PROYECTO FIN DE MÁSTER EN INGENIERÍA DE COMPUTADORES

Máster en Investigación en Informática, Facultad de Informática, Universidad Complutense de Madrid

Evaluación de Expresiones Regulares sobre Hardware Reconfigurable

Autor: Ignacio Martín Santamaría

Director: Marcos Sánchez-Élez Martín

Curso académico 2009-2010

Autorización de difusión

El abajo firmante, matriculado en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "Evaluación de Expresiones Regulares sobre Hardware Reconfigurable", realizado durante el curso académico 2009-2010 bajo la dirección de Marcos Sánchez-Élez Martín, en el Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Ignacio Martín Santamaría

CONTENIDOS

RESUMEN

Los últimos avances en redes y subsistemas de almacenamiento continúan aumentando la velocidad a la que los flujos de datos deben ser procesados tanto internamente como en redes de computadores. Una proporción relativamente alta de la carga computacional en este tipo de aplicaciones puede implicar hacer correspondencias con expresiones regulares. Dado que el conjunto de patrones a ser examinado se espera que continúe creciendo y cambiando a lo largo del tiempo, el hardware que lo procese debe ser reconfigurable, de forma que pueda adaptarse dinámicamente a los requerimientos de esta correspondencia de patrones. Las FPGAs son ideales para este propósito.

En este trabajo, se ha creado un generador de código, programado en un lenguaje de alto nivel. Este generador de código toma un conjunto de expresiones regulares y crea el código VHDL que describe el sistema que realiza la correspondencia de patrones. La principal ventaja de este esquema es que es posible beneficiarse simultáneamente de las ventajas de las implementaciones software y hardware. Por tanto, se tiene la flexibilidad proporcionada por el uso del software para describir expresiones regulares y el rendimiento alcanzado cuando el producto final es una implementación hardware del sistema.

ABSTRACT

The latest developments on networks and memory subsystems, push up the speed requirements for applications that process data streams transmitted. A significant part of the computational cost of this type of applications may imply regular expressions matching. Since the number of patterns to be examined is expected to keep increasing and changing along the time, the hardware required to process them should be as much flexible as possible. This means that the hardware will be able to adapt itself dynamically to the requirements of pattern matching. Then, the FPGAs are the best solution for this objective.

In this work, a code generator has been created. It has been programmed on a high level language. This code generator takes a set of regular expressions and creates the VHDL code that describes the system that performs the pattern matching. This code is synthesized by a commercial tool into a FPGA. The main advantage of this schema is that it possible to take advantage of the capabilities of software and hardware implementations. Therefore, we have the flexibility provided by the use of software to describe regular expressions, and the performance reached when the final product is a hardware implementation of the system.

LISTA DE PALABRAS CLAVE

Reconocedor de expresiones regulares, correspondencia de patrones, generador de código, FPGA, inspección de paquetes, Telecomunicaciones, expresión regular, VHDL.

KEY WORDS LIST

Regular expression matching, pattern matching, code generator. FPGA, deep packet inspection, Telecommunications, regular expression, VHDL.

CAPÍTULO 1: INTRODUCCIÓN

En este capítulo se van a describir los elementos que afectan a la clasificación de paquetes de datos en la industria de las Telecomunicaciones.

En primer lugar, se analizarán las FPGAs, plataforma elegida para la implementación de los reconocedores. Se comenzará por su historia, pasando a sus características, funcionamiento, arquitectura interna, programación y aplicaciones. Por último se reseñarán brevemente los principales fabricantes, mostrando las distintas decisiones de diseño por las que han optado.

A continuación, se pasará a estudiar otro elemento fundamental: las expresiones regulares. Al igual que en el punto anterior, se comenzará por su historia y características más destacadas, para pasar a ver su relevancia en varios campos de la tecnología. Posteriormente se evaluará su validez como solución para el problema que se está tratando.

Finalmente, en el último punto se analizará la idoneidad del binomio FPGAs + Expresiones Regulares como respuesta al problema de la clasificación de paquetes en el entorno de las Telecomunicaciones.

HARDWARE RECONFIGURABLE: FPGAs

Según la descripción expuesta en [8], una FPGA (del inglés *Field Programmable Gate Array*) es un dispositivo semiconductor que contiene bloques lógicos cuya interconexión y funcionalidad se puede programar. La lógica programable puede reproducir desde funciones tan sencillas como las llevadas a cabo por una puerta lógica o un sistema combinacional hasta complejos sistemas en un chip.

Historia. ¿Por qué surgieron las FPGAs?

Tal y como se explica en [10], a principios de los 80 los circuitos LSI (*Large Scale Integrated circuits*) suponían la base de los circuitos lógicos en grandes sistemas. Microprocesadores, controladores de E/S etc.… fueron implementas usando tecnología de fabricación de circuitos integrados. Además era necesaria cierta "lógica pegamento" o interconexiones para comunicar grandes circuitos integrados para:

- Generar señales de control globales (resets, por ejemplo)
- Transportar señales de control de un subsistema a otro

Típicamente, los sistemas consistían en unos pocos componentes LSI y un gran número de SSI (*Small Scale Integrated circuits*) y MSI (*Medium Scale Integrated circuits*).

Un primer intento para resolver este problema llevó al desarrollo de circuitos integrados a medida. Esto reducía la complejidad del sistema y el coste de manufacturación, así como mejoraba el rendimiento. No obstante, tienen desventajas: son relativamente muy caros de desarrollar e introducen un retardo en el tiempo de llegada al mercado (time to market) debido al mayor tiempo de diseño. Hay dos tipos de costes asociados en el desarrollo de circuitos integrados a medida:

- Coste de desarrollo y diseño
- Coste de fabricación

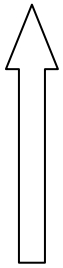
Por tanto, los circuitos integrados a medida sólo eran viables para productos con un alto volumen de manufacturación y donde el tiempo de llegada al mercado no fuera determinante.

Las FPGAs fueron inventadas en el año 1984 por Ross Freeman y Bernard Vonderschmitt, cofundadores de Xilinx, y fueron introducidas como una alternativa

a los circuitos integrados a medida para implementar el sistema completo en un chip y para proporcionar la flexibilidad de la reprogramación al usuario. Su introducción resultó en una mejora de la densidad con respecto a un conjunto componentes SSI/MSI individuales. Otra ventaja de las FPGAs respecto a los circuitos integrados es que, con la ayuda de herramientas de CAD (Computer Aided Design), los circuitos pueden ser implementados en un corto espacio de tiempo.

Actualmente las FPGAs están ganado terreno a los ASICs, son más lentas y tienen un mayor consumo de potencia, por lo que no pueden obtener las mismas prestaciones que los ASIC. A pesar de esto, las FPGAs tienen las ventajas de ser reprogramables (lo que añade una enorme flexibilidad al flujo de diseño), sus costes de desarrollo y adquisición son mucho menores para pequeñas cantidades de dispositivos, y el tiempo de desarrollo es también menor.

En cuanto a los microprocesadores, estos son los dispositivos más versátiles del mercado y pueden desempeñar cualquier tarea. Como contrapartida, esa versatilidad conlleva un diseño muy poco optimizado para el algoritmo que se esté ejecutando, lo que se traduce un rendimiento menor al de las FPGAs y ASICs. Además, el consumo de potencia es el mayor de todos. Los microprocesadores son muy utilizados porque ofrecen un comportamiento suficientemente bueno en muchos ámbitos, pero no son la mejor opción para tareas que requieran un cálculo intensivo.



Rendimiento	Número de ERs	Coste Unidad	Time To Market
ASIC	ASIC	FPGA	ASIC
FPGA	FPGA	MICRO	FPGA
MICRO	MICRO	ASIC	MICRO

Figura 1 Comparativa entre FPGAs, ASICs y microprocesadores

Características y funcionamiento

La información de este capítulo ha sido principalmente extraída de [9].

Todas las FPGAs independientemente del fabricante poseen unas características comunes relacionadas con la funcionalidad de las celdas y las interconexiones entre ellas.

Las FPGAs tienen una jerarquía de interconexiones programables que permite que los bloques lógicos de una FPGA sean interconectados según la necesidad del diseñador del sistema. Estos bloques lógicos e interconexiones pueden ser programados después del proceso de manufactura por el usuario/diseñador, así que el FPGA puede desempeñar cualquier función lógica necesaria.

Una tendencia reciente ha sido combinar los bloques lógicos e interconexiones de los FPGA con microprocesadores y periféricos relacionados para formar un «Sistema programable en un chip». Ejemplo de tales tecnologías híbridas pueden ser encontradas en los dispositivos Virtex-II PRO y Virtex-4 de Xilinx, los cuales incluyen uno o más procesadores PowerPC embebidos junto con la lógica del FPGA. Otra alternativa es hacer uso de núcleos de procesadores

implementados haciendo uso de la lógica del FPGA. Esos núcleos incluyen los procesadores MicroBlaze y PicoBlaze de Xilinx, Nios y Nios II de Altera, y los procesadores de código abierto LatticeMicro32 y LatticeMicro8.

Muchas FPGA modernas soportan la reconfiguración parcial del sistema, permitiendo que una parte del diseño sea reprogramada, mientras las demás partes siguen funcionando. Este es el principio de la idea de la «computación reconfigurable», o los «sistemas reconfigurables».

Los **bloques lógicos** de una FPGA se implementan usando varias puertas con un bajo fan-in, lo que proporciona un diseño más compacto, comparado con una implementación con dos niveles de lógica AND-OR. Hay dos aspectos que son configurables en este tipo de sistemas:

1. La intersección entre bloques lógicos

2. La función de cada bloque lógico

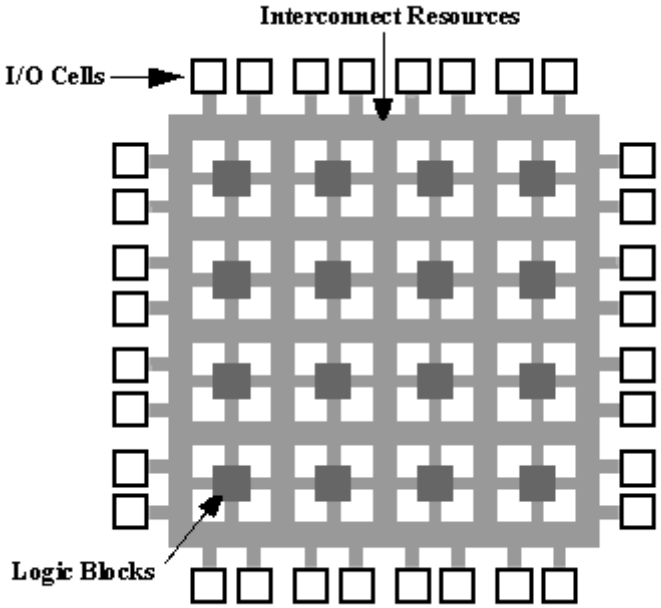


Figura 2 Esquema del hardware de una FPGA. Imagen tomada de [10]

El **enrutamiento** en las FPGAs consiste en segmentos de cables de longitud variable que pueden ser interconectados mediante switches programables eléctricamente. Es un aspecto especialmente relevante, ya que la densidad de los bloques lógicos usados en una FPGA depende del número y longitud de los segmentos de cable usados para el enrutamiento. De hecho, la cantidad de segmentos utilizados para la interconexión es una solución de compromiso entre la densidad de los bloques lógicos y el área usada para enrutamiento.

Las FPGAs también se pueden diferenciar por utilizar diferentes **tecnologías de memoria**:

- Volátiles: Basadas en RAM. Su programación se pierde al quitar la alimentación. Requieren una memoria externa no volátil para configurarlas al arrancar (antes o durante el reset).

- No Volátiles: Basadas en ROM. Hay de dos tipos, las reprogramables y las no reprogramables.

    - Reprogramables: Basadas en EPROM o flash. Éstas se pueden borrar y volver a reprogramar aunque con un límite de unos 10.000 ciclos.

    - No Reprogramables: Basadas en fusibles. Solo se pueden programar una vez, lo que las hace poco recomendables para trabajos en laboratorios.

Diseño

El diseño se realiza mediante la definición de la función lógica que realizará cada uno de los CLB, seleccionar el modo de trabajo de cada IOB e interconectarlos.

El diseñador cuenta con la ayuda de entornos de desarrollo especializados en el diseño de sistemas a implementar en una FPGA. Un diseño puede ser capturado ya sea como esquemático, o haciendo uso de un lenguaje de programación especial. Estos lenguajes de programación especiales son conocidos como HDL o *Hardware Description Language*. Los HDLs más utilizados son:

- VHDL
- Verilog
- ABEL

En un intento de reducir la complejidad y el tiempo de desarrollo en fases de prototipado rápido, y para validar un diseño en HDL, existen varias propuestas y niveles de abstracción del diseño.

Flujo de diseño: una de las ventajas más importantes de los diseños basados en FPGAs es que los usuarios pueden crearlos usando herramientas de CAD. El flujo de diseño genérico sobre una FPGA incluye los siguientes pasos:

1. **Diseño del sistema**: En esta etapa el diseñador tiene qué decidir qué parte de la funcionalidad desea implementar en la FPGA y cómo integrar dicha funcionalidad con el resto del sistema.

2. **Integración de la E/S con el resto del sistema**: los flujos de entrada de la FPGA están integrados con el resto del PCB (Printed Circuit Board), lo que permite el diseño del PCB en una etapa temprana del proceso. Los fabricantes de FPGAs proporcionan herramientas de automatización para el diseño de la E/S.

3. **Descripción del diseño**: el diseñador describe la funcionalidad del sistema bien usando editores de esquemáticos o usando un lenguaje de descripción de alto nivel como VHDL o Verilog.

4. **Síntesis**: una vez que el diseño ha sido definido, las herramientas de CAD son usadas para implementar el diseño en una FPGA concreta. La síntesis incluye optimizaciones genéricas, de consumo etc. y está seguida por la ubicación y el enrutamiento. Al final de este proceso, se obtiene un archivo bit stream.

5. **Verificación del diseño**: un simulador procesa el bit stream para simular la funcionalidad del diseño y reportar errores en el comportamiento deseado del diseño. También pueden usarse herramientas específicas para estudiar el comportamiento temporal del sistema y sus posibles restricciones, tanto internas como específicas de la FPGA en el que será implementado. Finalmente, el diseño es cargado en la FPGA final y el testeo es realizado en un entorno real.

Aplicaciones

Cualquier circuito de aplicación específica puede ser implementado en un FPGA, siempre y cuando esta disponga de los recursos necesarios. Las aplicaciones donde más comúnmente se utilizan los FPGA incluyen a los DSP (procesamiento digital de señales), radio definido por software, sistemas aeroespaciales y de defensa, prototipos de ASICs, sistemas de imágenes para medicina, sistemas de visión para computadoras, reconocimiento de voz, bioinformática, emulación de hardware de computadora, entre otras. Cabe notar que su uso en otras áreas es cada vez mayor, sobre todo en aquellas aplicaciones que requieren un alto grado de paralelismo.

