UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

PROYECTO FIN DE CARRERA

GENERACIÓN DE VECTORES PARA LA VERIFICACIÓN DE CIRCUITOS BOOLEANOS MEDIANTE MILP

Titulación: Ingeniería en Electrónica

Autor: María de las Nieves Gloria Hernández González

Tutores: Dr. Carlos Javier Sosa González

Dr. Carlos Salvador Betancor Martín

Fecha: Junio 2017

UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

PROYECTO FIN DE CARRERA

GENERACIÓN DE VECTORES PARA LA VERIFICACIÓN DE CIRCUITOS BOOLEANOS MEDIANTE MILP

HOJA DE FIRMAS

Alumna

Fdo.: María de las Nieves Gloria Hernández González

Tutor **Tutor**

Fdo.: Dr. Carlos Javier Sosa González Fdo.: Dr. Carlos Salvador Betancor Martín

Fecha: Junio 2017

UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

PROYECTO FIN DE CARRERA

GENERACIÓN DE VECTORES PARA LA VERIFICACIÓN DE CIRCUITOS BOOLEANOS MEDIANTE MILP

HOJA DE EVALUACIÓN

Calificación: _____

Presidente

Fdo.:

Vocal **Secretario/a**

Fdo.: Fdo.:

Fecha: Junio 2017

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

MEMORIA

RESUMEN

Este Proyecto Fin de Carrera presenta una metodología para la generación de vectores de verificación basada en Programación Lineal Entera Mixta. La metodología propuesta da lugar a una herramienta que hace uso de diversos programas desarrollados, en lenguaje de programación C, a lo largo de este PFC y que se ejecutan a nivel de línea de comandos.

La herramienta desarrollada permite generar patrones para la verificación en la que se especifican, por ejemplo, el porcentaje de entradas y/o salidas que han de fijarse. Esta herramienta se ha evaluado mediante un banco de pruebas estándar ISCAS'85, empleado extensamente en la bibliografía.

Se han escogido de dicho banco de pruebas los circuitos combinacionales más representativos del estado del arte. La complejidad de los mismos va desde unas pocas puertas, C17 con 6 puertas, a circuitos de complejidad media con aproximadamente 3500 puertas como es el caso del C7552. En términos de entradas/salidas, la complejidad alcanza las 233 entradas para el C2670 ó 5 entradas para el C17 y desde las 2 salidas con que cuenta C17 hasta las 140 salidas para C2670 .

Todas las pruebas realizadas demuestran la eficiencia de la metodología propuesta y la validez de la herramienta desarrollada. Los tiempos requeridos para obtener los vectores de verificación se encuentran siempre por debajo de 1,5 segundos en todas las pruebas realizadas en este PFC, salvo en contadas ocasiones que se corresponden con el circuito C7552 que requiere aproximadamente 33 segundos. Esa diferencia de tiempos es debida principalmente a la complejidad de interconexión de las puertas de dicho circuito de referencia.

Se puede concluir que el desarrollo de este PFC ha logrado satisfactoriamente los objetivos planteados inicialmente, sirviendo como punto de partida de una futura tesis doctoral en base a los resultados obtenidos.

ABSTRACT

This PFC presents a methodology for the generation of verification patterns using Mixed Integer Linear Programming. Based on proposed methodology, this PFC develops a verification tool made by several programs developed in C programming language. The verification tool is executed at the shell command line interface.

The developed tool allows to generate verification patterns which are specified for example with a percentage of inputs and/or outputs to be asserted. This verification tool has been evaluated using ISCAS'85 standard testbenches, widely used in the literature.

The most representative combinational circuits of the state of the art have been chosen from testbenches. Their complexity ranges from a few gates, C17 with 6 gates, to medium complexity circuits with approximately 3500 gates as is the case of the C7552. In terms of inputs/outputs, the complexity reaches 233 inputs for the C2670 or 5 inputs for the C17 and from the 2 outputs with C17 up to the 140 outputs for C2670.

All tests demonstrate the efficiency and usefulness of the proposed methodology and developed verification tool. The time required to obtain the verification patterns was always less than 1.5 seconds in all tests performed in this PFC. Only for circuit C7552, the required time to obtain the solution rises to approximately 33 seconds. This difference is mainly due to the complexity of its internal interconnection in this reference circuit.

It can be concluded that the development of this PFC has satisfactorily achieved the objectives set out initially, serving as a starting point for a future PhD. based on the obtained results.

ACRÓNIMOS

AI	Artificial Intelligence
ASCII	American Standard Code for Information Interchange
BDD	Binary Decision Diagram
BLIF	Berkeley Logic Interchange Format
BMC	Bounded Model Checking
CAD	Computer-Aided Design
CI	Circuito Integrado
CNF	Conjunctive Normal Form
CTL	Computational Tree Logic
DFT	Design For Testability
DIMACS	Center for Discrete Mathematics and Theoretical Computer Science
DUT	Device Under Test
EC	Equivalence Checking
EDA	Electronic Design Automation
EITE	Escuela de Ingeniería de Telecomunicación y Electrónica
IP	Integer Programming
ISCAS	International Symposium on Circuits And Systems
LP	Linear Programming
LSI	Large Scale integration
MBTG	Model Based Test Generation
MC	Model Checking
MCNC	Microelectronics Center of North Carolina
MILP	Mixed Integer Linear Programming

MIP	Mixed Integer Programming
MIQP	Mixed Integer Quadratic Programming
MSI	Medium Scale integration
PC	Property Checking
PFC	Proyecto Fin de Carrera
PLA	Programmable Logic Array
Prolog	PROgrammation en LOGique
QCP	Quadratically Constrained Programming
RTPG	Random Test Pattern Generation
SAT	Satisfiability
SoC	System on Chip
SSI	Small Scale integration
TPG	Test Pattern Generation
ULPGC	Universidad de Las Palmas de Gran Canaria
ULSI	Ultra Large Scale Integration
Verilog	VERIfication LOGic
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLSI	Very Large Scale Integration

CAPÍTULO 1
INTRODUCCIÓN Y ANTECEDENTES

1.1 INTRODUCCIÓN

En el Diseño Electrónico, comprobar las funcionalidades implementadas es primordial [1]. Es bien conocido que un fallo de diseño a nivel de puerta, bloque, módulo, interfaz, subsistema o sistema puede ocasionar cuantiosas pérdidas económicas. En el mejor de los casos y en las etapas tempranas del diseño, la detección de un error implica un retraso de tiempo y una recodificación parcial del código bajo comprobación. En el peor de los casos, estos fallos pueden ocasionar un retraso considerable o incluso la cancelación del proyecto [2].

Por otro lado, se define el time-to-market como aquel periodo de tiempo que marca el momento idóneo para introducir un producto en el mercado. Si el producto se introduce con demasiada antelación a un time-to-market lo más probable es que dicho producto genere bajas ventas pues el mercado no está preparado. Cuando se da el caso contrario, es decir, el producto se introduce

posteriormente a su time-to-market, probablemente la competencia tenga acaparada gran parte de la cuota de mercado lo que implicaría nuevamente un bajo número de ventas.

Si se une la necesidad de cumplir el time-to-market al incremento de la complejidad de las funcionalidades incluidas en los circuitos integrados actuales [3][4], el desarrollo de un nuevo producto conlleva grandes riesgos que hay que minimizar. A modo de resumen, teniendo en cuenta las presiones temporales, pues los proyectos se acotan en el tiempo, y el volumen de funcionalidades más su complejidad, es necesario por lo general dedicar a verificación un 70% del esfuerzo total empleado en el desarrollo del producto.

Como puede verse en la Tabla 1, los sistemas digitales cada vez contienen un mayor número de elementos lógicos [5], por lo que resulta muy complejo probar todos y cada uno de los estados lógicos posibles y en algunos casos es prácticamente imposible, ya que los tiempos requeridos para realizar ese tipo de simulaciones y verificar el correcto funcionamiento del circuito se disparan exponencialmente. Por ello, se ha optado por aplicar la verificación funcional, que se basa en la cobertura de los puntos más significativos o de mayor interés en el diseño de un sistema digital, con los cuales se puede garantizar la correcta funcionalidad de todo el sistema.

Escala de Integración	Año	Nº Transistores	Nº Puertas Lógicas
SSI	1964	1 a 10	1 a 12
MSI	1968	10 a 500	13 a 99
LSI	1971	500 a 20.000	100 a 9.999
VLSI	1980	20.000 a 1.000.000	10.000 a 99.999
ULSI	1984	1.000.000 y más	100.000 y más

Tabla 1: Generaciones de CIs

La verificación de un circuito integrado no se limita a un simple paso dentro del flujo de diseño. Puesto que en todos los pasos del flujo de diseño de un circuito integrado puede surgir un fallo, se ha de ejecutar un proceso de comprobación (verificación) tras cada paso en el que evolucione el diseño (Figura 1).

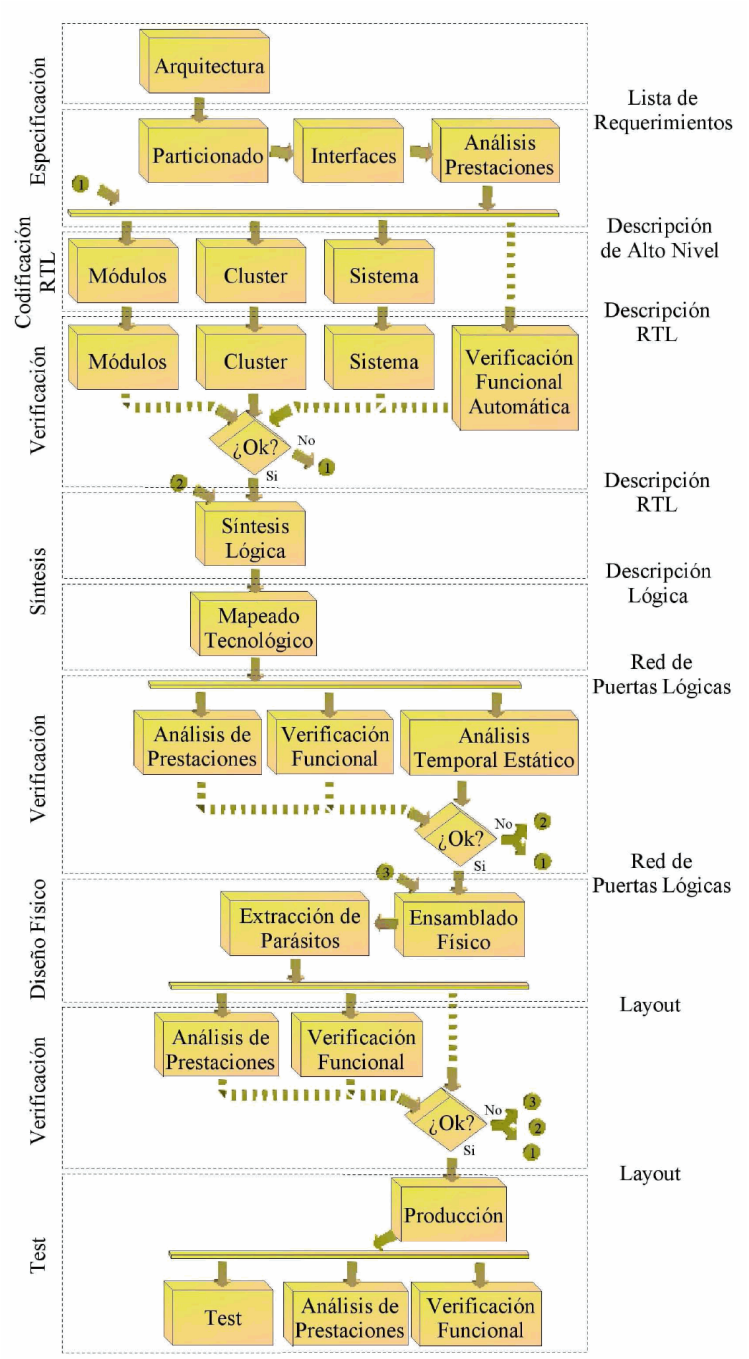


Figura 1: Flujo de diseño de un circuito integrado [6]

A partir de un conjunto de especificaciones en las que se detallan las características del circuito integrado a desarrollar, la verificación del mismo ha de plantear los distintos casos a comprobar en todos los niveles. Estos casos se denominan esquinas críticas (corner cases) [7]. A medida que el desarrollo avanza, su complejidad se ve incrementada. Del estudio de la literatura al respecto, se deduce que la complejidad de los casos críticos definidos en la verificación aumenta

en relación a la complejidad de la etapa del diseño. Además, esta complejidad es abordada introduciendo diversas capas de abstracción.

1.2 ANTECEDENTES

Existen múltiples aportaciones en el estado del arte en el área de verificación en todos los niveles del flujo de diseño. Por ejemplo, destacan la generación de vectores, las métricas de cobertura y los modelos para acelerar el proceso de verificación.

En particular, un área muy estudiada es la generación de vectores. Conocida una funcionalidad, es muy sencillo generar estímulos de entrada a ese circuito y obtener su respuesta en las salidas. Lo complicado es el número mínimo de esos patrones de entrada/salida para asegurar un máximo de comprobación en el menor tiempo. Este problema es hoy día el paradigma de la verificación, es decir, la generación de vectores de máxima cobertura.

A nivel de módulo o incluso bloque, el incremento del número de puertas complica la generación de esos vectores de máxima cobertura. La literatura aborda este problema mediante la definición del llamado problema de satisfacibilidad (satisfiability) o comúnmente llamado SAT. Se define un SAT como el problema de determinar si el comportamiento implementado en un circuito conocido permite fijar unas entradas determinadas y que su respuesta en las salidas sean unos valores conocidos.

Por otro lado, los problemas SAT han sido ampliamente estudiados en el terreno matemático. La descripción del problema SAT se hace mediante ecuaciones algebraicas que modelan las puertas, la definición de las entradas fijadas y las salidas esperadas [8]. Los métodos matemáticos más recurridos en la literatura comprenden, por ejemplo, Programación Lineal Entera Mixta (MILP), Programación Cuadrática Entera Mixta (MIQP) [9] o Prolog [10].

1.3 OBJETIVOS

Este Proyecto Fin de Carrera plantea la creación de herramientas que faciliten la generación de patrones de verificación de máxima cobertura para circuitos booleanos empleando MILP.

El objetivo principal es crear un entorno de desarrollo para la generación de patrones de verificación. Este entorno se ejecutará a nivel de línea de comandos y se desarrollará completamente en C. Su ejecución está prevista para máquinas Linux o Unix.

Como objetivos secundarios, pero no menos importantes, se plantea que se haga uso de un formato de entrada estándar. El empleo de un formato estándar permitirá utilizar esta herramienta con los diversos lenguajes de descripción hardware que se emplean en la literatura como, por ejemplo, VHDL, Verilog, BLIF y PLA, entre otros. Igualmente, en base a la literatura, se ha de establecer el banco de circuitos de prueba para comprobar las prestaciones del entorno.

También se han de desarrollar los modelos de puerta Booleana que permitan traducir el modelo circuital (netlist) a un modelo basado en MILP.

1.4 PETICIONARIO

Actúa como peticionario de este Proyecto de Fin de Carrera la Escuela de Ingeniería de Telecomunicación y Electrónica (EITE), de la Universidad de Las Palmas de Gran Canaria (ULPGC), siendo la realización de este proyecto requisito para la obtención del título de Ingeniero en Electrónica.

1.5 ORGANIZACIÓN DEL DOCUMENTO

Este Proyecto Fin de Carrera está estructurado en cuatro bloques: Memoria, Presupuesto, Pliego de Condiciones y Anexos. Además, se adjunta un CD con la información contenida en el presente documento.

1.5.1 MEMORIA

La Memoria de este Proyecto Fin de Carrera está estructurada en 6 capítulos y la bibliografía.

- **Capítulo 1: Introducción y antecedentes**. En el capítulo introductorio de la memoria se detallan los antecedentes, así como los objetivos del proyecto, su peticionario y la estructura del documento.
- **Capítulo 2: Verificación de circuitos**. Se describen diferentes técnicas para la verificación de circuitos, centrándose en las relativas a la verificación de circuitos booleanos.
- **Capítulo 3: Satisfacibilidad**. Se explican los fundamentos de satisfacibilidad sobre los que se sustenta el trabajo, así como los formatos usados como entrada para los SAT-solvers.
- **Capítulo 4: Método desarrollado**. Se describe la organización del sistema diseñado, detallando las distintas partes desarrolladas.
- **Capítulo 5: Resultados experimentales**. Se expone la experimentación realizada con el banco de pruebas.
- **Capítulo 6: Conclusiones y líneas futuras**. Finalmente, se detallan las conclusiones obtenidas durante el desarrollo del proyecto y se presentan algunas líneas de trabajo que pueden derivarse de su realización.
- **Bibliografía.** En la bibliografía se detallan las referencias empleadas para la realización y la redacción del presente proyecto.

1.5.2 PRESUPUESTO

En el presupuesto se cuantifican los costes del desarrollo del presente Proyecto Fin de Carrera.

1.5.3 PLIEGO DE CONDICIONES

En este apartado se detallan las condiciones bajo las cuales se han desarrollado las diferentes etapas del presente Proyecto Fin de Carrera.

1.5.4 ANEXO I: CÓDIGOS

Se muestra el código en lenguaje C de los programas que han sido desarrollados en este proyecto.

1.5.5 ANEXO II: BANCO DE PRUEBAS

Se hace una relación de los circuitos de referencia obtenidos con este PFC.

CAPÍTULO 2
VERIFICACIÓN DE CIRCUITOS

2.1 INTRODUCCIÓN

Como se ha mencionado previamente, este PFC se centra en el desarrollo de herramientas para la generación de patrones de verificación y test de circuitos booleanos. Con el fin de abordar de forma confiable la temática de este PFC, antes de proseguir detallando en profundidad el desarrollo del mismo, se hace imprescindible la lectura de este capítulo que versa sobre el estudio del arte.

En primer lugar, y de forma simplificada, se exponen las diversas metodologías empleadas en la verificación/test de circuitos, aportando las razones por las que se deba emplear una u otra técnica y haciendo especial hincapié en aquellos métodos de verificación en que se encuadran los trabajos del presente PFC.

Por otro lado, se explican las principales técnicas de generación de vectores de verificación/test, detallando los conceptos de modelos de fallos, métodos algorítmicos y cobertura de fallos.

2.2 VERIFICACIÓN DE CIRCUITOS

En la actualidad, dado que los circuitos pueden contener varios cientos de millones de transistores, la verificación de los diseños actuales se hace cada vez más difícil. Además, se ha observado que la verificación se convierte en el principal cuello de botella en los flujos de diseño, de forma que hasta el 80% de los costes globales de diseño se deben a la verificación (Tabla 2). Esta es una de las razones por las que en los últimos años se han propuesto infinidad de métodos/técnicas como alternativas y/o complementos a la simulación clásica [11].

Se define el concepto de cobertura de verificación/test de un diseño como el grado de seguridad sobre el correcto funcionamiento de un diseño. Si se tiene en cuenta el constante incremento de funcionalidades en las nuevas generaciones de circuitos integrados, el número de estas crece de forma exponencial. Por ello, asegurar el correcto funcionamiento de las mismas mediante simulación clásica se torna una tarea cuanto menos ardua, si no imposible, en términos de tiempos de ejecución razonables.

Entendiendo que es imposible alcanzar el cien por cien de confianza en la cobertura de verificación/test, es ahí donde entran en juego esas técnicas que permiten reducir el universo de simulaciones a un conjunto finito y bien definido que permita alcanzar un grado de confianza/cobertura aceptable.

Una primera metodología sencilla como aproximación aceptable consiste en la creación de un Testplan o el llamado Plan de Verificación/Test. Este es un documento en el que se detallan cada una de las funcionalidades a comprobar y el conjunto de pruebas que se han de realizar. Adicionalmente, este documento tendría que indicar el nivel de prioridad de dicha prueba. Las pruebas o casos críticos se han de priorizar en varios niveles (por ejemplo: críticos, medio y bajo) en los que se explicitaría qué casos han de comprobarse de forma ineludible, cuales son

opcionales y cuales prescindibles. Este documento ha de presentar una estimación del tiempo requerido para la ejecución de cada uno de los casos así como el grado de confianza alcanzado.

La comprobación de cada uno de esos casos conlleva la ineludible tarea de generación de los estímulos de entrada así como la obtención de las respuestas a obtener por parte del diseño. Si se realiza una comprobación basada en simulación, dichos estímulos de entrada y respuesta en las salidas se traducen en un conjunto de patrones de verificación (vectores de datos booleanos) en sus entradas y salidas, respectivamente.

La simulación de casos críticos, por sí sola no puede garantizar una cobertura suficiente del diseño, lo que conduce a errores que pueden permanecer sin ser detectados. La correcta generación de esos vectores de verificación/test son la clave de su correcta consecución.

Puesto que la simulación no puede garantizar el comportamiento correcto, y la simulación exhaustiva es a menudo imposible, como alternativa se han propuesto técnicas de verificación formal. En lugar de simular un diseño, la comprobación del mismo es probada por técnicas formales. Hay muchas áreas diferentes donde estos enfoques se pueden utilizar, como la comprobación de equivalencia, la comprobación de propiedades o la simulación simbólica. Estos métodos se han aplicado con éxito en muchos proyectos industriales y se han convertido en la técnica más avanzada en varios campos.

	1970	1975	1980	1985	1990	1995	2000
Complejidad	SSI	MSI	LSI	VLSI	ULSI		SoC
Transistores	100	1K	10K	100K	1M	10M	100M
Memorias	256	1K	4-16K	64-256K	1-16M	64M	1-16G
Velocidad		1MHz	10MHz	30MHz	100MHz	1GHz	10GHz
Terminales		32	64	128	256	512	1024
Coste test/Coste total	5%	10%	20%	40%	60%	70%	80%

Tabla 2: Evolución temporal en el diseño de CIs [12]

2.2.1 VERIFICACIÓN FORMAL

La idea principal de la verificación formal de un diseño es probar su funcionalidad sin recurrir a simulación. Para el proceso de comprobación se han

propuesto diferentes técnicas. La mayoría de ellos trabajan en el dominio booleano, como diagramas binarios de decisión (BDDs) o solucionadores SAT (SAT-solvers). Los escenarios típicos de verificación de hardware, en los que se aplican técnicas de verificación formal, son el chequeo de equivalencia (EC) y la verificación de propiedades (PC), también llamada verificación de modelos (MC).

El objetivo de EC es asegurar la equivalencia de dos descripciones de circuitos. Estos circuitos pueden darse en diferentes niveles de abstracción, es decir, nivel de transferencia de registro o nivel de puerta. Las etapas principales de un chequeador de equivalencia son las siguientes (Figura 2) [13]:

1. Traducir ambos diseños a un formato propio.
2. Establecer la correspondencia entre los dos diseños en una fase de coincidencia.
3. Probar equivalencia o inequivalencia.
4. En caso de una inequivalencia se genera una incidencia y comienza la fase de depuración.

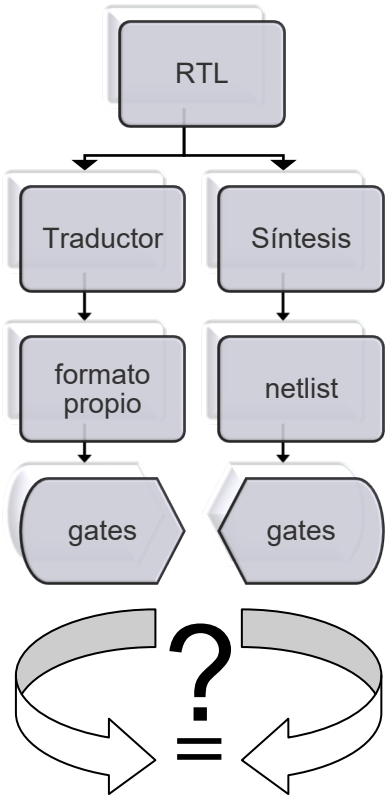


Figura 2: Chequeo de Equivalencia (EC)

El circuito se considera puramente combinacional al modelar los elementos de estado como entradas y salidas primarias adicionales. Este modelado puede dar lugar a casos que no son accesibles durante el funcionamiento normal del circuito.

