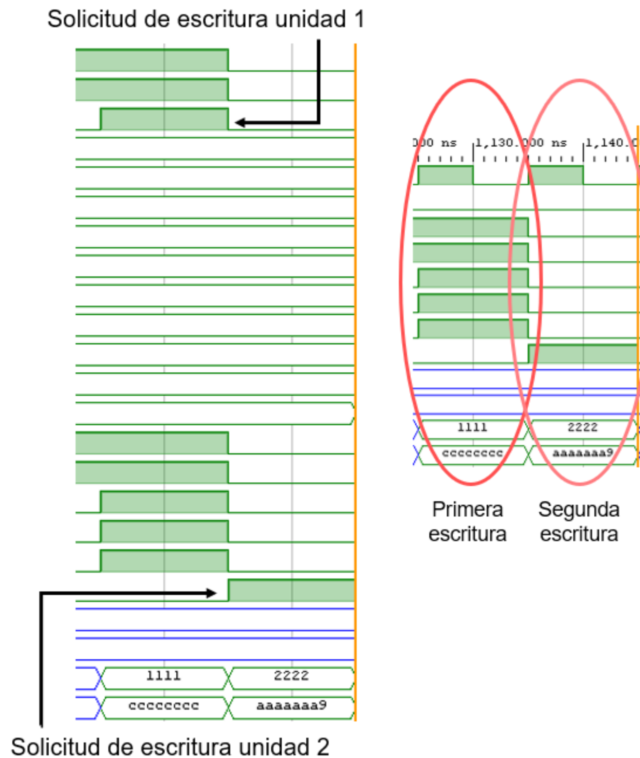


Figura 4.5 Solicitud de escritura por dos unidades de ejecución al mismo tiempo.

En la figura 4.6 se muestra el proceso de ejecución de una instrucción con almacenamiento en memoria en cada unidad de las dos que conforman el modelo a verificar. Dado que los operandos vienen de memoria se tiene un tiempo de carga dependiendo del nivel del que proceden, una vez que están disponibles se establece en ‘1’ una señal que indica que puede iniciar la ejecución y por tanto empieza el tiempo de μops, cuando este tiempo (en ciclos de reloj) concluye se establece en ‘1’ otra señal que indica que es momento de realizar la escritura en la dirección que corresponda. Por último, se indica cuando la escritura ya se ha realizado.

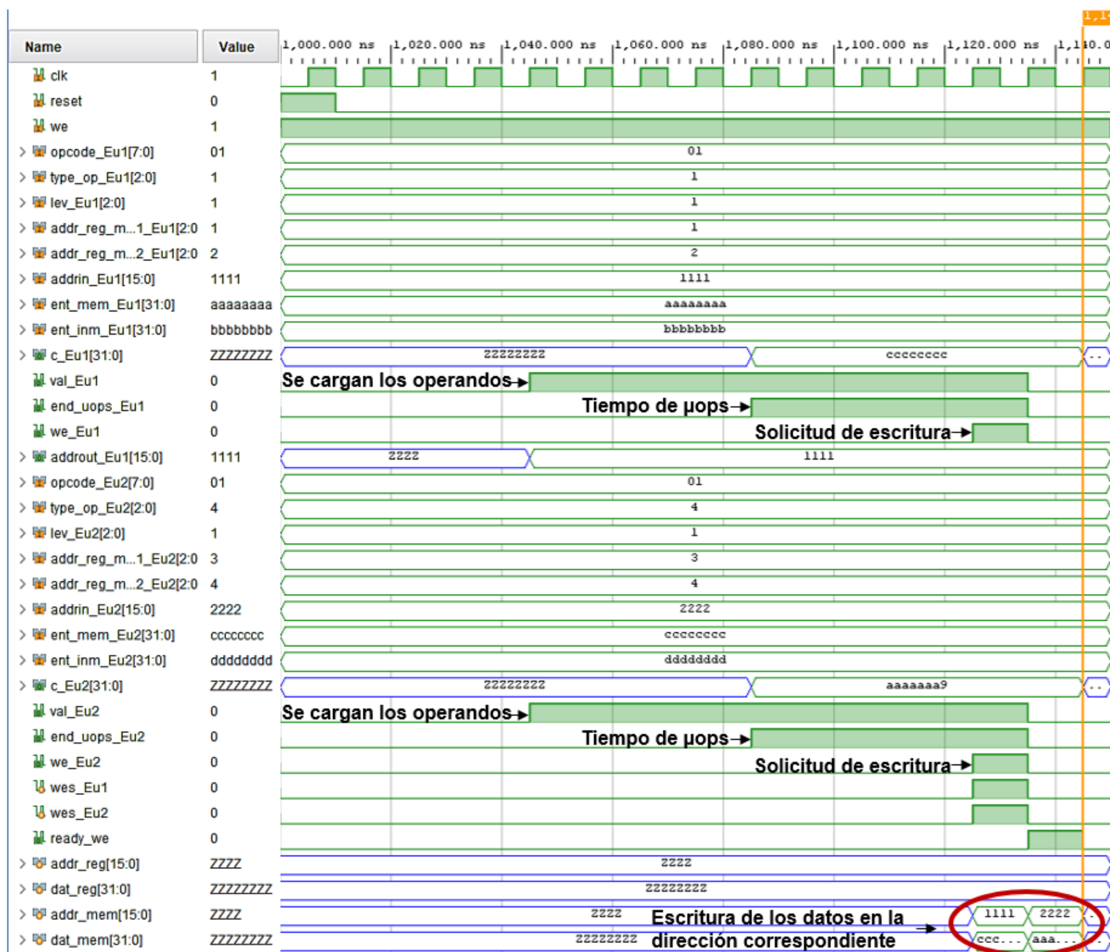


Figura 4.6 Simulación de ejecución de una instrucción con escritura a memoria.

4.2 Verificación funcional del modelo comportamental

El diagrama RTL del modelo comportamental derivado del código de la figura 4.1 contiene dos unidades aritmético – lógicas, dos unidades 'Load' para carga de datos y una unidad 'Store' para almacenamiento y es el dispositivo que se conecta al ambiente de verificación con las modificaciones de la figura 3.6 donde se observa que cada unidad aritmética – lógica tiene su propio *'Generator', 'Driver', Monitor', 'Scoreboard'* y la interfaz de conexión. En la figura 4.7 se presenta el diagrama de conexiones entre el dispositivo bajo verificación y el ambiente de verificación.

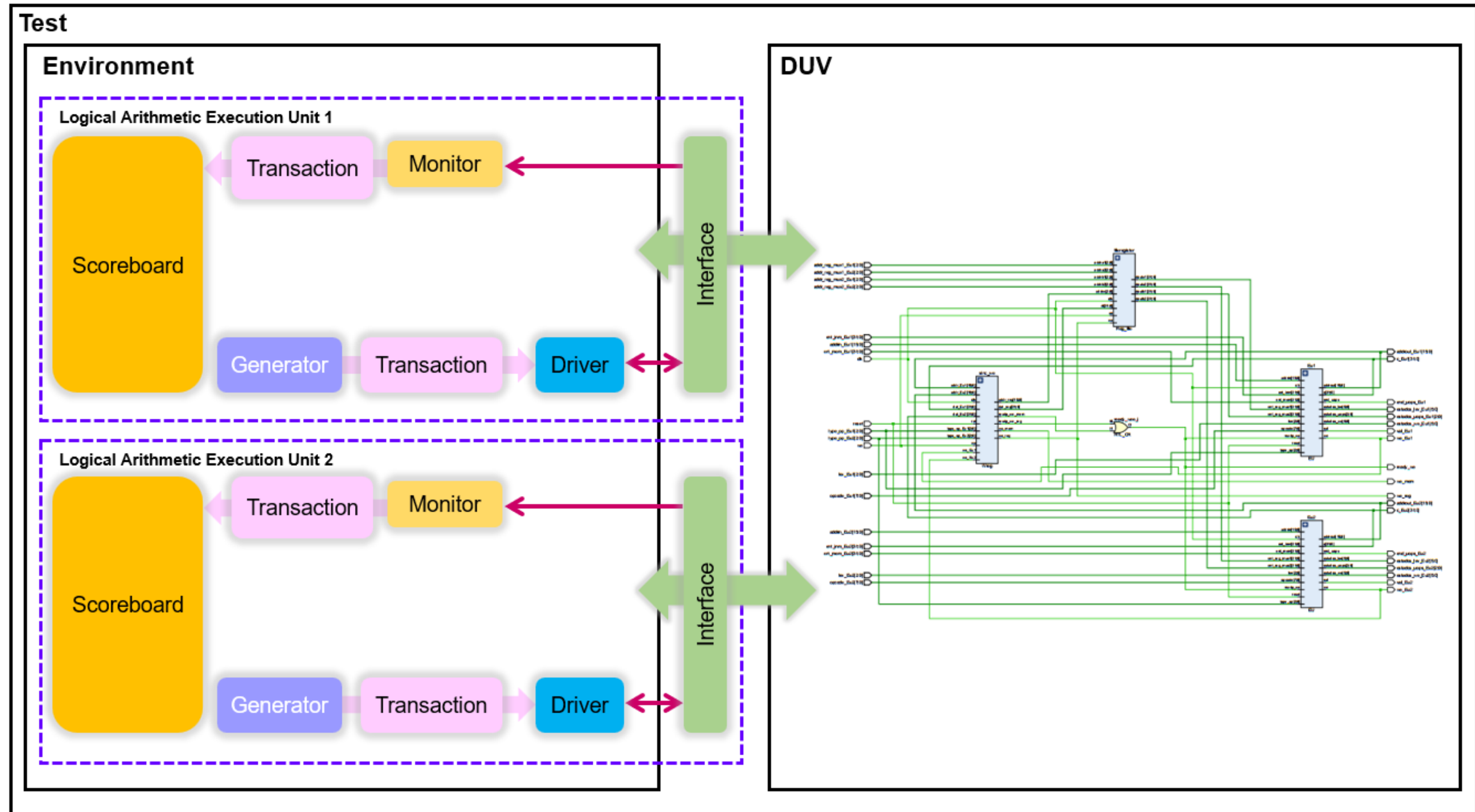


Figura 4.7 Diagrama del ambiente de verificación y dispositivo bajo verificación conectados.