Existe código fuente disponible (bajo licencia GNU GPL) de sistemas como microprocesadores, microcontroladores, filtros, módulos de comunicaciones y memorias, entre otros. Estos códigos se llaman cores.

Fabricantes

Actualmente, en la industria de las FPGA hay dos grandes fabricantes de FPGAs de propósito general que están a la cabeza del mercado, Xilinx y Altera, y un conjunto de otros competidores que se diferencian por ofrecer dispositivos de capacidades únicas. La información de cada fabricante ha sido obtenida de su página web (referencias de la [1] a la [7], por este orden).

- Xilinx es la mayor productora mundial de lógica programable y la inventora de las FPGAs. Una de sus mayores virtudes es que no sólo fabrica los chips, sino que también ofrece software (Xilinx ISE Design Suite), IPs, tarjetas y kits orientados a las aplicaciones objetivo.

  En cuanto a las FPGAs, actualmente Xilinx desarrolla dos familias:

  - Virtex: orientadas al alto rendimiento.

  - Spartan: más económicas y aptas para aplicaciones que requieran costes reducidos.

- Altera: la otra gran empresa del sector. Junto con Xilinx, acaparan la mayor parte del mercado. Al igual que Xilinx ofrece IPs y software, Quartus II Design Software, aunque este no está tan depurado como el Xilinx ISE. También ofrece dos líneas de FPGAs de diferentes características:

  - Stratix: los dispositivos de Altera de mayor tamaño y prestaciones

- Ciclone y Arria: de menor coste y enfocadas a aplicaciones con requerimientos más contenidos.

- Lattice Semiconductor: aparte de CPLDs y SPLDs, Lattice también fabrica FPGAs. Sus productos están principalmente orientados al bajo coste y consumo, aunque también fabrica dispositivos de alto rendimiento.

- Actel: ofrece FPGAs basadas en tecnología Flash reprogramable. Su seña de identidad es el bajo consumo y esa es la directiva bajo la cual están diseñados todos sus productos, comercializados con la serie Igloo.

- QuickLogic: ofrece FPGAs basadas en antifusibles (programables una sola vez).

- Atmel: esta empresa está enfocada en proveer microcontroladores AVR con FPGAs, todo en el mismo encapsulado.

- Achronix Semiconductor: desarrolla FGPAs capaces de trabajar a muy altas frecuencias. Su chip Speedster funciona a 1.5GHz.

INTRODUCCIÓN A LAS EXPRESIONES REGULARES

Una expresión regular es un conjunto de símbolos y caracteres que describe un conjunto de cadenas sin enumerar sus elementos. Su potencia a la hora de describir conjuntos regulares hace que sean utilizadas en varios dominios científicos, entre ellos la Informática y las Telecomunicaciones.

Historia

Según [12], los orígenes de las expresiones regulares residen en la teoría de autómatas y en la teoría de lenguaje formal. Estos campos estudian modelos de computación (autómatas) y formas de clasificar los lenguajes formales. En la década de 1950, el matemático Stephen Cole Kleene describió estos modelos usando su notación matemática llamada conjuntos regulares. El lenguaje SNOBOL fue una implementación temprana de la correspondencia de patrones, pero no idéntica a las expresiones regulares. Ken Thompson integró la notación de Kleene en el editor QED como manera de encontrar cadenas en un texto. Posteriormente añadió esta posibilidad al editor de Unix *ed*, lo que finalmente desembocó en el uso de de expresiones regulares en la popular herramienta de búsqueda *grep*.

Por otro lado, el uso de expresiones regulares en estándares de información estructurada para modelado de documentos y bases de datos comenzó en los años sesenta y se expandió en los ochenta, cuando estándares de la industria tales como ISO SGML las consolidaron ([12]).

Debido a la variedad de situaciones y de entornos en los que las expresiones regulares son utilizadas, especialmente entre distintas aplicaciones de entornos Unix, habían proliferado una variedad de sintaxis y funcionalidades. El estándar POSIX 1003.2 del IEEE trata de remediar esta heterogeneidad y ofrecer un modelo homogéneo. Define el conjunto mínimo de metacaracteres, u operadores, que se debe soportar, así como la función de cada uno de estos ([12]).

Utilización

Una expresión regular describe un conjunto de cadenas. Normalmente se utilizan para dar una descripción concisa de un conjunto, sin tener que enumerar todos los elementos.

En los últimos años y gracias al desarrollo de Internet, han encontrado un nuevo campo en el que están muy presentes. Su potencia para representar una enorme variedad de cadenas de una forma compacta ha hecho que sean utilizadas a la hora de encontrar código malicioso o spam. De la misma manera, los filtros y los robots emplean estos mecanismos para detectar elementos potencialmente dañinos.

Sintaxis

La sintaxis es una variación de la expuesta en [11]. La mayoría de formalizaciones proporcionan las siguientes construcciones, que pueden ser combinadas para formar expresiones arbitrariamente complejas:

- Alternación: Una barra vertical separa las alternativas. Por ejemplo, "casa|coche" se corresponde con *casa* o *coche*.
- Cuantificación: un cuantificador tras un carácter especifica la frecuencia con la que éste puede ocurrir. Los cuantificadores más comunes son +, ? y *:
    - +: El signo más indica que el carácter al que sigue debe aparecer al menos una vez. Por ejemplo, "bis+abuelo" describe el conjunto infinito *bisabuelo*, *bisbisabuelo*, *bisbisbisabuelo*, etcétera.
    - ?: El signo de interrogación indica que el carácter al que sigue puede aparecer como mucho una vez. Por ejemplo, "ob?scuro" se corresponde con *oscuro* y *obscuro*.
    - *: El asterisco, también llamado cierre de Kleene, indica que el carácter al que sigue puede aparecer cero, una, o más veces. Por ejemplo, "0*42" se corresponde con *42*, *042*, *0042*, *00042*, etcétera.

- Agrupación: Los paréntesis pueden usarse para definir el ámbito y precedencia de los demás operadores. Por ejemplo, "(p|m)adre" es lo mismo que "padre|madre", y "(des)?amor" se corresponde con *amor* y con *desamor*.

IMPORTANCIA DEL RECONOCIMIENTO DE ERs EN LAS TELECOMUNICACIONES

Tal y como se explica en [14], los avances en redes y subsistemas de almacenamiento continúan aumentando la velocidad a la que los flujos de datos deben ser procesados tanto internamente como en redes de computadores. Mientras tanto, el contenido de estos flujos de datos está sujeto a un escrutinio cada vez mayor y a distintos niveles. Los patrones de interés pueden incluir cadenas sencillas, como “perro” o “gato”, por ejemplo; o secuencias más complejas de reconocer como números de tarjetas de crédito, valores de divisas, números de teléfono, que requieren un mecanismo de especificación más general. Para tales aplicaciones, el lenguaje de especificación de patrones más ampliamente utilizado son las expresiones regulares.

Para algunas aplicaciones, como el filtrado de cabeceras de paquetes, la localización de un patrón dado puede estar *anclada*, en el sentido de que una correspondencia se da sólo si el patrón empieza o termina en un conjunto de localizaciones prescritas dentro del flujo de datos. Pero lo más común es que el filtrado deba hacerse sobre conjuntos de paquetes o flujos de datos desestructurados.

Algunas aplicaciones requieren el análisis concurrente de miles de patrones en cada byte del flujo de datos. Por ejemplo: sistemas de detección de intrusos y prevención, que operan usando un número de expresiones del orden de 10000, sistemas de monitorización de email, que pueden escanear correos salientes buscando contenido ilegal o inapropiado; filtros de spam, que imponen patrones específicos para cada usuario en el correo entrante; escáner de virus, que examinan los archivos en busca de señales de programas que se sabe que

son dañinos…En todos estos casos el conjunto de patrones a reconocer es frecuentemente modificado y puede crecer de forma muy rápida [13].

Las máquinas de hoy no pueden procesar semejante carga de correspondencia de patrones, dada la velocidad de los flujos de datos originados en las redes de alta velocidad y los sistemas de almacenamiento de alta productividad.

CASO A ESTUDIO

Uno de los campos más importantes en el que las expresiones regulares desempeñan una labor fundamental es la inspección profunda de paquetes (DPI, *Deep Packet Inspection*). El tráfico es sometido a un análisis, que puede ser realizado a distintos niveles, y es clasificado haciendo correspondencia entre un conjunto de patrones o cadenas representados en forma de expresiones regulares. Dicha clasificación puede ser empleada para diferenciar distintas clases de servicio, identificar código malicioso (dentro de un Network Intrusion Detection System, por ejemplo)… Una proporción relativamente alta de la carga computacional en este tipo de aplicaciones puede implicar hacer correspondencias con expresiones regulares, lo que consume un elevado porcentaje del tiempo de computación, ya que el conjunto de dichos conjuntos que están siendo procesados en un determinado momento en un procesador de DPI pueden crecer muy rápidamente. Ya que las velocidades de las redes tienen que satisfacer las demandas de los usuarios de servicios de calidad más alta, crecientes volúmenes de tráfico multimedia o ubicuidad de la conexión a internet, será necesaria una mayor capacidad de procesamiento de tráfico.

El uso de una solución hardware para el reconocedor de patrones se antoja muy interesante. Dado que el conjunto de patrones a ser examinado se espera que continúe creciendo y cambiando a lo largo del tiempo, el hardware que lo procese debe ser reconfigurable, de forma que pueda adaptarse dinámicamente a los requerimientos de esta correspondencia de patrones rápida. Las FPGAs son

ideales para este propósito, debido a su capacidad de procesamiento en paralelo y a su potencial de reconfiguración.

Las expresiones regulares ofrecen ventajas tales como la compactación de varias expresiones en una sola que puede representar un gran número de cadenas. De esta manera, las soluciones hardware para correspondencia de patrones están siendo muy utilizadas, especialmente explotando las FGPAs en dominios de aplicaciones que hagan un uso importante de las expresiones regulares, tales como procesadores de DPI o datamining, donde se requieren actualizaciones frecuentes de los patrones de referencia.

CAPÍTULO 2: TRABAJO RELACIONADO

Actualmente existen otras alternativas tecnológicas para el problema del reconocimiento de expresiones regulares para Deep Packet Inspection:

Entre las soluciones software destaca especialmente Snort [15]. Este proyecto Open Source es un sistema de detección y prevención de intrusiones (IDS/IPS) basado en reglas. Es la alternativa software más extendida y es frecuentemente utilizada para medir la validez de una implementación hardware.

En cuanto a implementación hardware, sobresalen dos fabricantes:

- NetLogic [16]: esta empresa ha desarrollado un producto, NETL7, diseñado para realizar inspección de paquetes a altas velocidades. Sus chips son capaces de inspeccionar 10000 firmas en paralelo a una velocidad de 2.6 Gbps, utilizando cuatro cores.

- Procera [17]: ofrece la plataforma Packet Logic, que, entre otras tareas, también realiza labores de Deep Packet Inspection. Es capaz de reconocer 1000 expresiones regulares diferentes a 5 Gbps.

En el campo de la investigación, también se han propuesto otras alternativas, entre las que, por relación directa con este trabajo, destacan las siguientes:

- En [18], Bonesana, Paolieri y Santambrogio proponen un framework completo para correspondencia de patrones. Plantea el diseño de un procesador de propósito general, adaptado a los requerimientos de la correspondencia de patrones, conjuntamente con un compilador inspirado en diseños VLIW. El procesador sería programado usando expresiones regulares como lenguaje de programación y tendría una arquitectura segmentada. Uno de los objetivos principales es

explotar el paralelismo propio de las FPGAs, por lo que proponen un esquema para que se procese más de un carácter en cada ciclo.

- Bispo, Sourdis, Cardoso y Vassiliadis ofrecen en [19] como alternativa una propuesta similar a la expuesta en este trabajo. En primer lugar, defienden el uso de expresiones regulares en lugar de patrones fijos como medio para detectar secuencias. Sugieren el uso de una herramienta generadora de código, que, dada una expresión regular, describe el sistema que la reconoce. En cambio, los circuitos necesarios para la implementación son más complejos, contando con un decodificador 8 a 256 bits, por ejemplo. Por otra parte, sugieren técnicas para reducir el área empleada, mejorando de este modo la implementación en la lógica reconfigurable.

- P. Sutton, en [20], continúa las ideas expuestas por Sidhu y Prasanna en [21], utilizado como base para este trabajo. En este artículo, plantea la posibilidad de decodificar más de un carácter por cada ciclo, extendiendo para ello el método de construcción modular. Así mismo, introduce el concepto de decodificación parcial, en el que unidades de reconocimiento de caracteres son compartidas. El número de señales necesarias para el enrutamiento en la FPGA pasan por un proceso en el cual su número se reduce, con el objetivo de disminuir el tamaño de los decodificadores

CAPÍTULO 3: RECONOCEDOR DE EXPRESIONES REGULARES

En este capítulo se va a estudiar el diseño del reconocedor de expresiones regulares y el proceso que se ha seguido hasta obtener su diseño. El trabajo está basado en la propuesta de R. Sidhu y V.K. Prasanna en [21]. Se comenzará por una explicación general del sistema y de sus principales características. A grandes rasgos, el análisis podría dividirse en dos bloques:

Generador de código: se tratará la sintaxis que tienen que seguir las expresiones regulares, para luego pasar al algoritmo generador del código VHDL, explicando cuál es el procedimiento para, a partir de una expresión regular, obtener la arquitectura del hardware que la reconoce.

Reconocedor de expresiones regulares: se analizará el hardware descrito por el código VHDL, empezando por el sistema completo, para posteriormente pasar a analizar de forma individual los elementos que lo componen.