A diferencia de EC, donde se consideran dos circuitos, para PC se da un solo circuito y las propiedades se formulan en un "lenguaje de verificación" específico. Se demuestra entonces formalmente si estas propiedades se mantienen en todas las circunstancias. Mientras que la verificación de modelos "clásicos" basada en CTL (Computational Tree Logic) sólo puede aplicarse a diseños de tamaño mediano, los enfoques basados en BMC (Bounded Model Checking), como se describe en [14], dan muy buenos resultados cuando se usan para bloques completos con hasta 100k puertas.

Sin embargo, todos estos enfoques pueden tener problemas causados por la complejidad, esto es, si el circuito se hace demasiado grande o si la función representada resulta ser compleja para los métodos formales. El segundo problema surge a menudo en casos de aritmética compleja, como multiplicadores.

Estos problemas han conducido a la propuesta de métodos híbridos, como por ejemplo, simulación simbólica y verificación de aserción. Estos métodos intentan salvar la brecha entre la simulación y las pruebas de corrección, pero también hacen uso de técnicas de verificación formal.

2.3 GENERACIÓN DE PATRONES DE VERIFICACIÓN Y/O TEST

Aunque BDDs y SAT son las técnicas más populares en la verificación hardware, y se han aplicado a muchos dominios, todavía hay gran cantidad de investigación en curso. Además de los enfoques monolíticos clásicos, las herramientas modernas de EC utilizan enfoques múltiples que combinan diferentes técnicas, como: SAT, BDD, reescritura de términos, generación de patrones de test y simulación con patrones de test aleatorios.

La Generación de Patrones de Test (TPG) es un método de automatización de diseño electrónico utilizado para encontrar una secuencia de entrada (o test) que, cuando se aplica a un circuito digital, permita que el equipo de prueba distinga

entre el comportamiento correcto del circuito y el comportamiento defectuoso causado por defectos.

Los primeros patrones de test consistían en un archivo formado por vectores binarios escritos manualmente. Posteriormente, se desarrollaron generadores de vectores de test, como los generadores de pruebas aleatorias (RTPG), los generadores de test basados en modelos (MBTG), llegando a los actuales generadores de patrones de test aleatorios con restricciones mediante el uso de solucionadores de restricciones.

Los patrones generados se usan para probar dispositivos semiconductores después de la fabricación, o para ayudar a determinar la causa del fallo (análisis de fallos). La eficacia de la TPG se mide por el número de defectos modelados, o modelos de fallo, detectables y por el número de patrones generados. Estas métricas generalmente indican la calidad de la prueba (mayor con más detecciones de fallos) y el tiempo de aplicación del test (mayor con más patrones). La eficiencia de la TPG es otra consideración importante que está influenciada por el modelo de fallo considerado, el tipo de circuito bajo test, el nivel de abstracción utilizado para representar el circuito bajo prueba y la calidad requerida del test.

2.3.1 FUNDAMENTOS PARA LA GENERACIÓN DE PATRONES DE TEST

Un defecto es un error causado en un dispositivo durante el proceso de fabricación. Un modelo de fallo es una descripción matemática de cómo un defecto altera el comportamiento del diseño. Los valores lógicos observados en las salidas primarias del dispositivo, al aplicar un patrón de prueba a algún dispositivo bajo prueba (DUT), se denominan salida de ese patrón de prueba. La salida de un patrón de test, cuando se prueba un dispositivo sin fallos que funcione exactamente como se diseñó, se denomina salida esperada de ese patrón de prueba. Un fallo se dice que es detectado por un patrón de prueba si la salida de ese patrón de test, cuando se prueba un dispositivo que tiene solo un fallo, es diferente de la salida esperada. El proceso TPG para un fallo selectivo consta de dos fases: activación de fallos y propagación de fallos. La activación de fallos establece un valor de señal en el sitio del modelo de fallos que es opuesto al valor producido por el modelo de fallo. La

propagación de fallos mueve el valor de señal resultante o el efecto del fallo hacia adelante, sensibilizando un camino desde el sitio del fallo a una salida primaria.

La TPG no puede encontrar una prueba para un fallo en al menos dos casos. En primer lugar, el fallo puede ser intrínsecamente indetectable, de tal manera que no existen patrones que puedan detectar ese fallo en particular. El ejemplo clásico de esto es un circuito redundante, diseñado de tal manera que ningún fallo único hace que la salida cambie. En tal circuito, cualquier fallo único será intrínsecamente indetectable.

En segundo lugar, es posible que exista un patrón de detección, pero el algoritmo no puede encontrar uno. Puesto que el problema de la TPG es NP-completo (por reducción del problema de satisfacibilidad booleano) habrá casos donde existen patrones, pero la TPG abandona, ya que tardaría demasiado tiempo en encontrarlos.

2.3.2 MODELOS DE FALLOS

Un modelo de fallo es un modelo matemático de algo que podría ir mal en la construcción u operación del DUT. A partir del modelo, el diseñador o usuario puede predecir las consecuencias de este fallo en particular. Los modelos de fallo pueden usarse en casi todas las ramas de la ingeniería.

Puede hacerse la suposición de fallo único, es decir, que solo se produce un fallo en el circuito. Si se definen k posibles tipos de fallos en el modelo de fallos y el circuito tiene n líneas de señal, por suposición de fallo único, el número total de fallos únicos es $k \times n$. O puede hacerse la asunción de múltiples de fallos, esto es que se contempla la posibilidad de que puedan ocurrir varios fallos en el circuito.

La magnitud del problema de la verificación hace necesaria la utilización de modelos de fallos que permitan determinar con precisión los fallos posibles y contar su número, encontrar vectores de test para cada fallo, conocer qué fallos detecta cada vector de test y calcular la cobertura de fallos, es decir, el conjunto de fallos que pueden ser detectados por un conjunto dado de vectores de test.

2.3.2.1 Colapso de fallos equivalentes

Los fallos equivalentes producen el mismo comportamiento defectuoso para todos los patrones de entrada. Cualquier fallo del conjunto de fallos equivalentes puede representar el conjunto completo. En este caso, se requieren pruebas de fallos mucho menores que k × n para un circuito con n líneas de señal. Eliminar fallos equivalentes de un conjunto completo de fallos se denomina colapso de fallos.

2.3.2.2 Modelo de fallos por bloqueo Stuck-at

Es el modelo de fallos más utilizado en las últimas décadas. En este modelo, se supone que una de las líneas de señal de un circuito está atascada a un valor lógico fijo, independientemente de las entradas que se suministran al circuito. Por lo tanto, si un circuito tiene n líneas de señal, existen potencialmente 2n atascos definidos en el circuito, algunos de los cuales pueden ser vistos como equivalentes a otros. El modelo de fallos stuck-at es un modelo de error lógico porque no hay información de retardo asociado con la definición del fallo. También se denomina modelo de fallo permanente, porque se supone que el defecto es permanente, en contraste con los fallos intermitentes, que pueden ocurrir esporádicamente de forma aleatoria y transitoria, dependiendo de las condiciones de funcionamiento (por ejemplo, temperatura, tensión de alimentación) o en los valores de datos (estados de alta o baja tensión) en las líneas de señal circundantes. El modelo stuck-at es estructural porque se define a partir de un modelo de circuito estructural a nivel de puerta.

Un conjunto de patrones con 100% de cobertura del modelo de fallos stuck-at consiste en pruebas para detectar todos los fallos posibles en un circuito.

2.3.3 MÉTODOS ALGORÍTMICOS

La prueba de circuitos integrados VLSI (Very Large Scale Integration) con una alta cobertura de fallos es una tarea difícil debido a la complejidad de los mismos. Por lo tanto, se han desarrollado muchos métodos TPG diferentes para abordar circuitos combinacionales y secuenciales.

Los primeros algoritmos de generación de test, tales como la diferencia booleana y la proposición literal, no eran prácticos para implementar en una computadora.

El Algoritmo D fue el primer algoritmo de generación de test práctico en términos de requerimientos de memoria. Fue propuesto por Roth en 1966 e introdujo la Notación D que sigue siendo utilizada en la mayoría de los algoritmos TPG. Este algoritmo intenta propagar el valor de fallo stuck-at a una salida primaria.

La toma de decisiones orientada por el camino (PODEM) es una mejora sobre el Algoritmo D. PODEM fue creado en 1981 por Prabhu Goel, cuando se evidenciaron las limitaciones del Algoritmo D que no podía con los circuitos que surgieron por las innovaciones del diseño.

El Algoritmo FAN, orientado por Fan-Out, es una mejora sobre PODEM. Limita el espacio de búsqueda TPG para reducir el tiempo de cálculo y acelera el proceso.

También se usan métodos basados en la satisfacibilidad booleana para generar vectores de prueba.

La generación de test Pseudoaleatorios es el método más simple de crear tests. Utiliza un generador de números pseudoaleatorios para generar vectores de test y se basa en la simulación lógica, para conseguir buenos resultados de funcionamiento, y simulación de fallos, para calcular la cobertura de fallos de los vectores generados.

2.4 COBERTURA DE FALLOS

La cobertura de fallos se refiere al porcentaje de algún tipo de fallo que se puede detectar durante el test. Una alta cobertura de fallos es particularmente valiosa durante la prueba de fabricación, y técnicas tales como diseño para test (DFT) y la generación automática de patrones de test se utilizan para aumentarla.

En la electrónica se mide la cobertura de fallos stuck-at fijando cada pin del modelo hardware a '0' lógico y a '1' lógico, respectivamente, y ejecutando los

vectores de test. Si al menos una de las salidas difiere de lo que se espera, se dice que se detecta el fallo. Conceptualmente, el número total de ciclos de simulación es el doble del número de pines, ya que cada pin es fijado de las dos maneras, y ambos fallos deben ser detectados. Sin embargo, hay muchas optimizaciones que pueden reducir el cálculo necesario. En particular, a menudo muchos fallos que no interactúan pueden ser simulados en una ejecución y cada simulación puede ser terminada tan pronto como se detecta un fallo.

Un test con una determinada cobertura de fallos se alcanza cuando se puede detectar por lo menos un porcentaje especificado de todos los fallos posibles. Si no se consigue, existen al menos tres opciones. En primer lugar, el diseñador puede aumentar o mejorar el conjunto de vectores, tal vez utilizando una herramienta de generación de patrones de prueba automática más eficaz. En segundo lugar, el circuito puede ser redefinido para una mejor detección de fallos (mejor controlabilidad y observabilidad). En tercer lugar, el diseñador puede simplemente aceptar la menor cobertura.

2.5 CONCLUSIONES

Uno de los principales desafíos en la línea de producción de circuitos integrados es el uso de procedimientos adecuados y prácticos para probar circuitos en diferentes etapas del proceso de fabricación para mejorar el rendimiento y la fiabilidad de los productos finales. Con los recientes avances en las tecnologías de semiconductores y el aumento continuo de la densidad de circuitos de los chips VLSI, el costo de las pruebas aumenta rápidamente. La prueba de circuitos digitales es el proceso de aplicar algunos vectores de prueba a las entradas del circuito y observar las salidas. El tamaño del conjunto de pruebas afecta el coste total de las pruebas del circuito.

Este análisis se utiliza con el objetivo de generar nuevos vectores de prueba altamente eficaces. La relación entre los fallos focalizados y el total de fallos posibles tiene efecto en la mejora de la cobertura de fallos.

Por último y a modo de reflexión, mientras la verificación de circuitos se refiere al conjunto de tareas encaminadas a comprobar un diseño desde un punto de vista puramente funcional y el test se centra en chequear defectos físicos, ambos procesos del ciclo de diseño se encuentran íntimamente interrelacionados. Por ejemplo, gran parte de las pruebas a las que se somete el diseño en su verificación son reutilizadas en el test. Esta dualidad de ciertas metodologías/técnicas de verificación/test hace que, por ejemplo, la inmensa mayoría de los generadores de patrones de verificación/test sean aplicados en ambos ámbitos.

En particular, los generadores de patrones propuestos en este PFC son aplicables tanto a la verificación como al test de circuitos digitales booleanos.

CAPÍTULO 3
SATISFACIBILIDAD (SAT)

3.1 INTRODUCCIÓN

La satisfacibilidad (SAT) es un problema clásico tanto en el área de las matemáticas como en la ingeniería. Hoy día se resuelven muchos problemas cotidianos mediante SAT-solvers. La complejidad de los nuevos problemas, que surgen a diario en este ámbito, requieren de mayores recursos computacionales que a su vez crea nuevos retos en la obtención de la solución a dichos problemas de forma eficiente y más veloz en términos de memoria requerida y esfuerzo computacional.

Su aplicabilidad se debe probablemente a que la SAT se encuentra en el cruce de la lógica, la teoría de grafos, la informática, la ingeniería informática y la investigación operativa. Por lo tanto, muchos problemas originados en uno de estos campos suelen tener múltiples traducciones a SAT y existen muchas herramientas

matemáticas disponibles para que el SAT-solver pueda conseguir soluciones con un rendimiento mejorado.

En este capítulo se introducen conceptos teóricos sobre la satisfacibilidad, resaltando los aspectos aplicables al modelado y posterior resolución de circuitos booleanos. También se describen los formatos que van a ser usados para el modelado, con la finalidad de emplear SAT-solvers para obtener los vectores de verificación de los circuitos. Para tener éxito en esto, deben ser introducidas las ideas que han surgido durante los numerosos intentos de razonar con la lógica, requiriendo una terminología y una perspectiva que se ha desarrollado en los últimos dos milenios.

Por último, también se describe la metodología y usos de la programación lineal entera mixta (MILP), así como el formato de descripción que utiliza.

3.2 SATISFACIBILIDAD BOOLEANA (SAT)

La lógica proposicional ha sido reconocida a lo largo de los siglos como una de las piedras angulares del razonamiento en filosofía y matemáticas. A lo largo de la historia, su formalización en álgebra booleana fue acompañada gradualmente por el reconocimiento de que una amplia gama de problemas combinatorios puede expresarse como problemas de satisfacibilidad proposicional (SAT). Debido a estas dos funciones, SAT se ha convertido en una disciplina científica madura con múltiples facetas. Iniciado por el trabajo de Cook [15], que estableció su estatus NP-completo en 1971, SAT se ha convertido en un problema de referencia para una enorme variedad de especificaciones de gran complejidad.

Simultáneamente, muchos problemas del mundo real se formalizaron como problemas de decisión SAT utilizando diferentes técnicas de codificación. Esto incluye problemas de verificación en software y hardware, planificación, programación y diseño combinatorio.

Debido a las implicaciones prácticas potenciales, desde los comienzos de la informática ha estado en marcha una búsqueda intensiva de cómo se podrían resolver los problemas SAT de manera automatizada. En 1957, Allen Newell y Herb

Simon introdujeron uno de los primeros programas de inteligencia artificial (AI), la Máquina de Lógica Teórica [16], para probar los teoremas proposicionales del "Principia Mathematica" de Whitehead y Russell. Poco después, en 1960, Martin Davis y Hillary Putnam introdujeron su ahora famoso procedimiento de decisión para problemas de satisfacibilidad proposicional [17]. En 1962 siguieron Martin Davis, George Logemann y Donald Loveland con una versión más eficiente del espacio [18].

El problema de satisfacibilidad booleana consiste en determinar si existe una asignación de variables satisfactoria para una función booleana, o determinar que no existe esta posibilidad. SAT tiene muchas aplicaciones directas en el campo de la automatización del diseño electrónico (EDA), incluyendo la generación de patrones de prueba, análisis de tiempos, verificación de lógica, pruebas funcionales, etc. SAT pertenece a la clase de problemas NP-completos, con soluciones algorítmicas que pueden tener una complejidad exponencial en el peor de los casos. Este problema ha sido y sigue siendo un área importante de investigación porque las técnicas eficientes de SAT pueden tener un gran impacto en el rendimiento de muchas herramientas EDA actuales.

Con respecto a las aplicaciones en VLSI CAD (Computer-Aided Design), la mayoría de las instancias de formulaciones SAT comienzan a partir de una descripción abstracta del circuito, para lo cual se necesita validar un valor de salida requerido. La formulación resultante se mapea a continuación sobre una instancia de SAT, típicamente usando formulación en formato de Forma Normal Conjuntiva (CNF).

Los SAT-solvers basados en CNF pueden aplicarse directamente a los circuitos booleanos transformando todo el circuito en fórmulas CNF.

La mayoría de los SAT-solvers operan sobre problemas especificados en formato CNF. Esta forma consiste en el AND lógico de una o más cláusulas, que son a su vez un OR lógico de uno o más literales. El literal comprende la unidad lógica fundamental en el problema, siendo simplemente una instancia de una variable o su complemento. Todas las funciones booleanas se pueden describir en el formato

CNF. La ventaja de CNF es que, en esta forma, para que el problema sea satisfecho, cada cláusula individual debe ser satisfecha.

3.3 FORMATO DE DESCRIPCIÓN PARA SAT: CNF

Antes de que un problema combinatorio pueda ser resuelto por los métodos actuales de SAT, normalmente debe ser transformado (codificado) a formato CNF, que consiste en una conjunción de cláusulas Ci, donde cada cláusula Ci es una disyunción de literales Lj, y cada literal Lj es una variable booleana o su negación. CNF tiene la ventaja de ser un formato muy simple, que conduce a la fácil implementación de algoritmos de búsqueda. Además, es un formato de archivo común. Lamentablemente, hay varias maneras de codificar la mayoría de los problemas y pocas directrices sobre cómo elegir entre ellos, sin embargo, la elección de la codificación puede ser tan importante como la elección del algoritmo de búsqueda.

Este formato se utiliza para definir una expresión booleana, escrita en CNF, que se puede utilizar como entrada para el problema de satisfacibilidad (SAT). Este problema considera el caso en el que se usan N variables booleanas para formar una expresión booleana que implica negación (NOT), conjunción (AND) y disyunción (OR). El problema es determinar si hay alguna asignación de valores a las variables booleanas que hace que la fórmula sea verdadera. Es algo así como tratar de mover un montón de interruptores para encontrar la combinación que hace que una bombilla se encienda.

En resumen, para que un problema SAT sea resuelto es común requerir que la expresión booleana sea escrita en formato CNF, que consiste en:

1. *cláusulas* unidas por AND;
2. cada cláusula, a su vez, consiste en *literales* unidos por OR;
3. cada literal es el nombre de una variable (*un literal positivo*) o el nombre de una variable precedida por NOT (*un literal negativo*).

En la Figura 3 se muestra un ejemplo de una expresión booleana de 3 variables y 2 cláusulas.

```
( x(1) OR ( NOT x(3) ) )
AND
( x(2) OR x(3) OR ( NOT x(1) ) )
```

Figura 3: Expresión booleana

El formato de archivo CNF es un formato de archivo ASCII.

1. El archivo puede comenzar con líneas de comentario. El primer carácter de cada línea de comentario debe ser una letra minúscula "c". Las líneas de comentario normalmente se sitúan en una sección al principio del archivo, pero se les permite aparecer en todo el archivo.

2. Las líneas de comentario son seguidas por la línea de "problema". Esto comienza con la letra minúscula "p" seguida por un espacio, seguido por el tipo de problema, que para los archivos CNF es "cnf", seguido por el número de variables seguido por el número de cláusulas.

3. El resto del archivo contiene líneas que definen las cláusulas, una por una.

4. Una cláusula se define enumerando el índice de cada literal positivo y el índice negativo de cada literal negativo. Los índices comienzan en `1` y, por razones obvias, el índice `0` no está permitido.

5. La definición de una cláusula puede extenderse más allá de una sola línea de texto.

6. La definición de una cláusula se termina por un valor final de "`0`".

7. El archivo finaliza después de que se define la última cláusula.

Algunas curiosidades a tener en cuenta incluyen:

- La definición de la siguiente cláusula comienza normalmente en una nueva línea, pero puede seguir, en la misma línea, al "0" que marca el final de la cláusula anterior.
- En algunos ejemplos de archivos CNF, la definición de la última cláusula no termina con un "0" final.
- En algunos ejemplos de archivos CNF, no se sigue la regla de numeración de las variables de 1 a N. El archivo puede declarar que hay 10 variables, por ejemplo, pero permitir que se numeren de 2 a 11.

3.3.1 EJEMPLO CNF

En la Figura 4 se muestra el archivo CNF que corresponde a la expresión booleana simple mostrada en la Figura 3.

```
c simple_v3_c2.cnf
p cnf 3 2
1 -3 0
2 3 -1 0
```

Figura 4: Codificación en formato CNF

3.3.2 FORMATO DE ARCHIVO DIMACS PARA CNF

En el DIMACS Challenge de 1993 [19] fue ideado un formato de archivo para problemas de SAT en CNF y ha sido seguido desde entonces. También se propuso un formato para el SAT general, pero no parece haber sido adoptado en general. Tener un formato de archivo común ha facilitado la recolección de problemas benchmark de SAT en el sitio web de SATLIB, los desafíos de Beijing y las competiciones regulares de SAT-solvers han estimulado una gran cantidad de investigación sobre algoritmos eficientes e implementaciones.

En este formato al principio hay un preámbulo que contiene información sobre el archivo, como pueden ser su nombre, procedencia, etc. Se trata de líneas de

comentario, que son opcionales y comienzan con la letra "c" como puede verse en el ejemplo mostrado en la Figura 5.

```
c Este es un ejemplo de una línea de comentario.
```

Figura 5: Línea de comentario en formato CNF

Por convenio, la siguiente línea debe describir la dimensión del problema, indicando la cantidad de variables y cláusulas que tiene el problema codificado en CNF. En la codificación CNF solo hay una línea de este tipo y comienza con la letra "p". El formato de esta línea es el que se muestra en la Figura 6.

```
p cnf variables cláusulas
```

Figura 6: Línea de problema en formato CNF

Como ya se ha dicho, esta línea indica el número de variables y cláusulas booleanas en el archivo, donde las variables y cláusulas son enteros positivos, siendo las variables numeradas desde `1` hasta `variables`.

El resto del archivo contiene las cláusulas. Cada cláusula está representada por una lista de enteros, seguida de un cero que representa el final de la cláusula. Los números enteros deben estar separados por espacios, tabuladores o salto de línea. Un número entero positivo `i` representa un literal positivo con la variable número `i`, mientras que un número entero negativo `-i` representa un literal negativo con la variable número `i`. Por ejemplo, la línea mostrada en la Figura 7 representa la cláusula `v(1) OR v(3) OR ( NOT v(7) )`.

```
1 3 -7 0
```

Figura 7: Cláusula en formato CNF

Las cláusulas pueden extenderse por más de una línea; los espacios y tabuladores pueden insertarse libremente entre enteros, y tanto el orden de literales en una cláusula como el orden de cláusulas en el archivo son irrelevantes en cuanto a la descripción del problema, pero tienen cierta importancia en cuanto a que los SAT-solvers pueden tardar más o menos tiempo en resolver un problema dependiendo del orden de las cláusulas y sus literales.

3.3.3 PROPIEDADES DESEABLES DE LAS CODIFICACIONES CNF

Hay algunas características de una codificación CNF que la hacen mejor que otra. A continuación, se exponen características que se han propuesto como deseables.

3.3.3.1 Tamaño de codificación

Por el tamaño de una codificación se puede estimar el número de sus cláusulas, literales o variables. En primer lugar, se considera el número de cláusulas. Existen diversas técnicas para reducir la cantidad de cláusulas en un archivo CNF, posiblemente convirtiendo una codificación grande intratable en una más manejable.

En segundo lugar, se tiene en cuenta el número total de literales en la codificación, que se calcula sumando las longitudes de la cláusula, lo cual también tiene un fuerte efecto sobre los *overheads* de tiempo de ejecución. Como medida de la memoria utilizada por una codificación es más preciso que el número de cláusulas.

En tercer lugar, se considera el número de variables en la codificación. Analizando el tiempo de ejecución de los algoritmos SAT, en términos de número de variables en la fórmula, parece útil minimizar el número de variables. Por otra parte, puede haber razones de modelado para no minimizar el número de variables, ya que puede ser más conveniente para expresar las limitaciones del problema.

En la práctica, reducir el tamaño de una codificación no es garantía de rendimiento, independientemente de cómo se mida. Sin embargo, las codificaciones pequeñas valen la pena porque los resultados computacionales pueden ser muy

impredecibles y, si no hay más diferencias que el tamaño, es preferible una codificación más pequeña.