4.2.1 Restricciones propuestas

Con base en el conjunto de instrucciones de la tabla 3.3 de la sección 3.2.6 se propone probar instrucciones en general que cubran todos los tipos de operaciones disponibles, usando una distribución en los niveles de memoria que priorice el nivel 1 de caché, también se propone tener rangos alto, medio y bajo en las localidades de memoria.

Otros casos de restricciones son donde solo se ejecuten instrucciones que escriban a registros o instrucciones que solo escriban a memoria, de esta manera se tendrá un mayor control de los grupos y puntos de cobertura.

4.2.2 Plan de verificación

En la sección anterior se plantea el utilizar rangos en las localidades de memoria y la distribución de los niveles de esta. Para lograr un porcentaje satisfactorio en la cobertura se requiere escribir correctamente los grupos y puntos de cobertura, a continuación, se nombran algunas estrategias usadas en la cobertura.

- 'Bins' para cubrir cada rango de memoria contemplado (alto, medio, bajo)
- Distribuir los 'bins' según los requerimientos de cobertura especificados en las restricciones.
- Corroborar la distribución de memoria propuesta.
- Controlar el número de ocurrencias con que cada punto se considera cubierto.
- Control del peso de cada punto de cobertura.

4.2.3 Caso de instrucciones generales

En la sección 4.2.1 se menciona que uno de los casos a probar es donde se generen instrucciones en general, es decir, pueden ser ejecutadas todas las instrucciones de la tabla 3.3 en el apartado del catálogo de instrucciones disponibles. Las restricciones que se consideran en esta ejecución son la distribución de los niveles de memoria, que son nivel 0 (RAM) 1, 2, 3 de memoria caché y 4 que corresponde a registros, dado que en un proceso real es fundamental que los datos estén disponibles lo más rápido posible se prioriza el nivel 1 de memoria caché y las instrucciones que operan sobre registros, también se considera un nivel bajo en los domicilios de memoria. Para la restricción de memoria se utiliza únicamente la palabra reservada 'dist' como en el listado de código 4.1 donde 'lev_Eux' corresponde a la variable de nivel en la unidad de ejecución, la 'x' en los nombres de variables e interfaces hace alusión a la similitud en las unidades Eu1 y Eu2.

Listado de código 4.1 Ejemplo de uso de la palabra reservada 'dist'.

```
constraint mem_lev_Eux {lev_Eux dist {0,1,2,3,4};}
```

En la figura 4.8 se presenta un reporte de cobertura donde se observa un ejemplo del caso propuesto de instrucciones generales. En este reporte hay cuatro grupos de cobertura, el primero llamado 'istruc_type' pretende recabar la información de cuantas instrucciones que operan a registros (bin 'Reg') o a memoria (bin 'Mem') se ejecutan, siendo dos el número mínimo que cada uno debe ser cubierto.

El segundo grupo 'Istruc_lev_x' contiene la información de las instrucciones que contienen registros y que son de nivel 1 de memoria caché, cada uno debe ser cubierto al menos cinco veces, la variable de interés para la detección del nivel de memoria es 'lev_Eux'. El tercer grupo cubre los niveles

2, 3 y RAM con un número mínimo de dos veces a ser cubiertos. La separación de la información de los niveles de memoria tiene como principal objetivo el de controlar el número de veces que cada grupo y punto se cubren para obtener altos porcentajes en la cobertura.

El cuarto grupo ‘Address_mem’ corresponde al nivel bajo en los domicilios de memoria. Las especificaciones de los puntos de cobertura descritos para los niveles de memoria se encuentran en el listado de código 4.2.

Listado de código 4.2 Puntos de cobertura para el caso de restricciones en general en niveles de memoria.

```
Istruc_levx: coverpoint i_intf_Eux.lev_Eux
    {
      option.weight = 30;
      option.at_least = 5;
      bins registros = {4};
      bins lev1  = {1};
    }
Istruc_lev: coverpoint i_intf_Eux.lev_Eux
    {
      option.weight = 20;
      option.at_least = 2;
      bins lev2  = {2};
      bins lev3  = {3};
      bins lev_RAM  = {0};
    }
```

```
====================================================================
|          Covergroup             |   Hits   |  Goal /  |  Status   |
|                                 |          | At Least |           |
====================================================================
| TYPE /tbench_top/EU1            |  78.333% | 100.000% | Uncovered |
====================================================================
| INSTANCE Eu1_cov                |  78.333% | 100.000% | Uncovered |
|---------------------------------|----------|----------|-----------|
| COVERPOINT Eu1_cov::Istruc_type | 100.000% | 100.000% | Covered   |
|---------------------------------|----------|----------|-----------|
| bin Reg                         |       13 |        2 | Covered   |
| bin Mem                         |        5 |        2 | Covered   |
|---------------------------------|----------|----------|-----------|
| COVERPOINT Eu1_cov::Istruc_lev1 |  50.000% | 100.000% | Uncovered |
|---------------------------------|----------|----------|-----------|
| bin registros                   |       11 |        5 | Covered   |
| bin lev1                        |        2 |        5 | Uncovered |
|---------------------------------|----------|----------|-----------|
| COVERPOINT Eu1_cov::Istruc_lev  |  66.666% | 100.000% | Uncovered |
|---------------------------------|----------|----------|-----------|
| bin lev2                        |        1 |        2 | Uncovered |
| bin lev3                        |        2 |        2 | Covered   |
| bin lev_RAM                     |        2 |        2 | Covered   |
|---------------------------------|----------|----------|-----------|
| COVERPOINT Eu1_cov::Address_mem | 100.000% | 100.000% | Covered   |
|---------------------------------|----------|----------|-----------|
| bin low                         |        6 |        2 | Covered   |
====================================================================
```

Figura 4.8 Cobertura del caso de instrucciones generales.

Se puede apreciar en el reporte de la figura 4.8 que los *'bins'* correspondientes al caso de nivel 1 y dos de memoria no están cubiertos y el impacto en el porcentaje de cobertura final es tal que solo se alcanza un 78.3% del 100% requerido.

En el apartado 3.4 sobre la estrategia de verificación a seguir se habla sobre la reescritura de puntos y grupos de cobertura con el fin de controlar la información que reúnen y la cantidad de veces que se deben cubrir, para el caso de las instrucciones en general los puntos de cobertura para los niveles de memoria se reescriben como en el listado de código 4.3.

Listado de código 4.3 Puntos de cobertura modificados para el caso de restricciones en general en niveles de memoria.

```
Istruc_levx: coverpoint i_intf_Eux.lev_Eux
    {
      option.weight = 30;
      option.at_least = 5;
      bins registros = {4};
      bins lev1  = {1};
    }
Istruc_lev: coverpoint i_intf_Eux.lev_Eux
    {
      option.weight = 20;
      option.at_least = 2;
      bins lev2  = {2};
      bins lev3  = {3};
    }
Istruc_RAM: coverpoint i_intf_Eux.lev_Eux
    {
      option.weight = 10;
      option.at_least = 2;
      bins lev_RAM  = {0};
    }
```