DESCRIPCIÓN GENERAL

La principal característica diferenciadora de esta arquitectura respecto a otras alternativas propuestas es la inclusión de un generador de código. Esta capa por encima del reconocedor de expresiones regulares propiamente dicho, abre la puerta a multitud de posibilidades. En la **Figura3** se puede ver un esquema del funcionamiento de la arquitectura:

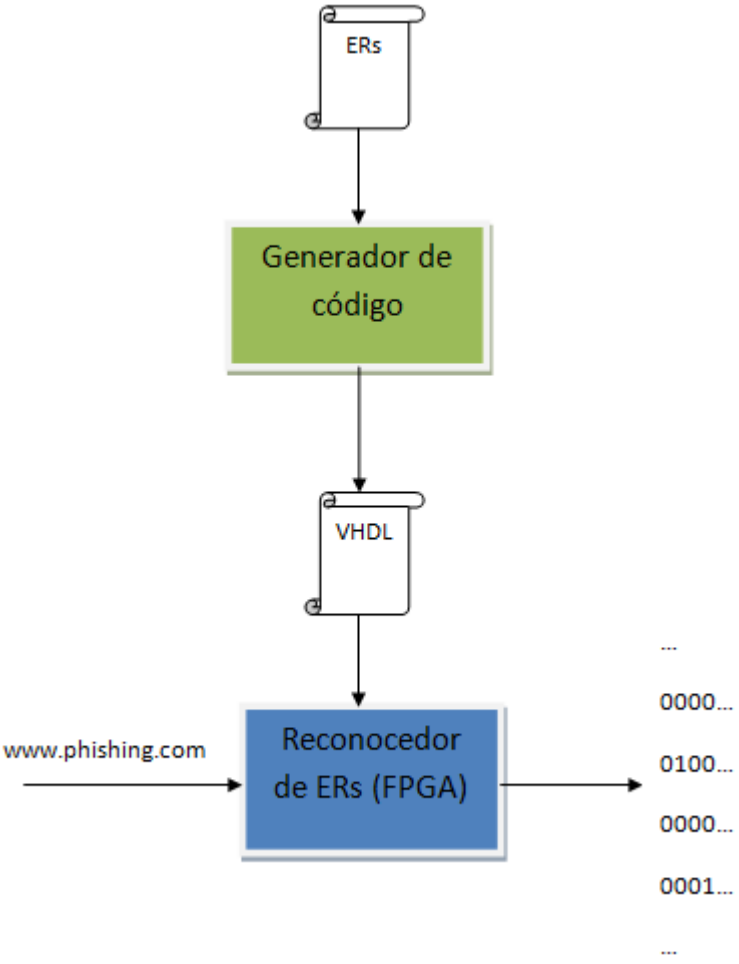


Figura 3 Esquema del funcionamiento de la arquitectura

El fichero de texto con las expresiones regulares pasa al generador de código. Este es una aplicación escrita en java que implementa el algoritmo descrito por Sidhu y Prasanna [21] y que genera como salida un fichero VHDL que describe la arquitectura que reconoce las expresiones regulares del fichero de entrada. La idea de crear una herramienta generadora está expuesta en [22] y [23].

La principal característica del generador de código es que introduce una nueva capa sobre el código VHDL, aunando las ventajas de las implementaciones hardware y software. Con este esquema, se dispone por un lado de la flexibilidad y facilidad de codificación que proporciona el reconocimiento de expresiones regulares por software; y, por otra parte, el producto final es una implementación

hardware del sistema, con las evidentes ventajas en cuanto a rendimiento que esto supone.

El generador, al estar escrito en un lenguaje de alto nivel como es Java, proporciona muchos beneficios:

- Facilidad en la gestión del programa: desde estructuras de control de flujo que serían imposibles, o muy incómodas, en VHDL, hasta el uso de objetos o manejo de ficheros.

- Análisis exhaustivo: al no estar encorsetado por las limitaciones de un lenguaje de descripción hardware, es posible realizar un análisis todo lo fino que se desee para, de esta forma, generar un código VHDL más optimizado.

- Posibilidades de gestión y optimización: tareas como cambiar la implementación de un módulo, introducir uno nuevo o incluso la opción de realizar estas y otras tareas de forma remota, no supone ningún problema.

GENERADOR DE CÓDIGO

Sintaxis de las ERs

La sintaxis soportada es esencialmente la misma que la descrita en el Capítulo 1. Tan sólo hay que reseñar que la operación a+ es internamente sustituida por a(a*), que es equivalente y simplifica la implementación. En cualquier caso, este hecho es totalmente transparente al usuario y no afecta de ningún modo al uso de la sintaxis.

Cabe señalar que se ha introducido la posibilidad de crear clases de expresiones regulares. Por ejemplo, las expresiones se pueden agrupar en función del tipo de tráfico que identifiquen. Habría una clase que reconocería el

intercambio de archivos P2P, otra el streaming de vídeo, etc.… La adición de esta etiqueta identificadora permite que se aglutinen múltiples funcionalidades en un solo fichero de expresiones regulares. Para seleccionar qué clase se quiere implementar, tan sólo hay que indicar al generador de código la etiqueta de dicha clase.

Algoritmo

El algoritmo acepta una expresión regular en forma postfija y genera un código VHDL capaz de reconocerla. En nuestro generador de código, la entrada de datos es un fichero de texto con las expresiones regulares expresadas en notación tradicional, por lo que es necesario hacer una conversión a notación postfija. Esto se realiza mediante el recorrido en postorden de la expresión. Una vez en la notación adecuada, la expresión regular se va parseando, y en función del símbolo de su tipo se realiza un conjunto de operaciones u otro. El algoritmo utiliza una pila y se basa en las siguientes rutinas y subrutinas de ubicación:

- place_char, place_|, place_· y place_*: colocan, respectivamente, los bloques lógicos que implementan el reconocimiento de caracteres (bloque básico) y las operaciones asociadas con los metacaracteres |, · y *. En todos los casos se devuelve una referencia p a la estructura colocada.

- route1 y route2: crean conexiones bidireccionales entre las estructuras lógicas referenciadas como argumentos. route1(q, p) conecta la salida o de q con la entrada i1 de p y la salida o de p con la entrada i1 de q. Similarmente, route2 crea conexiones a los puertos i2 y o2 de p.

- Las últimas dos líneas del algoritmo crean las conexiones i y o para la estructura lógica del último símbolo de la expresión regular.

Algoritmo de generación del código. Extraído de [21]:

```
for (i=1; i<regexp_len; i++){
      switch (regexp[i]){
            case char: place_char(regexp[i],&p);
                        push(p);
            case |:     place_|(&p);
                        pop(&p1);
                        route1(p,p1);
                        pop(&p2);
                        route2(p,p2);
                        push(p2);
            case ·:     place_·(&p);
                        pop(&p1);
                        route1(p,p1);
                        pop(&p2);
                        route2(p,p2);
                        push(p2);
            case *:     place_*(&p);
                        pop(&p1);
                        route1(p,p1);
                        push(p);
            }
}
pop(&p);
route_input_high(p);
route_output_ff(p);
```

RECONOCEDOR DE EXPRESIONES REGULARES

Visión general

El objetivo del sistema es, dado un flujo continuo de caracteres, identificar si en ese flujo existe alguna subcadena que se corresponda con una de las expresiones regulares soportadas. Cada expresión regular es implementada de forma individual por un reconocedor específico. Por tanto, en el sistema habrá tantos reconocedores como expresiones regulares sean soportadas . Para ajustarnos a la terminología más extendida, vamos a denominar a estos reconocedores *engines*. Cada uno de estos engines es capaz de reconocer una expresión regular en concreto y está diseñado específicamente para reconocer tal expresión. Una vez configurado, no es posible cambiar la expresión regular reconocida. La complejidad de cada engine depende por completo de la estructura y longitud de la expresión regular que lo define. Cada engine recibe como entrada un flujo de caracteres, cada uno representado por un byte, de forma síncrona, lo procesa e indica con un bit de salida si en ese flujo de caracteres se ha encontrado alguna cadena que se corresponda con la expresión regular que se reconoce.

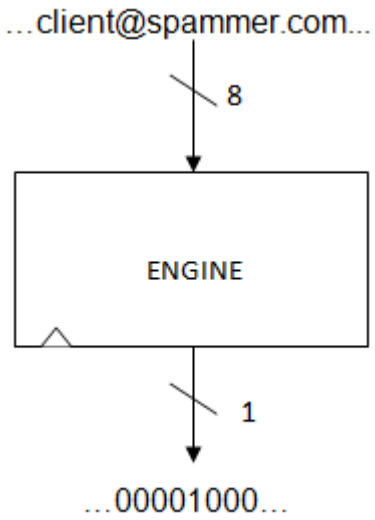


Figura 4 Esquema del engine

Aprovechando el paralelismo propio de este tipo de arquitecturas, la FPGA tendrá sintetizados en su lógica un gran número de estos engines. Cada engine trabaja de forma independiente al resto de los módulos y no necesita de ningún tipo de intercomunicación. Un posible esquema de la arquitectura sería el siguiente:

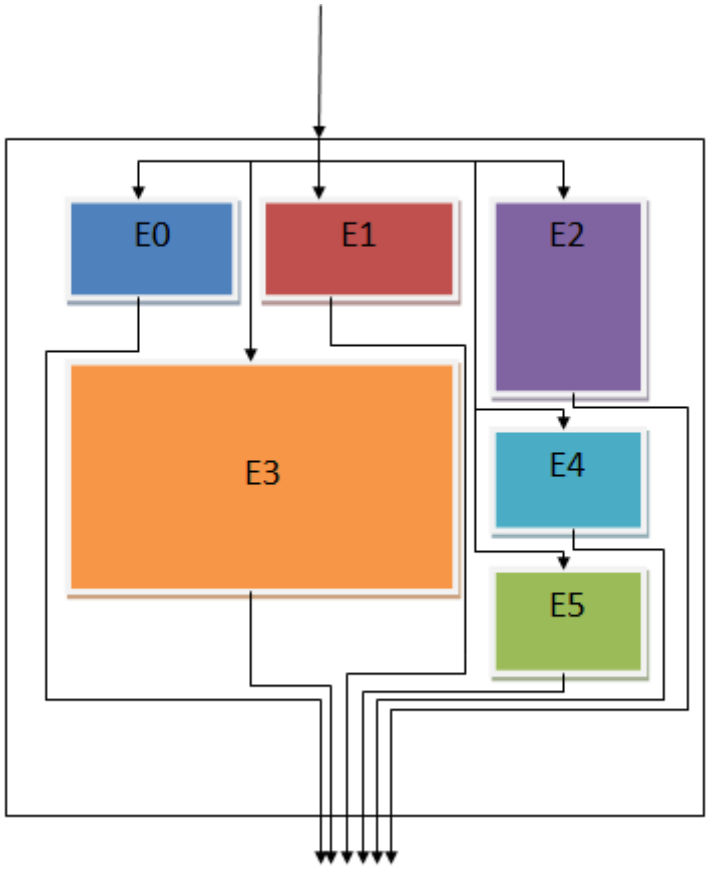


Figura 5 Posible implementación del sistema

Como se puede ver, el tamaño y la forma de los reconocedores no es homogéneo. Esto se debe a que la arquitectura de cada engine viene determinada por el algoritmo de generación del mismo, que se basa en la expresión regular a reconocer para definirla. Este algoritmo se estudiará con detenimiento más adelante.

La distribución de los engines en la FPGA está determinada por varios factores. En primer lugar, el área de la lógica reconfigurable disponible y su

topología, ya que habrá que situar en ella los distintos reconocedores con sus formas particulares. También hay que tener en cuenta el cableado necesario para conectar los engines con la entrada de datos global y a cada uno con su salida particular.

Descripción de los engines

La arquitectura de cada engine viene determinada por el algoritmo de generación del mismo, que se basa en la expresión regular a reconocer para definirla. El algoritmo se estudiará con detenimiento más adelante.

Los engines están compuestos por dos tipos de módulos:

- Módulos reconocedores de caracteres o Bloques Básicos
- Módulos implementadores de operaciones

Antes de continuar es necesario explicar lo que quiere decir que un bloque se encuentre en estado activo. Esto significa que la cadena procesada ha sido aceptada hasta ese punto. En caso de que el bloque B_i esté activo, este propagará al siguiente bloque B_{i+1} el resultado de la comparación del carácter a procesar (char_in) con $matching\_char_i$. En caso de resultar positiva, B_{i+1} pasará a estar en estado activo. En caso contrario, permanecerá inactivo. De cualquier manera, una vez transferido el estado activo de B_i a B_{i+1}, B_i volverá a estado inactivo. En caso de que B_i implemente un estado final del autómata, el resultado de la comparación será la salida del propio autómata.

Según el esquema utilizado, es posible que haya varios bloques activos simultáneamente. Eso significa que hay una cadena c_i que está siendo aceptada y que algunas subcadenas de c_i también se corresponden parcialmente con la expresión regular. Por ejemplo, para la expresión regular aa(b)*c, tanto la cadena “aab” como su subcadena “aa” llevarán a estados activos a dos bloques distintos

Bloques básicos

Su función es la de reconocer un carácter concreto, matching_char. Realiza una comparación entre el carácter a procesar, char_in, y matching_char. En caso de encontrarse en estado activo, el resultado de la comparación se propagará al siguiente bloque. Inmediatamente después, pasa a estado inactivo. Esta sería su estructura general:

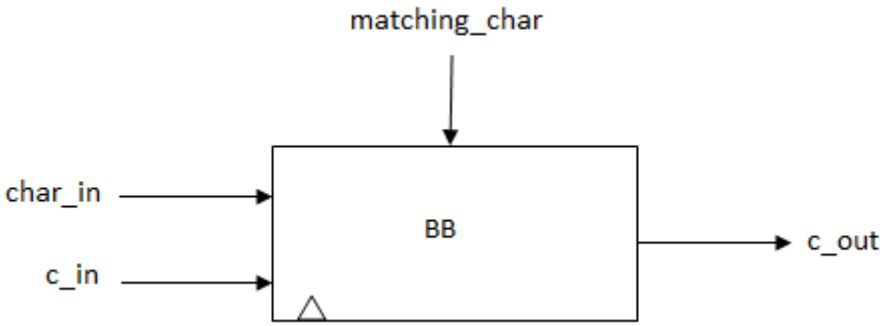


Figura 6 Esquema simple del bloque básico

Entradas del módulo:

- matching_char: es el carácter que reconoce el bloque básico. En principio, ese valor es fijo y no varía nunca en toda la vida del bloque básico, por lo que podría ser interno. Se ha decidido mantenerlo como entrada externa conservando, por tanto, la posibilidad de cambiarlo en ejecución porque esto permite acometer cierto tipo de optimizaciones que podrían ser tratadas en el futuro.

- char_in: carácter a comparar con matching_char.

- c_in: señal proveniente del módulo i-1 que indica si la cadena ha sido reconocida hasta ese punto de la ruta de datos.

- c_out: resultado de la comparación entre matching_char y char_in. Sólo se propaga si el estado de módulo es activo.

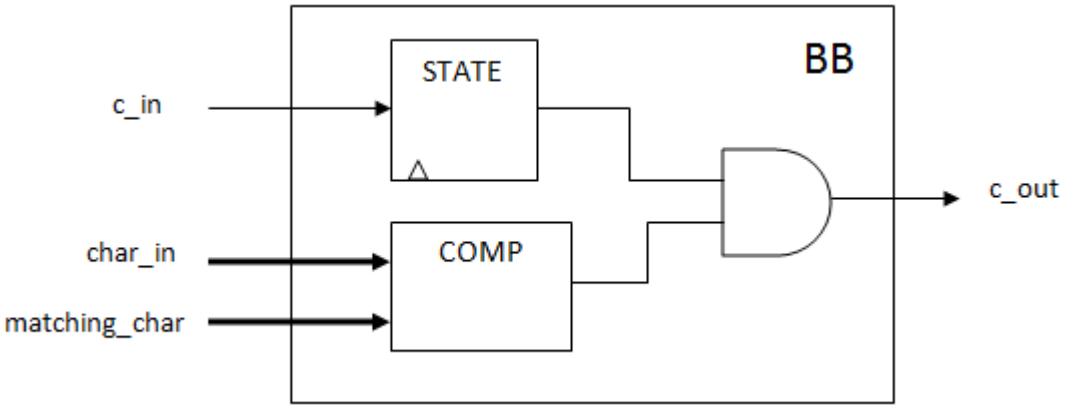


Figura 7 Esquema detallado del bloque básico

Bloques de operaciones

Estos módulos implementan el juego de operaciones soportado, a saber: *, ·, | y +, aunque esta última operación internamente traduce a+ por a·a*. Como se puede ver, son bloques con una arquitectura muy sencilla que básicamente constan de cables y dos puertas lógicas en el algún caso.