3.3.3.2 Densidad de la solución

Otra propiedad interesante de una codificación es su densidad de solución, que puede definirse como el número de soluciones dividido por 2^n, donde n es el número de variables SAT. La idea de que una mayor densidad de solución hace que un problema sea más fácil de resolver parece natural, y normalmente se supone que tiene al menos algún efecto en el rendimiento de la búsqueda. Podría esperarse que un algoritmo de búsqueda pueda resolver un problema (satisfactorio) más rápidamente si tiene mayor densidad de solución.

3.4 PROGRAMACIÓN LINEAL ENTERA MIXTA (MILP)

A lo largo de más de 50 años de existencia, la teoría de la programación lineal entera mixta (MILP) y la práctica se han desarrollado significativamente, y ahora es una herramienta indispensable en los negocios y la ingeniería. Dos razones para el éxito de MILP son los *solvers* basados en la programación lineal (LP) y la flexibilidad del modelado con MILP. Existen varios solucionadores extremadamente eficaces que incorporan muchas técnicas avanzadas y, desde sus primeras etapas, MILP se ha utilizado para modelar una amplia gama de aplicaciones

Un MILP es un LP más restringido por las restricciones de integralidad. Se dice que es un MILP cuando algunas, pero no todas, las variables están restringidas a ser enteros.

MILP maximiza o minimiza una función objetivo lineal sujeta a desigualdades lineales y restricciones de integridad en algunas de las variables.

La Programación Lineal (LP) se ocupa de maximizar o minimizar una ecuación sujeta a ciertos criterios o restricciones. Un ejemplo simple podría ser el dado en la Figura 8.

```
Max: z = 3x1 - x2 + 5x3
Tal que    x1 + 2x2 - 4x3 ≤ 7
           7x1 - 5x2 - x3 ≤ 2
           -x1 + 3x2 + 9x3 ≤ 13
           xi ≥ 0, i = 1,. . . , 3
```

Figura 8: Problema de Programación Lineal

El problema de la Figura 8 puede resolverse usando el Método Simplex para encontrar una solución de: `x = (x1, x2, x3) = (0.5, 0, 1.5)`.

En MILP algunas partes de la solución pueden estar restringidas a números enteros. Los MILP son LP que requieren que algunas o todas las variables sean enteros. Redondear simplemente la solución a menudo comprometerá la solución, ya que el redondeo puede no estar en la región factible.

Una forma de resolver MILP es enumerar todas las soluciones enteras en la región factible y verificar individualmente cada una de ellas para obtener la óptima. Sin embargo, a medida que las dimensiones del problema crezcan, la enumeración se revelará como un problema de complejidad computacional de clase NP complejo (NP-hard), lo que significa que el número de soluciones enteras factibles crecerá exponencialmente.

3.4.1 REGLAS DE SINTAXIS DEL FORMATO DE ARCHIVO LP

Cualquier cosa que siga una barra invertida (\) es un comentario y se ignorará hasta que se encuentre un final de línea. Las líneas en blanco también se ignoran. Las líneas en blanco y de comentario pueden colocarse en cualquier lugar y con la frecuencia que se desee en el archivo.

En general, el espacio en blanco entre caracteres es irrelevante, ya que se omite cuando se lee un archivo. Sin embargo, no se permite el espacio en blanco en las palabras clave utilizadas para introducir una nueva sección, como MAX, MIN, ST o BOUNDS. También las palabras clave deben estar separadas por espacio en blanco

del resto del archivo y deben estar al principio de una línea. La longitud máxima de línea permitida es de 255 caracteres.

Saltar espacios puede hacer que se interprete mal (y acepte) una entrada no válida, como la de la Figura 9. Si el usuario pretendía entrar en este ejemplo una restricción no lineal en que el producto de `x1` por `x2` fuera `0`, restricción que no es válida en formato LP, se interpretaría como una restricción lineal especificando que una variable denominada `x1x2` debe ser igual a cero.

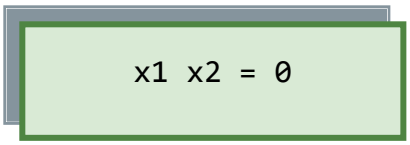


Figura 9: Uso de espacios en LP

La definición del problema debe comenzar con la palabra MINIMIZE o MAXIMIZE, MINIMUM o MAXIMUM, o las abreviaturas MIN o MAX, en cualquier combinación de mayúsculas y minúsculas. La palabra introduce la sección de función objetivo.

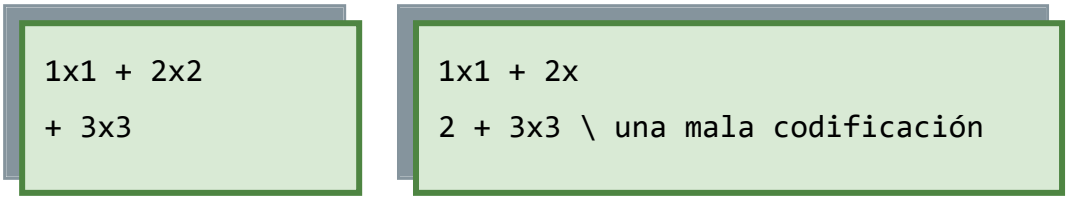
Las variables se pueden nombrar siempre que el nombre no exceda de 16 caracteres, todos los cuales deben ser alfanuméricos (a-z, A-Z, 0-9) o uno de estos símbolos: ! " # $ % & ( ) / , . ; ? @ _ ` ' { } | ~. Los nombres más largos se truncarán a 16. El nombre de una variable no puede comenzar con un número o un punto.

La letra E o e, sola o seguida por otros símbolos válidos, o seguida por otra E o e, debe evitarse ya que esta notación está reservada para entradas exponenciales. Por lo tanto, las variables no pueden llamarse e9, E-24, E8cats u otros nombres que podrían interpretarse como un exponente. Incluso nombres de variables como "entradas" o "ejemplo" pueden causar un error de lectura, dependiendo de su ubicación en una línea de entrada.

Además, los siguientes caracteres no son válidos en los nombres de las variables: ^, *, [y].

La definición de función objetivo debe seguir a MINIMIZE/MAXIMIZE. Se puede introducir en varias líneas siempre y cuando ni una variable, ni una constante,

ni un indicador de sentido sea dividido por un final de línea. Por ejemplo, la función objetivo `1x1 + 2x2 + 3x3` se puede introducir en la forma dada en la Figura 10(a), pero no de la forma de la Figura 10(b), porque este segundo modo divide el nombre de la variable x2 por un retorno.



(a) *(b)*

Figura 10: Uso de final de línea en LP

La función objetivo puede ser nombrada escribiendo un nombre y dos puntos antes de la función objetivo. El nombre de la función objetivo y los dos puntos deben aparecer en la misma línea. Los nombres de las funciones objetivo deben ajustarse a las mismas directrices que los nombres de las variables.

La sección de restricciones es introducida por las palabras clave SUBJECT TO. Esta expresión también puede aparecer como such that, st, S.T. o ST. En cualquier combinación de mayúsculas y minúsculas. Una de estas expresiones debe preceder a la primera restricción y estar separada de ella al menos por un espacio.

Cada definición de restricción debe comenzar en una nueva línea. Una restricción puede ser nombrada escribiendo un nombre y dos puntos antes de la restricción. El nombre de la restricción y los dos puntos deben aparecer en la misma línea. Los nombres de restricciones deben cumplir con las mismas directrices que los nombres de variables. Si no se especifican nombres de restricciones, se asignarán los nombres R1, R2, R3, etc.

Las restricciones se introducen de la misma manera que la función objetivo; sin embargo, una restricción debe ser seguida por una indicación de su sentido y un coeficiente a su derecha. El coeficiente del lado derecho debe escribirse en la misma línea que el indicador de sentido. Los indicadores de sentido aceptables son <, <=, =<,>,>=, => y =. Estos se interpretan como <=, <=, <=, >=, >=, >= y =, respectivamente.

Por ejemplo, en la Figura 11 hay una restricción con nombre:

```
time: x1 + x2 <= 10
```

Figura 11: Restricción con nombre en LP

La sección BOUNDS (límites) es opcional y sigue a la sección obligatoria de restricciones. Está precedida por la palabra bounds o bound en cualquier combinación de mayúsculas y minúsculas.

Cada definición de límites debe comenzar en una nueva línea. El formato para un límite es l_n<= x_n <= u_n excepto en los siguientes casos:

1. Los límites superior e inferior también se pueden introducir por separado como en la Figura 12.

l_n <= x_n

x_n <= u_n

Figura 12: Bounds en LP

2. Con el límite inferior predeterminado de 0 (cero) y el límite superior por defecto de +∞, permaneciendo en efecto hasta que el límite se cambia explícitamente.

3. Los límites que fijan una variable se pueden introducir como igualdades simples. Por ejemplo, x5 = 5.6 es equivalente a 5.6 <= x5 <= 5.6.

Los límites +∞ (infinito positivo) y -∞ (infinito negativo) deben introducirse como palabras: +infinity, -infinity, +inf, -inf.

Una variable con un límite inferior infinito negativo y un límite superior infinito positivo se puede introducir como free, en cualquier mezcla de mayúsculas

y minúsculas, con un espacio que separa el nombre de la variable y la palabra free. Por ejemplo, x7 free equivale a -infinity <= x7 <= +infinity.

El archivo debe terminar con la palabra end en cualquier combinación de mayúsculas y minúsculas, solo en una línea. Esta palabra no es estrictamente necesaria, pero se recomienda encarecidamente. Los archivos que se han dañado con frecuencia se pueden detectar por una última línea que falta.

Para especificar cualquiera de las variables como variables enteras generales, se agrega una sección GENERAL; para especificar cualquiera de las variables como variables binarias enteras, se añade una sección BINARY. Las secciones GENERAL y BINARY siguen la sección BOUNDS, si hay una; de lo contrario, siguen la sección de restricciones. Cualquiera de las secciones GENERAL o BINARY puede preceder a la otra. La sección GENERAL está precedida por la palabra GENERAL, GENERALS, o GEN que debe aparecer solo en una línea. La línea o líneas siguientes deben enumerar los nombres de todas las variables que deben restringirse a valores enteros generales, separados por al menos un espacio. La sección BINARY está precedida por la palabra BINARY, BINARIES o BIN que debe aparecer solo en una línea. La línea o líneas siguientes deben enumerar los nombres de todas las variables que deben restringirse a valores enteros binarios, separados por al menos un espacio. Las variables binarias reciben automáticamente los límites de 0 (cero) y 1 (uno), a menos que se especifiquen límites alternativos en la sección BOUNDS, en cuyo caso se emite un mensaje de advertencia.

En la Figura 13 se muestra un ejemplo de una formulación de problemas en formato LP donde x4 es un entero general.

Para especificar cualquiera de las variables como variables semi-continuas, es decir, como variables que pueden tomar el valor 0 o valores entre los límites inferiores y superiores especificados, se utiliza la sección SEMI-CONTINUOUS. Esta sección debe seguir las secciones BOUNDS, GENERALS y BINARIES. La sección SEMI-CONTINUOUS está precedida por la palabra clave SEMI-CONTINUOUS, SEMI o SEMIS. La línea o líneas siguientes deben enumerar los nombres de todas las

variables que se van a declarar semi-continuas, separadas por al menos un espacio como en la figura 14.

```
Maximize
     obj: x1 + 2 x2 + 3 x3 + x4
Subject To
     c1: - x1 + x2 + x3 + 10 x4 <= 20
     c2: x1 - 3 x2 + x3 <= 30
     c3: x2 - 3.5 x4 = 0
Bounds
     0 <= x1 <= 40
     2 <= x4 <= 3
General
     x4
End
```

Figura 13: Codificación LP

```
Semi-continuous
     x1 x2 x3
```

Figura 14: Variables semi-continuas en LP

Para especificar conjuntos especiales ordenados, hay que utilizar una sección SOS, que está precedida por la palabra clave SOS. La sección SOS debe seguir a las secciones Bounds, General, Binaries y Semi-Continuous. Los conjuntos ordenados especiales de tipo 1 requieren que, de las variables en el conjunto, una como máximo pueda ser distinta de cero. Los conjuntos ordenados especiales del tipo 2 requieren que como máximo dos variables en el conjunto puedan ser distintas de cero y, si hay dos distintas de cero, deben ser adyacentes. La adyacencia se define por los pesos, que deben ser únicos dentro de un conjunto dado a las variables. Los

pesos ordenados definen el orden del conjunto ordenado especial. El conjunto se especifica mediante un nombre de conjunto opcional seguido de dos puntos y, a continuación, una de las palabras clave S1 o S2 (que especifica el tipo) seguido de dos puntos dobles. Los nombres de los miembros del conjunto se enumeran en esta línea o líneas, con sus pesos. Los nombres y pesos de las variables están separados por dos puntos como puede verse en la Figura 15.

```
SOS
     set1: S1:: x1:10 x2:13
```

Figura 15: Conjuntos especiales ordenados en LP

3.5 LP-SOLVER CPLEX®

CPLEX® [20] es una herramienta para resolver problemas de optimización relacionados con LP. Específicamente, resuelve problemas de optimización con restricciones lineales o cuadráticas, donde el objetivo a optimizar puede expresarse como una función lineal o una función cuadrática convexa. Las variables en el problema pueden ser declaradas como continuas o restringidas para tomar solo valores enteros.

El Optimizador Interactivo de CPLEX® es un programa ejecutable que puede leer un problema interactivamente o de archivos en ciertos formatos estándar, resolver el problema y entregar la solución interactivamente o en archivos de texto. El programa consiste en el archivo `cplex.exe` en plataformas Windows o `cplex` en plataformas UNIX.

CPLEX® es, entre otras, una herramienta para resolver problemas de programación matemática en los cuales algunas o todas las variables deben asumir valores enteros en la solución. Tales problemas se conocen como programas enteros mixtos o MIP porque pueden combinar variables continuas y discretas (por ejemplo, enteras) en la función objetivo y en las restricciones. Los MIP con objetivos lineales se denominan programas lineales enteros mixtos o MILP, y los MIP con términos

objetivos cuadráticos se denominan programas cuadráticos enteros mixtos o MIQP. Del mismo modo, MIPs que también son cuadráticamente restringidos en el sentido de QCP se conocen como programas mixtos enteros cuadráticamente restringidos o MIQCPs.

Dentro de la categoría de programas enteros mixtos, hay dos tipos de variables enteras discretas: si los valores enteros de las variables discretas deben ser `0` (cero) o `1` (uno), entonces se les conoce como binarios; si los valores enteros no están restringidos de esa manera, se les conoce como variables enteras generales.

3.6 CONCLUSIONES

La satisfacibilidad está estrechamente relacionada con los conceptos centrales de la lógica, en los cuales tiene sus orígenes. Es más, la presencia de la satisfacibilidad en la lógica surge para modelar el pensamiento humano y el razonamiento científico a través de su uso en el diseño de computadoras y ahora se utiliza como herramienta de modelado para resolver gran variedad de problemas prácticos. Por todo ello, se ha revelado como una potente herramienta que permite modelar y resolver multitud de problemas, principalmente aquellos que comparten sus raíces en la lógica como es el caso del diseño y prueba de circuitos.

El formato de Forma Normal Conjuntiva (CNF) es el más ampliamente usado para describir los problemas que deberán resolver los SAT-solvers, razón por la que en este trabajo se toma como formato de partida. Una instancia de SAT es una fórmula de CNF Φ. El problema de satisfacibilidad es determinar si existe una asignación de valores a las variables de Φ que hace que Φ se evalúe a `1`.

Los modelos de programación entera surgen en prácticamente todas las áreas de aplicación de la programación matemática. Una amplia gama de problemas se puede modelar como problemas de Programación Lineal Entera Mixta (MILP). Se debe desarrollar una apreciación preliminar de la importancia de estos modelos.

CAPÍTULO 4
MÉTODO DESARROLLADO

4.1 INTRODUCCIÓN

En este Proyecto Fin de Carrera se propone un nuevo método para la generación de vectores para la verificación de circuitos booleanos haciendo uso de Programación Lineal Entera Mixta

Tras haber presentado los diferentes entornos en los que se enmarca este trabajo, se procede en este capítulo a exponer el método desarrollado. Para comenzar, se presenta el método de una forma genérica. Seguidamente, se explica cómo se ha hecho cada una de las partes que lo componen. A continuación, se muestran los modelos MILP desarrollados y los programas que han sido creados para describir los circuitos en MILP. Finalmente se explica el programa desarrollado para conseguir las soluciones de los circuitos y obtener vectores para la verificación de los mismos.

4.2 ENTORNO DE VERIFICACIÓN

Tal como se prevé en los objetivos, este entorno se ejecuta a nivel de línea de comandos y se ha desarrollado completamente en C. Su ejecución está prevista para máquinas Linux o Unix.

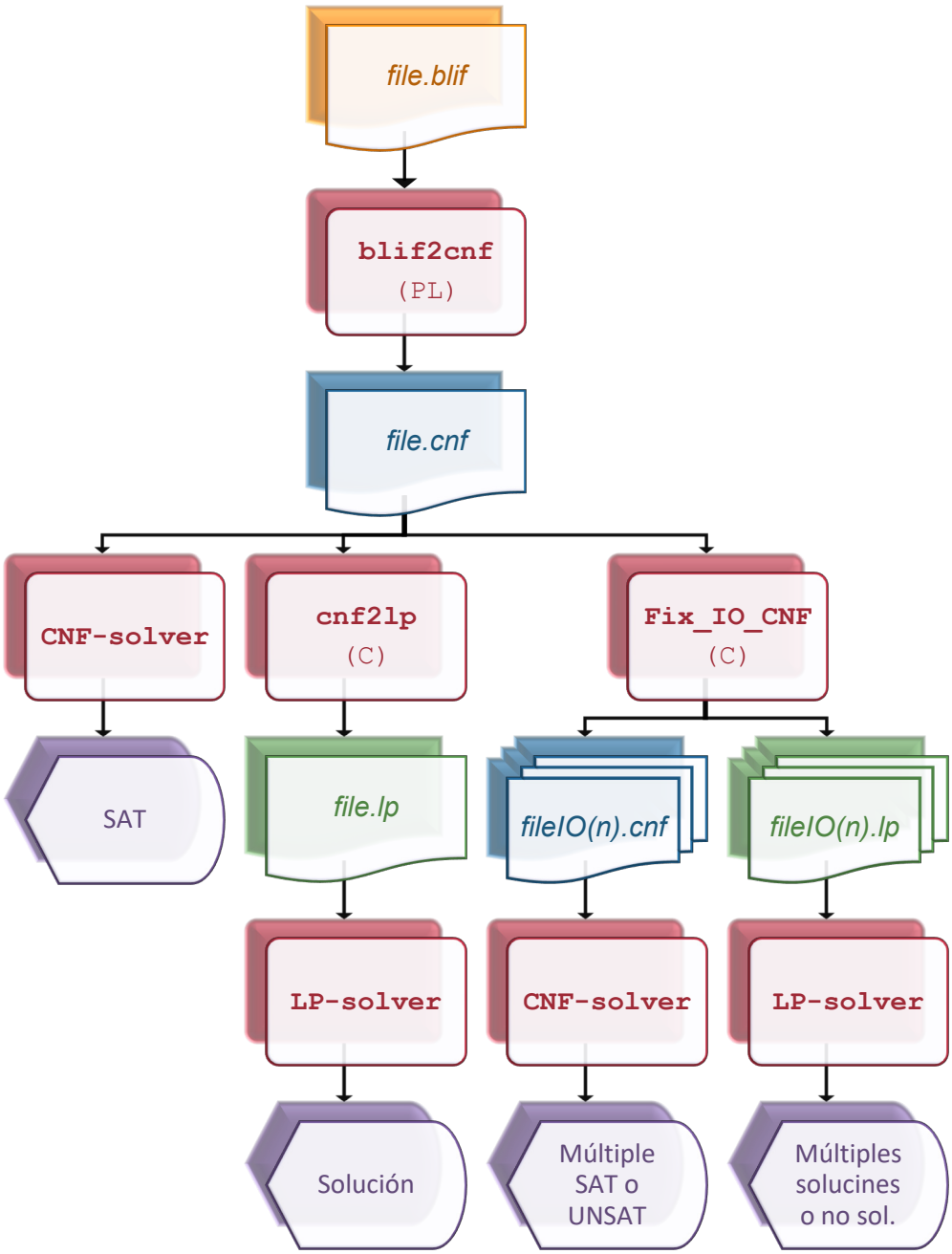


Figura 16: Entorno de Verificación

En la Figura 16 se muestra un diagrama de flujo en el que pueden observarse las potencialidades del entorno de verificación diseñado. Para explicar las

alternativas que ofrece, se muestra un ejemplo de circuito booleano al que se le aplica el método desarrollado.

El circuito elegido es `C17` (Figura 17), se trata de un circuito booleano compuesto por 6 puertas, y cuenta con 5 entradas y 2 salidas.

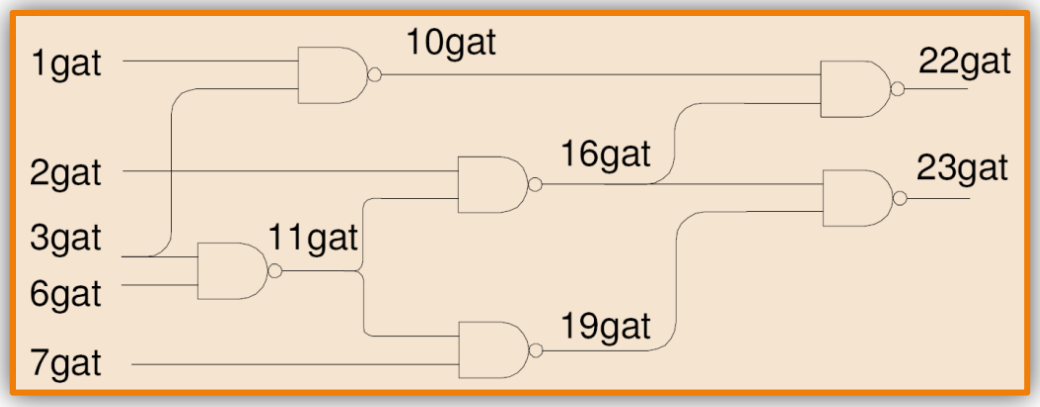


Figura 17: Circuito Booleano `C17`

La especificación de este circuito es un fichero en formato BLIF (Berkeley Logic Interchange Format), con nombre `C17.blif` (Figura 18).

```
# C17.blif
.model C17.iscas
.inputs 1GAT(0) 2GAT(1) 3GAT(2) 6GAT(3) 7GAT(4)
.outputs 22GAT(10) 23GAT(9)
.names 3GAT(2) 6GAT(3) 11GAT(5)
11 0
.names 1GAT(0) 3GAT(2) 10GAT(6)
11 0
.names 11GAT(5) 7GAT(4) 19GAT(7)
11 0
.names 2GAT(1) 11GAT(5) 16GAT(8)
11 0
.names 16GAT(8) 19GAT(7) 23GAT(9)
11 0
.names 10GAT(6) 16GAT(8) 22GAT(10)
11 0
.end
```

Figura 18: Codificación `C17.blif`

El circuito `C17`, descrito en el archivo `C17.blif`, es transformado al formato CNF con el script `blif2cnf.pl`. El resultado de esta transformación es el fichero

`C17.cnf` (Figura 19) que contiene la descripción del circuito (Figura 20) como un problema SAT en CNF compuesto por `30` cláusulas y `17` variables.

```
c CNF file generated from BLIF2CNF script.
c Original BLIF file was C17.blif
p cnf 17 30
1 -4 0
2 -4 0
4 -1 -2  0
-4 -3 0
3 4  0
5 -7 0
1 -7 0
7 -5 -1  0
-7 -6 0
6 7  0
3 -10 0
8 -10 0
10 -3 -8  0
-10 -9 0
9 10  0
11 -13 0
3 -13 0
13 -11 -3  0
-13 -12 0
12 13  0
12 -15 0
9 -15 0
15 -12 -9  0
-15 -14 0
14 15  0
6 -17 0
12 -17 0
17 -6 -12  0
-17 -16 0
16 17  0
```

Figura 19: Descripción `C17.cnf`

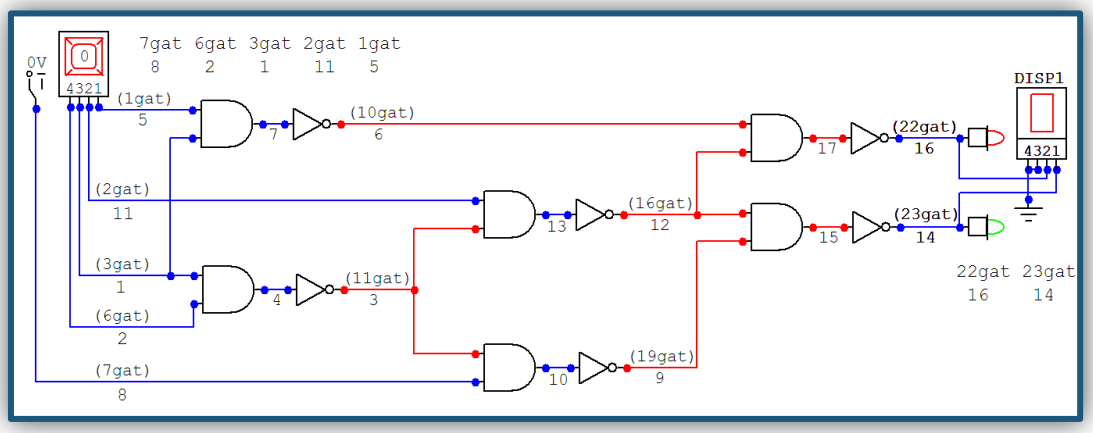


Figura 20: Circuito descrito en `C17.cnf`

Con este fichero `C17.cnf` primario se presentan tres opciones diferentes:

1. Resolverlo con un solucionador CNF.

2. Convertirlo a un problema de programación lineal en formato MILP, con el programa `cnf2lp.c`, diseñado para este PFC, obteniendo como resultado el archivo `C17.lp` (Figura 21), con la codificación del problema SAT en formato MILP, que contendrá tantas restricciones como cláusulas tenía el archivo `C17.cnf`, más una para la puerta AND final. Esto es así porque cada cláusula es modelada como una ecuación lineal. En el ejemplo del `C17.lp` habrá `31` restricciones.