También se plantea la modificación de las restricciones de las transacciones para ayudar a mejorar la cobertura y aumentar el número de casos de ejecución si es necesario. En la figura 4.9 se muestra el reporte de cobertura después de haber realizado cambios en los grupos y puntos de cobertura. El nivel de memoria 1 y 2 ya están cubiertos y se encuentra en otro grupo llamado 'Istruc_RAM' el nivel de memoria RAM. el porcentaje de cobertura alcanzado es de 100%.

```
=====================================================================
|          Covergroup              |   Hits   |  Goal /  | Status  |
|                                  |          | At Least |         |
=====================================================================
| TYPE /tbench_top/EU1             | 100.000% | 100.000% | Covered |
=====================================================================
| INSTANCE Eu1_cov                 | 100.000% | 100.000% | Covered |
|----------------------------------|----------|----------|---------|
| COVERPOINT Eu1_cov::Istruc_type  | 100.000% | 100.000% | Covered |
|----------------------------------|----------|----------|---------|
| bin Reg                          |       29 |        2 | Covered |
| bin Mem                          |       18 |        2 | Covered |
|----------------------------------|----------|----------|---------|
| COVERPOINT Eu1_cov::Istruc_lev1  | 100.000% | 100.000% | Covered |
|----------------------------------|----------|----------|---------|
| bin registros                    |       26 |        5 | Covered |
| bin lev1                         |       13 |        5 | Covered |
|----------------------------------|----------|----------|---------|
| COVERPOINT Eu1_cov::Istruc_lev   | 100.000% | 100.000% | Covered |
|----------------------------------|----------|----------|---------|
| bin lev2                         |        3 |        2 | Covered |
| bin lev3                         |        2 |        2 | Covered |
|----------------------------------|----------|----------|---------|
| COVERPOINT Eu1_cov::Istruc_RAM   | 100.000% | 100.000% | Covered |
|----------------------------------|----------|----------|---------|
| bin lev_RAM                      |        3 |        2 | Covered |
|----------------------------------|----------|----------|---------|
| COVERPOINT Eu1_cov::Address_mem  | 100.000% | 100.000% | Covered |
|----------------------------------|----------|----------|---------|
| bin low                          |       26 |        2 | Covered |
=====================================================================
```

Figura 4.9 Reporte de Cobertura del caso de instrucciones en general con modificaciones en los grupos de cobertura.

La restricción en memoria para el nuevo reporte considera una distribución pondera de los niveles de memoria como se propone en el listado de código 4.4.

Listado de código 4.4 Restricciones en la variable de nivel para el caso de restricciones en general.

```
Constraint mem_lev_Eux {lev_Eux
dist{0:=2,1:=3,2:=2,3:=2,4:=1};}
```

4.2.4 Caso de instrucciones que escriben a registros

Para este caso particular se deben ejecutar únicamente aquellas instrucciones cuyo resultado sea escrito en registros, lo que permite utilizar las siguientes combinaciones de operandos según la tabla 3.3 del catálogo de instrucciones disponibles en el modelo comportamental.

- Registro a registro
- Memoria a registro
- Inmediato a registro
- Registro (instrucciones de un solo operando)

Para escribir las restricciones de forma correcta y alcanzar la cobertura deseada se debe consultar también la tabla 3.2 del apartado 3.2.5 donde se encuentran los códigos en binario para cada tipo de instrucción antes mencionada. Los puntos de cobertura propuestos para las unidades Eu1 y Eu2 son los del listado de código 4.5, donde la ‘x’ en los nombres usados para las variables de las interfaces y unidades se refiere a que los grupos son iguales para ambas unidades.

Listado de código 4.5 Puntos de cobertura para el caso de restricciones que escriben a registros

```
   Istruc_type: coverpoint i_intf_Eux.type_op_Eux
     {
       option.weight = 50;
       option.at_least = 10;
       bins Reg = {0,2,3,5};
     }
  Istruc_lev: coverpoint i_intf_Eu1.lev_Eux
     {
       option.weight = 50;
       option.at_least = 15;
       bins registros = {4};
     }
```

Las restricciones en las transacciones consideran la jerarquía de la figura 3.9, donde se genera primero el código de instrucción con una distribución ponderada y con base en este se resuelve la generación del tipo de operación con las palabras reservadas *'solve'* y *'before'* como muestra el listado de código 4.6, además en el listado 4.7 se aprecia que únicamente se realiza la generación de tipos de escritura a registros según los identificadores de la tabla 3.2, donde los códigos son 0, 2, 3 y 5.

Listado de código 4.6 Restricciones para el caso de instrucciones que escriben a registros.

```
constraint codigo_Eux {opcode_Eux dist {'h05: =1,'h01:
=2,'h03: =2,'h25: =1,'h21: =2,'hff: =2,'h89: =2,'hc7:
=2,'hf7: =2,'h0d: =1,'h09: =2,'h0b: =2,'h2d: =1,'h81:
=2,'h29: =2,'h2b: =2,'h35: =1,'h31: =2,'h33: =2};}
constraint typ_op_Eux {type_op_Eux inside {0,1,2,3,4,5,6};}
constraint sab_Eux {solve opcode_Eux before type_op_Eux;}
```

Listado de código 4.7 Restricción para generación de tipos que escriben a registros.

```
constraint options_Eu1{
    (opcode_Eu1 =='h05) -> (type_op_Eu1 == 3);
    (opcode_Eu1 =='h01) -> (type_op_Eu1 inside {0});
    (opcode_Eu1 =='h03) -> (type_op_Eu1 inside {0});
    (opcode_Eu1 =='h25) -> (type_op_Eu1 == 3);
    (opcode_Eu1 =='h21) -> (type_op_Eu1 inside {0});
    (opcode_Eu1 =='hff) -> (type_op_Eu1 inside {6});
    (opcode_Eu1 =='h89) -> (type_op_Eu1 inside {0});
    (opcode_Eu1 =='hc7) -> (type_op_Eu1 inside {3});
    (opcode_Eu1 =='hf7) -> (type_op_Eu1 inside {5});
    (opcode_Eu1 =='h0d) -> (type_op_Eu1 == 3);
    (opcode_Eu1 =='h09) -> (type_op_Eu1 inside {0});
    (opcode_Eu1 =='h0b) -> (type_op_Eu1 inside {0});
    (opcode_Eu1 =='h2d) -> (type_op_Eu1 == 3);
    (opcode_Eu1 =='h81) -> (type_op_Eu1 inside {3});
    (opcode_Eu1 =='h29) -> (type_op_Eu1 inside {0});
    (opcode_Eu1 =='h2b) -> (type_op_Eu1 inside {0,2});
    (opcode_Eu1 =='h35) -> (type_op_Eu1 == 3);
    (opcode_Eu1 =='h31) -> (type_op_Eu1 inside {0});
    (opcode_Eu1 =='h33) -> (type_op_Eu1 inside {0,2});
           }
```

En la figura 4.10 se muestra el reporte de cobertura obtenido donde se aprecian los dos grupos de cobertura para la unidad de ejecución EU1, los resultados son idénticos para la unidad EU2, el primer grupo 'Istruc_type' cubre los tipos de instrucción de la tabla 3.2 donde se escribe en registros únicamente, mientras que el segundo grupo llamado 'Istruc_lev' cubre el nivel de dónde vienen los datos, es decir, nivel 0 (RAM) 1, 2, 3 de memoria caché y 4 que corresponde a registros. En casos aislados como el de este ejemplo es mucho más fácil definir aquellas restricciones que van a permitir alcanzar un mejor porcentaje de cobertura e identificar las dependencias entre las variables involucradas en este proceso, como puede observarse el porcentaje de cobertura alcanzado es de 100%.

```
===================================================================
|           Covergroup             |   Hits    |  Goal /  | Status   |
|                                  |           | At Least |          |
===================================================================
| TYPE /tbench_top/EU1             | 100.000% | 100.000% | Covered |
===================================================================
| INSTANCE Eu1_cov                 | 100.000% | 100.000% | Covered |
|----------------------------------|----------|----------|---------|
| COVERPOINT Eu1_cov::Istruc_type  | 100.000% | 100.000% | Covered |
|----------------------------------|----------|----------|---------|
| bin Reg                          |       21 |       10 | Covered |
|----------------------------------|----------|----------|---------|
| COVERPOINT Eu1_cov::Istruc_lev   | 100.000% | 100.000% | Covered |
|----------------------------------|----------|----------|---------|
| bin registros                    |       21 |       15 | Covered |
===================================================================
```

Figura 4.10 Reporte de cobertura del caso de instrucciones de escritura en registros.

Con ayuda del ambiente de verificación es posible monitorear los datos de entrada y salida de cada unidad y compararlos con los resultados correctos a obtener, tanto para la tarea de cobertura como para la tarea de monitoreo se hace énfasis en la correcta conexión de las señales en las interfaces de cada dispositivo como se muestra en la figura 2.3 del apartado 2.6 de las características del ambiente de verificación.

En la figura 4.11 se presentan los resultados obtenidos de la transacción marcada con el número 1, el código ‘0h0d’ corresponde a una operación lógica OR que al ser del tipo ‘0b11’ o ‘3’ se refiere al tipo inmediato a registro, el registro es el ‘0’ indicado como ‘addr_reg_mux1’ que por omisión tiene almacenado el mismo valor de su índice, el dato inmediato generado es ‘0hd9e30c3d’.

Con el número 2 se indica como llegan los datos hasta el ‘Driver’ y después con el número 3 hasta el ‘Monitor’, por último, en el ‘Scoreboard’ señalado con un número 4 se corrobora que todos los datos coinciden y que la operación tiene un resultado correcto, de no ser así se debe desplegar un letrero con la información correspondiente.