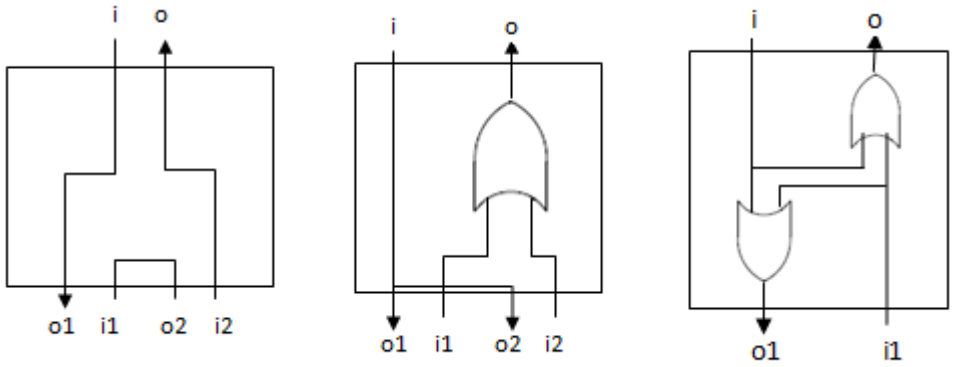


Figura 8 Esquemas de los bloques AND, OR y *, respectivamente

EJEMPLO PASO A PASO

En este apartado se va a ver con un ejemplo las distintas etapas del proceso que convierte un fichero con expresiones regulares en un sistema implementado en una FPGA que reconoce dichas expresiones regulares.

Para la realización de este ejemplo se ha creado un fichero de expresiones regulares que podría ser una versión reducida de un fichero real. Se han establecido cuatro clases distintas, cada una de ellas representando un tipo de tráfico diferente, que podrían requerir de distinto tratamiento por parte de la rede (en términos de volumen de volumen de datos o calidad de servicio, por ejemplo). Se utilizará la clase "`Mail`", que, para clarificar el ejemplo, contiene una sola expresión regular. Esta es de una complejidad superior a la habitual, pero se pretende mostrar que, aún así, el consumo de recursos de la FPGA es muy reducido

1. Fichero de expresiones regulares:

Para este ejemplo se ha creado un sencillo fichero de expresiones regulares. Se han establecido cuatro clases, cada una englobando una o más expresiones regulares. A continuación, se puede ver el fichero creado para el ejemplo:

```
FTP ftp;((port|22)+);
HTTP ((www.)|(http://));
Mail (((ba)|(be)|(bi)|(bo)|(bu))+)@((eritalk)|(ericsson))(.com);
Video ((vlc)|(streaming));
```

Por ejemplo, la clase FTP está diseñada para que se identifiquen los paquetes que contengan la cadena "`ftp`" o aquellos en los que aparezcan "`port`" o "`22`" en una o más ocasiones.

Para realizar las pruebas, se ha decidido utilizar la clase Mail, que contiene una expresión regular más compleja:

```
(((ba)|(be)|(bi)|(bo)|(bu))+)@((eritalk)|(ericsson))(.com);
```

Esta expresión regular reconocerá todas las cadenas que sigan la siguiente estructura: tienen que comenzar con al menos una aparición de cualquiera de las subcadenas "`ba`", "`be`", "`bi`", "`bo`", "`bu`". A continuación, el siguiente símbolo debe ser "`@`", seguido o bien de la subcadena "`ericsson`" o bien de "`eritalk`". Finalmente, la cadena debe terminar por "`.com`" para que sea reconocida por la expresión regular.

2. Generador de código:

Este fichero de expresiones regulares se le pasa como entrada al generador de código, así como la clase expresiones que reconocerá el sistema, Mail en este caso. Igualmente, se introduce el nombre que tendrá el fichero VHDL generado.

Para simplificar las pruebas y depuración del sistema, se ha añadido al código VHDL un banco de pruebas que simula el flujo de caracteres de entrada con la frecuencia y el retardo inicial del sistema deseados.

En este ejemplo, el flujo de entrada será la cadena "`aaababu@eritalk.com`", en la que debería reconocerse el segmento "`babu@eritalk.com`". Ya que el objetivo no es probar la máxima frecuencia a la que puede trabajar el sistema, los caracteres individuales se inyectarán cada 10ns, para alcanzar un rendimiento de 800 Mbps. Para que la señal de los caracteres se encuentre estable con la llegada del flanco de reloj, se introduce un retardo arbitrario de 2ns.

El código generado puede verse en el Apéndice.

3. Verificación del sistema

Con el objetivo de verificar que el código generado funciona, se ha utilizado el programa de simulación y depuración ModelSim. La herramienta de Xilinx, Xilinx ISE 11, también permite realizar este tipo de tareas, pero por familiaridad y sencillez de uso, se ha optado por utilizar ModelSim.

Se ha establecido la frecuencia de reloj a 10 MHz para que coincida con aquella con la que el banco de pruebas inyecta los caracteres.

Este es el resultado de la simulación:

Tras un retardo inicial de 10ns se estabilizan las señales. Los tres primeros caracteres son "`aaa`", que no se corresponden con la expresión regular reconocida. El siguiente carácter es "`b`", que sí que podría hacer correspondencia. Internamente, un módulo que reconoce "`b`" pasa a estado activo. En el siguiente ciclo, llega el carácter "`a`".

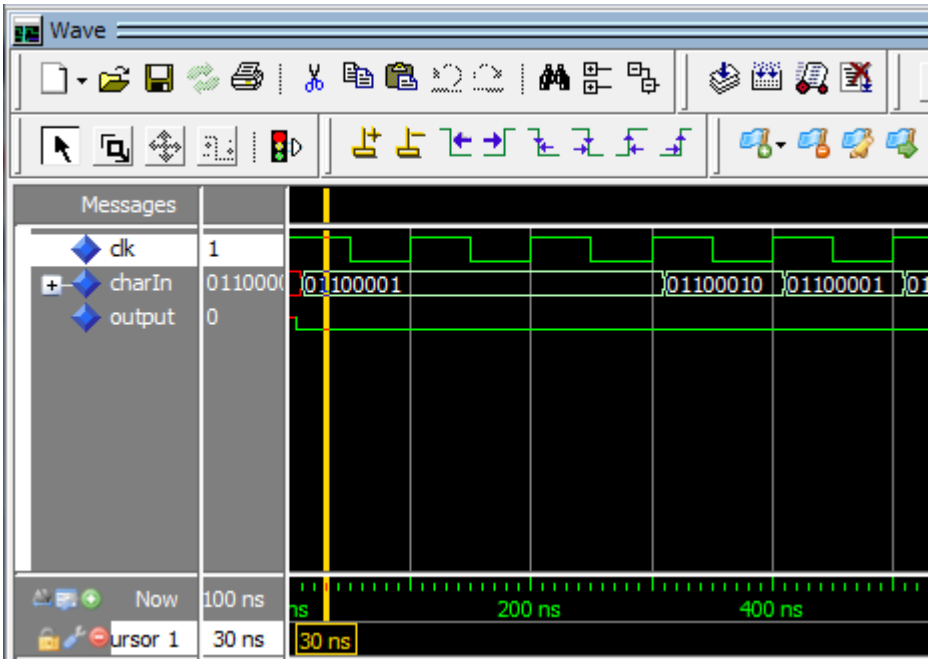


Figura 9 Primeros 5 ciclos de la verificación

Los caracteres que van llegando, “`bu@er`”, siguen haciendo avanzar al sistema en el reconocimiento de la expresión regular:

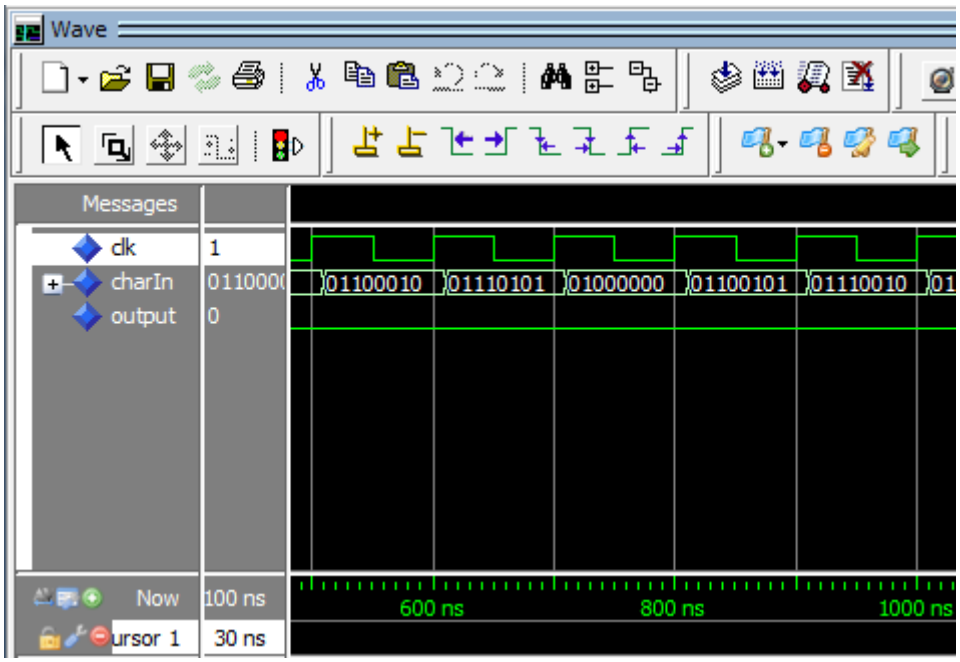


Figura 10 Ciclos del 6 al 10 de la verificación

En la Figura 11, los caracteres de entrada son: “`italk`”. La salida continúa a 0 porque no ha habido una correspondencia completa.

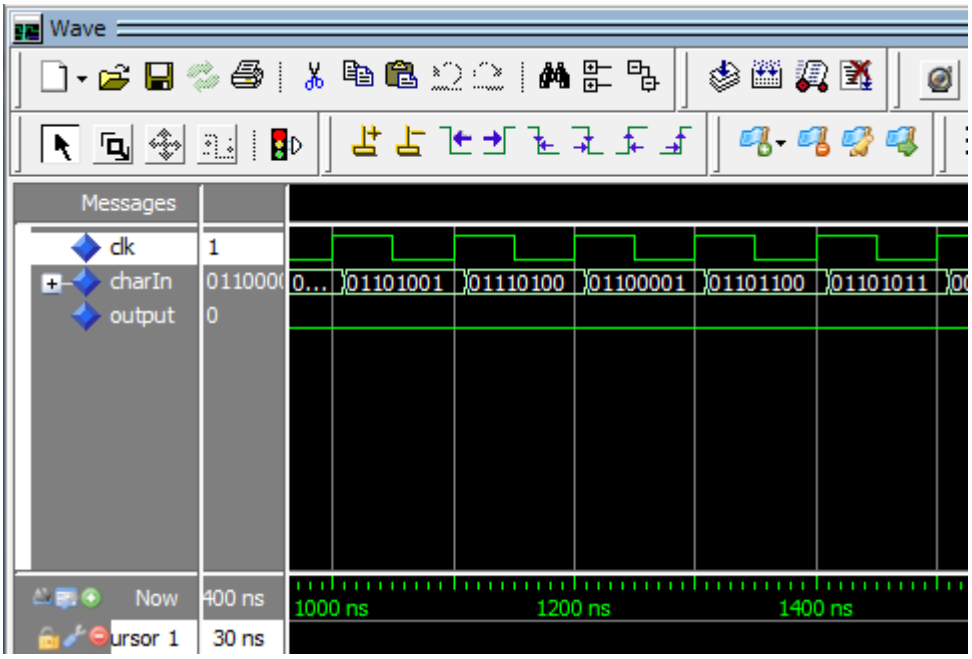


Figura 11 Ciclos del 11 al 15 de la verificación

Finalmente, con la llegada de los caracteres “.com”, se produce una correspondencia completa y la salida se pone a 1, indicando que se ha producido una correspondencia. En el siguiente ciclo vuelve a 0.

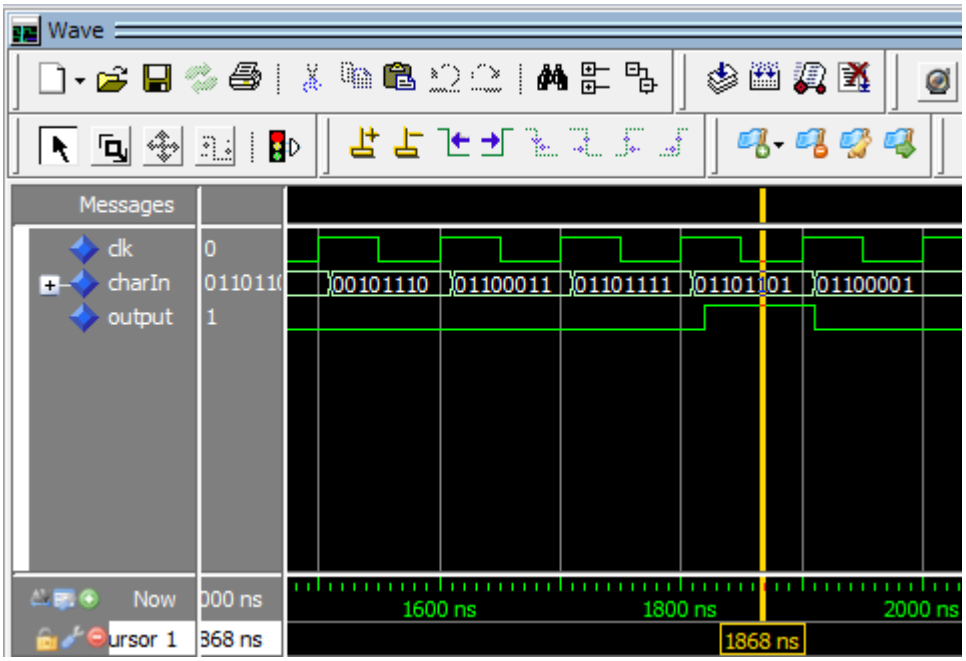


Figura 12 Ciclos del 16 al 20 de la verificación

4. Implementación con Xilinx

El último paso es la implementación en la FPGA. Dada la dificultad de depurar el diseño en una FPGA real, todas las pruebas, todas las pruebas han sido realizadas sobre la herramienta de Xilinx, que estima con una exactitud suficiente la cantidad de recursos necesarios para cada implementación. La plataforma de referencia seleccionada ha sido una Virtex 5 XC5VFX30T.

En las Tablas 1 y 2 pueden verse los datos que proporciona Xilinx con un resumen de los recursos utilizados para la síntesis del sistema. Como se puede apreciar, estos suponen un porcentaje muy pequeño del total de los recursos disponibles.