3. Fijar un porcentaje de entradas y/o salidas del circuito con el programa `Fix_IO_CNF.c`, también diseñado para este PFC. En esta última opción, se obtienen tantos ficheros como se indique, con diferentes valores de entradas y/o salidas fijadas según los porcentajes de entradas y salidas que se especifique que deban ser fijadas. Estos ficheros están en formato CNF y tendrán la misma cantidad de cláusulas que el CNF primario, más el número de entradas y salidas fijadas. El programa `Fix_IO_CNF.c` también ejecuta el programa `cnf2lp.c`, con lo que se obtienen también los ficheros en formato LP con las mismas entradas y/o salidas fijadas que en los archivos CNF. Los archivos LP tendrán, al igual que en el caso anterior, la misma cantidad de restricciones que cláusulas tiene el fichero CNF más una. El nombre de estos ficheros tiene el formato `C17_PI_PO_ixxx.cnf` y `C17_PI_PO_ixxx.lp`, donde `PI` es el porcentaje de entradas que se desea fijar, `PO` es el porcentaje de salidas a fijar e `ixxx` indica de qué iteración se trata, dado que se pueden pedir varias y todas deberán ser diferentes. En la Figura 23 se muestra un ejemplo para el circuito C17 con el 100% de las salidas fijadas.

El fichero en formato MILP generado en la segunda opción será resuelto con el LP-solver CPLEX, obteniendo una solución al circuito (Figura 22).

```
max obj: y1
subject to
c1: 2 y1 - aux0 - b1 + b4 = 1
c2: 2 y2 - aux1 - b2 + b4 = 1
c3: 4 y3 - 2 aux3 - aux2 - 2 b4 + b1 + b2 = 2
c4: 2 y4 - aux4 + b4 + b3 = 2
c5: 2 y5 - aux5 - b3 - b4 = 0
c6: 2 y6 - aux6 - b5 + b7 = 1
c7: 2 y7 - aux7 - b1 + b7 = 1
c8: 4 y8 - 2 aux9 - aux8 - 2 b7 + b5 + b1 = 2
c9: 2 y9 - aux10 + b7 + b6 = 2
c10: 2 y10 - aux11 - b6 - b7 = 0
c11: 2 y11 - aux12 - b3 + b10 = 1
c12: 2 y12 - aux13 - b8 + b10 = 1
c13: 4 y13 - 2 aux15 - aux14 - 2 b10 + b3 + b8 = 2
c14: 2 y14 - aux16 + b10 + b9 = 2
c15: 2 y15 - aux17 - b9 - b10 = 0
c16: 2 y16 - aux18 - b11 + b13 = 1
c17: 2 y17 - aux19 - b3 + b13 = 1
c18: 4 y18 - 2 aux21 - aux20 - 2 b13 + b11 + b3 = 2
c19: 2 y19 - aux22 + b13 + b12 = 2
c20: 2 y20 - aux23 - b12 - b13 = 0
c21: 2 y21 - aux24 - b12 + b15 = 1
c22: 2 y22 - aux25 - b9 + b15 = 1
c23: 4 y23 - 2 aux27 - aux26 - 2 b15 + b12 + b9 = 2
c24: 2 y24 - aux28 + b15 + b14 = 2
c25: 2 y25 - aux29 - b14 - b15 = 0
c26: 2 y26 - aux30 - b6 + b17 = 1
c27: 2 y27 - aux31 - b12 + b17 = 1
c28: 4 y28 - 2 aux33 - aux32 - 2 b17 + b6 + b12 = 2
c29: 2 y29 - aux34 + b17 + b16 = 2
c30: 2 y30 - aux35 - b16 - b17 = 0
c31: y1 + y2 + y3 + y4 + y5 + y6 + y7 + y8 + y9 + y10 + y11 + y12 + y13 +
y14 + y15
 + y16 + y17 + y18 + y19 + y20 + y21 + y22 + y23 + y24 + y25 + y26 + y27 +
y28 + y29 + y30
 = 30
bin
y1 y2 y3 y4 y5 y6 y7 y8 y9 y10 y11 y12 y13 y14 y15
 y16 y17 y18 y19 y20 y21 y22 y23 y24 y25 y26 y27 y28 y29 y30
aux0 aux1 aux2 aux3 aux4 aux5 aux6 aux7 aux8 aux9 aux10 aux11 aux12 aux13
aux14 aux15
 aux16 aux17 aux18 aux19 aux20 aux21 aux22 aux23 aux24 aux25 aux26 aux27
aux28 aux29 aux30
 aux31 aux32 aux33 aux34 aux35
b1 b2 b3 b4 b5 b6 b7 b8 b9 b10 b11 b12 b13 b14 b15
 b16 b17
end
```

Figura 21: Codificación MILP `C17.lp`

```
Welcome to CPLEX Interactive Optimizer 11.1.1
 with Simplex, Mixed Integer & Barrier Optimizers
Copyright (c) ILOG 1997-2008  CPLEX is a registered trademark of ILOG
Type 'help' for a list of available commands.
Type 'help' followed by a command name for more information on commands.
CPLEX> New value for time limit in seconds: 600
CPLEX> Problem 'C17.lp' read.
Read time =    0.00 sec.
CPLEX> Tried aggregator 2 times.
MIP Presolve eliminated 3 rows and 34 columns.
Aggregator did 2 substitutions.
Reduced MIP has 26 rows, 47 columns, and 90 nonzeros.
Presolve time =    0.00 sec.
Clique table members: 26.
MIP emphasis: balance optimality and feasibility.
MIP search method: dynamic search.
Parallel mode: none, using 1 thread.
Root relaxation solution time =    0.00 sec.
        Nodes                                         Cuts/
   Node  Left     Objective  IInf  Best Integer     Best Node    ItCnt
Gap
      0     0        cutoff                                         0
Solution pool: 1 solution saved.
MIP - Integer optimal solution:  Objective =  1.0000000000e+00
Solution time =    0.00 sec.  Iterations = 0  Nodes = 0
CPLEX> Incumbent solution
Variable Name           Solution Value
y1                            1.000000
b1                            1.000000
b4                            1.000000
y2                            1.000000
. . .
b16                           1.000000
y30                           1.000000
All other variables in the range 1-83 are 0.
```

Figura 22: Solución de `C17.lp` obtenida con CPLEX

Los ficheros en formato CNF producidos en la tercera opción pueden ser resueltos con un solucionador CNF, lo que en algunos casos resultará en que encuentra que el problema es SAT y en otros podría dar como resultado UNSAT, puesto que no encuentra una solución SAT con la combinación de entradas y/o salidas fijadas en esa iteración.

Los ficheros en formato MILP obtenidos en la tercera opción serán resueltos también con el solucionador de problemas de programación lineal CPLEX,

obteniéndose soluciones en los casos que existan (Figura 24). En el caso particular del circuito C17 siempre hay una solución posible para todas las combinaciones de entradas y para todas las combinaciones de salidas, pero hay combinaciones de entradas y salidas que no tendrían solución (Tabla 3).

ENTRADAS					SALIDAS	
7gat	6gat	3gat	2gat	1gat	22gat	23gat
8	2	1	11	5	16	14
0	0	0	0	0	0	0
0	0	0	0	1	0	0
0	0	0	1	0	1	1
0	0	0	1	1	1	1
0	0	1	0	0	0	0
0	0	1	0	1	1	0
0	0	1	1	0	1	1
0	0	1	1	1	1	1
0	1	0	0	0	0	0
0	1	0	0	1	0	0
0	1	0	1	0	1	1
0	1	0	1	1	1	1
0	1	1	0	0	0	0
0	1	1	0	1	1	0
0	1	1	1	0	0	0
0	1	1	1	1	1	0
1	0	0	0	0	0	1
1	0	0	0	1	0	1
1	0	0	1	0	1	1
1	0	0	1	1	1	1
1	0	1	0	0	0	1
1	0	1	0	1	1	1
1	0	1	1	0	1	1
1	0	1	1	1	1	1
1	1	0	0	0	0	1
1	1	0	0	1	0	1
1	1	0	1	0	1	1
1	1	0	1	1	1	1
1	1	1	0	0	0	0
1	1	1	0	1	1	0
1	1	1	1	0	0	0
1	1	1	1	1	1	0

Tabla 3: Combinaciones de entradas y salidas posibles para C17

```
max obj: y1

subject to
c1: 2 y1 - aux0 - b1 + b4 = 1
c2: 2 y2 - aux1 - b2 + b4 = 1
c3: 4 y3 - 2 aux3 - aux2 - 2 b4 + b1 + b2 = 2
c4: 2 y4 - aux4 + b4 + b3 = 2
c5: 2 y5 - aux5 - b3 - b4 = 0
c6: 2 y6 - aux6 - b5 + b7 = 1
c7: 2 y7 - aux7 - b1 + b7 = 1
c8: 4 y8 - 2 aux9 - aux8 - 2 b7 + b5 + b1 = 2
c9: 2 y9 - aux10 + b7 + b6 = 2
c10: 2 y10 - aux11 - b6 - b7 = 0
c11: 2 y11 - aux12 - b3 + b10 = 1
c12: 2 y12 - aux13 - b8 + b10 = 1
c13: 4 y13 - 2 aux15 - aux14 - 2 b10 + b3 + b8 = 2
c14: 2 y14 - aux16 + b10 + b9 = 2
c15: 2 y15 - aux17 - b9 - b10 = 0
c16: 2 y16 - aux18 - b11 + b13 = 1
c17: 2 y17 - aux19 - b3 + b13 = 1
c18: 4 y18 - 2 aux21 - aux20 - 2 b13 + b11 + b3 = 2
c19: 2 y19 - aux22 + b13 + b12 = 2
c20: 2 y20 - aux23 - b12 - b13 = 0
c21: 2 y21 - aux24 - b12 + b15 = 1
c22: 2 y22 - aux25 - b9 + b15 = 1
c23: 4 y23 - 2 aux27 - aux26 - 2 b15 + b12 + b9 = 2
c24: 2 y24 - aux28 + b15 + b14 = 2
c25: 2 y25 - aux29 - b14 - b15 = 0
c26: 2 y26 - aux30 - b6 + b17 = 1
c27: 2 y27 - aux31 - b12 + b17 = 1
c28: 4 y28 - 2 aux33 - aux32 - 2 b17 + b6 + b12 = 2
c29: 2 y29 - aux34 + b17 + b16 = 2
c30: 2 y30 - aux35 - b16 - b17 = 0
c31: b16 = 1
c32: b14 = 0
c33: y1 + y2 + y3 + y4 + y5 + y6 + y7 + y8 + y9 + y10 + y11 + y12 + y13 +
y14 + y15 + y16 + y17 + y18 + y19 + y20 + y21 + y22 + y23 + y24 + y25 + y26
+ y27 + y28 + y29 + y30 = 30

bin
y1 y2 y3 y4 y5 y6 y7 y8 y9 y10 y11 y12 y13 y14 y15
 y16 y17 y18 y19 y20 y21 y22 y23 y24 y25 y26 y27 y28 y29 y30
aux0 aux1 aux2 aux3 aux4 aux5 aux6 aux7 aux8 aux9 aux10 aux11 aux12 aux13
aux14 aux15 aux16 aux17 aux18 aux19 aux20 aux21 aux22 aux23 aux24 aux25
aux26 aux27 aux28 aux29 aux30 aux31 aux32 aux33 aux34 aux35
b1 b2 b3 b4 b5 b6 b7 b8 b9 b10 b11 b12 b13 b14 b15 b16 b17
end
```

Figura 23: Codificación MILP `C17_0_100_i001.lp`

```
Welcome to CPLEX Interactive Optimizer 11.1.1
 with Simplex, Mixed Integer & Barrier Optimizers
Copyright (c) ILOG 1997-2008  CPLEX is a registered trademark of ILOG
Type 'help' for a list of available commands.
Type 'help' followed by a command name for more information on commands.
CPLEX> New value for time limit in seconds: 600
CPLEX> Problem 'C17_0_100_i001.lp' read.
Read time =    0.00 sec.
CPLEX> Tried aggregator 1 time.
MIP Presolve eliminated 23 rows and 71 columns.
Aggregator did 6 substitutions.
Reduced MIP has 4 rows, 8 columns, and 12 nonzeros.
Presolve time =    0.00 sec.
Clique table members: 4.
MIP emphasis: balance optimality and feasibility.
MIP search method: dynamic search.
Parallel mode: none, using 1 thread.
Root relaxation solution time =    0.00 sec.
        Nodes                                         Cuts/
   Node  Left     Objective  IInf  Best Integer     Best Node    ItCnt
Gap
      0     0        cutoff                                         0
Solution pool: 1 solution saved.
MIP - Integer optimal solution:  Objective =  1.0000000000e+00
Solution time =    0.00 sec.  Iterations = 0  Nodes = 0
CPLEX> Incumbent solution
Variable Name           Solution Value
y1                            1.000000
b1                            1.000000
b4                            1.000000
y2                            1.000000
b2                            1.000000
y3                            1.000000
y4                            1.000000
. . .
. . .
. . .
y29                           1.000000
b16                           1.000000
y30                           1.000000
All other variables in the range 1-83 are 0.
```

Figura 24: Solución de `C17_0_100_i001.lp` obtenida con CPLEX

4.3 FORMATO DE ENTRADA DE CIRCUITOS DE REFERENCIA

Como objetivo secundario, se planteaba hacer uso de un formato de entrada estándar. El formato con el que se desarrollan las pruebas hechas en este trabajo es BLIF. No obstante, se podrían utilizar estas herramientas con otros lenguajes de descripción hardware que se emplean en la literatura, como por ejemplo VHDL o Verilog, puesto que hay disponibilidad de traductores que permiten el paso de los circuitos al formato CNF con el que trabajan los programas desarrollados para este proyecto.

4.4 MODELADO DE LÓGICA BOOLEANA MEDIANTE MILP

Una amplia gama de problemas se puede modelar como problemas de Programación Lineal Entera Mixta (MILP) usando técnicas de formulación estándar. Sin embargo, en algunos casos el MILP resultante puede ser demasiado débil o demasiado grande para ser resuelto eficazmente por los solucionadores de última generación. En este trabajo se han desarrollado técnicas avanzadas de formulación para modelar mediante MILP la lógica booleana, resultando en formulaciones más fuertes y más pequeñas que las descritas en la literatura para esta clase de problemas.

Es importante destacar que la calidad de los modelos MILP desarrollados es de la máxima importancia, puesto que influirá directamente en el tamaño de la codificación resultante y en el tiempo que empleará el solucionador en encontrar la respuesta adecuada para verificar el circuito.

En este apartado se presenta el modelado de la lógica booleana que se ha hecho en este PFC utilizando las restricciones de MILP.

Todas las entradas y salidas primarias de la red booleana se declaran como variables binarias, mientras que el resto de los nodos intermedios se pueden dejar como variables continuas. Debido a la naturaleza binaria de la red booleana, las variables continuas asumirán automáticamente valores binarios también. Las variables auxiliares usadas para el modelado de las puertas lógicas también se declaran como variables binarias.

4.4.1 PUERTA OR

Considérese una puerta OR de 2 entradas, con entradas (b_1, b_2) y salida (Y). Según su tabla de verdad, lo siguiente es siempre verdadero: $2 \cdot Y - aux = b_1 + b_2$, donde aux es una variable auxiliar de tipo binario. Independientemente del valor que tome aux, la salida Y debe tomar el valor 1 cuando alguna de las entradas es 1 para que la ecuación lineal se cumpla. En resumen, la restricción lineal para la puerta OR de 2 entradas es la que se muestra en (4.1).

$$2 \cdot Y = aux + b_1 + b_2 \tag{4.1}$$

Para comprobar (4.1) se realiza la Tabla 4, en la que se exploran todas las combinaciones posibles de las variables binarias Y, aux, b_1, y b_2. En dicha tabla se puede comprobar que solo hay una solución en que se cumple la ecuación lineal (4.1) para cada uno de los cuatro casos de la tabla de verdad de una puerta OR de dos entradas (Tabla 5).

OR2	$2 \cdot Y - aux = b_1 + b_2$					
b_1	**b_2**	**Y**	aux	$2 \cdot Y - aux$	$b_1 + b_2$	Válido
0	0	0	0	0	0	✓
0	0	0	1	-1	0	✗
0	0	1	0	2	0	✗
0	0	1	1	1	0	✗
0	1	0	0	0	1	✗
0	1	0	1	-1	1	✗
0	1	1	0	2	1	✗
0	1	1	1	1	1	✓
1	0	0	0	0	1	✗
1	0	0	1	-1	1	✗
1	0	1	0	2	1	✗
1	0	1	1	1	1	✓
1	1	0	0	0	2	✗
1	1	0	1	-1	2	✗
1	1	1	0	2	2	✓
1	1	1	1	1	2	✗

Tabla 4: Comprobación del modelado MILP de la puerta OR de dos entradas

OR2	$2\cdot Y - aux = b_1 + b_2$					
b_1	**b_2**	**Y**	aux	$2\cdot Y - aux$	$b_1 + b_2$	Válido
0	0	0	0	0	0	✓
0	1	1	1	1	1	✓
1	0	1	1	1	1	✓
1	1	1	0	2	2	✓

Tabla 5: Tabla de verdad de la puerta OR de dos entradas

Esto se puede extender fácilmente a una puerta OR con un número arbitrario de entradas, con una ecuación lineal como la que se detalla en (4.2).

$$2^n \cdot Y = \sum_{k=0}^{n-1} (2^k \cdot aux_k) + \sum_{i=1}^{2^n} b_i \tag{4.2}$$

En (4.2), Y es la salida y b_i la i-ésima entrada de la puerta OR. Esta ecuación (4.2) se cumpliría siempre y cuando el número de entradas de la puerta OR sea la potencia de dos 2^n.

Si, por el contrario, el número de entradas fuera menor que 2^n, pero mayor que 2^{n-1}, entonces se duplicaría el peso de tantas entradas como diferencia haya entre 2^n y el número de entradas que tenga la puerta, quedando como puede verse en (4.3).

$$2^n \cdot Y = \sum_{k=0}^{n-1} (2^k \cdot aux_k) + 2 \cdot \sum_{i=1}^{2^n-\beta} b_i + \sum_{j=2^n-\beta+1}^{\beta} b_j \tag{4.3}$$

En (4.3), β es el número de entradas de la puerta OR y debe cumplir la condición de que $2^n > \beta > 2^{n-1}$.

4.4.2 PUERTA AND

Para una puerta AND de 2 entradas (b_1, b_2) y salida (Y). Según su tabla de verdad, lo siguiente es siempre verdadero: $2\cdot Y + aux = b_1 + b_2$, donde aux es una variable auxiliar de tipo binario. Independientemente del valor que tome aux, la

salida Y tomará el valor 1 solamente cuando todas las entradas son 1 para que la ecuación lineal se cumpla. En resumen, la restricción lineal para la puerta AND de 2 entradas es la ecuación (4.4).

$$2 \cdot Y = b_1 + b_2 - aux \tag{4.4}$$

Para comprobar el modelo para una puerta AND de dos entradas mediante la ecuación MILP (4.4), se realiza la Tabla 6, en la que se exploran todas las combinaciones posibles de las variables binarias Y, aux, b_1 y b_2. En dicha tabla puede comprobarse que solo hay una solución en que se cumple la ecuación lineal (4.4) para cada uno de los cuatro casos de la tabla de verdad de una puerta AND de dos entradas (Tabla 7).

AND2	$2 \cdot Y + aux = b_1 + b_2$					
b_1	**b_2**	**Y**	aux	$2 \cdot Y + aux$	$b_1 + b_2$	Válido
0	0	0	0	0	0	✓
0	0	0	1	1	0	✗
0	0	1	0	2	0	✗
0	0	1	1	3	0	✗
0	1	0	0	0	1	✗
0	1	0	1	1	1	✓
0	1	1	0	2	1	✗
0	1	1	1	3	1	✗
1	0	0	0	0	1	✗
1	0	0	1	1	1	✓
1	0	1	0	2	1	✗
1	0	1	1	3	1	✗
1	1	0	0	0	2	✗
1	1	0	1	1	2	✗
1	1	1	0	2	2	✓
1	1	1	1	3	2	✗

Tabla 6: Comprobación del modelado MILP de la puerta AND de dos entradas

AND2	2·Y + aux = $b_1 + b_2$					
b_1	**b_2**	**Y**	aux	2·Y + aux	$b_1 + b_2$	Válido
0	0	0	0	0	0	✓
0	1	0	1	1	1	✓
1	0	0	1	1	1	✓
1	1	1	0	2	2	✓

Tabla 7: Tabla de verdad de la puerta AND de dos entradas

Esto puede extenderse a una puerta AND con un mayor número de entradas. De modo que si el número de entradas es la potencia de dos 2^n, se cumpliría la ecuación lineal (4.5), donde Y sería la salida y b_i la i-ésima entrada de la puerta AND.

$$2^n \cdot Y = \sum_{i=1}^{2^n} b_i - \sum_{k=0}^{n-1} (2^k \cdot aux_k) \tag{4.5}$$

Si el número de entradas fuera menor que 2^n, pero mayor que 2^{n-1}, entonces se duplicaría el peso de tantas entradas como diferencia haya entre 2^n y el número de entradas que tenga la puerta, quedando una puerta AND de β entradas modelada con la ecuación lineal (4.6), donde debe cumplirse que $2^n > \beta > 2^{n-1}$.

$$2^n \cdot Y = 2 \cdot \sum_{i=1}^{2^n-\beta} b_i + \sum_{j=2^n-\beta+1}^{\beta} b_j - \sum_{k=0}^{n-1} (2^k \cdot aux_k) \tag{4.6}$$

4.4.3 PUERTA NOT

Una puerta NOT con entrada b y salida Y puede modelarse con una ecuación lineal como la que se muestra en (4.7).

$$Y = 1 - b \tag{4.7}$$

Puede comprobarse que si b fuera 0, resultaría que para que se cumpliera la ecuación (4.7) tendría que ser Y=1. Asimismo, si b fuese 1, Y tendría que ser 0 para que se cumpliese (4.7).