1
```
-------------------------
- [ Generator_Eu1 ]
-------------------------
- value of opcode (Hex)= 0d
- type op (Bin) = 011
- level (Dec) = 4
- addr_reg_mux1 (Hex) = 0
- addr_reg_mux2 (Hex) = 4
- operando_inmediato (Hex) = d9e30c3d
- c (Hex) = 00000000
-------------------------
```

2
```
-------------------------
- Transaction Eu1 No = 3
-------------------------
- [ Driver_Eu1 ]
-------------------------
- value of opcode (Hex)= 0d
- type op (Bin) = 011
- level (Dec) = 4
- addr_reg_mux1 (Hex) = 0
- addr_reg_mux2 (Hex) = 4
- operando_inmediato (Hex) = d9e30c3d
- c (Hex) = d9e30c3d
-------------------------
```

3
```
-------------------------
- [ Monitor_Eu1 ]
-------------------------
- value of opcode (Hex)= 0d
- type op (Bin) = 011
- level (Dec) = 4
- addr_reg_mux1 (Hex) = 0
- addr_reg_mux2 (Hex) = 4
- operando_inmediato (Hex) = d9e30c3d
- c (Hex) = d9e30c3d
-------------------------
```

4
```
-------------------------
- [ Scoreboard_Eu1 ]
-------------------------
- value of opcode (Hex)= 0d
- type op (Bin) = 011
- level (Dec) = 4
- addr_reg_mux1 (Hex) = 0
- addr_reg_mux2 (Hex) = 4
- operando_inmediato (Hex) = d9e30c3d
- c (Hex) = d9e30c3d
-------------------------
```

Figura 4.11 Resultados de la ejecución de una instrucción que escribe a registros en 1) transacción, 2) ‘Driver’, 3) ‘Monitor’, 4) ‘Scoreboard’.

4.2.5 Caso de instrucciones que escriben en memoria

Al igual que en el apartado 4.2.4 de instrucciones que escriben en registros es necesario identificar las instrucciones de la tabla 3.3 que pueden ser ejecutadas con la característica de escritura a memoria. En el apartado 3.2.6 se especifica que no pueden ejecutarse instrucciones con dos operandos de memoria a la vez, acorde a la tabla 3.2 los tipos son los siguientes.

- Registro a memoria
- Inmediato a memoria
- Memoria (instrucciones de un solo operando)

Para la particularidad de escritura a memoria en este ejemplo se encuentran los puntos de cobertura para las unidades Eu1 y Eu2 en el listado de código 4.8, la 'x' en los nombres de interfaz unidad hace referencia a que los grupos y restricciones utilizadas son idénticos para las dos unidades.

Listado de código 4.8. Puntos de cobertura para el caso de instrucciones que escriben en memoria.

```
Istruc_type: coverpoint i_intf_Eux.type_op_Eux
    {
      option.weight = 50;
      option.at_least = 15;
      bins Mem  = {1,4,6};
    }
Istruc_lev_1: coverpoint i_intf_Eux.lev_Eux
    {
      option.weight = 30;
      option.at_least =10 ;
      bins lev1  = {1};
    }
Istruc_lev: coverpoint i_intf_Eu1.lex_Eux
    {
      option.weight = 20;
      option.at_least = 5;
      bins lev2  = {2};
```

```
    bins lev3  = {3};
    bins lev_RAM  = {0};
  }
```

Las restricciones consideradas en el caso de escritura a memoria también son generadas según la figura 3.9, dando prioridad al código de instrucción, una vez que esté resuelto se genera el tipo de acuerdo a la tabla 3.3 del conjunto de instrucciones disponibles en el generador usando las palabras reservadas *‘solve’* y *‘before’* como se observa en el listado de código 4.9.

Listado de código 4.9 Restricciones para el caso de instrucciones que escriben en memoria.

```
constraint codigo_Eux {opcode_Eux inside
{'h05,'h01,'h03,'h25,'h21,'hff,'h89,'hc7,'hf7,'h0d,'h09,'h0b,
'h2d,'h81,'h29,'h2b,'h35,'h31,'h33};}
constraint typ_op_Eux {type_op_Eux inside {0,1,2,3,4,5,6};}
constraint sab_Eux {solve opcode_Eux before type_op_Eux;}
```

En el listado 4.10 se aprecia que únicamente se realiza la generación de instrucciones de escritura a memoria de acuerdo a la tabla 3.3 del conjunto de instrucciones disponibles.

Listado de código 4.10 Restricciones para generación de instrucciones que escriben en memoria.

```
constraint codigo_Eu1 {opcode_Eu1 dist
{'h05:=0,'h01:=2,'h03:=0,'h25:=0,'h21:=2,'hff:=2,'h89:=2,'hc7
:=2,'hf7:=2,'h0d:=0,'h09:=2,'h0b:=0,'h2d:=0,'h81:=2,'h29:=2,'
h2b:=0,'h35:=0,'h31:=2,'h33:=0};}
```

Para una mayor certeza en la generación de instrucciones se utiliza también la restricción de tipos de solo escritura a memoria como se presenta en el listado de código 4.11, esta selección está basada en la tabla 3.3 de los códigos de identificación de tipos de operación, para este caso son los tipos 1, 4 y 6.

Listado de código 4.11 Fragmento de restricciones para generación de tipo de direccionamiento de instrucciones que escriben en memoria.

```
constraint options_Eu1{
  (opcode_Eu1 =='h01) -> (type_op_Eu1 inside {1});
  (opcode_Eu1 =='h21) -> (type_op_Eu1 inside {1});
  (opcode_Eu1 =='hff) -> (type_op_Eu1 inside {6});
  (opcode_Eu1 =='h89) -> (type_op_Eu1 inside {1});
  (opcode_Eu1 =='hc7) -> (type_op_Eu1 inside {4});
  (opcode_Eu1 =='hf7) -> (type_op_Eu1 inside {6});
  (opcode_Eu1 =='h09) -> (type_op_Eu1 inside {1});
  (opcode_Eu1 =='h81) -> (type_op_Eu1 inside {4});
  (opcode_Eu1 =='h29) -> (type_op_Eu1 inside {1});
  (opcode_Eu1 =='h31) -> (type_op_Eu1 inside {1});
        }
```

En la figura 4.12 se aprecia el reporte de cobertura obtenido para las instrucciones con escritura a memoria

En el primer grupo llamado 'Istruc_type' se reúne la información correspondiente a los tipos de operandos generados para la instrucción que de acuerdo con la tabla 3.2 del apartado 3.2.5 son los binarios '0b1', '0b4' y '0b110'.

Con el segundo grupo 'Istruc_lev_x' se pretende cubrir el nivel 1 de memoria caché debido a que las ocurrencias deben ser mayores que los niveles 2 y 3 de memoria caché y RAM, por lo que se estos se separan en un tercer grupo de cobertura llamado 'Istruc_lev', la prioridad de nivel 1 se asocia al proceso real de ejecución de una instrucción en un procesador, donde los datos deben estar a disposición con la mayor rapidez posible, es decir, en registros o en el nivel 1 de memoria caché.

Separar los diferentes niveles de memoria permite controlar la cantidad de veces o 'hits' que se deben cubrir en cada uno de ellos de manera independiente.

```
===================================================================
|           Covergroup             |   Hits   |  Goal /  | Status  |
|                                  |          | At Least |         |
===================================================================
| TYPE /tbench_top/EU1             | 100.000% | 100.000% | Covered |
===================================================================
| INSTANCE Eu1_cov                 | 100.000% | 100.000% | Covered |
|----------------------------------|----------|----------|---------|
| COVERPOINT Eu1_cov::Istruc_type  | 100.000% | 100.000% | Covered |
|----------------------------------|----------|----------|---------|
| bin Mem                          |       54 |       15 | Covered |
|----------------------------------|----------|----------|---------|
| COVERPOINT Eu1_cov::Istruc_lev_1 | 100.000% | 100.000% | Covered |
|----------------------------------|----------|----------|---------|
| bin lev1                         |       31 |       10 | Covered |
|----------------------------------|----------|----------|---------|
| COVERPOINT Eu1_cov::Istruc_lev   | 100.000% | 100.000% | Covered |
|----------------------------------|----------|----------|---------|
| bin lev2                         |       10 |        5 | Covered |
| bin lev3                         |        6 |        5 | Covered |
| bin lev_RAM                      |        7 |        5 | Covered |
===================================================================
| TYPE /tbench_top/EU2             | 100.000% | 100.000% | Covered |
===================================================================
| INSTANCE Eu2_cov                 | 100.000% | 100.000% | Covered |
|----------------------------------|----------|----------|---------|
| COVERPOINT Eu2_cov::Istruc_type  | 100.000% | 100.000% | Covered |
|----------------------------------|----------|----------|---------|
| bin Mem                          |       41 |       15 | Covered |
|----------------------------------|----------|----------|---------|
| COVERPOINT Eu2_cov::Istruc_lev_1 | 100.000% | 100.000% | Covered |
|----------------------------------|----------|----------|---------|
| bin lev1                         |       14 |       10 | Covered |
|----------------------------------|----------|----------|---------|
| COVERPOINT Eu2_cov::Istruc_lev   | 100.000% | 100.000% | Covered |
|----------------------------------|----------|----------|---------|
| bin lev2                         |        9 |        5 | Covered |
| bin lev3                         |       12 |        5 | Covered |
| bin lev_RAM                      |        6 |        5 | Covered |
===================================================================
```

Figura 4.12 Reporte de cobertura del caso de instrucciones con escritura a memoria.