Physical Resource Estimates

Site Type	Available	Required	% Util
LUT	20480	26	1
FD_LD	20480	18	1
SLICEL	3600	6	1
SLICEM	1520	3	1

Tabla 1 Tasas de ocupación de los recursos

Primitive Statistics

Primitive type	Count
LUT	26
FD_LD	18
MUXFX	1
CLK	1
IO	9
OTHERS	1

Tabla 2 Conteo de los recursos empleados

A continuación, en la Figura 13, se muestra la representación de la FPGA. Las líneas verdes representan el emplazamiento del reconocedor que ha hecho el software. Ocupa un área muy pequeña, aún a pesar de que, posiblemente, el cableado sintetizado no sea el más compacto sino el más sencillo o rápido.

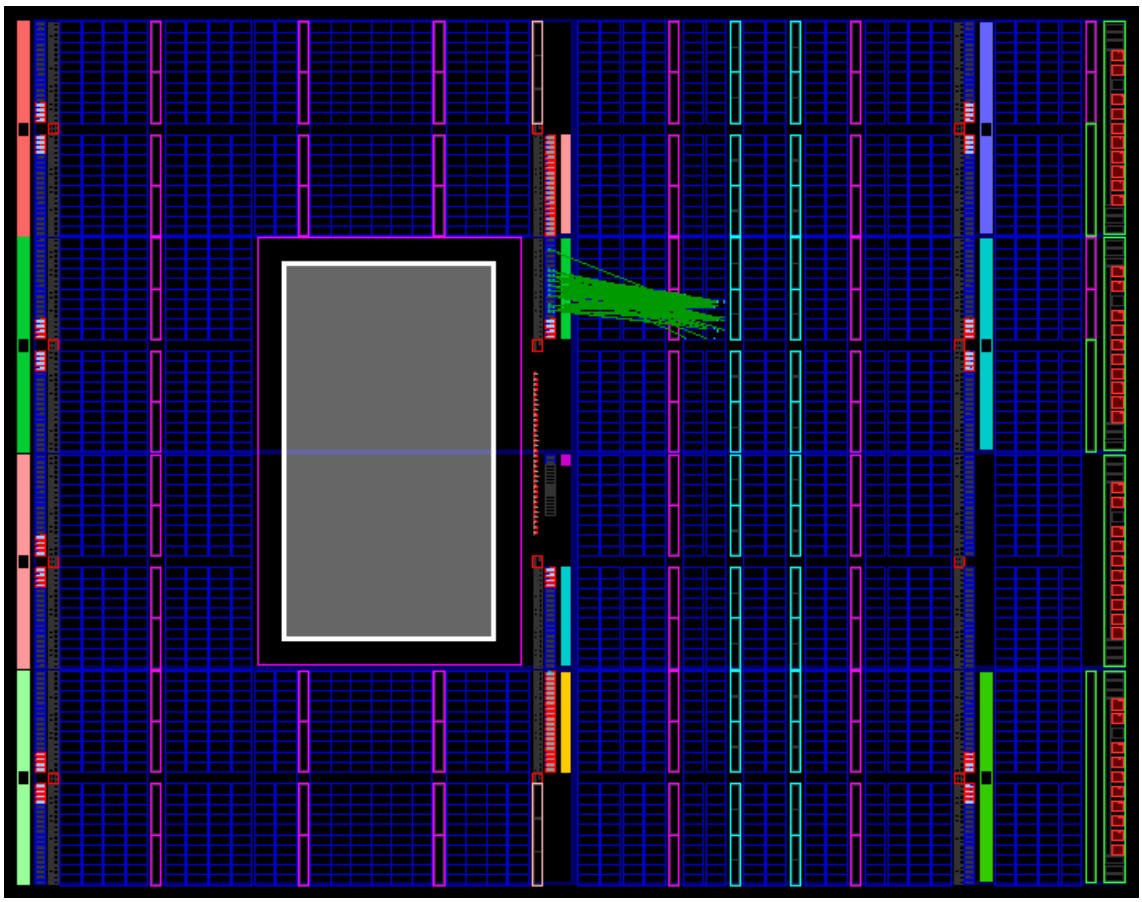


Figura 13 Esquema de la ocupación de los recursos de la FPGA

La herramienta Xilinx ISE 11 permite comprobar la interpretación HW realizada del código VHDL comprobando los esquemáticos generados. En este caso, en la Figura 14 se muestra el esquemático generado para el reconocedor

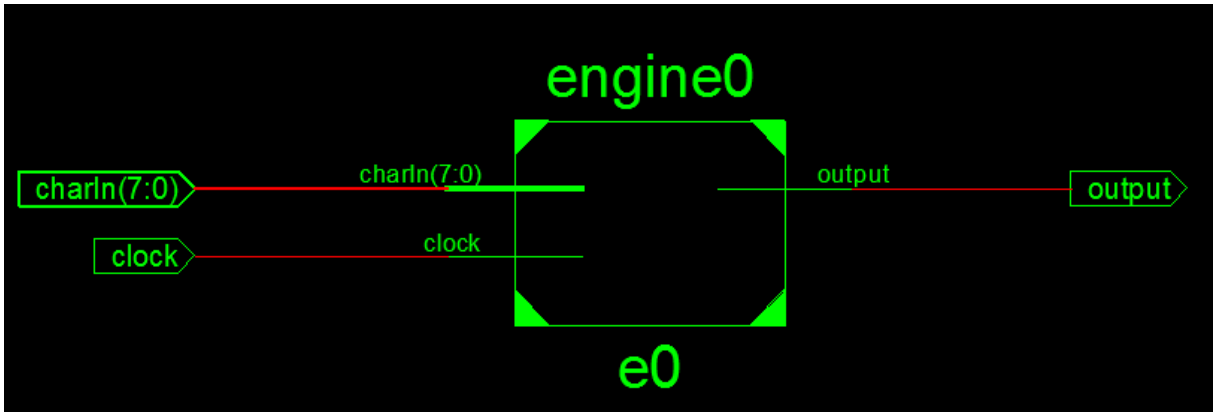


Figura 14 Esquemático generado para el reconocedor

Si se desciende más en la jerarquía, se puede ver la estructura interna del sistema implementado:

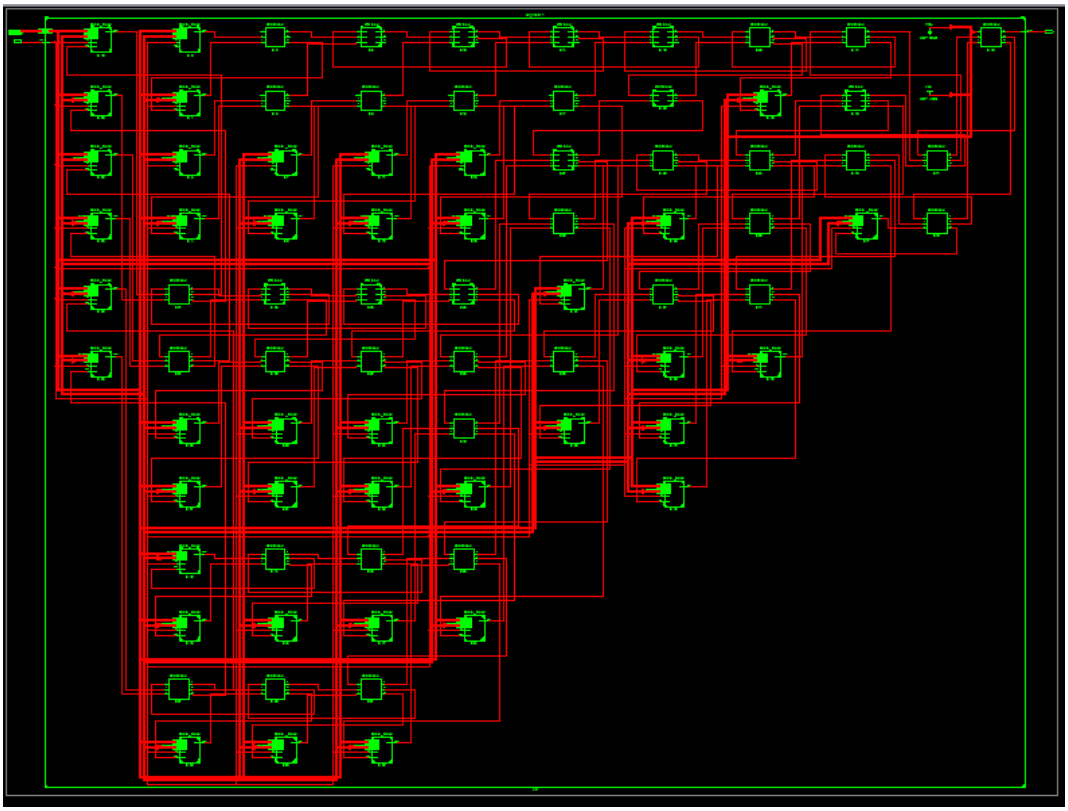


Figura 15 Vista general de la implementación

Si se observa con detenimiento, se pueden apreciar los componentes que forman el sistema: los bloques básicos, los de operaciones y el cableado.

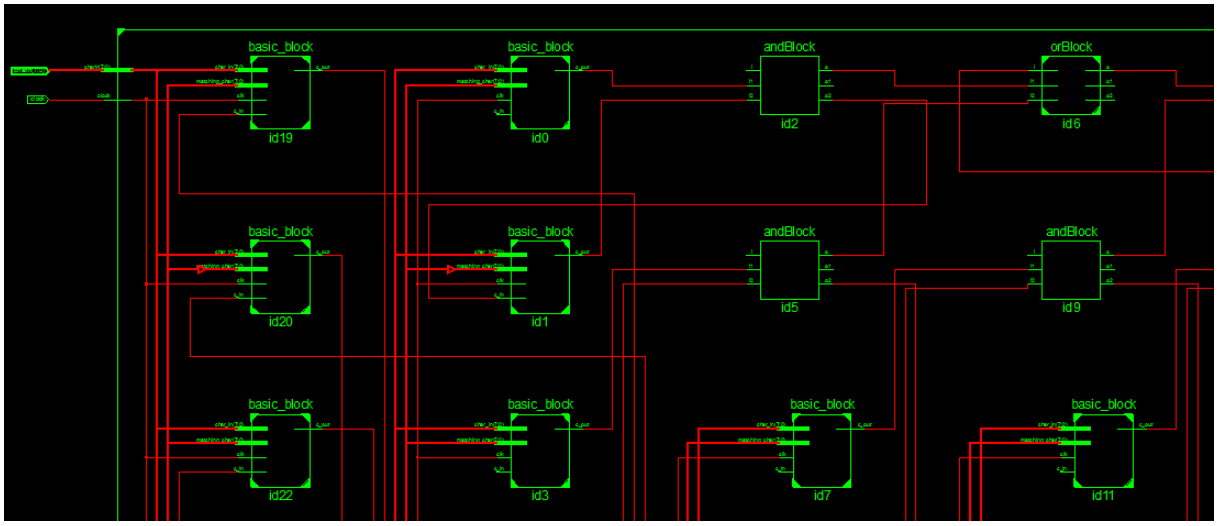


Figura 16 Vista más detallada de un sector

También es posible observar la estructura interna de los módulos. En la Figura 17, el bloque básico en detalle.

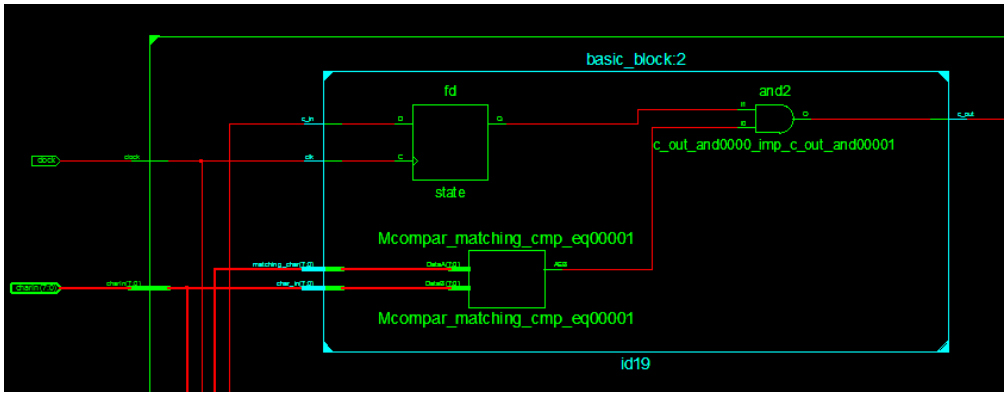


Figura 17 Estructura interna del bloque básico

CAPÍTULO 4: RESULTADOS EXPERIMENTALES

ESTUDIO DEL USO DE LOS RECURSOS DE LA FPGA

En un dispositivo comercial, son muchas las expresiones regulares que tienen que ser reconocidas simultáneamente. Para probar el impacto que un reconocedor con múltiples engines (uno por cada expresión regular) tiene sobre los recursos y área de la FPGA, se han diseñado unas pruebas que lo evalúan.

Por problemas de confidencialidad, ha sido imposible obtener un fichero de expresiones regulares real, con las expresiones que actualmente se utilizan en la industria, por lo que se ha tenido que crear uno ficticio. Este contiene una sola clase con 50 expresiones regulares, de una complejidad menor que la de la estudiada en el ejemplo paso a paso, pero más acorde con las necesidades en entornos reales.

Se comenzará, por tanto, con un reconocedor de 50 engines y se irá incrementando paulatinamente el número de estos. Dado que la arquitectura del reconocedor depende de forma fundamental de las expresiones regulares que este reconoce, para realizar las pruebas se ha tomado un conjunto base de 50 expresiones que ha sido replicado tantas veces como fuese necesario para crear los archivos de 100, 200, 500… etc. expresiones regulares. De esta forma, se mantiene la homogeneidad de las pruebas. Se continuará incrementando el número de expresiones regulares hasta que se alcance el límite impuesto por los recursos hardware, tanto de la FPGA como de la máquina sobre la que corre el reconocedor, esto es, 5000 expresiones.

La FPGA empleada para las pruebas es una Virtex 5 XC5VFX30T, un chip de media gama pero que, aún así, es capaz de reconocer un gran número de expresiones en paralelo.

50 Expresiones regulares

Llama la atención el reducido uso de los recursos, ya que la mayoría de los componentes de la FPGA tienen una tasa de utilización cercana al 1%. En cambio, destaca enormemente la utilización de los pares LUT-FF, con un porcentaje del 75%. La razón de este fenómeno reside en que la arquitectura de los bloques básicos, los que reconocen un carácter, está compuesta por un biestable acompañado de una lógica muy sencilla, a lo que se ajusta muy bien el par LUT-FF.