4.4.4 PUERTA NOR

Para una puerta NOR de 2 entradas, con entradas (b_1, b_2) y salida (Y). Teniendo en cuenta que será equivalente a una AND con todas sus entradas invertidas, puede comprobarse que, según su tabla de verdad, lo siguiente es siempre verdadero: $2 \cdot Y + aux = 2 - b_1 - b_2$, donde aux es una variable auxiliar de tipo binario. Independientemente del valor que tome aux, la salida Y tomará el valor 1 solamente cuando ninguna de las entradas es 1 para que la ecuación lineal se cumpla. En resumen, la restricción lineal para la puerta NOR de 2 entradas es la que se muestra en (4.8).

$$2 \cdot Y = 2 - b_1 - b_2 - aux \tag{4.8}$$

NOR2	$2 \cdot Y + aux = 2 - b_1 - b_2$					
b_1	**b_2**	**Y**	aux	$2 \cdot Y + aux$	$2 - b_1 - b_2$	Válido
0	0	0	0	0	2	✗
0	0	0	1	1	2	✗
0	0	1	0	2	2	✓
0	0	1	1	3	2	✗
0	1	0	0	0	1	✗
0	1	0	1	1	1	✓
0	1	1	0	2	1	✗
0	1	1	1	3	1	✗
1	0	0	0	0	1	✗
1	0	0	1	1	1	✓
1	0	1	0	2	1	✗
1	0	1	1	3	1	✗
1	1	0	0	0	0	✓
1	1	0	1	1	0	✗
1	1	1	0	2	0	✗
1	1	1	1	3	0	✗

Tabla 8: Comprobación del modelado MILP de la puerta NOR de dos entradas

Para comprobar el modelado (4.8) de una puerta NOR de dos entradas se realiza la Tabla 8, en la que se exploran todas las combinaciones posibles de las variables binarias Y, aux, b_1, y b_2. En dicha tabla puede comprobarse que solo hay

una solución en que se cumple la ecuación lineal (4.8) para cada uno de los cuatro casos de la tabla de verdad de una puerta NOR de dos entradas (Tabla 9).

NOR2	$2\cdot Y + aux = 2 - b_1 - b_2$					
b_1	**b_2**	**Y**	aux	$2\cdot Y + aux$	$2 - b_1 - b_2$	Válido
0	0	1	0	2	2	✓
0	1	0	1	1	1	✓
1	0	0	1	1	1	✓
1	1	0	0	0	0	✓

Tabla 9: Tabla de verdad de la puerta NOR de dos entradas

Esto puede extenderse a una puerta NOR con un mayor número de entradas. De modo que si el número de entradas es la potencia de dos 2^n, se cumpliría la ecuación lineal (4.9), donde Y sería la salida y b_i la i-ésima entrada de la puerta NOR.

$$2^n \cdot Y = 2^n - \sum_{i=1}^{2^n} b_i - \sum_{k=0}^{n-1} (2^k \cdot aux_k) \tag{4.9}$$

Si el número de entradas fuera menor que 2^n, pero mayor que 2^{n-1}, entonces se duplicaría el peso de tantas entradas como diferencia haya entre 2^n y el número de entradas que tenga la puerta, quedando una puerta NOR de β entradas modelada con la ecuación lineal (4.10), donde debe cumplirse que $2^n > \beta > 2^{n-1}$.

$$2^n \cdot Y = 2^n - 2 \cdot \sum_{i=1}^{2^n-\beta} b_i - \sum_{j=2^n-\beta+1}^{\beta} b_j - \sum_{k=0}^{n-1} (2^k \cdot aux_k) \tag{4.10}$$

4.4.5 PUERTA NAND

Considerando una puerta NAND de 2 entradas, con entradas (b_1, b_2) y salida (Y). Teniendo en cuenta que será equivalente a una OR con todas sus entradas invertidas, puede comprobarse que, según su tabla de verdad, lo siguiente es siempre verdadero: $2\cdot Y\text{-}aux=2\text{-}b_1\text{-}b_2$, donde aux es una variable auxiliar de tipo

binario. Independientemente del valor que tome aux, la salida Y debe tomar el valor 0 solamente cuando todas las entradas son 1 para que la ecuación lineal se cumpla. En resumen, la restricción lineal para la puerta NAND de 2 entradas es la que se muestra en (4.11).

$$2 \cdot Y = aux + 2 - b_1 - b_2 \tag{4.11}$$

Para comprobar el modelado de una puerta NAND de dos entradas con la ecuación lineal (4.11) se realiza la Tabla 10, en la que se exploran todas las combinaciones posibles de las variables binarias Y, aux, b_1, y b_2. En dicha tabla se puede comprobar que solo hay una solución en que se cumple la ecuación lineal (4.11) para cada uno de los cuatro casos de la tabla de verdad de una puerta NAND de dos entradas (Tabla 11).

NAND2	$2 \cdot Y - aux = 2 - b_1 - b_2$					
b_1	**b_2**	**Y**	aux	$2 \cdot Y - aux$	$2 - b_1 - b_2$	Válido
0	0	0	0	0	2	✗
0	0	0	1	-1	2	✗
0	0	1	0	2	2	✓
0	0	1	1	1	2	✗
0	1	0	0	0	1	✗
0	1	0	1	-1	1	✗
0	1	1	0	2	1	✗
0	1	1	1	1	1	✓
1	0	0	0	0	1	✗
1	0	0	1	-1	1	✗
1	0	1	0	2	1	✗
1	0	1	1	1	1	✓
1	1	0	0	0	0	✓
1	1	0	1	-1	0	✗
1	1	1	0	2	0	✗
1	1	1	1	1	0	✗

Tabla 10: Comprobación del modelado MILP de la puerta NAND de dos entradas

NAND2	2·Y - aux = 2 - b_1 - b_2					
b_1	**b_2**	**Y**	aux	2·Y - aux	2 - b_1 - b_2	Válido
0	0	1	0	2	2	✓
0	1	1	1	1	1	✓
1	0	1	1	1	1	✓
1	1	0	0	0	0	✓

Tabla 11: Tabla de verdad de la puerta NAND de dos entradas

Esto puede extenderse a una puerta NAND con un mayor número de entradas. De modo que si el número de entradas es la potencia de dos 2^n, se cumpliría la ecuación lineal (4.12), donde Y sería la salida y b_i la i-ésima entrada de la puerta NAND.

$$2^n \cdot Y = \sum_{k=0}^{n-1} (2^k \cdot aux_k) + 2^n - \sum_{i=1}^{2^n} b_i \qquad (4.12)$$

Si el número de entradas fuera menor que 2^n, pero mayor que 2^{n-1}, entonces se duplicaría el peso de tantas entradas como diferencia haya entre 2^n y el número de entradas que tenga la puerta, quedando una puerta NAND de β entradas modelada con la ecuación lineal (4.13), donde debe cumplirse que $2^n > \beta > 2^{n-1}$.

$$2^n \cdot Y = \sum_{k=0}^{n-1} (2^k \cdot aux_k) + 2^n - 2 \cdot \sum_{i=1}^{2^n-\beta} b_i - \sum_{j=2^n-\beta+1}^{\beta} b_j \qquad (4.13)$$

4.5 CONVERSIÓN DEL PROBLEMA SAT DE CNF A MILP

Una manera de traducir la lógica booleana de CNF a MILP consiste en mapear primero la parte booleana en una red NOT-OR-AND y, a continuación, derivar restricciones lineales para cada puerta lógica. De este modo se modela mediante MILP el OR lógico y el AND lógico, ya que el formato CNF consiste en el AND lógico de una o más cláusulas, que a su vez consisten en el OR lógico de uno o más literales, siendo cada literal una variable booleana o su negación.

Por tanto, se utilizará esa parte del modelado de la lógica booleana utilizando las restricciones de MILP.

4.5.1 CLÁUSULAS CNF

En el problema SAT expresado en formato CNF habrá infinidad de cláusulas que serán modeladas como puertas OR con ecuaciones MILP.

Como se ha visto previamente, una puerta OR con un número arbitrario de entradas β, tal que $2^n > \beta > 2^{n-1}$, puede modelarse utilizando las restricciones de MILP, con la ecuación lineal mostrada en (4.14). Donde Y es la salida y b_i la i-ésima entrada de la puerta OR.

$$2^n \cdot Y = \sum_{k=0}^{n-1} (2^k \cdot aux_k) + 2 \cdot \sum_{i=1}^{2^n-\beta} b_i + \sum_{j=2^n-\beta+1}^{\beta} b_j \qquad (4.14)$$

Si alguna entrada estuviera invertida se sustituiría en la ecuación lineal de la puerta OR con el modelado MILP de la puerta NOT (4.15).

$$\overline{b_i} = 1 - b_i \qquad (4.15)$$

4.5.2 AND LÓGICO

En el problema SAT expresado en formato CNF deben cumplirse todas las cláusulas, es decir, las salidas de todas las OR deben ser 1. Esto equivale a hacer el AND lógico de todas las cláusulas que han sido previamente modeladas como puertas OR con ecuaciones MILP. Este AND lógico también será modelado como una puerta AND con ecuaciones MILP. Esta única puerta AND es probable que tenga un gran número de entradas, que serán todas las cláusulas, aunque se sabe que ninguna de estas entradas estará invertida.

Como ya se ha visto, una puerta AND con un número arbitrario de entradas β, tal que $2^n > \beta > 2^{n-1}$, puede modelarse utilizando las restricciones de MILP,

quedando de la forma mostrada en la ecuación (4.16), donde Y es la salida y b_i la i-ésima entrada de la puerta AND.

$$2^n \cdot Y = 2 \cdot \sum_{i=1}^{2^n-\beta} b_i + \sum_{j=2^n-\beta+1}^{\beta} b_j - \sum_{k=0}^{n-1} (2^k \cdot aux_k) \qquad (4.16)$$

Pero dado que el problema SAT expresado en formato CNF requiere que todas las cláusulas se cumplan y por tanto todas las puertas OR modeladas con ecuaciones MILP deban tener salida igual a 1, esta puerta AND puede modelarse en ecuaciones MILP de una forma más sencilla, puesto que el problema es satisfacible solamente cuando todas las entradas de esta puerta AND final están a 1. Con ello este AND final puede modelarse con la ecuación MILP (4.17), donde ψ es la cantidad de cláusulas que tiene el problema e Y_i la i-ésima entrada de la puerta AND.

$$\sum_{i=1}^{\psi} Y_i = \psi \qquad (4.17)$$

4.5.3 CODIFICADOR DE CNF A MILP

Para este PFC se ha desarrollado un programa en C, cuyo código se encuentra en el fichero `cnf2lp.c`, para realizar la conversión del problema SAT en formato CNF a un problema de programación lineal entera mixta con restricciones MILP.

Este programa recibe como parámetros el nombre del fichero de entrada en formato CNF y el nombre del fichero de salida, que será el que se creará en MILP.

El algoritmo de este programa, que se explica a continuación, puede verse en Figura 25.

```
INICIO [algoritmo cnf2lp]
        ABRIR ficheroCNF
        CREAR ficheroLP
        LEER ficheroCNF
        ENTERO datos[] = ficheroCNF
        CERRAR ficheroCNF
        ESCRIBIR ficheroLP = "max obj: y1"
        ESCRIBIR ficheroLP = "subject to"
        ENTERO lin_LP = 0
        ENTERO x = 0
        ENTERO vector[]
        ENTERO n = 0
        ENTERO ent = 0
        DESDE i=0 HASTA FIN datos[] HACER
                MIENTRAS datos[i]<>0 HACER
                        vector[n]=datos[i]
                        SI datos[i] > ent ENTONCES
                                ent = datos[i]
                        FINSI
                        INCREMENTAR n
                        INCREMENTAR i
                REPETIR
                INCREMENTAR lin_LP
                ESCRIBIR ficheroLP ="c(lin_LP): "
                x=genera_OR(ficheroLP, vector[], n, lin_LP, x)
        SIGUIENTE
        ESCRIBIR ficheroLP = "c(lin_LP+1): y1"
        DESDE i=2 HASTA lin_LP HACER
                ESCRIBIR ficheroLP = " + y(i)"
        SIGUIENTE
        ESCRIBIR ficheroLP = " = (lin_LP)"
        ESCRIBIR ficheroLP = "bin"
        DESDE i=1 HASTA lin_LP HACER
                ESCRIBIR ficheroLP = "y(i) "
        SIGUIENTE
        DESDE i=0 HASTA x HACER
                ESCRIBIR ficheroLP = "aux(i) "
        SIGUIENTE
        DESDE i=1 HASTA ent HACER
                ESCRIBIR ficheroLP = "b(i) "
        SIGUIENTE
        ESCRIBIR ficheroLP = "end"
        CERRAR ficheroLP
FIN
```

Figura 25: Algoritmo cnf2lp

Primero, se abre el fichero CNF en modo lectura y se crea el fichero MILP. Seguidamente se leerá línea a línea el CNF y se irá interpretando de forma que se cargará en una lista de enteros, llamada *datos*, los números que contienen la información de los literales, con su signo incluido si lo tuvieran, así como el `0` que implica el final de cada cláusula.

A continuación se cerrará el fichero CNF y se comienza a escribir en el fichero MILP. Al principio se escribe el objetivo `"max obj: y1"`, que es trivial, dado que el objetivo real es resolver todo el circuito, razón por la que se ha puesto como objetivo maximizar `y1` que es la salida de la primera puerta OR que modela la primera cláusula de la codificación CNF. Esta salida `y1` tendrá que ser `1` al igual que todas las salidas de todas las cláusulas para que el problema sea SAT.

Lo siguiente que se escribe en el fichero es `"subject to"`, para proseguir con las restricciones, habrá una restricción por cada cláusula que hubiera en el CNF.

Puesto que cada cláusula acaba con un `0`, que está cargado en la lista datos, se pasarán los números cargados en *datos* a una lista de enteros llamada *vector* mientras no encuentre el `0`, que indica el final de la cláusula.

Cuando *vector* tiene cargados los literales de una cláusula, se pasa a la función *genera_OR*, que se encarga de generar el modelo con restricciones MILP para la cláusula cargada en *vector*. Esta cláusula consiste en el OR lógico de los literales invertidos o no, esto se identifica por el signo que figura en cada elemento de la lista *vector*.

En el siguiente paso se genera la restricción para el AND lógico de las salidas de todas las cláusulas, haciendo que la suma de todas estas salidas sea igual a la cantidad de cláusulas.

Finalmente, se declaran como *binary* todas las variables usadas, que son las salidas de las cláusulas OR (`y1, y2,…`), las variables (`b1, b2,…`) y las variables auxiliares necesarias para implementar el OR lógico de cada cláusula (`aux0, aux1,…`).

4.5.4 FUNCIÓN PARA GENERAR OR

La función *genera_OR*, cuyo algoritmo se muestra en la Figura 26, es utilizada en el programa `cnf2lp` para generar el modelo con restricciones MILP para cada cláusula del fichero CNF. Estas cláusulas consisten en el OR lógico de los literales invertidos o no.

```
ENTERO FUNCION genera_OR (ficheroLP, ENTERO vector[], ENTERO tam, ENTERO
lin_LP, ENTERO aux_ind)
INICIO
        ENTERO i
        ENTERO val = 0
        ENTERO max_tam = 0
        SI tam > 1 ENTONCES
                max_tam = 2^┌SQRT(tam)┐
                ESCRIBIR ficheroLP = "(max_tam) Y(lin_LP) "
                i = tam_max/2
                MIENTRAS i > 1 HACER
                        ESCRIBIR ficheroLP = "- (i) aux(aux_ind)"
                        i = i/2
                        INCREMENTAR aux_ind
                REPETIR
                ESCRIBIR ficheroLP = "- aux(aux_ind) "
                INCREMENTAR aux_ind
                DESDE i = 0 HASTA tam_max-tam HACER
                        SI vector[i] < 0 ENTONCES
                                ESCRIBIR ficheroLP = "+ 2 b(-vector[i])"
                                INCREMENTAR val
                                INCREMENTAR val
                        SINO
                                ESCRIBIR ficheroLP = "- 2 b(vector[i])"
                        FINSI
                SIGUIENTE
                ESCRIBIR ficheroLP = " = (val)"
        SINO
                SI vector[0] < 0
                        ESCRIBIR ficheroLP = "b(-vector[0]) = 0"
                SINO
                        ESCRIBIR ficheroLP = "b(vector[0]) = 1"
                FINSI
        FINSI
        DEVOLVER aux_ind
FIN FUNCION
```

Figura 26: Función `genera_OR`

4.5.5 FIJAR ENTRADAS Y/O SALIDAS EN CNF

Se trata de un programa en C, cuyo código se encuentra en el fichero `Fix_IO_CNF.c`, que se ha desarrollado en este PFC para que permita fijar algunas o todas las entradas y salidas en los circuitos en formato CNF.

```
INICIO [algoritmo Fix_IO_CNF]
        FICHERO ficheroCNFfix;  ENTERO iter = 0, i = 0, ent = 0, sal = 0, vector[]
        DESDE iter = 1 HASTA veces HACER
                CREAR ficheroCNFfix[iter]
                ABRIR ficheroCNForig
                COPIAR ficheroCNFfix[iter] = ficheroCNForig
                LEER ficheroCNForig
                MIENTRAS ficheroCNForig <> "entradas" HACER
                        LEER ficheroCNForig
                REPETIR
                MIENTRAS ficheroCNForig <> "salidas" HACER
                        ENTERO datosentradas[] = ficheroCNForig
                        INCREMENTAR entradas
                REPETIR
                MIENTRAS NO FIN ficheroCNForig HACER
                        ENTERO datossalidas[] = ficheroCNForig
                        INCREMENTAR salidas
                REPETIR
                CERRAR ficheroCNForig
                ESCRIBIR ficheroCNFfix[iter] = "Porcentaje de entradas fijadas: (Pentradas)"
                ESCRIBIR ficheroCNFfix[iter] = "Porcentaje de salidas fijadas: (Psalidas)"
                DESDE i = 1 HASTA ent*Pentradas/100 HACER
                        vector[i] = datosentradas[RANDOM(1..ent)]
                SIGUIENTE
                DESDE i = ent*Pentradas/100 HASTA ent*Pentradas/100+sal*Psalidas/100 HACER
                        vector[i] = datossalidas[RANDOM(1..sal)]
                SIGUIENTE
                DESDE i = 1 HASTA ent*Pentradas/100+sal*Psalidas/100 HACER
                        SI RANDOM(0,1) < 1
                                ESCRIBIR ficheroCNFfix[iter] = "-(vector[i]) 0"
                        SINO
                                ESCRIBIR ficheroCNFfix[iter] = "(vector[i]) 0"
                        FINSI
                SIGUIENTE
                CERRAR ficheroCNForig
                LLAMAR (cnf2lp)
        SIGUIENTE
FIN
```

Figura 27: Algoritmo `Fix_IO_CNF`

El algoritmo de este programa, que a continuación se explica, puede verse en la Figura 27.

Recibe como parámetros de entrada el porcentaje de las entradas y salidas que se desea que sean fijadas, así como el fichero CNF que se desea procesar y la cantidad de veces que se quiere que se realice una asignación aleatoria. Es importante destacar que la asignación aleatoria deberá ser diferente cada vez, para así proporcionar mayor cobertura.

Finalmente, este programa hace una llamada al programa `cnf2lp`, explicado anteriormente, de forma que se obtendrán como resultado la cantidad solicitada de ficheros en formato CNF con entradas y/o salidas diferentes fijadas, así como sus correspondientes ficheros traducidos al formato MILP con las mismas entradas y/o salidas fijadas.

4.6 CONCLUSIONES

En este capítulo se ha presentado el método de generación de vectores para verificación de circuitos booleanos mediante MILP desarrollado en este proyecto.

Se describe el entorno de verificación, así como los formatos de entrada para los circuitos.

También se expone el modelado de lógica booleana diseñado en este proyecto. Se han conseguido modelos MILP de un tamaño muy reducido, esto es de la máxima importancia, puesto que influirá directamente en el tamaño de la codificación resultante y en el tiempo que empleará el solucionador en encontrar la respuesta adecuada para verificar el circuito.

Se explica cómo se realiza la transformación del circuito, primeramente a un modelado como problema de satisfacibilidad en formato CNF, para posteriormente ser convertido a un problema de programación lineal en formato MILP.

Finalmente, se describen los programas que se han hecho para permitir estas transformaciones y la opción de fijar entradas y/o salidas en los circuitos en formato CNF.

CAPÍTULO 5
RESULTADOS EXPERIMENTALES

5.1 INTRODUCCIÓN

En el presente capítulo se muestran los resultados experimentales obtenidos en este PFC. Primero se describen las condiciones bajo las que se realizaron las pruebas y a continuación se procede a comparar las características de los distintos experimentos.

En concreto, se han realizado pruebas para validar la eficiencia del método desarrollado, comparando el comportamiento para distintos circuitos de diversa condición y complejidad. Las herramientas desarrolladas en este PFC se han puesto a prueba con conjuntos de entrenamiento escogidos por número de puertas y con diferentes porcentajes de entradas y/o salidas fijadas.

Finalmente, se presenta un estudio comparativo de los resultados obtenidos para diversos circuitos y un resumen de los resultados alcanzados.

5.2 DESCRIPCIÓN DE LOS EXPERIMENTOS

La experimentación se ha hecho con circuitos escogidos del banco de circuitos de referencia relacionados en el Anexo II. Se han escogido por ser de diferente naturaleza y complejidad para observar la eficiencia del método desarrollado enfrentado a variados niveles de dificultad.

Primeramente se hace la transformación de todos los circuitos del banco de pruebas a formato CNF con el script `blif2cnf.pl` (Figura 29). El resultado es el banco de pruebas en formato CNF que también se relaciona en el Anexo II.

Por ejemplo uno de los circuitos seleccionados de alta complejidad es el `C6288` (Figuras 28 y 30). Se trata de un Multiplicador de `16` bits con `32` entradas, `32` salidas y `2406` puertas lógicas.