En la figura 4.13 se encuentran indicados con el numero 1 los datos de una transacción con código de operación ‘0h21’ que según la tabla de instrucciones del apartado 3.2.6 corresponde a una operación lógica AND.

El tipo de operación es el binario ‘0b001’ o ‘1’ decimal por lo que se trata de una operación AND de registro a memoria, el dato ‘0h8dac4fa7’ proviene de nivel de memoria 2 y se encuentra en la localidad ‘0hafaa’. El registro a utilizar es el número 6 que tiene su propio índice almacenado y está identificado con el nombre ‘addr_reg_mux2’ que hace referencia al operando fuente.

1

```
-------------------------------------------
value of address (Hex) = afaa
dat (Hex) = 8dac4fa7
-------------------------------------------
-------------------------
- [ Generator_Eu1 ]
-------------------------
- value of opcode (Hex)= 21
- type op (Bin) = 001
- level (Dec) = 2
- addr_reg_mux1 (Hex) = 2
- addr_reg_mux2 (Hex) = 6
- operando_inmediato (Hex) = 77757175
- c (Hex) = 00000000
-------------------------
```

2

```
-------------------------
- Transaction Eu1 No = 1
-------------------------
- [ Driver_Eu1 ]
-------------------------
- value of opcode (Hex)= 21
- type op (Bin) = 001
- level (Dec) = 2
- addr_reg_mux1 (Hex) = 2
- addr_reg_mux2 (Hex) = 6
- operando_inmediato (Hex) = 77757175
- c (Hex) = 04244626
-------------------------
```

3

```
-------------------------
- [ Monitor_Eu1 ]
-------------------------
- value of opcode (Hex)= 21
- type op (Bin) = 001
- level (Dec) = 2
- addr_reg_mux1 (Hex) = 2
- addr_reg_mux2 (Hex) = 6
- operando_inmediato (Hex) = 77757175
- c (Hex) = 04244626
-------------------------
```

4

```
-------------------------
- [ Scoreboard_Eu1 ]
-------------------------
- value of opcode (Hex)= 21
- type op (Bin) = 001
- level (Dec) = 2
- addr_reg_mux1 (Hex) = 2
- addr_reg_mux2 (Hex) = 6
- operando_inmediato (Hex) = 77757175
- c (Hex) = 04244626
-------------------------
```

Figura 4.13 Resultados de la ejecución de una instrucción que escribe en memoria en 1) transacción, 2) ‘Driver’, 3) ‘Monitor’, 4) ‘Scoreboard’.

Con el número 2 se indica que la transacción está en el ‘Driver’ con los datos correctos, después pasa al ‘Monitor’ indicado con el número 3 y finalmente en el ‘Scoreboard’ se observa que los resultados coinciden.

4.2.6 Pruebas extendidas

Como se menciona anteriormente en este documento la verificación no solo tiene el objetivo de detectar errores en el diseño, sino tiene también el propósito de que el dispositivo que se está verificando cumpla correctamente con las especificaciones dadas.

Si bien la mayoría de las pruebas que se realizan tienen como objetivo lograr la cobertura de casos difíciles de alcanzar, es importante poner especial interés en la variabilidad de los datos con los que se prueba, un ejemplo claro de esto es el procesador Pentium de Intel que en 1994 tuvo un fallo en una operación de división de punto flotante que le costó a la empresa miles de millones de dólares [14] .

Dentro de las prácticas que se usan en verificación es usual la ejercitación del diseño realizando diferentes cantidades de iteraciones, esto ayuda a que exista una variación significativa en los datos, lo que permite también el control de estos, priorizando o descartando aquellos casos que el verificador considere apropiados según los resultados previamente obtenidos. Algunas de las pruebas de datos que se realizan son con datos intermedios, extremos, o realizando un corrimiento de unos y ceros en los bits de la variable.

En los apartados anteriores se describe el procedimiento para los casos de ejecución de instrucciones que solo escriben en memoria o solo a registros donde se utiliza un número limitado de iteraciones, a continuación, se describe la ejecución de un mayor número de iteraciones en los casos propuestos.

4.2.6.1 Instrucciones con escritura a registros.

Para las pruebas con más iteraciones en el caso de instrucciones que escriben a registros se agrega un nuevo grupo de cobertura denominado 'EUx_dat' con el que se pretende monitorear cuatro 'bins' para los datos generados en los domicilios de memoria. Los 'bins' son 'haaaaaaaa', 'h55555555', 'h00000000' y 'h11111111', los cuales representan variaciones en niveles '1' y '0' de cada uno de los 32 bits que conforman el dato, el grupo de cobertura se presenta en el listado de código 4.12. y es similar para las unidades EU1 y EU2.

Listado de código 4.12 Grupo de cobertura de datos (Instrucciones de escritura a registros).

```
covergroup EUx_dat () @(posedge i_intf_Eux.we_Eux);
  option.name = "Eux_cov_dat";
  option.per_instance = 1;
   option.goal         = 100;//objetivo de cobertura
   Dat_mem: coverpoint i_intf_Eux.ent_mem_Eux
  {
    option.weight = 100;
    option.at_least = 150;

    bins h55555555  = {'h55555555};
    bins haaaaaaaa  = {'haaaaaaaa};
    bins h00000000  = {'h00000000};
    bins h11111111  = {'h11111111};
  }
endgroup
```

En la figura 4.14 se observa el reporte de cobertura que se obtiene de la ejecución de 80 iteraciones en la unidad EU1 y 100 iteraciones para la unidad EU2, el reporte en 4.15 corresponde a 450 y 500 ejecuciones respectivamente, por último, en 4.16 se aprecia el reporte con 800 iteraciones para EU1 y 1000 para EU2, en todos se alcanza un porcentaje de cobertura de 100%.

COVERGROUP COVERAGE

a)

Covergroup	Hits	Goal / At Least	Status
TYPE /tbench_top/EU1	100.000%	100.000%	Covered
INSTANCE Eu1_cov	100.000%	100.000%	Covered
COVERPOINT Eu1_cov::Istruc_type	100.000%	100.000%	Covered
bin Reg	76	20	Covered
COVERPOINT Eu1_cov::Istruc_lev	100.000%	100.000%	Covered
bin registros	70	25	Covered
TYPE /tbench_top/EU1_dat	100.000%	100.000%	Covered
INSTANCE Eu1_cov_dat	100.000%	100.000%	Covered
COVERPOINT Eu1_cov_dat::Dat_mem	100.000%	100.000%	Covered
bin h55555555	20	15	Covered
bin haaaaaaaa	18	15	Covered
bin h00000000	19	15	Covered
bin h11111111	23	15	Covered

b)

Covergroup	Hits	Goal / At Least	Status
TYPE /tbench_top/EU2	100.000%	100.000%	Covered
INSTANCE Eu2_cov	100.000%	100.000%	Covered
COVERPOINT Eu2_cov::Istruc_type	100.000%	100.000%	Covered
bin Reg	96	20	Covered
COVERPOINT Eu2_cov::Istruc_lev	100.000%	100.000%	Covered
bin registros	89	25	Covered
TYPE /tbench_top/EU2_dat	100.000%	100.000%	Covered
INSTANCE Eu2_cov_dat	100.000%	100.000%	Covered
COVERPOINT Eu2_cov_dat::Dat_mem	100.000%	100.000%	Covered
bin h55555555	25	20	Covered
bin haaaaaaaa	27	20	Covered
bin h00000000	24	20	Covered
bin h11111111	24	20	Covered

Figura 4.14 Reporte de Cobertura para instrucciones que escriben a Registros con a) 80 iteraciones para EU1 y b) 100 iteraciones para EU2.