Device Utilization Summary			
Slice Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	68	20,480	1%
Number used as Flip Flops	68		
Number of Slice LUTs	90	20,480	1%
Number used as logic	90	20,480	1%
Number using O6 output only	90		
Number of occupied Slices	57	5,120	1%
Number of occupied SLICEMs	0	1,520	0%
Number of LUT Flip Flop pairs used	90		
Number with an unused Flip Flop	22	90	24%
Number with an unused LUT	0	90	0%
Number of fully used LUT-FF pairs	68	90	75%
Number of unique control sets	8		
Number of slice register sites lost to control set restrictions	20	20,480	1%
Number of bonded IOBs	10	360	2%
Number of BUFG/BUFGCTRLs	1	32	3%
Number used as BUFGs	1		
Average Fanout of Non-Clock Nets	4.66		

Tabla 3 Uso de los recursos para el reconocedor de 50 expresiones regulares

En cuanto a la utilización del área, se observa que sigue suponiendo una fracción relativamente pequeña de la FPGA.

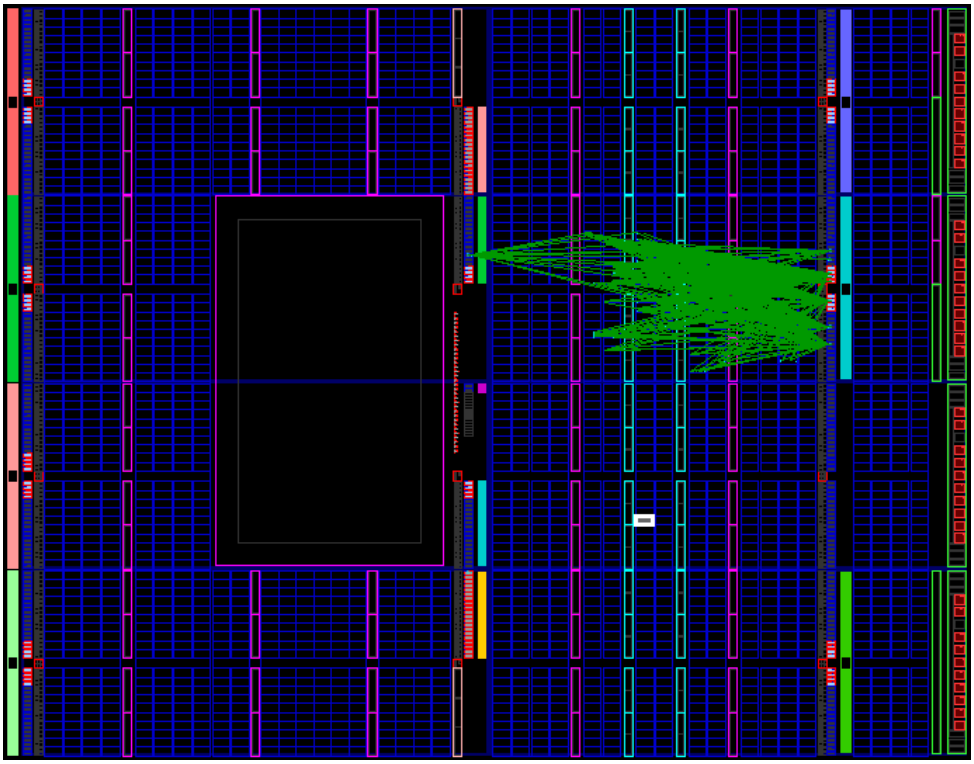


Figura 18 Esquema de la utilización de la FPGA para 50 expresiones regulares

100 Expresiones regulares

Si se observan las tasas de ocupación de los recursos se aprecia que en el reconocedor de 100 engines se mantienen en niveles similares a los del de 50 engines. Por otra parte, el número de pares LUT-FF ha crecido respecto a la versión anterior. Esto se explica porque la herramienta de síntesis detecta que va a ser una estructura muy apropiada para la arquitectura a implementar y configura la FPGA para que disponga de un mayor número.

Device Utilization Summary			
Slice Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	84	20,480	1%
Number used as Flip Flops	84		
Number of Slice LUTs	106	20,480	1%
Number used as logic	106	20,480	1%
Number using O6 output only	106		
Number of occupied Slices	58	5,120	1%
Number of occupied SLICEMs	0	1,520	0%
Number of LUT Flip Flop pairs used	106		
Number with an unused Flip Flop	22	106	20%
Number with an unused LUT	0	106	0%
Number of fully used LUT-FF pairs	84	106	79%
Number of unique control sets	6		
Number of slice register sites lost to control set restrictions	8	20,480	1%
Number of bonded IOBs	10	360	2%
Number of BUFG/BUFGCTRLs	1	32	3%
Number used as BUFGs	1		
Average Fanout of Non-Clock Nets	4.42		

Tabla 4 Uso de los recursos para el reconocedor de 100 expresiones regulares

En lo que a la ocupación del área se refiere, a simple vista no se aprecian grandes cambios, salvo, tal vez, una mayor densidad de elementos en un área ligeramente mayor.

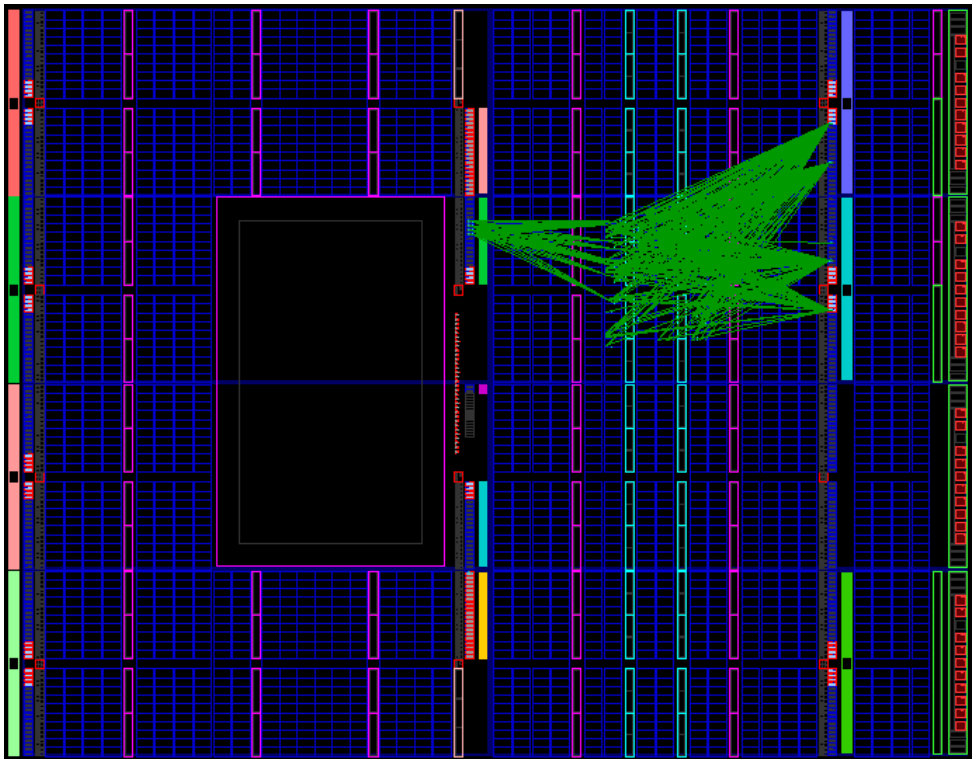


Figura 19 Esquema de la utilización de la FPGA para 100 expresiones regulares

200 Expresiones regulares

Se mantiene la tendencia en el uso de los recursos: el porcentaje de pares LUT-FF crece ligeramente a la vez que aumenta su número (149 respecto a los 106 que había para la mitad de engines)

Device Utilization Summary			
Slice Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	123	20,480	1%
Number used as Flip Flops	123		
Number of Slice LUTs	149	20,480	1%
Number used as logic	149	20,480	1%
Number using O6 output only	149		
Number of occupied Slices	89	5,120	1%
Number of occupied SLICEMs	0	1,520	0%
Number of LUT Flip Flop pairs used	149		
Number with an unused Flip Flop	26	149	17%
Number with an unused LUT	0	149	0%
Number of fully used LUT-FF pairs	123	149	82%
Number of unique control sets	7		
Number of slice register sites lost to control set restrictions	17	20,480	1%
Number of bonded IOBs	10	360	2%
Number of BUFG/BUFGCTRLs	1	32	3%
Number used as BUFGs	1		
Average Fanout of Non-Clock Nets	4.52		

Tabla 5 Uso de los recursos para el reconocedor de 200 expresiones regulares

En cuanto al área, sí que se aprecia un aumento significativo de los sectores utilizados, aparte de una redistribución del espacio ocupado.

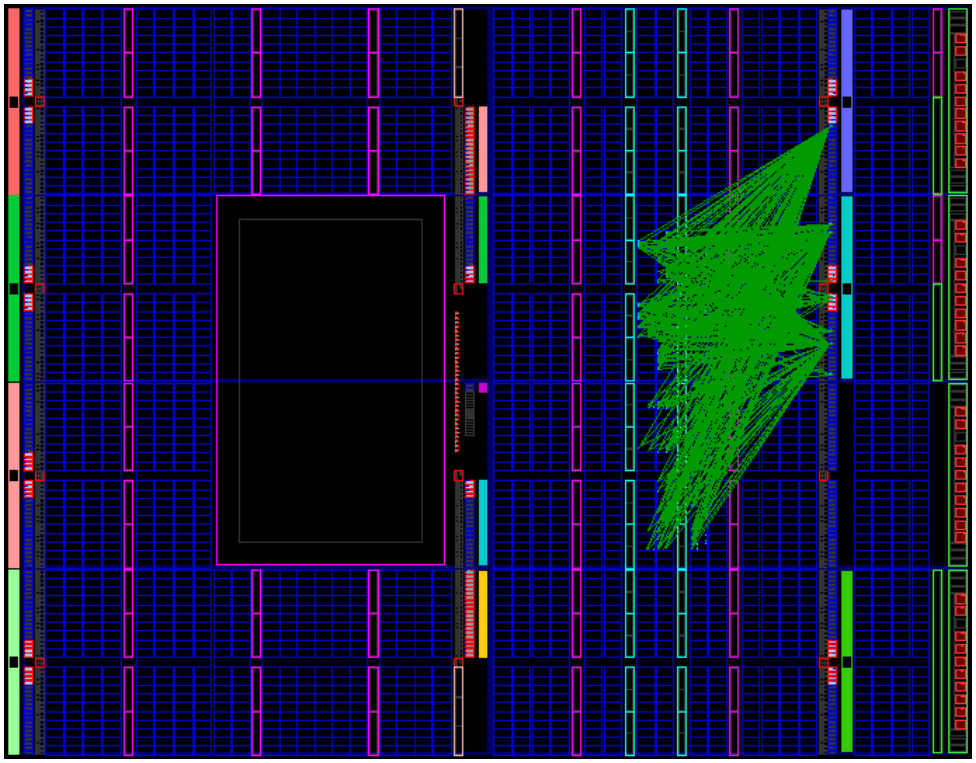


Figura 20 Esquema de la utilización de la FPGA para 200 expresiones regulares

500 Expresiones regulares

El número de pares LUT-FF asciende hasta 258, aumentando el porcentaje de utilización hasta un 87%. El resto de parámetros se mantienen en torno al 1%, aunque se sigue incrementando su número. Por ejemplo, el número de registros slice ha pasado desde los 68 de la versión de 50 engines hasta los 227 de la de 500.

Device Utilization Summary			
Slice Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	227	20,480	1%
Number used as Flip Flops	227		
Number of Slice LUTs	258	20,480	1%
Number used as logic	258	20,480	1%
Number using O6 output only	258		
Number of occupied Slices	94	5,120	1%
Number of occupied SLICEMs	0	1,520	0%
Number of LUT Flip Flop pairs used	258		
Number with an unused Flip Flop	31	258	12%
Number with an unused LUT	0	258	0%
Number of fully used LUT-FF pairs	227	258	87%
Number of unique control sets	6		
Number of slice register sites lost to control set restrictions	13	20,480	1%
Number of bonded IOBs	10	360	2%
Number of BUFG/BUFGCTRLs	1	32	3%
Number used as BUFGs	1		
Average Fanout of Non-Clock Nets	4.32		

Tabla 6 Uso de los recursos para el reconocedor de 500 expresiones regulares

El porcentaje de área de la FPGA utilizado comienza a ser importante, ocupando ya buena parte de la misma.

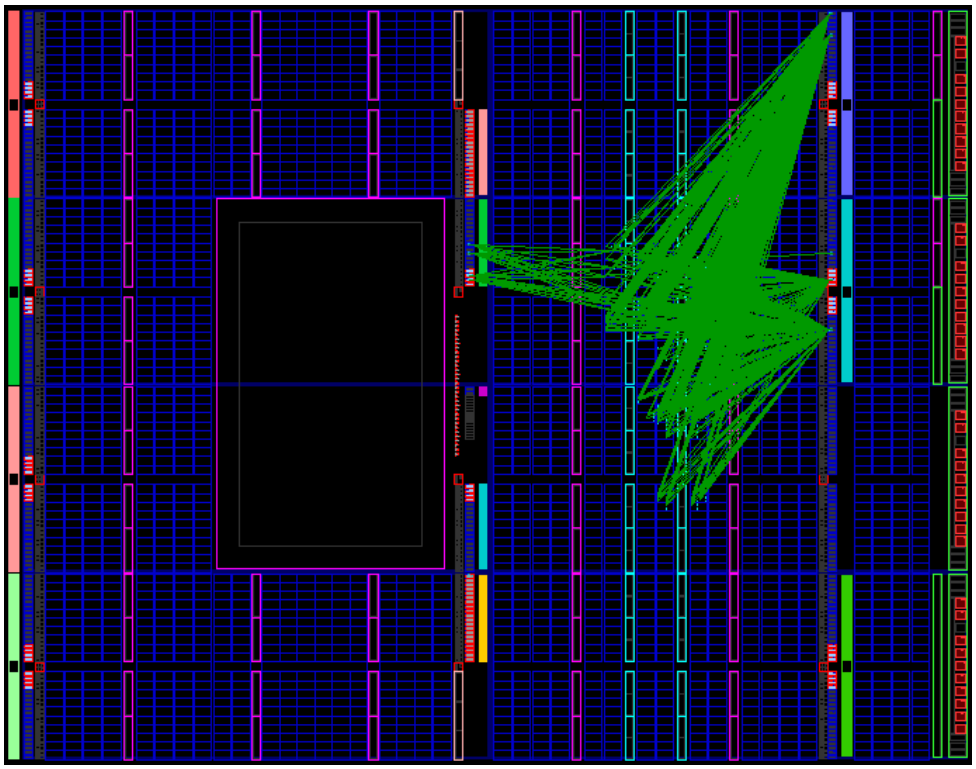


Figura 21 Esquema de la utilización de la FPGA para 500 expresiones regulares

1000 Expresiones regulares

El porcentaje de ocupación de los pares LUT-FF supera ya el 90%. El resto de los parámetros continúan con su ascenso mucho más moderado y, aunque su número sigue creciendo paulatinamente, la tasa de ocupación tan sólo ha crecido en un punto respecto a los números del reconocedor de 50 expresiones regulares.