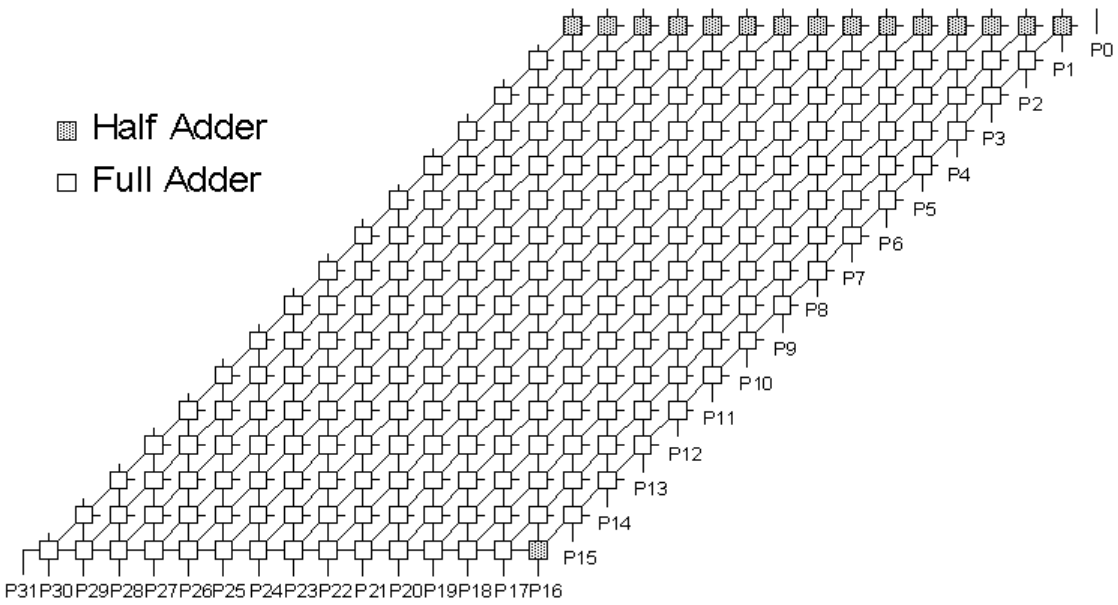


Figura 28: Multiplicador `16x16 ISCAS-85 C6288`

Figura 29: Codificación de `C6288.blif` a `C6288.cnf`

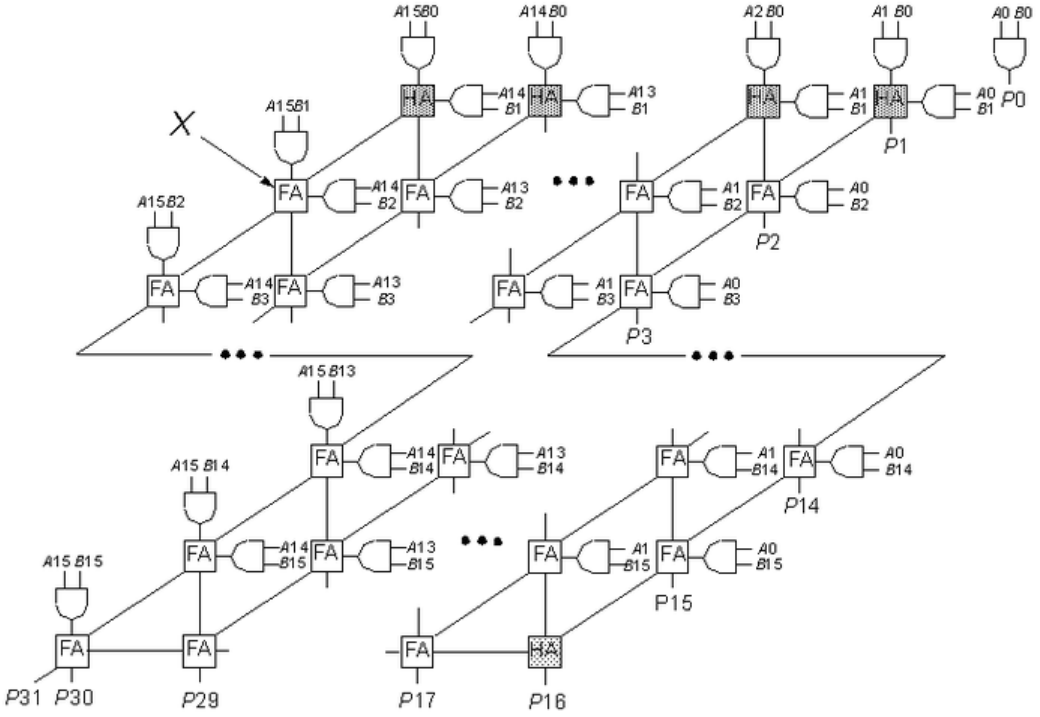


Figura 30: Representación alternativa del Multiplicador 16x16 ISCAS-85 C6288

Siguiendo con este ejemplo, el circuito es codificado a formato CNF, obteniéndose una descripción del circuito como un problema SAT. Esta codificación contará con 12048 cláusulas y 4864 variables (Figura 31).

```
c CNF file generated from BLIF2CNF script.
c Original BLIF file was C6288.blif
p cnf 4864 12048
1 -4 0
2 -4 0
4 -1 -2  0
-4 3 0
-3 4  0
1 -7 0
5 -7 0
7 -1 -5  0
.
.
.
-4861 -4864 0
4864 4859 4861  0
-4864 4863 0
-4863 4864  0
```

Figura 31: Codificación del Multiplicador de 16 bits C6288.cnf

El fichero `C6288.cnf` es transformado en un problema de programación lineal en formato MILP con el programa `cnf2lp.c` (Figura 32). El resultado es el fichero `C6288.lp` que tendrá tantas ecuaciones de restricciones como cláusulas tenía el archivo `C6288.cnf`, más una para la puerta AND final.

Figura 32: Codificación de `C6288.cnf` a `C6288.lp`

El circuito definido como un problema de programación lineal es resuelto con CPLEX (Figura 33) y se mide el tiempo que tarda en obtenerse una solución.

Figura 33: Resolución de `C6288.lp` con CPLEX

Por otra parte, en el fichero `C6288.cnf` se fijan diferentes porcentajes de entradas y salidas, con el programa `Fix_IO_CNF.c` (Figura 34). El resultado son los ficheros `C6288_PI_PO_ixxx.cnf` y `C6288_PI_PO_ixxx.lp`, donde `PI` es el porcentaje de entradas que se desea fijar, `PO` es el porcentaje de salidas a fijar e `ixxx` indica de qué iteración se trata, dado que se pueden pedir varias y todas deberán ser diferentes. El número de cláusulas y restricciones aumentará en la cantidad de entradas y salidas fijadas.

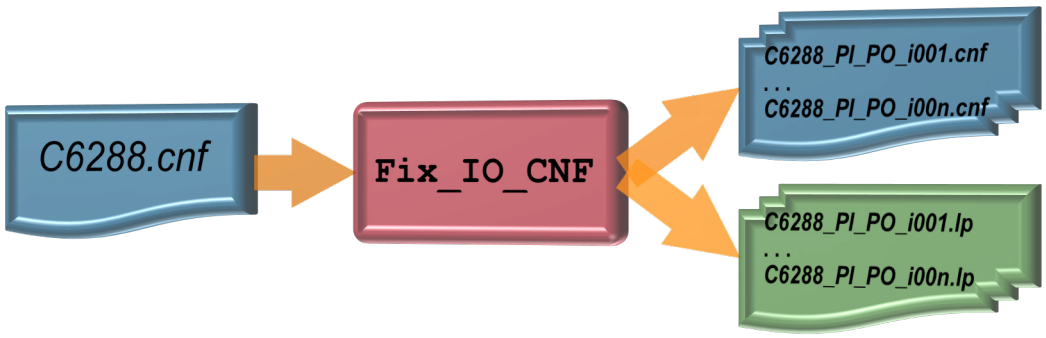


Figura 34: Fijar entradas y/o salidas en el circuito `C6288`

Los circuitos, con las entradas y/o salidas fijadas a unos y ceros, definidos en formato de programación lineal son resueltos con CPLEX y se mide el tiempo que tarda en obtenerse una solución en cada caso.

5.3 CIRCUITOS EMPLEADOS

Se han hecho las pruebas con un conjunto bien establecido de circuitos de referencia (benchmarks) [7]. El banco de pruebas empleado, proporciona un conjunto de circuitos de referencia común para la comunidad científica con el objeto de probar y comparar diferentes aplicaciones y problemas en igualdad de condiciones. Normalmente, estos circuitos del banco de pruebas, se encuentran descritos mediante un modelo estructural. El circuito de referencia se expresa en algún lenguaje que especifica las líneas de entradas y salidas, así como las señales del sistema y sus componentes. De la serie de referencia ISCAS'85, el conjunto de circuitos más conocidos y más empleados en esta área, comprende un total de `11` circuitos combinacionales (`C1355, C17, C1908, C2670, C3540, C432, C499, C5315, C6288, C7552` y `C880`). En este banco de pruebas, el nombre de los circuitos comienza por la letra C a la que acompaña un número, dicho número se corresponde con el número de líneas de señal (nets) del circuito. El más simple de ellos es `C17`, que ya ha sido descrito en el capítulo anterior como ejemplo, con su correspondiente diagrama esquemático.

5.4 RESULTADOS

Con el objeto de comprobar la versatilidad de la metodología propuesta y desarrollada en este PFC se procede a ejecutar un conjunto de pruebas sobre los circuitos ISCAS'85 descritos en el apartado anterior. Las pruebas consisten en hacer para cada circuito todo el procedimiento explicado en el apartado 5.2 para las siguientes condiciones:

- **Sin fijar ninguna entrada ni salida.** Esta prueba evalúa la complejidad de la formulación propuesta en función de la propia complejidad circuital.

- **Con porcentajes de salidas fijadas**: `10%, 30%, 50%, 70%, 90%` y `100%`. Estas pruebas permiten evaluar la versatilidad de la metodología propuesta para encontrar vectores de entrada a los circuitos combinacionales cuando se requiere una determinada respuesta en sus salidas.

- **Con porcentajes de entradas fijadas:** `10%, 30%, 50%, 70%, 90%` y `100%`. Estas pruebas evalúan la capacidad de la metodología propuesta para encontrar el vector de entrada y su respuesta ante determinados estímulos.

A continuación se presentan los resultados en forma de tablas, para posteriormente comentar los resultados más destacados de las mismas.

Nombre	Funcionalidad	Entradas	Salidas	Puertas	Entradas/Salidas sin Fijar	
					Restricciones	Tiempo (s)
C1355	Error Correcting	41	32	546	2703	0,073
C17	Logic	5	2	6	31	0,005
C1908	Error Correcting	33	25	880	4139	0,107
C2670	ALU and Control	233	140	1193	5656	1,431
C3540	ALU and Control	50	22	1669	7947	0,223
C432	Priority Decoder	36	7	160	889	0,030
C499	Error Correcting	41	32	202	1431	0,044
C5315	ALU and Selector	178	123	2307	11308	0,340
C6288	16-bit Multiplier	32	32	2406	12049	0,295
C7552	ALU and Control	207	108	3512	16681	33,12
C880	ALU and Control	60	26	383	1879	0,049

Tabla 12: Resolución de circuitos sin fijar entradas ni salidas

Esta primera tabla 12 presenta los resultados de aplicar la metodología propuesta a los circuitos más conocidos del banco de prueba ISCAS'85. Es resaltable que la metodología propuesta basada en Programación Lineal ha sido capaz de abordar cada uno de los circuitos sin problemas. Muestra de ello es que el circuito

más complejo, identificado como C7552 lo ha resuelto en 33,12 segundos. Este circuito se describe con la metodología desarrollada en 16681 ecuaciones lineales, posee 207 entradas y 108 salidas para un total de 3512 puertas.

Este circuito complejo contrasta con otros como puede ser el C6288 o el C5315 en los que, teniendo un número inferior de puertas (2406 y 2307 respectivamente), los tiempos de ejecución para estos últimos rondan los pocos cientos de milisegundos (295 y 340 respectivamente). Ello es debido única y exclusivamente a la complejidad circuital y no tan directamente al número de puertas de dichos circuitos.

Todas las pruebas realizadas en esta propuesta han sido resueltas en menos de 1,5 segundos para circuitos de hasta 2500 puertas.

Nombre	Funcionalidad	Entradas	Salidas	Puertas	Fijadas 10% de Entradas	
					Restricciones	Tiempo (s)
C1355	Error Correcting	41	32	546	2707	0,076
C17	Logic	5	2	6	32	0.004
C1908	Error Correcting	33	25	880	4142	0,111
C2670	ALU and Control	233	140	1193	5679	0,158
C3540	ALU and Control	50	22	1669	7952	0,246
C432	Priority Decoder	36	7	160	893	0,041
C499	Error Correcting	41	32	202	1435	0,041
C5315	ALU and Selector	178	123	2307	11326	1,496
C6288	16-bit Multiplier	32	32	2406	12052	0,389
C7552	ALU and Control	207	108	3512	16702	15,301
C880	ALU and Control	60	26	383	1885	0,047

Tabla 13: Resolución de circuitos con 10% de entradas fijadas

Nombre	Funcionalidad	Entradas	Salidas	Puertas	Fijadas 30% de Entradas	
					Restricciones	Tiempo (s)
C1355	Error Correcting	41	32	546	2715	0,074
C17	Logic	5	2	6	33	0,004
C1908	Error Correcting	33	25	880	4149	0,106
C2670	ALU and Control	233	140	1193	5726	0,149
C3540	ALU and Control	50	22	1669	7962	0,197
C432	Priority Decoder	36	7	160	900	0,033
C499	Error Correcting	41	32	202	1443	0,040
C5315	ALU and Selector	178	123	2307	11361	0,352
C6288	16-bit Multiplier	32	32	2406	12059	0,442
C7552	ALU and Control	207	108	3512	16743	0,577
C880	ALU and Control	60	26	383	1897	0,043

Tabla 14: Resolución de circuitos con 30% de entradas fijadas

Nombre	Funcionalidad	Entradas	Salidas	Puertas	Fijadas 50% de Entradas	
					Restricciones	Tiempo (s)
C1355	Error Correcting	41	32	546	2724	0,068
C17	Logic	5	2	6	34	0,003
C1908	Error Correcting	33	25	880	4156	0,091
C2670	ALU and Control	233	140	1193	5773	0,126
C3540	ALU and Control	50	22	1669	7972	0,179
C432	Priority Decoder	36	7	160	907	0,019
C499	Error Correcting	41	32	202	1452	0,072
C5315	ALU and Selector	178	123	2307	11397	0,303
C6288	16-bit Multiplier	32	32	2406	12065	0,506
C7552	ALU and Control	207	108	3512	16785	0,609
C880	ALU and Control	60	26	383	1909	0,036

Tabla 15: Resolución de circuitos con 50% de entradas fijadas

Nombre	Funcionalidad	Entradas	Salidas	Puertas	Fijadas 70% de Entradas	
					Restricciones	Tiempo (s)
C1355	Error Correcting	41	32	546	2732	0,061
C17	Logic	5	2	6	35	0,003
C1908	Error Correcting	33	25	880	4162	0,084
C2670	ALU and Control	233	140	1193	5819	0,107
C3540	ALU and Control	50	22	1669	7982	0,183
C432	Priority Decoder	36	7	160	914	0,016
C499	Error Correcting	41	32	202	1460	0,036
C5315	ALU and Selector	178	123	2307	11433	0,264
C6288	16-bit Multiplier	32	32	2406	12071	0,474
C7552	ALU and Control	207	108	3512	16826	0,393
C880	ALU and Control	60	26	383	1921	0,032

Tabla 16: Resolución de circuitos con 70% de entradas fijadas

Nombre	Funcionalidad	Entradas	Salidas	Puertas	Fijadas 90% de Entradas	
					Restricciones	Tiempo (s)
C1355	Error Correcting	41	32	546	2740	0,049
C17	Logic	5	2	6	36	0,002
C1908	Error Correcting	33	25	880	4169	0,063
C2670	ALU and Control	233	140	1193	5866	0,083
C3540	ALU and Control	50	22	1669	7992	0,118
C432	Priority Decoder	36	7	160	921	0,019
C499	Error Correcting	41	32	202	1468	0,028
C5315	ALU and Selector	178	123	2307	11468	0,202
C6288	16-bit Multiplier	32	32	2406	12078	0,505
C7552	ALU and Control	207	108	3512	16867	0,279
C880	ALU and Control	60	26	383	1933	0,026

Tabla 17: Resolución de circuitos con 90% de entradas fijadas

Nombre	Funcionalidad	Entradas	Salidas	Puertas	Fijadas 100% de Entradas	
					Restricciones	Tiempo (s)
C1355	Error Correcting	41	32	546	2744	0,027
C17	Logic	5	2	6	36	0,002
C1908	Error Correcting	33	25	880	4172	0,043
C2670	ALU and Control	233	140	1193	5889	0,056
C3540	ALU and Control	50	22	1669	7997	0,080
C432	Priority Decoder	36	7	160	925	0,014
C499	Error Correcting	41	32	202	1472	0,015
C5315	ALU and Selector	178	123	2307	11486	0,116
C6288	16-bit Multiplier	32	32	2406	12081	0,126
C7552	ALU and Control	207	108	3512	16888	0,187
C880	ALU and Control	60	26	383	1939	0,019

Tabla 18: Resolución de circuitos con 100% de entradas fijadas

El conjunto de pruebas basado en fijar las entradas se presenta en las Tablas 13 y sucesivas hasta la Tabla 18. Se puede apreciar que a medida que crece el porcentaje de entradas fijadas, la aplicación de la metodología propuesta resuelve con mayor velocidad. Si tenemos en cuenta que la descripción circuital se traduce a ecuaciones lineales, la fijación de entradas equivale en ese sistema de ecuaciones lineales a fijar como constante variables del problema. Por consiguiente, se reduce el número de variables y por tanto se reduce también en igual medida el conjunto de ecuaciones a resolver.

Véase como, por ejemplo, el circuito `C7552` pasa a resolverse en aproximadamente `15` segundos cuando se le fija tan solo el `10%` de sus entradas (véase Tabla 13 para más detalle). En el resto de casos los tiempos no superan `1.5` segundos.

En las siguientes tablas se muestran los resultados obtenidos fijando diferentes porcentajes de las salidas de los circuitos.

Nombre	Funcionalidad	Entradas	Salidas	Puertas	Fijadas 10% de Salidas	
					Restricciones	Tiempo (s)
C1355	Error Correcting	41	32	546	2706	0,243
C17	Logic	5	2	6	32	0,004
C1908	Error Correcting	33	25	880	4142	0,144
C2670	ALU and Control	233	140	1193	5670	1,464
C3540	ALU and Control	50	22	1669	7949	4,653
C432	Priority Decoder	36	7	160	890	0,029
C499	Error Correcting	41	32	202	1434	0,065
C5315	ALU and Selector	178	123	2307	11320	0,262
C6288	16-bit Multiplier	32	32	2406	12052	20,777
C7552	ALU and Control	207	108	3512	16692	37,568
C880	ALU and Control	60	26	383	1882	0,121

Tabla 19: Resolución de circuitos con 10% de salidas fijadas

Nombre	Funcionalidad	Entradas	Salidas	Puertas	Fijadas 30% de Salidas	
					Restricciones	Tiempo (s)
C1355	Error Correcting	41	32	546	2713	0,083
C17	Logic	5	2	6	32	0,004
C1908	Error Correcting	33	25	880	4147	0,967
C2670	ALU and Control	233	140	1193	5698	0,034
C3540	ALU and Control	50	22	1669	7954	2,927
C432	Priority Decoder	36	7	160	891	0,027
C499	Error Correcting	41	32	202	1441	0,119
C5315	ALU and Selector	178	123	2307	11345	0,069
C6288	16-bit Multiplier	32	32	2406	12059	55,076
C7552	ALU and Control	207	108	3512	16713	8,411
C880	ALU and Control	60	26	383	1887	0,052

Tabla 20: Resolución de circuitos con 30% de salidas fijadas

Nombre	Funcionalidad	Entradas	Salidas	Puertas	Fijadas 50% de Salidas	
					Restricciones	Tiempo (s)
C1355	Error Correcting	41	32	546	2719	0,079
C17	Logic	5	2	6	32	0,004
C1908	Error Correcting	33	25	880	4152	1,134
C2670	ALU and Control	233	140	1193	5726	0,036
C3540	ALU and Control	50	22	1669	7958	6,665
C432	Priority Decoder	36	7	160	893	0,022
C499	Error Correcting	41	32	202	1447	0,144
C5315	ALU and Selector	178	123	2307	11370	0,072
C6288	16-bit Multiplier	32	32	2406	12065	197,753
C7552	ALU and Control	207	108	3512	16735	0,109
C880	ALU and Control	60	26	383	1892	0,033

Tabla 21: Resolución de circuitos con 50% de salidas fijadas

Nombre	Funcionalidad	Entradas	Salidas	Puertas	Fijadas 70% de Salidas	
					Restricciones	Tiempo (s)
C1355	Error Correcting	41	32	546	2725	0,079
C17	Logic	5	2	6	32	0,004
C1908	Error Correcting	33	25	880	4157	1,091
C2670	ALU and Control	233	140	1193	5754	0,034
C3540	ALU and Control	50	22	1669	7962	4,638
C432	Priority Decoder	36	7	160	894	0,024
C499	Error Correcting	41	32	202	1453	0,270
C5315	ALU and Selector	178	123	2307	11394	0,070
C6288	16-bit Multiplier	32	32	2406	12071	>600
C7552	ALU and Control	207	108	3512	16757	0,108
C880	ALU and Control	60	26	383	1897	0,068

Tabla 22: Resolución de circuitos con 70% de salidas fijadas

Nombre	Funcionalidad	Entradas	Salidas	Puertas	Fijadas 90% de Salidas	
					Restricciones	Tiempo (s)
C1355	Error Correcting	41	32	546	2732	0,076
C17	Logic	5	2	6	33	0,004
C1908	Error Correcting	33	25	880	4162	1,013
C2670	ALU and Control	233	140	1193	5782	0,034
C3540	ALU and Control	50	22	1669	7967	2,487
C432	Priority Decoder	36	7	160	895	0,017
C499	Error Correcting	41	32	202	1460	0,046
C5315	ALU and Selector	178	123	2307	11419	0,071
C6288	16-bit Multiplier	32	32	2406	12078	>600
C7552	ALU and Control	207	108	3512	16778	0,103
C880	ALU and Control	60	26	383	1902	0,023

Tabla 23: Resolución de circuitos con 90% de salidas fijadas

Nombre	Funcionalidad	Entradas	Salidas	Puertas	Fijadas 100% de Salidas	
					Restricciones	Tiempo (s)
C1355	Error Correcting	41	32	546	2735	1,728
C17	Logic	5	2	6	33	0,004
C1908	Error Correcting	33	25	880	4164	0,131
C2670	ALU and Control	233	140	1193	5796	0,037
C3540	ALU and Control	50	22	1669	7969	0,051
C432	Priority Decoder	36	7	160	896	0,015
C499	Error Correcting	41	32	202	1463	0,047
C5315	ALU and Selector	178	123	2307	11431	0,070
C6288	16-bit Multiplier	32	32	2406	12081	>600
C7552	ALU and Control	207	108	3512	16789	0,101
C880	ALU and Control	60	26	383	1905	0,041

Tabla 24: Resolución de circuitos con 100% de salidas fijadas

En las Tablas 19 a 24 se presentan los resultados de fijar un porcentaje del conjunto de salidas. Para todas las pruebas se puede determinar que, a medida que se aumenta el conjunto de salidas fijas, el problema gana complejidad y tarda más en resolverse. Hay que tener en cuenta que las salidas a valor fijo se determinan aleatoriamente en las pruebas hasta alcanzar el porcentaje definido. Este procedimiento de inicialización no evita la aparición de patrones de salida inviables, entendiendo como "patrón de salida inviable" aquel conjunto de valores de salida que no cuentan con un patrón de entrada que aplicado al circuito bajo prueba lo obtenga. Con el objeto de evitar un bucle de espera infinito por el efecto indeseado expuesto, se ha procedido a incluir un tiempo límite (timeout) de ejecución de 600 segundos (10 minutos). Alcanzado este tiempo, el sistema de verificación se para y genera un error por tiempo.

En las Tablas 22 a 24 se puede observar la aparición de dicho error por tiempo en los resultados referentes al circuito `C6288`.

5.5 CONCLUSIONES

En este quinto capítulo se presentan el conjunto de pruebas que se han realizado para evaluar la bondad de la metodología de generación de vectores para la verificación de circuitos booleanos mediante MILP.

Las pruebas se han basado en el banco de pruebas estándar ISCAS'85, de donde se han escogido los circuitos combinacionales más empleados en la literatura descrita en el estado del arte en el área de verificación. La complejidad de los circuitos abarca desde unas pocas puertas (`C17`) hasta circuitos de complejidad mediana con miles de ellas (`C7552`, con `3500` puertas aproximadamente).

De las pruebas se desprende que no solo la complejidad del circuito medida en términos de puertas ha de tenerse en cuenta, a la hora de predecir el tiempo requerido para obtener los vectores de verificación, sino que se ha de incluir la complejidad del interconexionado entre dichas puertas. Prueba de ello son los resultados obtenidos con los circuitos `C7552` y `C6288`.

También es reseñable que los tiempos para obtener los vectores de verificación son inversamente proporcionales al porcentaje de entradas que se especifiquen como fijas y que dicha relación es directamente proporcional al conjunto de salidas que sean forzadas. Además, se puede determinar si una especificación de entradas/salidas fijas es viable o no mediante el empleo de un tiempo límite de ejecución.

CAPÍTULO 6
CONCLUSIONES Y LÍNEAS FUTURAS

6.1 INTRODUCCIÓN

Finalmente, en este último capítulo se realiza un análisis de los objetivos alcanzados a lo largo del desarrollo del proyecto y si estos se corresponden con los objetivos fijados al inicio.

Por otro lado, se presenta un pequeño resumen del proyecto y las conclusiones que se han alcanzado tras el desarrollo del mismo y tras la realización de los distintos experimentos con los circuitos de referencia utilizados. A partir de las conclusiones obtenidas se proponen posibles líneas futuras de trabajo para la continuación en la investigación de este tipo de métodos.