COVERGROUP COVERAGE

a)

Covergroup	Hits	Goal / At Least	Status
TYPE /tbench_top/EU1	100.000%	100.000%	Covered
INSTANCE Eu1_cov	100.000%	100.000%	Covered
COVERPOINT Eu1_cov::Istruc_type	100.000%	100.000%	Covered
bin Reg	451	100	Covered
COVERPOINT Eu1_cov::Istruc_lev	100.000%	100.000%	Covered
bin registros	425	150	Covered
TYPE /tbench_top/EU1_dat	100.000%	100.000%	Covered
INSTANCE Eu1_cov_dat	100.000%	100.000%	Covered
COVERPOINT Eu1_cov_dat::Dat_mem	100.000%	100.000%	Covered
bin h55555555	109	100	Covered
bin haaaaaaaa	148	100	Covered
bin h00000000	117	100	Covered
bin h11111111	100	100	Covered

b)

Covergroup	Hits	Goal / At Least	Status
TYPE /tbench_top/EU2	100.000%	100.000%	Covered
INSTANCE Eu2_cov	100.000%	100.000%	Covered
COVERPOINT Eu2_cov::Istruc_type	100.000%	100.000%	Covered
bin Reg	480	100	Covered
COVERPOINT Eu2_cov::Istruc_lev	100.000%	100.000%	Covered
bin registros	443	150	Covered
TYPE /tbench_top/EU2_dat	100.000%	100.000%	Covered
INSTANCE Eu2_cov_dat	100.000%	100.000%	Covered
COVERPOINT Eu2_cov_dat::Dat_mem	100.000%	100.000%	Covered
bin h55555555	115	100	Covered
bin haaaaaaaa	144	100	Covered
bin h00000000	132	100	Covered
bin h11111111	109	100	Covered

Figura 4.15 Reporte de Cobertura para instrucciones que escriben a Registros con a) 450 iteraciones para EU1 y b) 500 iteraciones para EU2.

COVERGROUP COVERAGE

a)

Covergroup	Hits	Goal / At Least	Status
TYPE /tbench_top/EU1	100.000%	100.000%	Covered
INSTANCE Eu1_cov	100.000%	100.000%	Covered
COVERPOINT Eu1_cov::Istruc_type	100.000%	100.000%	Covered
bin Reg	910	200	Covered
COVERPOINT Eu1_cov::Istruc_lev	100.000%	100.000%	Covered
bin registros	860	250	Covered
TYPE /tbench_top/EU1_dat	100.000%	100.000%	Covered
INSTANCE Eu1_cov_dat	100.000%	100.000%	Covered
COVERPOINT Eu1_cov_dat::Dat_mem	100.000%	100.000%	Covered
bin h55555555	182	150	Covered
bin haaaaaaaa	224	150	Covered
bin h00000000	366	150	Covered
bin h11111111	185	150	Covered

b)

Covergroup	Hits	Goal / At Least	Status
TYPE /tbench_top/EU2	100.000%	100.000%	Covered
INSTANCE Eu2_cov	100.000%	100.000%	Covered
COVERPOINT Eu2_cov::Istruc_type	100.000%	100.000%	Covered
bin Reg	961	200	Covered
COVERPOINT Eu2_cov::Istruc_lev	100.000%	100.000%	Covered
bin registros	892	250	Covered
TYPE /tbench_top/EU2_dat	100.000%	100.000%	Covered
INSTANCE Eu2_cov_dat	100.000%	100.000%	Covered
COVERPOINT Eu2_cov_dat::Dat_mem	100.000%	100.000%	Covered
bin h55555555	238	200	Covered
bin haaaaaaaa	275	200	Covered
bin h00000000	245	200	Covered
bin h11111111	242	200	Covered

Figura 4.16 Reporte de Cobertura para instrucciones que escriben a Registros con a) 800 iteraciones para EU1 y b) 1000 iteraciones para EU2.

En la tabla 4.1 se presenta un resumen del número de iteraciones que se ejecutan en la unidad EU1 y EU2 con el número de veces que se solicita que sean cubiertos de acuerdo a los datos especificados en los 'bins' 'haaaaaaaa', 'h55555555', 'h00000000' y 'h11111111' del grupo de cobertura 'EUx_dat' agregado para cubrir las variaciones en los datos de los domicilios de memoria.

Tabla 4.1 Resumen de número de iteraciones con datos específicos en instrucciones que escriben a registros en las unidades de ejecución EU1 y EU2.

	'Bins'	Número de iteraciones					
		80		450		800	
		Cubierto	Min.	Cubierto	Min.	Cubierto	Min.
EU1	haaaaaaaa	20	15	109	100	182	150
	h55555555	18		148		224	
	h00000000	19		117		366	
	h11111111	23		100		185	
		Número de iteraciones					
		100		500		1000	
		Cubierto	Min.	Cubierto	Min.	Cubierto	Min.
EU2	haaaaaaaa	25	20	115	100	238	200
	h55555555	27		144		275	
	h00000000	24		132		245	
	h11111111	24		109		242	

Del número total de iteraciones por unidad se cubren al 100% los cuatro 'bins' del grupo 'EUx_dat'. El número de ejecuciones en la unidad EU1 es menor que el número en EU2 en ciertos casos debido a que después de un determinado número de estas se alcanza el tiempo de ejecución máximo o el número de líneas que se pueden imprimir en la herramienta de simulación de

EdaPlayground, por lo que se opta por realizar cantidades diferentes en cada unidad.

4.2.6.2 Instrucciones de escritura a memoria.

En el caso de instrucciones que escriben en memoria también se agrega un nuevo grupo de cobertura denominado ‘EUx_dat_mem’ con el que se realiza un control sobre los datos que se generan para el contenido de los domicilios de memoria, los ‘bins’ son ‘haaaaaaaa’, ‘h55555555’, ‘h00000000’ y ‘h11111111’, lo que significa variaciones de ‘1’ y ‘0’ en cada uno de los 32 bits del dato, el grupo de cobertura es similar para las unidades EU1 y EU2 y se presenta en el listado de código 4.13.

Listado de código 4.13 Grupo de cobertura de datos (Instrucciones de escritura a registros).

```
covergroup EUx_dat_mem () @(posedge i_intf_Eux.we_Eux);
  option.name = "Eu2_cov_dat";
  option.per_instance = 1;
   option.goal         = 100;//objetivo de cobertura
   Dat_mem: coverpoint i_intf_Eux.ent_mem_Eux
  {
    option.weight = 100;
    option.at_least = 100;

    bins h55555555  = {'h55555555};
    bins haaaaaaaa  = {'haaaaaaaa};
    bins h00000000  = {'h00000000};
    bins h11111111  = {'h11111111};
  }
endgroup
```

En la figura 4.17 se muestra el reporte de cobertura como resultado de la ejecución de instrucciones que escriben en memoria con 100 iteraciones para ambas unidades EU1 y EU2, en 4.18 se observa el reporte para 500 ejecuciones en las dos unidades y en 4.19 la ejecución de 800 instrucciones en EU1 y 1000 en EU2.

COVERGROUP COVERAGE

a)

Covergroup	Hits	Goal / At Least	Status
TYPE /tbench_top/EU1	100.000%	100.000%	Covered
INSTANCE Eu1_cov	100.000%	100.000%	Covered
COVERPOINT Eu1_cov::Istruc_type	100.000%	100.000%	Covered
bin Mem	103	20	Covered
COVERPOINT Eu1_cov::Istruc_lev_1	100.000%	100.000%	Covered
bin lev1	37	20	Covered
COVERPOINT Eu1_cov::Istruc_lev	100.000%	100.000%	Covered
bin lev2	19	15	Covered
bin lev3	28	15	Covered
bin lev_RAM	19	15	Covered
TYPE /tbench_top/EU1_dat	100.000%	100.000%	Covered
INSTANCE Eu1_cov_dat	100.000%	100.000%	Covered
COVERPOINT Eu1_cov_dat::Dat_mem	100.000%	100.000%	Covered
bin h55555555	20	15	Covered
bin haaaaaaaa	19	15	Covered
bin h00000000	24	15	Covered
bin h11111111	39	15	Covered

b)

Covergroup	Hits	Goal / At Least	Status
TYPE /tbench_top/EU2	100.000%	100.000%	Covered
INSTANCE Eu2_cov	100.000%	100.000%	Covered
COVERPOINT Eu2_cov::Istruc_type	100.000%	100.000%	Covered
bin Mem	100	25	Covered
COVERPOINT Eu2_cov::Istruc_lev_1	100.000%	100.000%	Covered
bin lev1	27	25	Covered
COVERPOINT Eu2_cov::Istruc_lev	100.000%	100.000%	Covered
bin lev2	25	10	Covered
bin lev3	36	10	Covered
bin lev_RAM	12	10	Covered
TYPE /tbench_top/EU2_dat	100.000%	100.000%	Covered
INSTANCE Eu2_cov_dat	100.000%	100.000%	Covered
COVERPOINT Eu2_cov_dat::Dat_mem	100.000%	100.000%	Covered
bin h55555555	30	20	Covered
bin haaaaaaaa	22	20	Covered
bin h00000000	27	20	Covered
bin h11111111	21	20	Covered

Figura 4.17 Reporte de Cobertura para instrucciones que escriben en memoria con a) 100 iteraciones para EU1 y b) 100 iteraciones para EU2.