Device Utilization Summary			
Slice Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	398	20,480	1%
Number used as Flip Flops	398		
Number of Slice LUTs	435	20,480	2%
Number used as logic	435	20,480	2%
Number using O6 output only	435		
Number of occupied Slices	144	5,120	2%
Number of occupied SLICEMs	0	1,520	0%
Number of LUT Flip Flop pairs used	435		
Number with an unused Flip Flop	37	435	8%
Number with an unused LUT	0	435	0%
Number of fully used LUT-FF pairs	398	435	91%
Number of unique control sets	6		
Number of slice register sites lost to control set restrictions	14	20,480	1%
Number of bonded IOBs	10	360	2%
Number of BUFG/BUFGCTRLs	1	32	3%
Number used as BUFGs	1		
Average Fanout of Non-Clock Nets	4.35		

Tabla 7 Uso de los recursos para el reconocedor de 1000 expresiones regulares

A simple vista, la distribución del área sigue sin sufrir cambios reseñables.

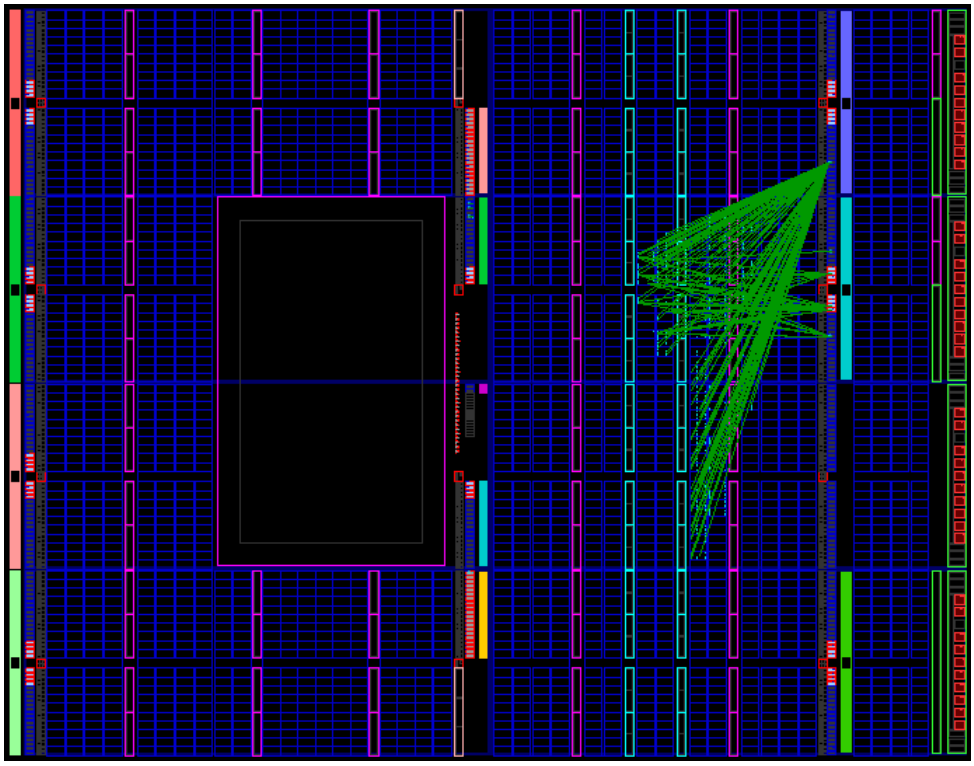


Figura 22 Esquema de la utilización de la FPGA para 1000 expresiones regulares

2000 Expresiones regulares

Al llegar a los 2000 engines, la FPGA comienza a saturarse. Estructuras como los registros slice aumentan el porcentaje de ocupación a un ritmo mayor que la tendencia que habían mostrado en las pruebas anteriores. Los pares LUT-FF alcanzan el 94% de ocupación (781 elementos).

Device Utilization Summary			
Slice Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	739	20,480	3%
Number used as Flip Flops	739		
Number of Slice LUTs	781	20,480	3%
Number used as logic	781	20,480	3%
Number using O6 output only	781		
Number of occupied Slices	215	5,120	4%
Number of occupied SLICEMs	0	1,520	0%
Number of LUT Flip Flop pairs used	781		
Number with an unused Flip Flop	42	781	5%
Number with an unused LUT	0	781	0%
Number of fully used LUT-FF pairs	739	781	94%
Number of unique control sets	6		
Number of slice register sites lost to control set restrictions	13	20,480	1%
Number of bonded IOBs	10	360	2%
Number of BUFG/BUFGCTRLs	1	32	3%
Number used as BUFGs	1		
Average Fanout of Non-Clock Nets	4.42		

Tabla 8 Uso de los recursos para el reconocedor de 2000 expresiones regulares

La distribución del área mantiene el patrón que había mantenido hasta el momento.

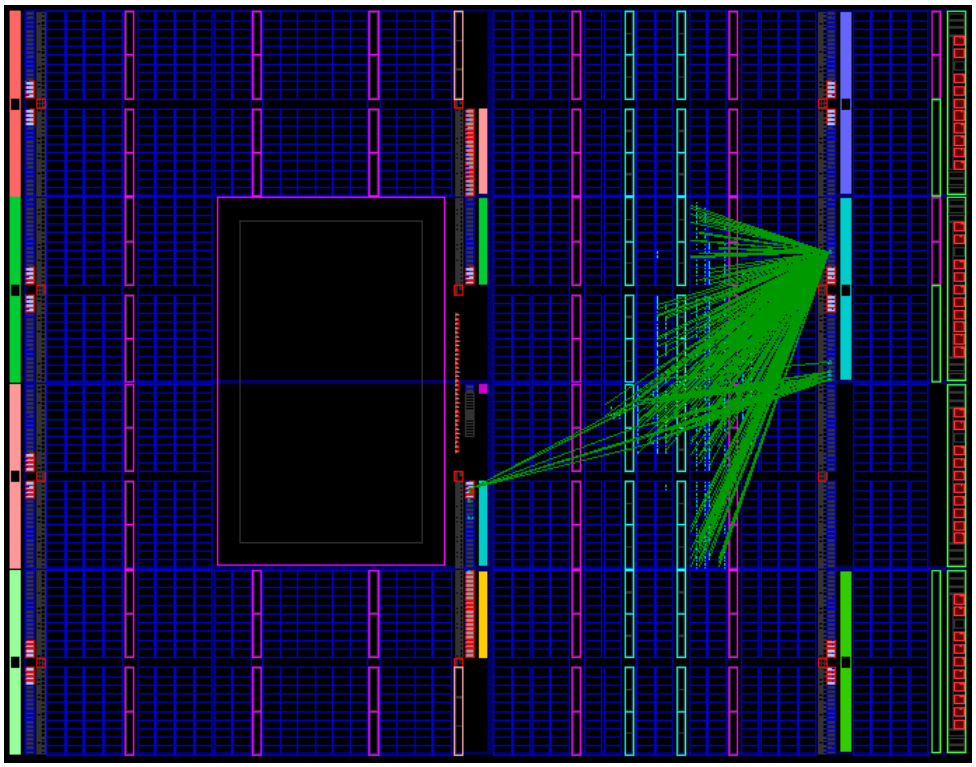


Figura 23 Esquema de la utilización de la FPGA para 2000 expresiones regulares

5000 Expresiones regulares

Se ve claramente que la FPGA ha llegado a un punto de saturación. Pese a multiplicar prácticamente por 2,5 el número de pares LUT-FF, el porcentaje de ocupación es del 95%. En cuanto al resto de elementos, estos han continuado con el notable incremento en la tasa de utilización que ya se iba apreciando en la prueba anterior

Device Utilization Summary			
Slice Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	1,759	20,480	8%
Number used as Flip Flops	1,759		
Number of Slice LUTs	1,834	20,480	8%
Number used as logic	1,833	20,480	8%
Number using O6 output only	1,833		
Number used as exclusive route-thru	1		
Number of route-thrus	1	40,960	1%
Number using O6 output only	1		
Number of occupied Slices	558	5,120	10%
Number of occupied SLICEMs	0	1,520	0%
Number of LUT Flip Flop pairs used	1,834		
Number with an unused Flip Flop	75	1,834	4%
Number with an unused LUT	0	1,834	0%
Number of fully used LUT-FF pairs	1,759	1,834	95%
Number of unique control sets	6		
Number of slice register sites lost to control set restrictions	13	20,480	1%
Number of bonded IOBs	10	360	2%
Number of BUFG/BUFGCTRLs	1	32	3%
Number used as BUFGs	1		
Average Fanout of Non-Clock Nets	4.46		

Tabla 9 Uso de los recursos para el reconocedor de 5000 expresiones regulares

En cuanto a la distribución del área, la tendencia continúa siendo la misma, habiéndose ocupado ya una gran parte de la FPGA. Debido a la densidad de las conexiones, la herramienta de síntesis ha dejado de representarlas.

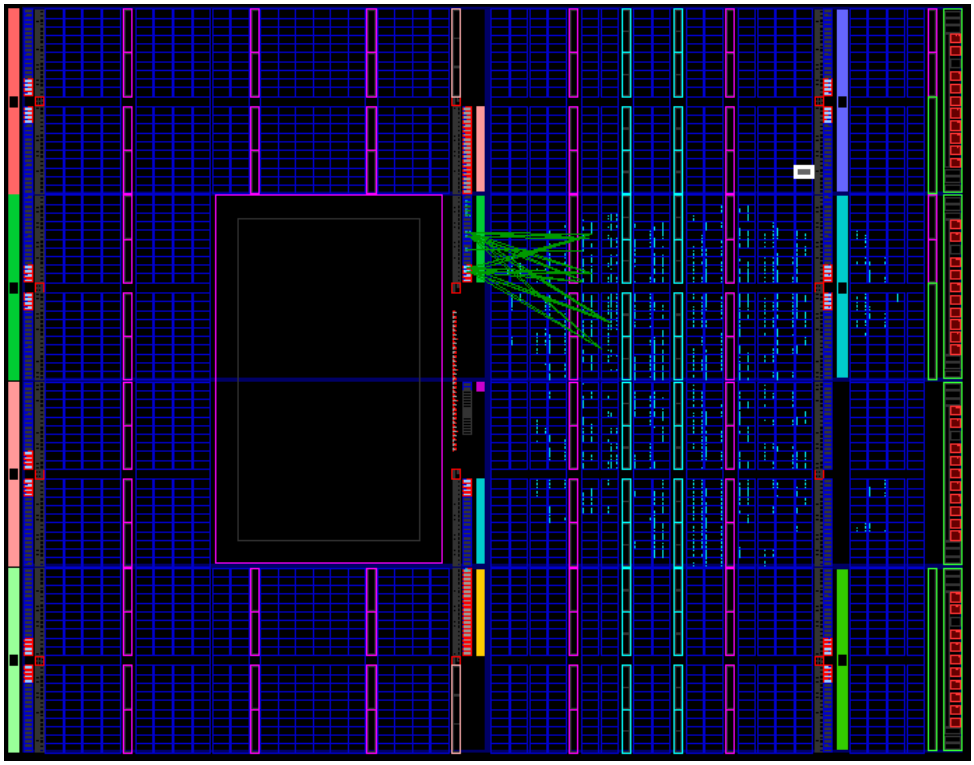


Figura 24 Esquema de la utilización de la FPGA para 5000 expresiones regulares

EVALUACIÓN DE LA VALIDEZ DE LA SOLUCIÓN PROPUESTA PARA UN ENTORNO REAL

Tras estudiar el impacto de los reconocedores de distintos tamaños en una FPGA, se han realizado otro tipo de pruebas orientadas a evaluar la viabilidad de la solución propuesta en entornos reales para tareas de Deep Packet Inspection, estudiando aspectos tales como el máximo número de expresiones regulares que podrían reconocerse en paralelo o el mayor ancho de banda que podría conseguirse.

Características de las distintas familias

En primer lugar, se han comparado varias familias de Virtex, la gama alta de las FPGAs de Xilinx. Las familias de Xilinx se diferencian entre sí en la tecnología de fabricación utilizada, la arquitectura hardware de las celdas básicas y en los elementos hardware extra que poseen. Se ha estudiado el máximo número de engines que podrían estar funcionando en paralelo, así como el ancho de banda teórico que podría alcanzarse. Así mismo, el número de celdas informa del tamaño de la FPGA y el número de pines I/O de la conectividad máxima posible.

Dispositivo	Número de celdas	Pines I/O	Número de engines en paralelo	Ancho de banda
Virtex 4 - XC4VLX100	110592	960	2700	4Gbps
Virtex 4 - XC4VLX160	152064	960	3700	4Gbps
Virtex 4 - XC4VLX200	200448	960	4900	4Gbps
Virtex 5 - XC5VTX150T	92800	680	3500	6Gbps
Virtex 5 - XC5VTX240T	149760	680	5700	6Gbps
Virtex 6 - XC6VC195XT	124800	600	4600	7Gbps
Virtex 6 - XC6VC240XT	150000	600	5500	7Gbps

Tabla 10 Comparativa entre las distintas familias de Virtex

La Virtex 6 prácticamente dobla el ancho de banda respecto a la Virtex 4, aunque el número de expresiones regulares reconocidas simultáneamente es bastante similar, al menos entre los chips tope de gama.

Una de las características más destacables del hardware reconfigurable es la flexibilidad, que permite obtener otras soluciones hardware centradas en aspectos concretos de rendimiento de nuestro sistema. De esta manera si la prioridad del sistema es reconocer expresiones lo más rápido posible, pero se puede reducir el número de expresiones diferentes a reconocer, se puede duplicar el rendimiento obtenido como muestra la Tabla 11, utilizando una única FPGA. Para todos los ejemplos de aquí en adelante vamos a utilizar la FPGA de Xilinx Virtex 5 - XC5VTX240T, por ser actualmente la de mayor tamaño del mercado. Los resultados obtenidos son extrapolables al resto de FPGAs presentes en la Tabla 10.

Optimización del rendimiento

La idea para aumentar el rendimiento del sistema, cuyos resultados se muestran en la Tabla 11, consiste en replicar los módulos que reconocen un cierto número de expresiones regulares en la FPGA e introducir cada cadena de caracteres nueva a reconocer en un módulo diferente.

Dispositivo	Número de reconocimientos distintos en paralelo	Ancho de banda
Virtex 5 - XC5VTX240T	5700	6Gbps
Virtex 5 - XC5VTX240T	2800	12Gbps
Virtex 5 - XC5VTX240T	1400	24Gbps

Tabla 11 Máximo rendimiento

Maximización del número de expresiones regulares reconocidas

Si, por el contrario, el sistema está orientado al reconocimiento del mayor número posible de expresiones regulares, es posible aprovechar la capacidad de reconfiguración parcial de la FPGA para ir cargando solapadamente nuevas configuraciones para reconocer nuevas expresiones. Los datos presentes en la Tabla 10 están calculados para cadenas de al menos 16 caracteres. Si este tamaño tuviera que ser menor para poder solapar de una manera efectiva la reconfiguración parcial con la ejecución, los tamaños de reconocimientos distintos en paralelo deberían ser otros, afectando por lo tanto al rendimiento obtenido.