6.2 ADECUACIÓN A LOS OBJETIVOS

En este documento de memoria de Proyecto Fin de Carrera se detalla una herramienta/metodología que permite la generación de patrones para la verificación de circuitos Booleanos empleando Programación Lineal Entera Mixta (MILP).

Se ha logrado el objetivo principal de crear un entorno de desarrollo para la generación de patrones de verificación. La herramienta se ha implementado mediante diversos programas desarrollados en C que se ejecutan en la línea de comandos (csh o bash). Se ha probado tanto en entornos Linux (Red Hat Enterprise Linux Server 5.11) así como Unix (Solaris 10).

También se ha alcanzado el objetivo secundario de hacer uso de un formato de entrada estándar, ya que se ha dotado a la herramienta desarrollada en este PFC de una interfaz estándar de especificación de circuitos Booleanos como es el formato BLIF. La elección de dicho formato se basó en la universalidad del mismo y la existencia de conversores tanto libres, basados en políticas GNU, como en comerciales (Cadence [21] y Synopsys [22]).

El objetivo de desarrollar modelos de puertas booleanas para traducir el modelo circuital a un modelo MILP también se ha logrado, puesto que se han desarrollado y probado diversos modelos de puerta que permiten describir de forma óptima el circuito mediante Programación Lineal Entera Mixta.

Por último, la herramienta/metodología propuesta se ha evaluado mediante el uso de un banco de pruebas estándar de referencia en el estado del arte.

6.3 CONCLUSIONES

A continuación se recopila cada una de las conclusiones a las que se ha llegado en este PFC y que ya han sido expuestas en los correspondientes capítulos.

Del capítulo 2, sobre verificación de circuitos, indicar que uno de los principales desafíos en la línea de producción de circuitos integrados es el uso de procedimientos adecuados y prácticos para probar circuitos en diferentes etapas del proceso de fabricación para mejorar el rendimiento y la fiabilidad de los productos

finales. Con los recientes avances en las tecnologías de semiconductores y el aumento continuo de la densidad de circuitos de los chips VLSI, el costo de las pruebas aumenta rápidamente. La prueba de circuitos digitales es el proceso de aplicar algunos vectores de prueba a las entradas del circuito y observar las salidas. El tamaño del conjunto de pruebas afecta el coste total de las pruebas del circuito.

Este análisis se utiliza con el objetivo de generar nuevos vectores de prueba altamente eficaces. La relación entre los fallos focalizados y el total de fallos posibles tiene efecto en la mejora de la cobertura de fallos.

Por último, y a modo de reflexión, mientras la verificación de circuitos se refiere al conjunto de tareas encaminadas a comprobar un diseño desde un punto de vista puramente funcional y el test se centra en chequear defectos físicos, ambos procesos del ciclo de diseño se encuentran íntimamente interrelacionados. Por ejemplo, gran parte de las pruebas a las que se somete el diseño en su verificación son reutilizadas en el test. Esta dualidad de ciertas metodologías/técnicas de verificación/test hace que por ejemplo, la inmensa mayoría de los generadores de patrones de verificación/test sean aplicados en ambos ámbitos.

En particular, los generadores de patrones propuestos en este PFC son aplicables tanto a la verificación como al test de circuitos digitales booleanos.

En el capítulo 3, sobre satisfacibilidad, se concluye que la satisfacibilidad está estrechamente relacionada con los conceptos centrales de la lógica, en los cuales tiene sus orígenes. Es más, la presencia de la satisfacibilidad en la lógica surge para modelar el pensamiento humano y el razonamiento científico a través de su uso en el diseño de computadoras y ahora se utiliza como herramienta de modelado para resolver gran variedad de problemas prácticos. Por todo ello, se ha revelado como una potente herramienta que permite modelar y resolver multitud de problemas, principalmente aquellos que comparten sus raíces en la lógica como es el caso del diseño y prueba de circuitos.

El formato de CNF es el más ampliamente usado para describir los problemas que deberán resolver los SAT-solvers, razón por la que en este trabajo se toma como formato de partida. Una instancia de SAT es una fórmula de CNF Φ. El problema de

satisfacibilidad es determinar si existe una asignación de valores a las variables de Φ que hace que Φ se evalúe a 1.

También es reseñable, como se ha indicado en el capítulo 4, que una amplia gama de problemas pueden ser modelados como problemas de Programación Lineal Entera Mixta (MILP) usando técnicas de formulación estándar. Sin embargo, en algunos casos el MILP resultante puede ser demasiado débil o demasiado grande para ser resuelto eficazmente por los solucionadores de última generación. En este trabajo se han desarrollado técnicas avanzadas de formulación para modelar mediante MILP la lógica booleana, de forma que resultan en formulaciones más fuertes y más pequeñas que las descritas en la literatura para esta clase de problemas.

Es importante destacar que la calidad de los modelos MILP desarrollados es de la máxima importancia, puesto que influirá directamente en el tamaño de la codificación resultante y en el tiempo que empleará el solucionador en encontrar la respuesta adecuada para verificar el circuito.

En el quinto capítulo se presentan el conjunto de pruebas que se han realizado para evaluar la bondad de la metodología de generación de vectores para la verificación de circuitos booleanos mediante MILP.

Las pruebas se han basado en el banco de pruebas estándar ISCAS'85, de donde se han escogido los circuitos combinacionales más empleados en la literatura. La complejidad de los circuitos abarca desde unas pocas puertas (C17) hasta circuitos de complejidad mediana con miles de ellas (C7552, con 3500 puertas aproximadamente).

De las pruebas se desprende que no solo la complejidad del circuito medida en términos de puertas ha de tenerse en cuenta, a la hora de predecir el tiempo requerido para obtener los vectores de verificación, sino que se ha de incluir la complejidad del interconexionado entre dichas puertas. Prueba de ello son los resultados obtenidos con los circuitos C7552 y C6288.

También es reseñable que los tiempos para obtener los vectores de verificación son inversamente proporcionales al porcentaje de entradas que se especifiquen como fijas y que dicha relación es directamente proporcional al

conjunto de salidas que sean forzadas. Además, se puede determinar si una especificación de entradas/salidas fijas es viable o no mediante el empleo de un tiempo límite de ejecución.

6.4 LÍNEAS FUTURAS

A partir de las conclusiones del presente proyecto, se plantean nuevas líneas de trabajo que parten de estas o que usan estas como base en su desarrollo:

- **Circuitos Secuenciales**: Este PFC aborda la problemática de la generación de vectores para verificar circuitos combinacionales. La evolución natural de la lógica combinacional, lleva a pensar que es posible aplicar la metodología propuesta a circuito secuenciales. En ese caso el problema se identificaría como la generación de la secuencia de vectores de entrada que permiten emplazar al circuito secuencial en un estado deseado.
- **Generación Asistida**: La generación de vectores se automatizaría mediante una aplicación de nivel superior que gestionase la herramienta de generación desarrollada en este PFC.
- **Test asistido por verificación**: La metodología propuesta puede ser ligeramente modificada para permitir la re-utilización de los patrones de verificación para el proceso de test. En ese caso habría de introducirse criterios de observabilidad y detección de fallos en las puertas.

BIBLIOGRAFÍA

[1] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*, 2nd ed. New York: Oxford University Press, 2010.

[2] J. Sosa, J. A. Montiel-Nelson, and S. Nooshabadi, "Application of Genetic Algorithm in Computing the Tradeoffs Between Power Consumption Versus Delay in Digital Integrated Circuit Design," *Microelectronics J.*, vol. 41, no. 2–3, pp. 135–141, 2010.

[3] S. Kwak, D. Har, J. G. Lee, and J. A. Lee, "Design of Heterogeneous Adders Based on Power-Delay Tradeoffs," in *2008 Fifth IEEE International Symposium on Embedded Computing*, 2008, pp. 223–226.

[4] S. Kwak, J. G. Lee, E. G. Jung, D. Har, M. D. Ercegovac, and J. A. Lee, "Exploration of Power-Delay Trade-Offs with Heterogeneous Adders by Integer Linear Programming," vol. 18, no. 4, pp. 787–800, 2009.

[5] J. Sosa, J. A. Montiel-Nelson, H. Navarro, and J. C. García, "Minimum Power Consumption Optimization Using Genetic Algorithms," in *Evolutionary and Deterministic Methods for Design, Optimization and Control with Applications to Industrial and Societal Problems*, no. September 2005, Munich, 2005.

[6] C. J. Sosa González and J. A. Montiel Nelson, "Metodología para la Verificación de Sistemas de Conmutación de Paquetes," Universidad de Las Palmas de Gran Canaria, 2006.

[7] S. Yang, "Logic Synthesis and Optimization Benchmarks User Guide Version 3.0," no. 919, pp. 0–44, 1991.

[8] Z. Zeng, P. Kalla, and M. Ciesielski, "LPSAT: a unified approach to RTL satisfiability," in *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, 2001, pp. 398–402.

[9] ILOG Inc., *ILOG CPLEX: Using the CPLEX linear optimizer and mixed integer optimizer*. Sunnyvale, CA: ILOG CPLEX Division, 2008.

[10] P. Deransart, R. S. Scowen, C. Biro, A. A. Ed-Dbali, and L. Cervoni, *Prolog: The Standard: Reference Manual*. Springer Berlin Heidelberg, 1996.

[11] R. Drechsler, Ed., *Advanced Formal Verification*. Boston: Kluwer Academic Publishers, 2004.

[12] M. L. Bushnell and V. D. Agrawal, *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Kluwer Academic Publishers, 2000.

[13] R. Drechsler and S. Höreth, "Gatecomp: Equivalence Checking of Digital Circuits in an Industrial Environment," *Int. Work. Boolean Probl.*, pp. 195–200, 2002.

[14] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, "Symbolic model checking using SAT procedures instead of BDDs," *Proc. 36th ACM/IEEE Conf. Des. Autom. Conf. - DAC '99*, pp. 317–320, 1999.

[15] S. A. Cook, "The Complexity of Theorem-proving Procedures," in *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, 1971, pp. 151–158.

[16] A. Newell, J. C. Shaw, and H. A. Simon, "Empirical explorations of the logic theory machine," in *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability on - IRE-AIEE-ACM '57 (Western)*, 1957, pp. 218–230.

[17] M. Davis and H. Putnam, *Feasible computational methods in the propositional calculus*. Rensselaer Polytechnic Institute, Research Division, 1958.

[18] M. Davis, G. Logemann, and D. Loveland, "A Machine Program for Theorem-proving," *Commun. ACM*, vol. 5, no. 7, pp. 394–397, 1962.

[19] D. J. Johnson and M. A. Trick, Eds., *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, Workshop, October 11-13, 1993*. Boston, MA, USA: American Mathematical Society, 1996.

[20] *ILOG CPLEX 7.0 Optimization User's Manual*. Incline Village, NV: ILOG Inc., 2000.

[21] *OVM User Guide*. Cadence Design Systems y Mentor Graphics, 2011.

[22] *Synopsys User Manual*, V1.0. Synopsys Inc.

PRESUPUESTO

P.1. INTRODUCCIÓN

El presupuesto realizado para el presente PFC está basado en el documento orientativo facilitado por el Colegio Oficial de Ingenieros de Telecomunicación (COIT) en 2008, al no disponer la titulación de Ingeniería en Electrónica de un colegio propio.

Hay que indicar que el motivo por el que este presupuesto toma como referencia una lista publicada en 2008 se debe a las modificaciones introducidas en el ordenamiento jurídico y a la actuación de los colegios profesionales en la ley 25/2009 del 22 de diciembre, por la cual se liberalizan los honorarios profesionales y ya no es posible seguir publicando este tipo de listas por parte del COIT.

Los costes asociados al desarrollo del proyecto se han organizado en siete secciones:

- Recursos humanos.
- Recursos hardware.
- Recursos software.
- Material fungible y gastos generales.
- Redacción del proyecto.
- Coste del visado.
- Coste total.

A continuación, se detallan cada uno de estos apartados, calculando los costes asociados a cada uno e indicando el presupuesto total del proyecto.

P.2. COSTES DE RECURSOS HUMANOS

El coste de recursos humanos está relacionado con el tiempo empleado por el personal necesario para el desarrollo del presente PFC. En este sentido, se ha estimado el tiempo de empleo del proyectando, que tiene encargadas las tareas propias del desarrollo del trabajo.

De acuerdo a las tarifas sugeridas por la documentación del Colegio Oficial de Ingenieros de Telecomunicación, se establece que el cálculo de honorarios para trabajos a tiempo completo, expresados en euros, se calcule mediante la ecuación:

$$H = C \cdot 75 \cdot Hn + C \cdot 95 \cdot He$$

Donde:

- C: factor de corrección dependiente del número de horas trabajadas (Tabla 25).
- H: honorarios totales.
- Hn: número de horas normales trabajadas.
- He: número de horas especiales.

Número de horas	Factor de corrección
Menos de 36	1
36-72	0,9
72-108	0,8
108-144	0,7
144-180	0,65
180-360	0,6
360-510	0,55
510-720	0,5
720-1080	0,45
Más de 1080	0,4

Tabla 25: Factor de corrección según el número de horas trabajadas

La carga laboral del Ingeniero durante la realización del presente PFC ha sido de 8 horas diarias a razón de 20 días mensuales durante 9 meses, con lo que el número total de horas trabajadas es de 1.440 horas. Dado que exceden de 1.080

horas de trabajo, es necesario aplicar un coeficiente corrector de 0,4 sobre el número de horas trabajadas. Como no se han realizado trabajos fuera del horario laboral, no se considera la contribución a la ecuación de las horas especiales.

$$H = C \cdot 75 \cdot Hn = 0{,}4 \cdot 75 \cdot (8 \cdot 20 \cdot 9) = 43.200\ €$$

Por lo tanto, el coste total de los honorarios asciende a CUARENTA Y TRES MIL DOSCIENTOS EUROS (43.200,00 €).

P.3. COSTES DE RECURSOS HARDWARE

El coste de los recursos hardware viene determinado por los equipos informáticos empleados para la realización del presente PFC y que se listan a continuación:

- Portátil Intel Core i7.
- Impresora láser profesional.

Debido a que algunos de estos recursos son compartidos por varios usuarios, el coste asociado se debe calcular en función del número de dichos usuarios y del periodo de amortización aplicado. En este sentido, se ha estimado un periodo de amortización de tres años para el portátil y cinco para la impresora. Además se ha considerado que la impresora es utilizada por 30 personas mientras que el portátil es de uso personal. Teniendo en cuenta estos datos, el coste se calcula en la Tabla 26.

Recurso Hardware	Coste unitario	Amortización	Tiempo empleado	Nº usuarios	Coste
Portátil Intel i7	810 €	36 meses	9 meses	1	202,50 €
Impresora láser profesional	3500 €	60 meses	9 meses	30	17,50 €
				TOTAL	220,00 €

Tabla 26: Costes asociados a los recursos hardware

El coste asociado a los recursos hardware asciende a DOSCIENTOS VEINTE EUROS (220,00 €).

P.4. COSTES DE RECURSOS SOFTWARE

El coste de recursos software (Tabla 27) se obtiene a partir del valor de las licencias y el mantenimiento de cada uno de los programas utilizados. Se considera que el período de amortización es de doce meses, de los cuales se ha usado nueve, por lo que el factor de amortización es de 0,75.

Recurso Software	Coste unitario	Amortización	Tiempo empleado	Coste
Microsoft Windows 8.1	220 €	12 meses	9 meses	165,00 €
Microsoft Office Profesional 2010	439 €	12 meses	9 meses	329,25 €
			TOTAL	494,25 €

Tabla 27: Costes asociados a los recursos software

El coste asociado a los recursos software asciende a CUATROCIENTOS NOVENTA Y CUATRO EUROS CON VEINTICINCO CÉNTIMOS (494,25 €).

P.5. COSTES DE MATERIAL FUNGIBLE Y GASTOS GENERALES

En este apartado se incluyen todos los materiales necesarios para la edición del proyecto. En la Tabla 28 se resume el coste debido a estos recursos.

Concepto	Coste unitario	Cantidad	Coste
Paquetes de papel DIN-A4 80 gr/m^2	4 €	1	4 €
Coste de impresión	250 €	1	250 €
Coste de encuadernación	15 €	3	45 €
CD-R	0,50 €	3	1,50 €
		TOTAL	300,50 €

Tabla 28: Costes de material fungible y gastos generales.

Por lo tanto, el coste total de material fungible asciende a la cantidad de TRESCIENTOS EUROS CON CINCUENTA CÉNTIMOS (300,50 €).

P.6. COSTES DE REDACCIÓN

El coste de la redacción del presente proyecto se calcula mediante la aplicación de la ecuación:

$$R = 0{,}07 \cdot P \cdot C$$

Donde:

- R: coste de la redacción.
- P: presupuesto que se obtiene sumando las cantidades obtenidas en los apartados anteriores.
- C: coeficiente de ponderación en función del coste del proyecto.

Teniendo en consideración estos datos, el valor de P es:

$$P = 43.200 + 220 + 494{,}25 + 300{,}50 = 44.214{,}75\ €$$

Para este valor de P, el coeficiente C vale 0,8. Por tanto, el valor de R es:

$$R = 0{,}07 \cdot 44.214{,}75 \cdot 0{,}8 = 2.476{,}03\ €$$

Por lo que el coste total de la redacción del proyecto asciende a la cantidad de DOS MIL CUATROCIENTOS SETENTA Y SEIS EUROS CON TRES CÉNTIMOS (2.476,03 €).

P.7. COSTE DEL VISADO

Los derechos de visado del COIT se calculan con la expresión:

$$V = 0{,}006 \cdot P \cdot C$$

Donde:

- V: coste del visado.
- P: presupuesto del proyecto.
- C: factor de corrección dependiente del número de horas trabajadas. Como el número de horas es superior a 1080 y empleando la Tabla 25 este factor es de 0,4.

Por lo que el valor de V es de:

$$V = 0{,}006 \cdot 44.214{,}75 \cdot 0{,}4 = 106{,}12\ €$$

Con esto, los costes de visado ascienden a la cantidad de CIENTO SEIS EUROS CON DOCE CÉNTIMOS (106,12€).

P.8. PRESUPUESTO TOTAL DEL PROYECTO

Finalmente, se muestra el resumen de todos los costes parciales del presupuesto en la Tabla 29.

Concepto	Importe
Recursos Humanos	43.200 €
Recursos Hardware	220 €
Recursos Software	494,25 €
Material fungible y gastos generales	300,50 €
Redacción	2.476,03 €
Visado	106,12 €
TOTAL (sin IGIC)	46.796,90 €

Tabla 29: Presupuesto total del proyecto

Al resultado de la suma de los diferentes costes se le debe añadir un 7% debido al IGIC.

Dña. María de las Nieves Gloria Hernández González declara que el presupuesto del Proyecto Fin de Carrera "Generación de Vectores para la Verificación de Circuitos Booleanos mediante MILP" asciende a una cantidad total de CINCUENTA MIL SETENTA Y DOS EUROS CON SESENTA Y OCHO CÉNTIMOS (50.072,68 €).

Fdo. María de las Nieves Gloria Hernández González
Las Palmas de G.C., junio de 2017

PLIEGO DE CONDICIONES

PC.1. INTRODUCCIÓN

En el pliego de condiciones se describen las condiciones que se exigen en este Proyecto Fin de Carrera para su correcta reproducibilidad. A continuación se describe de forma muy breve el conjunto de los componentes hardware y software empleados durante la realización del proyecto.

PC.2. RECURSOS HARDWARE

- Acceso a servidor de cómputo Linux Sun Fire X2200:
    - 2 Procesadores AMD Opteron 2214 Dual Core.
    - 2,2 GHz.
    - 8 GB RAM.
    - Gibabit Ethernet.
    - Red Hat Enterprise Linux Server 5.11.
- Ordenador personal ASUS X556U. Ordenador portátil en el que se han instalado los clientes X2Go y BitVise SSH, para la conexión remota. Así como, El Microsoft Office, para la edición del documento. Las principales características de este PC son:
    - Procesador Intel Core i7-6500U/BGA.
    - 8 GB de memoria RAM.
    - 500 GB de disco duro.

PC.3. RECURSOS SOFTWARE

- BitVise SSH Client 7.31.
- Cliente X2Go Qt version 4.8.6.
- Compilador de C y C++ gcc de GNU.
- Script en PERL blif2cnf.pl.

– Microsoft Office.

PC.4. CIRCUITOS DE REFERENCIA

Circuitos Benchmarks en formato BLIF de ISCAS'85, MCNC'91.

Nombre	Funcionalidad	Nombre	Funcionalidad
9symml	Count Ones	**count**	Counter
C1355	Error Correcting	**cu**	Logic
C17	Logic	**decod**	Decoder
C1908	Error Correcting	**des**	Data Encription
C2670	ALU and Control	**example2**	Logic
C3540	ALU and Control	**f51ml**	Arithmetic
C432	Priority Decoder	**frg1**	Logic
C499	Error Correcting	**frg2**	Logic
C5315	ALU and Selector	**lal**	Logic
C6288	16-bit Multiplier	**majority**	Voter
C7552	ALU and Control	**mux**	Mux
C880	ALU and Control	**my_adder**	Adder
alu2	ALU	**pair**	Logic
alu4	ALU	**parity**	Parity
apex6	Logic	**pcle**	Logic
apex7	Logic	**pcler8**	Logic
b1	Logic	**pm1**	Logic
b9	Logic	**rot**	Logic
c8	Logic	**sct**	Logic
cc	Logic	**tcon**	Logic
cht	Logic	**term1**	Logic
cm138a	Logic	**too_large**	Logic
cm150a	Logic	**ttt2**	Logic
cm151a	Logic	**unreg**	Logic
cm162a	Logic	**vda**	Logic
cm163a	Logic	**x1**	Logic
cm42a	Logic	**x2**	Logic
cm82a	Logic	**x3**	Logic
cm85a	Logic	**x4**	Logic
cmb	Logic	**z4ml**	2-bit Adder
comp	Logic		