COVERGROUP COVERAGE

a)

Covergroup	Hits	Goal / At Least	Status
TYPE /tbench_top/EU1	100.000%	100.000%	Covered
INSTANCE Eu1_cov	100.000%	100.000%	Covered
COVERPOINT Eu1_cov::Istruc_type	100.000%	100.000%	Covered
bin Mem	490	100	Covered
COVERPOINT Eu1_cov::Istruc_lev_1	100.000%	100.000%	Covered
bin lev1	165	100	Covered
COVERPOINT Eu1_cov::Istruc_lev	100.000%	100.000%	Covered
bin lev2	76	50	Covered
bin lev3	158	50	Covered
bin lev_RAM	91	50	Covered
TYPE /tbench_top/EU1_dat	100.000%	100.000%	Covered
INSTANCE Eu1_cov_dat	100.000%	100.000%	Covered
COVERPOINT Eu1_cov_dat::Dat_mem	100.000%	100.000%	Covered
bin h55555555	121	100	Covered
bin haaaaaaaa	101	100	Covered
bin h00000000	106	100	Covered
bin h11111111	162	100	Covered

b)

Covergroup	Hits	Goal / At Least	Status
TYPE /tbench_top/EU2	100.000%	100.000%	Covered
INSTANCE Eu2_cov	100.000%	100.000%	Covered
COVERPOINT Eu2_cov::Istruc_type	100.000%	100.000%	Covered
bin Mem	500	100	Covered
COVERPOINT Eu2_cov::Istruc_lev_1	100.000%	100.000%	Covered
bin lev1	161	150	Covered
COVERPOINT Eu2_cov::Istruc_lev	100.000%	100.000%	Covered
bin lev2	90	50	Covered
bin lev3	177	50	Covered
bin lev_RAM	72	50	Covered
TYPE /tbench_top/EU2_dat	100.000%	100.000%	Covered
INSTANCE Eu2_cov_dat	100.000%	100.000%	Covered
COVERPOINT Eu2_cov_dat::Dat_mem	100.000%	100.000%	Covered
bin h55555555	110	100	Covered
bin haaaaaaaa	143	100	Covered
bin h00000000	121	100	Covered
bin h11111111	126	100	Covered

Figura 4.18 Reporte de Cobertura para instrucciones que escriben en memoria con a) 500 iteraciones para EU1 y b) 500 iteraciones para EU2.

COVERGROUP COVERAGE

a)

Covergroup	Hits	Goal / At Least	Status
TYPE /tbench_top/EU1	100.000%	100.000%	Covered
INSTANCE Eu1_cov	100.000%	100.000%	Covered
COVERPOINT Eu1_cov::Istruc_type	100.000%	100.000%	Covered
bin Mem	750	100	Covered
COVERPOINT Eu1_cov::Istruc_lev_1	100.000%	100.000%	Covered
bin lev1	176	100	Covered
COVERPOINT Eu1_cov::Istruc_lev	100.000%	100.000%	Covered
bin lev2	83	50	Covered
bin lev3	138	50	Covered
bin lev_RAM	353	50	Covered
TYPE /tbench_top/EU1_dat	100.000%	100.000%	Covered
INSTANCE Eu1_cov_dat	100.000%	100.000%	Covered
COVERPOINT Eu1_cov_dat::Dat_mem	100.000%	100.000%	Covered
bin h55555555	131	100	Covered
bin haaaaaaaa	111	100	Covered
bin h00000000	122	100	Covered
bin h11111111	386	100	Covered

b)

Covergroup	Hits	Goal / At Least	Status
TYPE /tbench_top/EU2	100.000%	100.000%	Covered
INSTANCE Eu2_cov	100.000%	100.000%	Covered
COVERPOINT Eu2_cov::Istruc_type	100.000%	100.000%	Covered
bin Mem	1000	100	Covered
COVERPOINT Eu2_cov::Istruc_lev_1	100.000%	100.000%	Covered
bin lev1	345	150	Covered
COVERPOINT Eu2_cov::Istruc_lev	100.000%	100.000%	Covered
bin lev2	186	50	Covered
bin lev3	330	50	Covered
bin lev_RAM	139	50	Covered
TYPE /tbench_top/EU2_dat	100.000%	100.000%	Covered
INSTANCE Eu2_cov_dat	100.000%	100.000%	Covered
COVERPOINT Eu2_cov_dat::Dat_mem	100.000%	100.000%	Covered
bin h55555555	233	200	Covered
bin haaaaaaaa	256	200	Covered
bin h00000000	236	200	Covered
bin h11111111	275	200	Covered

Figura 4.19 Reporte de Cobertura para instrucciones que escriben en memoria con a) 800 iteraciones para EU1 y b)1000 iteraciones para EU2.

En la tabla 4.2 se encuentra el resumen del número de ejecuciones en las unidades EU1 y EU2 de instrucciones que escriben a memoria, la tabla contiene el número de veces que se solicita que los 'bins' sean cubiertos de acuerdo a los datos especificados que son 'haaaaaaaa', 'h55555555', 'h00000000' y 'h11111111' del grupo de cobertura 'EUx_dat' agregado.

Tabla 4.2 Resumen de número de iteraciones con datos específicos en instrucciones que escriben en memora en las unidades de ejecución EU1 y EU2.

	'Bins'	Número de iteraciones					
		100		500		600	
		Cubierto	Min.	Cubierto	Min.	Cubierto	Min.
EU1	haaaaaaaa	10	15	121	100	131	100
	h55555555	19		101		111	
	h00000000	24		106		122	
	h11111111	39		162		386	
		Número de iteraciones					
		100		500		1000	
		Cubierto	Min.	Cubierto	Min.	Cubierto	Min.
EU2	haaaaaaaa	30	20	110	100	233	200
	h55555555	22		143		256	
	h00000000	27		121		236	
	h11111111	21		126		275	

De acuerdo con la tabla y a los reportes de las figuras 4.17, 4.18 y 4.19, el número de ejecuciones cubierto por unidad es mayor que el mínimo solicitado, por lo que se observa claramente que se alcanzan porcentajes de cobertura de 100%.

Al igual que en el caso de las instrucciones que escriben a registros, el número de ejecuciones difiere para cada unidad debido a los tiempos de

ejecución o el número de líneas de impresión permitidos en la herramienta de simulación de EdaPlayground. En las instrucciones donde se involucra el acceso a memoria se observa que la ejecución de instrucciones requiere más tiempo de acuerdo al nivel de donde provienen, ya sea nivel 1, 2, 3 o RAM, por lo que el número de ejecuciones es significativamente distinto en ambas unidades en algunos casos.

Capítulo 5

Conclusiones

A medida que surgen nuevos avances en el desarrollo de diseño y fabricación de semiconductores aparecen también nuevos retos en los procesos internos que esto conlleva. La tarea de verificar un circuito es tan compleja como el diseño lo sea, y al pasar los años, con el aumento en la cantidad de transistores que un dispositivo puede tener, esta complejidad crece también de manera significativa, ya que alcanzar los objetivos de cobertura puede consumir más tiempo del que se dispone en términos de oportunidad y costo.

En la presente tesis se presenta una estrategia de verificación basada en la metodología de verificación aleatoria restringida, con el objetivo de producir altos porcentajes en la cobertura de un diseño, que para este caso particular es el modelo comportamental de un procesador con múltiples unidades de ejecución, el cual está desarrollado de acuerdo a la estructura necesaria para la ejecución de instrucciones con distintas características de fácil manejo en el ambiente de verificación propuesto.

Los resultados obtenidos mediante la estrategia propuesta muestran que dirigir los estímulos hacia las características de interés en el dispositivo bajo verificación presenta mejores porcentajes en la cobertura a diferencia de

los estímulos no dirigidos, también se puede observar que realizar pruebas de casos separados permite definir más estrechamente las características de interés, para este trabajo se utilizan los casos con todo el conjunto de instrucciones permitido por el modelo comportamental, instrucciones que escriben a registros e instrucciones que escriben en memoria, los reportes de verificación que se presentan para estos casos demuestran que se alcanzan los porcentajes de cobertura deseados.

La verificación de un diseño es tan importante como el diseño en sí, y es necesario encontrar metodologías que permitan que el proceso sea cada vez más eficiente, ya que de esto depende en gran medida el tiempo en que un dispositivo puede pasar a la etapa de fabricación.

Si bien es posible encontrar gran cantidad de información acerca del tema de verificación es importante recalcar que la mayoría de las veces es difícil que dicha información sea detallada, profunda, completa y que vaya más allá de los límites del conocimiento básico, ya que la mayoría de los esfuerzos dedicados a esta área provienen principalmente del ámbito industrial, sin embargo, es importante conocer cada vez más como se puede incursionar en este tema de gran importancia.

Por otra parte, gran fracción del conjunto de trabajos académicos aplicados en el área de verificación utilizan diseños más simples, por lo que los métodos que se pueden aplicar para verificación y cobertura suelen tener un alcance más limitado. El modelo que se utiliza en este trabajo de tesis cuenta con las características necesarias para emular el comportamiento de un procesador con múltiples unidades de ejecución, que también puede expandirse a la cantidad de unidades que el verificador desee, sin embargo, en el caso ideal, contar con el diseño real de un dispositivo puede proveer una gran cantidad de oportunidades para explorar.

5.1 Trabajo futuro

Como trabajo futuro se propone:

- La sistematización de la estrategia de verificación desarrollada en este trabajo de tesis, que pueda dar lugar a la automatización parcial o total de esta.
- Probar otras herramientas de síntesis y verificación.
- Ampliar el conjunto de instrucciones que el modelo comportamental del procesador puede ejecutar.