Tabla 30: Relación de Benchmarks ISCAS'85

ANEXO I:
CÓDIGOS

AI.1. CÓDIGO C CNF2LP.C

Código desarrollado en lenguaje C para realizar el paso del problema de Satisfacibilidad del formato CNF al MILP.

```c
#include <stdio.h>
#include <stdlib.h>

#define MAX 15

/*****************************************************************************************/
/*                                      gen_or_gate                                      */
/*****************************************************************************************/

int gen_or_gate (FILE* lpFile, int* vect, int size, int LP_line, int aux_index)
{

    int i;
    int val = 0; //inv count

    if (size == 1)
    {
        if (vect[0] < 0)  //unset
            fprintf(lpFile, "b%d = 0\n", -vect[0]);
        else              //set
            fprintf(lpFile, "b%d = 1\n", vect[0]);
    }

    else if (size == 2)
    {
        fprintf(lpFile, "2 y%d - aux%d ", LP_line, aux_index++);

        for (i=0; i<size; i++)
        {
            if (vect[i] < 0)  //inv
            {
                fprintf(lpFile, "+ b%d ", -vect[i]);
                val++;
            }
            else
                fprintf(lpFile, "- b%d ", vect[i]);
        }

        fprintf(lpFile, "= %d\n", val);
    }

    else if (size <= 4)
    {
        fprintf(lpFile, "4 y%d - 2 aux%d - aux%d ", LP_line, aux_index++, aux_index++);

        for (i=0; i<(4-size); i++)
        {
            if (vect[i] < 0)  //inv
            {
                fprintf(lpFile, "+ 2 b%d ", -vect[i]);
                val++;
                val++;
            }
            else
                fprintf(lpFile, "- 2 b%d ", vect[i]);
        }

        for (i=(4-size); i<size; i++)
        {
            if (vect[i] < 0)  //inv
            {
                fprintf(lpFile, "+ b%d ", -vect[i]);
                val++;
            }
```

```
            else
                fprintf(lpFile, "- 2 b%d ", vect[i]);
        }

        for (i=(4-size); i<size; i++)
        {
            if (vect[i] < 0)  //inv
            {
                fprintf(lpFile, "+ b%d ", -vect[i]);
                val++;
            }
            else
                fprintf(lpFile, "- b%d ", vect[i]);
        }

        fprintf(lpFile, "= %d\n", val);
    }

    else if (size <= 8)
    {
        fprintf(lpFile, "8 y%d - 4 aux%d - 2 aux%d - aux%d ", LP_line, aux_index++, aux_index+
+, aux_index++);

        for (i=0; i<(8-size); i++)
        {
            if (vect[i] < 0)  //inv
            {
                fprintf(lpFile, "+ 2 b%d ", -vect[i]);
                val++;
                val++;
            }
            else
                fprintf(lpFile, "- 2 b%d ", vect[i]);
        }

        for (i=(8-size); i<size; i++)
        {
            if (vect[i] < 0)  //inv
            {
                fprintf(lpFile, "+ b%d ", -vect[i]);
                val++;
            }
            else
                fprintf(lpFile, "- b%d ", vect[i]);
        }

        fprintf(lpFile, "= %d\n", val);
    }

    else if (size <= 16)
    {
        fprintf(lpFile, "16 y%d - 8 aux%d - 4 aux%d - 2 aux%d - aux%d ", LP_line, aux_index+
+, aux_index++, aux_index++, aux_index++);

        for (i=0; i<(16-size); i++)
        {
            if (vect[i] < 0)  //inv
            {
                fprintf(lpFile, "+ 2 b%d ", -vect[i]);
                val++;
                val++;
            }
            else
                fprintf(lpFile, "- 2 b%d ", vect[i]);
        }

        for (i=(16-size); i<size; i++)
        {
            if (vect[i] < 0)  //inv
            {
                fprintf(lpFile, "+ b%d ", -vect[i]);
                val++;
            }
            else
                fprintf(lpFile, "- b%d ", vect[i]);
        }
```

```
        fprintf(lpFile, "= %d\n", val);
    }

    else if (size <= 32)
    {
        fprintf(lpFile, "32 y%d - 16 aux%d - 8 aux%d - 4 aux%d - 2 aux%d - aux%d ", LP_line,
aux_index++, aux_index++, aux_index++, aux_index++, aux_index++);

        for (i=0; i<(32-size); i++)
        {
            if ( (i%MAX) == 0 )
                if (vect[i] < 0)  //inv
                {
                    fprintf(lpFile, "+ 2 b%d\n ", -vect[i]);
                    val++;
                    val++;
                }
                else
                    fprintf(lpFile, "- 2 b%d\n ", vect[i]);
            else
                if (vect[i] < 0)  //inv
                {
                    fprintf(lpFile, "+ 2 b%d ", -vect[i]);
                    val++;
                    val++;
                }
                else
                    fprintf(lpFile, "- 2 b%d ", vect[i]);
        }

        for (i=(32-size); i<size; i++)
        {
            if ( (i%MAX) == 0 )
                if (vect[i] < 0)  //inv
                {
                    fprintf(lpFile, "+ b%d\n ", -vect[i]);
                    val++;
                }
                else
                    fprintf(lpFile, "- b%d\n ", vect[i]);
            else
                if (vect[i] < 0)  //inv
                {
                    fprintf(lpFile, "+ b%d ", -vect[i]);
                    val++;
                }
                else
                    fprintf(lpFile, "- b%d ", vect[i]);
        }

        fprintf(lpFile, "= %d\n", val);
    }

    return aux_index;
}
```

```
/*****************************************************************************************************/
/*                                          main                                                     */
/*****************************************************************************************************/

int main(int argc, char *argv[])
{
    if(argc != 3)
    {
        printf("ERROR1\n");
        exit(1);
    }

    char *cnfFilename = argv[1];
    FILE *cnfFile;

    char *lpFilename = argv[2];
    FILE *lpFile;

    if ((cnfFile = fopen(cnfFilename, "r")) == NULL || (lpFile = fopen(lpFilename, "w")) ==
NULL)
    {
        printf("ERROR2: Open file error\n");
        exit(2);
    }

    printf("CNF file opened: %s\n", cnfFilename);
    printf("LP file opened: %s\n", lpFilename);

//  Read cnf file

//  Lee linea a linea. Maximo 50 argumentos por linea

    char *line = NULL;
    int n = 0;
    int arg[50], args_assigned;

    int i = 0;
    int argst = 0;
    int inputs = 0;
    int* data = NULL;
    int* vector = NULL;

    while(getline(&line, &n, cnfFile) != -1)
    {
        args_assigned = sscanf(line, "%d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %
d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %
d", &arg[0], &arg[1], &arg[2], &arg[3], &arg[4], &arg[5], &arg[6], &arg[7], &arg[8], &arg[9],
&arg[10], &arg[11], &arg[12], &arg[13], &arg[14], &arg[15], &arg[16], &arg[17], &arg[18], &arg
[19], &arg[20], &arg[21], &arg[22], &arg[23], &arg[24], &arg[25], &arg[26], &arg[27], &arg
[28], &arg[29], &arg[30], &arg[31], &arg[32], &arg[33], &arg[34], &arg[35], &arg[36], &arg
[37], &arg[38], &arg[39], &arg[40], &arg[41], &arg[42], &arg[43], &arg[44], &arg[45], &arg
[46], &arg[47], &arg[48], &arg[49]);

        if (args_assigned > 49 && arg[49] != 0)
        {
            printf("ERROR3: MAX args_assigned = %d\n arg[49] = %d\n", args_assigned, arg[49]);
            exit(3);
        }

        if (args_assigned > 0)
        {
            argst += args_assigned;

            int* tmp_ptr = realloc(data, argst * sizeof(int));

            if (tmp_ptr == NULL)
            {
                printf("ERROR4: Memory allocation error\n");
                exit(4);
            }

            data = tmp_ptr;
```

```
            for (i = 0; i < args_assigned; i++)
            {
                data[argst - args_assigned + i] = arg[i];

                if (inputs < arg[i])
                    inputs = arg[i];
            }

        }

    }

    fclose(cnfFile);
    printf("CNF file closed: %s\n", cnfFilename);

    if (data[argst-1] != 0)    //If cnf_file has not last 0
    {
        argst ++;

        int* tmp_ptr = realloc(data, argst * sizeof(int));

        if (tmp_ptr == NULL)
        {
            printf("ERROR4: Memory allocation error\n");
            exit(4);
        }

        data = tmp_ptr;

        data[argst-1] = 0;

    }

//  Write lp file

    fprintf(lpFile, "max obj: y1\n\n");
    fprintf(lpFile, "subject to\n\n");

    int LP_line = 0; //AND_inputs
    int x = 0;   //aux_index

    char* inputsvector = calloc((inputs+1), sizeof(char));      //To load used inputs

    for (i = 0; i < argst; i++)  //Write OR gates
    {
        n = 0;  //n = OR size

        while ( data[i] != 0 )
        {
            if (data[i]>0)      //Load used inputs
                inputsvector[data[i]] = '1';
            else
                inputsvector[-data[i]] = '1';

            int* tmp_ptr = realloc(vector, (n+2) * sizeof(int));
            if (tmp_ptr == NULL)
            {
                printf("ERROR4: Memory allocation error\n");
                exit(4);
            }

            vector = tmp_ptr;

            vector[n] = data[i];

//      printf(" vector[%d] = %d    data[%d] = %d \n", n, vector[n], i, data[i]);

            n++;
            i++;
        }

        vector[n] = data[i];    //Load final 0

//      printf(" vector[%d] = %d    data[%d] = %d \n", n, vector[n], i, data[i]);
```

```
        fprintf(lpFile, "c%d: ", ++LP_line);

//      printf(" OR size = %d \n", n);

        if ( n <= 32)
            x = gen_or_gate (lpFile, vector, n, LP_line, x);
        else
        {
            printf("ERROR5: > MAX OR size = %d\n", n);
            exit(5);
        }

    }

//  Write SUM final_AND_gate

    fprintf(lpFile, "\nc%d: y1", (LP_line+1) );

    for (i = 2; i < (LP_line+1); i++)
        if ( (i%MAX) != 0 )
            fprintf(lpFile, " + y%d", i);
        else
            fprintf(lpFile, " + y%d\n", i);

    fprintf(lpFile, " = %d\n", (LP_line) );

//  Write binvars "y"

    fprintf(lpFile, "\n\nbin\n\n");

    for (i = 1; i < (LP_line+1); i++)
        if ( (i%MAX) != 0 )
            fprintf(lpFile, "y%d ", i);
        else
            fprintf(lpFile, "y%d\n ", i);

    fprintf(lpFile, "\n");

//  Write binvars "aux"

    for (i = 0; i < x; i++)
        if ( (i%MAX) != 0 || i == 0 )
            fprintf(lpFile, "aux%d ", i);
        else
            fprintf(lpFile, "aux%d\n ", i);

    fprintf(lpFile, "\n");

//  Write binvars "b"

    for (i = 1; i < (inputs+1); i++)
        if (inputsvector[i] == '1')
            if ( (i%MAX) != 0 )
                fprintf(lpFile, "b%d ", i);
            else
                fprintf(lpFile, "b%d\n ", i);

//      Free memory

        free(inputsvector);
    free(data);
    free(vector);

//  Write "end"

    fprintf(lpFile, "\n\nend\n");

    fclose(lpFile);
    printf("LP file closed: %s\n", lpFilename);

    exit(0);
}
```

AI.2. CÓDIGO C FIX_IO_CNF.C

Código desarrollado en lenguaje C para fijar entradas y/o salidas de los circuitos de referencia.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

int main(int argc, char *argv[])
{
    if(argc != 5)
    {
                printf("ERROR1\n");
        exit(1);
    }

    char *cnfFilenameOrig = argv[1];    //Original CNF File
    FILE *cnfFileOrig;

    FILE *cnfFileFix;                           //Modified CNF File with Inputs and Outputs
fixed

    int PInputs = atoi(argv[2]);        //Percentage of Inputs to be fixed

    int POutputs = atoi(argv[3]);       //Percentage of Outputs to be fixed

    int Iterations = atoi(argv[4]);     //Times to repeat the fixing

        if ((PInputs > 100) || (POutputs > 100))
        {
        printf("ERROR1: Max Percentage is 100\n");
        exit(1);
        }

        int iteration = 0;

        for (iteration = 1; iteration <= Iterations; iteration++)
        {
            char cp_File[100] = "cp ";          //Copy original cnf file to fixed cnf file
            char tmpFilename[50] = "F_";
            char cnfFilenameFix[50] = "./FixedCNF/";    //Modified CNF File with Inputs and
Outputs fixed

            char lpFilenameFix[100] = "./FixedCNF/FixedLP/";    //LP File with Inputs and
Outputs fixed
                char cmd_cnf2lp[150] = "../cnf2lp ";
            strcat (cp_File, cnfFilenameOrig);
            strcat (cp_File, " ");

            int l = strlen (cnfFilenameOrig);

            strncpy (tmpFilename, cnfFilenameOrig, (l-4));

                strcat (tmpFilename, "_");
                strcat (tmpFilename, argv[2]);
                strcat (tmpFilename, "_");
                strcat (tmpFilename, argv[3]);

                strcat (tmpFilename, "_i");
                char aiteration[5] = "";
                sprintf(aiteration, "%03d", iteration);
                strcat (tmpFilename, aiteration);

                strcat (lpFilenameFix, tmpFilename);
                strcat (lpFilenameFix, ".lp");

//              printf("fichero LP %d: %s\n", iteration, lpFilenameFix);
```

```
            strcat (tmpFilename, ".cnf");

            strcat (cnfFilenameFix, tmpFilename);

        strcat (cp_File, cnfFilenameFix);

        printf("\nNew File %d: %s\n", iteration, cp_File);

        system (cp_File);

        if ((cnfFileOrig = fopen(cnfFilenameFix, "r")) == NULL )
        {
            printf("ERROR2: Open file error to read\n");
            exit(2);
        }

        if ((cnfFileFix = fopen(cnfFilenameFix, "a")) == NULL)
        {
            printf("ERROR2: Open file error to append\n");
            exit(2);
        }

        char *line = NULL;
        int n = 0;
        int args_assigned = 0;
        char car_comment = '0';
        char string[7];
        int arg = 0;

        int i = 0;
        int inputs = 0;
        int outputs = 0;
        int* dataINPUTS = NULL;
        int* dataOUTPUTS = NULL;

        while(getline(&line, &n, cnfFileOrig) != -1)
        {
                    sscanf(line, "%c %s", &car_comment, string);
                    if ((car_comment == 'c') && (strcmp(string,"Inputs") == 0))     //
Found Inputs
                    {
                while(getline(&line, &n, cnfFileOrig) != -1)
                    {
                                    arg = 0;
                                    args_assigned = sscanf(line, "%c %s %d",
&car_comment, string, &arg);
                                    if (strcmp(string,"Outputs") != 0)      //Still Inputs
                                    {
                                    if ((args_assigned == 3) && (arg != 0))
                                    {
                                            inputs ++;

                                    int* tmp_ptr = realloc(dataINPUTS, inputs * sizeof
(int));

                                    if (tmp_ptr == NULL)
                                            {
                                                    printf("ERROR4: Memory
allocation error\n");
                                            exit(4);
                                            }

                                            dataINPUTS = tmp_ptr;
                                            dataINPUTS[inputs - 1] = arg;
                                    }
                                    }
                            else // Found Outputs
                                    {
                            while(getline(&line, &n, cnfFileOrig) != -1)
                                    {
                                            arg = 0;
                                            args_assigned = sscanf(line, "%c %s %
d", &car_comment, string, &arg);
```

```
                                        if ((args_assigned == 3) && (arg != 0))
                                        {
                                                outputs ++;

                                        int* tmp_ptr = realloc(dataOUTPUTS, outputs *
sizeof(int));

                                        if (tmp_ptr == NULL)
                                                {
                                                        printf("ERROR4:
Memory allocation error\n");
                                                exit(4);
                                                }

                                                dataOUTPUTS = tmp_ptr;
                                                dataOUTPUTS[outputs - 1] = arg;
                                        }
                                        }
                                }
                        }
                        }
                }

            fclose(cnfFileOrig);

        //      Append fixed inputs and outputs to cnfFileFix

            fprintf(cnfFileFix, "\nc Percentage Inputs Fixed: %d\nc Percentage Outputs Fixed:
%d\n", PInputs, POutputs);

            int* vector = calloc(((inputs*PInputs/100)+(outputs*POutputs/100)), sizeof
(int)); //To load inputs and outputs to fix
                int* posinputs = calloc((inputs*PInputs/100), sizeof(int));             //
Position in dataINPUTS to fix
                int* posoutputs = calloc((outputs*POutputs/100), sizeof(int));  //Position in
dataOUTPUTS to fix
            int j;
            char repeated = '1';

        //      Load inputs to fix in vector

                if (PInputs == 100)
                {
                        for (i=0; i<inputs; i++)
                                vector[i] = dataINPUTS[i];
                }
                else
                {
                        for (i=0; i<(inputs*PInputs/100); i++)
                        {
                                posinputs[i] = (rand() % inputs);
                        }

                    while (repeated == '1')
                    {
                        repeated = '0';
                        for (i=0; i<(inputs*PInputs/100); i++)
                        {
                            for (j=i+1; j<(inputs*PInputs/100); j++)
                            {
                                if (posinputs[i] == posinputs[j] && posinputs[i] !=
(inputs-1))
                                { //Repeated
                                    posinputs[i] = posinputs[i] +1;
                                    repeated = '1';
                                }
                                if (posinputs[i] == posinputs[j] && posinputs[i] ==
(inputs-1))
                                { //Repeated and is the last
                                    posinputs[i] = 0;
                                    repeated = '1';
                                }
                            }
                        }
                    }
```

```
                for (i=0; i<(inputs*PInputs/100); i++)
                {
                                vector[i] = dataINPUTS[posinputs[i]];
                        }
                }

        //      Load outputs to fix in vector

                repeated = '1';

                if (POutputs == 100)
                {
                        for (i=(inputs*PInputs/100); i<((inputs*PInputs/100)+outputs); i++)
                                vector[i] = dataOUTPUTS[i-(inputs*PInputs/100)];

                }
                else
                {
                        for (i=0; i<(outputs*POutputs/100); i++)
                        {
                                posoutputs[i] = (rand() % outputs);
                        }

                    while (repeated == '1')
                    {
                        repeated = '0';
                        for (i=0; i<(outputs*POutputs/100); i++)
                        {
                            for (j=i+1; j<(outputs*POutputs/100); j++)
                            {
                                if (posoutputs[i] == posoutputs[j] && posoutputs[i] !=
(outputs-1))
                                { //Repeated
                                    posoutputs[i] = posoutputs[i] +1;
                                    repeated = '1';
                                }
                                if (posoutputs[i] == posoutputs[j] && posoutputs[i] ==
(outputs-1))
                                { //Repeated and is the last
                                    posoutputs[i] = 0;
                                    repeated = '1';
                                }
                            }
                        }
                    }
                for (i=(inputs*PInputs/100); i<((inputs*PInputs/100)+(outputs*POutputs/100));
i++)
                {
                                vector[i] = dataOUTPUTS[posoutputs[i-(inputs*PInputs/100)]];
                        }
                }

        //      Write random fixed inputs and outputs to cnfFileFix

            for (i=0; i<((inputs*PInputs/100)+(outputs*POutputs/100)); i++)
            {
                        n = rand() % 2;
                        printf("vector[%d] = %d <-- %d\n", i, vector[i], n);

                        if (n == 0)
                        {
                        fprintf(cnfFileFix, "c unset %d\n", vector[i]);
                        fprintf(cnfFileFix, "-%d 0\n", vector[i]);
                }
                        else
                        {
                        fprintf(cnfFileFix, "c set %d\n", vector[i]);
                        fprintf(cnfFileFix, "%d 0\n", vector[i]);
                }
                }
```

```
        //      Free memory

                free(dataINPUTS);
            free(dataOUTPUTS);
            free(posinputs);
            free(posoutputs);
            free(vector);

            fclose(cnfFileFix);

            printf("CNF file with I/O fixed closed: %s\n", cnfFilenameFix);

        //      Call cnf2lp

            strcat (cmd_cnf2lp, cnfFilenameFix);
            strcat (cmd_cnf2lp, " ");
            strcat (cmd_cnf2lp, lpFilenameFix);

                printf("Comando cnf2lp %d: %s\n", iteration, cmd_cnf2lp);

                system (cmd_cnf2lp);

        }
}
```

ANEXO II:
BANCO DE PRUEBAS

AII.1. CIRCUITOS BENCHMARKS EN FORMATO BLIF

Se trata de los circuitos de referencia de ISCAS'85, MCNC'91 en formato BLIF.

Tabla 31: Relación de Benchmarks en formato BLIF

Fichero	Funcionalidad	Entradas	Salidas	Puertas
9symml.blif	Count Ones	9	1	43
C1355.blif	Error Correcting	41	32	546
C17.blif	Logic	5	2	6
C1908.blif	Error Correcting	33	25	880
C2670.blif	ALU and Control	233	140	1193
C3540.blif	ALU and Control	50	22	1669
C432.blif	Priority Decoder	36	7	160
C499.blif	Error Correcting	41	32	202
C5315.blif	ALU and Selector	178	123	2307
C6288.blif	16-bit Multiplier	32	32	2406
C7552.blif	ALU and Control	207	108	3512
C880.blif	ALU and Control	60	26	383
alu2.blif	ALU	10	6	335
alu4.blif	ALU	14	8	681
apex6.blif	Logic	135	99	452
apex7.blif	Logic	49	37	176
b1.blif	Logic	3	4	13
b9.blif	Logic	41	21	125
c8.blif	Logic	28	18	164
cc.blif	Logic	21	20	47
cht.blif	Logic	47	36	229
cm138a.blif	Logic	6	8	17
cm150a.blif	Logic	21	1	69
cm151a.blif	Logic	12	2	33
cm162a.blif	Logic	14	5	43
cm163a.blif	Logic	16	5	42
cm42a.blif	Logic	4	10	17
cm82a.blif	Logic	5	3	27
cm85a.blif	Logic	11	3	38
cmb.blif	Logic	16	4	41
comp.blif	Logic	32	3	151
count.blif	Counter	35	16	143
cu.blif	Logic	14	11	48
decod.blif	Decoder	5	16	22

Fichero	Funcionalidad	Entradas	Salidas	Puertas
des.blif	Data Encription	256	245	~4000
example2.blif	Logic	85	66	277
f51ml.blif	Arithmetic	8	8	43
frg1.blif	Logic	28	3	105
frg2.blif	Logic	143	139	1004
lal.blif	Logic	26	19	114
majority.blif	Voter	5	1	9
mux.blif	Mux	21	1	91
my_adder.blif	Adder	33	17	223
pair.blif	Logic	173	137	1434
parity.blif	Parity	16	1	68
pcle.blif	Logic	19	9	68
pcler8.blif	Logic	27	17	84
pm1.blif	Logic	16	13	39
rot.blif	Logic	135	107	691
sct.blif	Logic	19	15	91
tcon.blif	Logic	17	16	41
term1.blif	Logic	34	10	358
too_large.blif	Logic	38	3	578
ttt2.blif	Logic	24	21	200
unreg.blif	Logic	36	16	97
vda.blif	Logic	17	39	581
x1.blif	Logic	51	35	285
x2.blif	Logic	10	7	42
x3.blif	Logic	135	99	715
x4.blif	Logic	94	71	369
z4ml.blif	2-bit Adder	7	4	20

AII.2. CIRCUITOS BENCHMARKS EN FORMATO CNF

Son los circuitos de referencia de ISCAS'85, MCNC'91 transformados al formato CNF con indicación de las entradas y salidas primarias del circuito original.

Tabla 32: Relación de Benchmarks en formato CNF

Fichero	Funcionalidad	Entradas	Salidas	Puertas
9symml.cnf	Count Ones	9	1	43
C1355.cnf	Error Correcting	41	32	546
C17.cnf	Logic	5	2	6
C1908.cnf	Error Correcting	33	25	880
C2670.cnf	ALU and Control	233	140	1193
C3540.cnf	ALU and Control	50	22	1669
C432.cnf	Priority Decoder	36	7	160
C499.cnf	Error Correcting	41	32	202
C5315.cnf	ALU and Selector	178	123	2307
C6288.cnf	16-bit Multiplier	32	32	2406
C7552.cnf	ALU and Control	207	108	3512
C880.cnf	ALU and Control	60	26	383
alu2.cnf	ALU	10	6	335
alu4.cnf	ALU	14	8	681
apex6.cnf	Logic	135	99	452
apex7.cnf	Logic	49	37	176
b1.cnf	Logic	3	4	13
b9.cnf	Logic	41	21	125
c8.cnf	Logic	28	18	164
cc.cnf	Logic	21	20	47
cht.cnf	Logic	47	36	229
cm138a.cnf	Logic	6	8	17
cm150a.cnf	Logic	21	1	69
cm151a.cnf	Logic	12	2	33
cm162a.cnf	Logic	14	5	43
cm163a.cnf	Logic	16	5	42
cm42a.cnf	Logic	4	10	17
cm82a.cnf	Logic	5	3	27
cm85a.cnf	Logic	11	3	38
cmb.cnf	Logic	16	4	41
comp.cnf	Logic	32	3	151
count.cnf	Counter	35	16	143
cu.cnf	Logic	14	11	48

Fichero	Funcionalidad	Entradas	Salidas	Puertas
decod.cnf	Decoder	5	16	22
des.cnf	Data Encription	256	245	~4000
example2.cnf	Logic	85	66	277
f51ml.cnf	Arithmetic	8	8	43
frg1.cnf	Logic	28	3	105
frg2.cnf	Logic	143	139	1004
lal.cnf	Logic	26	19	114
majority.cnf	Voter	5	1	9
mux.cnf	Mux	21	1	91
my_adder.cnf	Adder	33	17	223
pair.cnf	Logic	173	137	1434
parity.cnf	Parity	16	1	68
pcle.cnf	Logic	19	9	68
pcler8.cnf	Logic	27	17	84
pm1.cnf	Logic	16	13	39
rot.cnf	Logic	135	107	691
sct.cnf	Logic	19	15	91
tcon.cnf	Logic	17	16	41
term1.cnf	Logic	34	10	358
too_large.cnf	Logic	38	3	578
ttt2.cnf	Logic	24	21	200
unreg.cnf	Logic	36	16	97
vda.cnf	Logic	17	39	581
x1.cnf	Logic	51	35	285
x2.cnf	Logic	10	7	42
x3.cnf	Logic	135	99	715
x4.cnf	Logic	94	71	369
z4ml.cnf	2-bit Adder	7	4	20