Página en blanco intencionalmente

Lista de figuras

Figura 1.1 Progreso del proceso de Verificación, derivado de [2] ... 3
Figura 1.2 Número medio de ingenieros máximos por proyecto, derivado de [3]. ... 4
Figura 2.1 Conceptos de Cobertura. ... 11
Figura 2.2 Reporte de cobertura del listado de código 2.2. ... 17
Figura 2.3 Estructura de un ambiente de verificación ... 18
Figura 2.4 Estructura de un sumador de 4 bits. ... 19
Figura 3.1 Estructura del modelo comportamental, a) Unidades de ejecución '*Execution units*', b) Módulo de memoria '*Memory Module'*, c) Generadores de instrucciones '*Instruction Generators*'. ... 26
Figura 3.2 Formato general de Instrucción. ... 29
Figura 3.3 Formato de Instrucción. ... 29
Figura 3.4 Ciclo de Instrucción. ... 33
Figura 3.5 Flujo de ejecución de una instrucción de registro a registro. ... 34
Figura 3.6 Ambiente de verificación modificado. ... 36
Figura 3.7 Estrategia de Verificación. ... 37
Figura 3.8 Sumador de 32 Bits. ... 39
Figura 3.9 Prioridad en la generación aleatoria. ... 41
Figura 4.1 Diagrama RTL del modelo comportamental derivado del código. 45
Figura 4.2 Diagrama detallado del módulo 'EU1'. ... 46
Figura 4.3 Detalle del módulo 'File_Register' ... 47
Figura 4.4 Detalle del módulo 'Sinc_we'. ... 48

Figura 4.5 Solicitud de escritura por dos unidades de ejecución al mismo tiempo. 49
Figura 4.6 Simulación de ejecución de una instrucción con escritura a memoria. 50
Figura 4.7 Diagrama del ambiente de verificación y dispositivo bajo verificación conectados... 51
Figura 4.8 Cobertura del caso de instrucciones generales. 55
Figura 4.9 Reporte de Cobertura del caso de instrucciones en general con modificaciones en los grupos de cobertura. 57
Figura 4.10 Reporte de cobertura del caso de instrucciones de escritura en registros. 60
Figura 4.11 Resultados de la ejecución de una instrucción que escribe a registros en 1) transacción, 2) 'Driver', 3) 'Monitor', 4) 'Scoreboard'............. 61
Figura 4.12 Reporte de cobertura del caso de instrucciones con escritura a memoria. 65
Figura 4.13 Resultados de la ejecución de una instrucción que escribe en memoria en 1) transacción, 2) 'Driver', 3) 'Monitor', 4) 'Scoreboard'. 66
Figura 4.14 Reporte de Cobertura para instrucciones que escriben a Registros con a) 80 iteraciones para EU1 y b) 100 iteraciones para EU2. 69
Figura 4.15 Reporte de Cobertura para instrucciones que escriben a Registros con a) 450 iteraciones para EU1 y b) 500 iteraciones para EU2. 70
Figura 4.16 Reporte de Cobertura para instrucciones que escriben a Registros con a) 800 iteraciones para EU1 y b) 1000 iteraciones para EU2. 71
Figura 4.17 Reporte de Cobertura para instrucciones que escriben en memoria con a) 100 iteraciones para EU1 y b) 100 iteraciones para EU2. 74
Figura 4.18 Reporte de Cobertura para instrucciones que escriben en memoria con a) 500 iteraciones para EU1 y b) 500 iteraciones para EU2. 75
Figura 4.19 Reporte de Cobertura para instrucciones que escriben en memoria con a) 800 iteraciones para EU1 y b)1000 iteraciones para EU2. 76

Lista de tablas

Tabla 3.1 Ciclos de reloj para niveles de memoria 1, 2, 3 y RAM. 28
Tabla 3.2 Código de identificación del tipo de operación. 29
Tabla 3.3 Conjunto de instrucciones disponibles en el generador aleatorio. 30
Tabla 4.1 Resumen de número de iteraciones con datos específicos en instrucciones que escriben a registros en las unidades de ejecución EU1 y EU2. 72
Tabla 4.2 Resumen de número de iteraciones con datos específicos en instrucciones que escriben en memora en las unidades de ejecución EU1 y EU2. 77

Página en blanco intencionalmente

Lista de listados de código

Listado de código 2.1 Declaración de un grupo de cobertura con puntos de cobertura y *'bins'*. 12
Listado de código 2.2 Grupo de cobertura propuesto para una memoria de 4 bits de domicilio. 15
Listado de código 2.3 Restricciones en las variables 'addr' y 'di' (domicilio y dato respectivamente) 16
Listado de código 2.4 Declaración de la interfaz del sumador de la figura 2.4 20
Listado de código 2.5 Transacciones de los estímulos para el sumador de la figura 2.4. 20
Listado de código 2.6 Generador de transacciones del sumador de la figura 2.4. 21
Listado de código 2.7 Extracto de código de 'Driver' para el sumador de la figura 2.4. 22
Listado de código 2.8 Extracto de código de 'Monitor' para el sumador de la figura 2.4. 23
Listado de código 2.9 Extracto de código de 'Scoreboard' para el sumador de la figura 2.4. 23
Listado de código 3.1 Ejemplo del uso de las palabras reservadas 'solve' y 'before'. 42
Listado de código 4.1 Ejemplo de uso de la palabra reservada 'dist'. 53
Listado de código 4.2 Puntos de cobertura para el caso de restricciones en general en niveles de memoria. 54

Listado de código 4.3 Puntos de cobertura modificados para el caso de restricciones en general en niveles de memoria. 56
Listado de código 4.4 Restricciones en la variable de nivel para el caso de restricciones en general. 57
Listado de código 4.5 Puntos de cobertura para el caso de restricciones que escriben a registros... 58
Listado de código 4.6 Restricciones para el caso de instrucciones que escriben a registros. 59
Listado de código 4.7 Restricción para generación de tipos que escriben a registros. 59
Listado de código 4.8. Puntos de cobertura para el caso de instrucciones que escriben en memoria. 62
Listado de código 4.9 Restricciones para el caso de instrucciones que escriben en memoria. 63
Listado de código 4.10 Restricciones para generación de instrucciones que escriben en memoria. 63
Listado de código 4.11 Fragmento de restricciones para generación de tipo de direccionamiento de instrucciones que escriben en memoria....................... 64
Listado de código 4.12 Grupo de cobertura de datos (Instrucciones de escritura a registros). 68
Listado de código 4.13 Grupo de cobertura de datos (Instrucciones de escritura a registros). 73

Bibliografía

[1] C. Spear, SystemVerilog for Verification: A Guide to Learning the Testbench Language Features, Marlboro: Springer, 2008.

[2] Acellera Systems Initiative, "Design and verification conference," 2018. [Online]. Available: https://dvcon-proceedings.org/document/formal-verification-tutorial-breaking-through-the-knowledge-barrier/. [Accessed Agosto 2022].

[3] H. Foster, "SIEMENS Digital Industries Software," 12 Diciembre 2022. [Online]. Available: https://blogs.sw.siemens.com/verificationhorizons/2022/12/12/part-8-the-2022-wilson-research-group-functional-verification-study/.

[4] H. Foster, "SIEMENS Digital Industries Software," 6 Enero 2021. [Online]. Available: https://blogs.sw.siemens.com/verificationhorizons/2021/01/06/part-8-the-2020-wilson-research-group-functional-verification-study/.

[5] IEEE, "IEEE Standard for SystemVerilog-Unified Hardware Design, Specification, and Verification Language," pp. 1-1315, 2018.

[6] Doulos, "EdaPlayground," [Online]. Available: https://www.edaplayground.com/x/dhCJ. [Accessed 2023 Agosto 27].

[7] Doulos, "EdaPlayground," [Online]. Available: https://www.edaplayground.com/x/TVyJ. [Accessed 27 Agosto 2023].

[8] "Verification Guide," [Online]. Available: https://verificationguide.com/systemverilog-examples/systemverilog-testbench-example-adder/. [Accessed 6 Septiembre 2023].

[9] Intel® corporation, "Intel," [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html. [Accessed 27 Agosto 2023].

[10] A. Fog, "Software Optimization Resources," [Online]. Available: https://www.agner.org/optimize/#manuals. [Accessed Agosto].

[11] A. Fog, "Software Optimization Resources," [Online]. Available: https://www.agner.org/optimize/microarchitecture.pdf. [Accessed Agosto 2023].

[12] H. Murdoca, Principios de Arquitectura de Computadoras, Prentice Hall, 2002.

[13] "Verification Guide," [Online]. Available: https://verificationguide.com/systemverilog-examples/systemverilog-testbench-example-adder/. [Accessed 23 Agosto 2023].

[14] The New York Times, "Flaw Undermines Accuracy of Pentium Chips," p. 1, 24 Noviembre 1994.