Dispositivo	Número de reconocimientos distintos en paralelo	Ancho de banda
Virtex 5 - XC5VTX240T	2800	6Gbps
Virtex 5 - XC5VTX240T	5700	3Gbps
Virtex 5 - XC5VTX240T	8600	1.5Gbps
Virtex 5 - XC5VTX240T	11500	0.75Gbps

Tabla 12 Mayor número de reconocimientos en paralelo con un solo dispositivo

Sin embargo existe otra opción para reconocer un mayor número de expresiones regulares sin afectar al rendimiento total del sistema. Dicha solución consiste en reconocer las expresiones utilizando varias FPGAs trabajando en paralelo. En el ejemplo de la Tabla 13 se utilizan distintas configuraciones de reconocimiento con hasta 4 FPGAs.

Dispositivo	Número de dispositivos	Número de reconocimientos distintos en paralelo	Ancho de banda
Virtex 5 - XC5VTX240T	2	5700	12Gbps
Virtex 5 - XC5VTX240T	4	5700	24Gbps
Virtex 5 - XC5VTX240T	2	11500	6Gbps
Virtex 5 - XC5VTX240T	4	11500	12Gbps
Virtex 5 - XC5VTX240T	4	23000	6Gbps

Tabla 13 Mayor número de reconocimientos en paralelo con varios dispositivo

CAPÍTULO 5: TRABAJO FUTURO

La inclusión de un generador de código VHDL escrito en un lenguaje de alto nivel como es Java, abre la puerta a multitud de optimizaciones. La potencia y flexibilidad de este tipo de lenguajes permitiría hacer de forma sencilla cosas tales como actualizaciones del software o una interfaz gráfica. Aunque, tal vez, la posibilidad más atractiva para una empresa de Telecomunicaciones sea la de la gestión remota del sistema. Sería factible realizar muchas tareas sin tener que desplazar a un técnico hasta el cliente, con las complicaciones y coste que esto supone. A continuación, algunas opciones que resultarían interesantes:

Actualización de módulos: un cambio en el diseño de un módulo, ya sea porque se han realizado optimizaciones o porque se han detectado fallos en el diseño, podría trasladarse de forma inmediata al sistema en cuestión. De hecho, esta y otras actualizaciones podrían hacerse en caliente y con un coste en rendimiento prácticamente nulo. De igual manera, resultaría totalmente transparente para el cliente.

Inclusión de nuevas funcionalidades: continuando con la idea de la actualización de módulos, sería posible la creación de una funcionalidad totalmente nueva, con la que el sistema no contaba al principio, y su integración inmediata.

Librería de módulos: otra de las posibilidades consistiría en la creación de una librería de módulos, de manera que se pudiesen escoger aquellos que se adaptasen a las necesidades específicas de cada cliente, potenciando en unos casos la productividad y en otros el rendimiento, por ejemplo.

El esquema resultaría especialmente interesante para una empresa de Telecomunicaciones porque el coste de desarrollo de los módulos resultaría muy bajo, ya que las FPGAs resultan unas plataformas ideales para este cometido y,

pese a suponer una implementación puramente hardware, no acarrean los costes derivados del prototipado de ASICs.

Independientemente de aquellas nacidas a partir de la utilidad del generador de código, es posible acometer otro tipo de optimizaciones que habilitarían nuevas características muy valiosas:

Reconfiguración dinámica: otra de las funcionalidades que permiten las FPGAs es la reconfiguración dinámica. Esta característica posibilita que la FPGA sea reconfigurada cen caliente, ya sea completamente o sólo en algunos sectores. Una de las mejoras que podría realizarse haciendo uso de esta cualidad, es que el sistema se fuese adaptando dinámicamente al tipo de tráfico que está procesando. Por ejemplo: la FPGA podría comenzar el procesamiento con un conjunto muy heterogéneo de engines que reconociesen una amplia gama de expresiones regulares. A medida que se fuese pasando el tiempo, podría detectarse qué tipo de engines están siendo más utilizados y crear más instancias de estos que sustituyan a aquellos engines en desuso. Por ejemplo: podría detectarse que el tráfico es mayoritariamente de tipo P2P y streaming. Entonces, se mulitiplicarían el número de engines destinados a la identificación de estos tipos de tráfico, incrementando notablemente el rendimiento.

Mayor robustez: las FPGAs actuales permiten la desactivación de sectores. Esto es muy útil en caso de que ocurriese una avería hardware, a que se podría detectar el área afectada y aislarla, de manera que el sistema en ningún caso quedase totalmente inutilizado. Al estar las empresas de telecomunicaciones sujetas a unos requisitos de funcionamiento muy estrictos (regla de los cinco nueves), esta robustez añadida puede ser clave.

Ahorro energético: la misma funcionalidad de las FPGAs que permite aislar un sector si está averiado, podría aprovecharse para desconectar los sectores de la FPGA que no estén siendo utilizados. Dichos sectores se reactivarían dinámicamente cuando fuese necesario. Esta medida aumentaría el ahorro energético e iría en la línea del Green Computing que están siguiendo muchas empresas.

APÉNDICE

Código VHDL generado para la expresión regular "babu@eritalk.com", del apartado **Ejemplo paso a paso** del **Capítulo 3**.

```
library ieee;
use ieee.std_logic_1164.all;


ENTITY basic_block IS
PORT (clk: IN std_logic; matching_char : IN std_logic_vector(7 downto 0); char_in: IN
std_logic_vector (7 downto 0); c_in: IN std_logic; c_out: OUT std_logic);
END;

ARCHITECTURE behavior OF basic_block IS
SIGNAL state: std_logic;
SIGNAL matching: std_logic;
BEGIN
PROCESS (state, matching)
BEGIN
IF (matching ='1' AND state='1') THEN
c_out <= '1';
ELSE
c_out <= '0';
END IF;
END PROCESS;
PROCESS(clk, c_in)
BEGIN
IF (clk'EVENT AND clk='1') THEN
state <= c_in;
END IF;
END PROCESS;
PROCESS(char_in, matching_char)
BEGIN
IF (matching_char = char_in) THEN
matching <= '1';
ELSE
matching <= '0';
END IF;
END PROCESS;
END behavior;



library ieee;
use ieee.std_logic_1164.all;


ENTITY andBlock IS
PORT (i: IN std_logic; i1: IN std_logic; i2: IN std_logic; o: OUT std_logic; o1: OUT
std_logic; o2: OUT std_logic);
END;

ARCHITECTURE structuralAnd OF andBlock IS
BEGIN

PROCESS(i, i1, i2)
BEGIN
o1 <= i;
o2 <= i1;
```

```
o <= i2;
END PROCESS;

END structuralAnd;


library ieee;
use ieee.std_logic_1164.all;

ENTITY orBlock IS
PORT (i: IN std_logic; i1: IN std_logic; i2: IN std_logic; o: OUT std_logic; o1: OUT
std_logic; o2: OUT std_logic);
END;

ARCHITECTURE structuralOr OF orBlock IS
BEGIN

PROCESS(i, i1, i2)
BEGIN
o1 <= i;
o2 <= i;
o <= i1 OR i2;
END PROCESS;

END structuralOr;


library ieee;
use ieee.std_logic_1164.all;

ENTITY starBlock IS
PORT (i: IN std_logic; i1: IN std_logic; o: OUT std_logic; o1: OUT std_logic);
END;

ARCHITECTURE structuralStar OF starBlock IS
BEGIN

PROCESS(i, i1)
BEGIN
o1 <= i OR i1;
o <= i1 OR i;
END PROCESS;

END structuralStar;



library ieee;
use ieee.std_logic_1164.all;


ENTITY engine0 IS
PORT (clock: IN std_logic; charIn: IN std_logic_vector(7 downto 0); output: OUT std_logic);
END;

ARCHITECTURE structuralEngine OF engine0 IS

COMPONENT basic_block
PORT (clk: IN std_logic; matching_char : IN std_logic_vector(7 downto 0); char_in : IN
std_logic_vector(7 downto 0); c_in: IN std_logic; c_out: OUT std_logic);
END COMPONENT;
COMPONENT andBlock
PORT (i: INOUT std_logic; i1: INOUT std_logic; i2: INOUT std_logic; o: INOUT std_logic; o1:
INOUT std_logic; o2: INOUT std_logic);
END COMPONENT;
COMPONENT orBlock
PORT (i: INOUT std_logic; i1: INOUT std_logic; i2: INOUT std_logic; o: INOUT std_logic; o1:
INOUT std_logic; o2: INOUT std_logic);
END COMPONENT;
COMPONENT starBlock
PORT (i: INOUT std_logic; i1: INOUT std_logic; o: INOUT std_logic; o1: INOUT std_logic);
END COMPONENT;

SIGNAL id0_i: std_logic;
SIGNAL id0_i1: std_logic;
SIGNAL id0_o: std_logic;
SIGNAL id0_o1: std_logic;
```

```
SIGNAL id1_i: std_logic;
SIGNAL id1_i1: std_logic;
SIGNAL id1_o: std_logic;
SIGNAL id1_o1: std_logic;
SIGNAL id2_i: std_logic;
SIGNAL id2_i1: std_logic;
SIGNAL id2_i2: std_logic;
SIGNAL id2_o: std_logic;
SIGNAL id2_o1: std_logic;
SIGNAL id2_o2: std_logic;
SIGNAL id3_i: std_logic;
SIGNAL id3_i1: std_logic;
SIGNAL id3_o: std_logic;
SIGNAL id3_o1: std_logic;
SIGNAL id4_i: std_logic;
SIGNAL id4_i1: std_logic;
SIGNAL id4_o: std_logic;
SIGNAL id4_o1: std_logic;
SIGNAL id5_i: std_logic;
SIGNAL id5_i1: std_logic;
SIGNAL id5_i2: std_logic;
SIGNAL id5_o: std_logic;
SIGNAL id5_o1: std_logic;
SIGNAL id5_o2: std_logic;
SIGNAL id6_i: std_logic;
SIGNAL id6_i1: std_logic;
SIGNAL id6_i2: std_logic;
SIGNAL id6_o: std_logic;
SIGNAL id6_o1: std_logic;
SIGNAL id6_o2: std_logic;
SIGNAL id7_i: std_logic;
SIGNAL id7_i1: std_logic;
SIGNAL id7_o: std_logic;
SIGNAL id7_o1: std_logic;
SIGNAL id8_i: std_logic;
SIGNAL id8_i1: std_logic;
SIGNAL id8_o: std_logic;
SIGNAL id8_o1: std_logic;
SIGNAL id9_i: std_logic;
SIGNAL id9_i1: std_logic;
SIGNAL id9_i2: std_logic;
SIGNAL id9_o: std_logic;
SIGNAL id9_o1: std_logic;
SIGNAL id9_o2: std_logic;
SIGNAL id10_i: std_logic;
SIGNAL id10_i1: std_logic;
SIGNAL id10_i2: std_logic;
SIGNAL id10_o: std_logic;
SIGNAL id10_o1: std_logic;
SIGNAL id10_o2: std_logic;
SIGNAL id11_i: std_logic;
SIGNAL id11_i1: std_logic;
SIGNAL id11_o: std_logic;
SIGNAL id11_o1: std_logic;
SIGNAL id12_i: std_logic;
SIGNAL id12_i1: std_logic;
SIGNAL id12_o: std_logic;
SIGNAL id12_o1: std_logic;
SIGNAL id13_i: std_logic;
SIGNAL id13_i1: std_logic;
SIGNAL id13_i2: std_logic;
SIGNAL id13_o: std_logic;
SIGNAL id13_o1: std_logic;
SIGNAL id13_o2: std_logic;
SIGNAL id14_i: std_logic;
SIGNAL id14_i1: std_logic;
SIGNAL id14_i2: std_logic;
SIGNAL id14_o: std_logic;
SIGNAL id14_o1: std_logic;
SIGNAL id14_o2: std_logic;
SIGNAL id15_i: std_logic;
SIGNAL id15_i1: std_logic;
SIGNAL id15_o: std_logic;
SIGNAL id15_o1: std_logic;
SIGNAL id16_i: std_logic;
SIGNAL id16_i1: std_logic;
SIGNAL id16_o: std_logic;
```

```
SIGNAL id16_o1: std_logic;
SIGNAL id17_i: std_logic;
SIGNAL id17_i1: std_logic;
SIGNAL id17_i2: std_logic;
SIGNAL id17_o: std_logic;
SIGNAL id17_o1: std_logic;
SIGNAL id17_o2: std_logic;
SIGNAL id18_i: std_logic;
SIGNAL id18_i1: std_logic;
SIGNAL id18_i2: std_logic;
SIGNAL id18_o: std_logic;
SIGNAL id18_o1: std_logic;
SIGNAL id18_o2: std_logic;
SIGNAL id19_i: std_logic;
SIGNAL id19_i1: std_logic;
SIGNAL id19_o: std_logic;
SIGNAL id19_o1: std_logic;
SIGNAL id20_i: std_logic;
SIGNAL id20_i1: std_logic;
SIGNAL id20_o: std_logic;
SIGNAL id20_o1: std_logic;
SIGNAL id21_i: std_logic;
SIGNAL id21_i1: std_logic;
SIGNAL id21_i2: std_logic;
SIGNAL id21_o: std_logic;
SIGNAL id21_o1: std_logic;
SIGNAL id21_o2: std_logic;
SIGNAL id22_i: std_logic;
SIGNAL id22_i1: std_logic;
SIGNAL id22_o: std_logic;
SIGNAL id22_o1: std_logic;
SIGNAL id23_i: std_logic;
SIGNAL id23_i1: std_logic;
SIGNAL id23_o: std_logic;
SIGNAL id23_o1: std_logic;
SIGNAL id24_i: std_logic;
SIGNAL id24_i1: std_logic;
SIGNAL id24_i2: std_logic;
SIGNAL id24_o: std_logic;
SIGNAL id24_o1: std_logic;
SIGNAL id24_o2: std_logic;
SIGNAL id25_i: std_logic;
SIGNAL id25_i1: std_logic;
SIGNAL id25_i2: std_logic;
SIGNAL id25_o: std_logic;
SIGNAL id25_o1: std_logic;
SIGNAL id25_o2: std_logic;
SIGNAL id26_i: std_logic;
SIGNAL id26_i1: std_logic;
SIGNAL id26_o: std_logic;
SIGNAL id26_o1: std_logic;
SIGNAL id27_i: std_logic;
SIGNAL id27_i1: std_logic;
SIGNAL id27_o: std_logic;
SIGNAL id27_o1: std_logic;
SIGNAL id28_i: std_logic;
SIGNAL id28_i1: std_logic;
SIGNAL id28_i2: std_logic;
SIGNAL id28_o: std_logic;
SIGNAL id28_o1: std_logic;
SIGNAL id28_o2: std_logic;
SIGNAL id29_i: std_logic;
SIGNAL id29_i1: std_logic;
SIGNAL id29_i2: std_logic;
SIGNAL id29_o: std_logic;
SIGNAL id29_o1: std_logic;
SIGNAL id29_o2: std_logic;
SIGNAL id30_i: std_logic;
SIGNAL id30_i1: std_logic;
SIGNAL id30_o: std_logic;
SIGNAL id30_o1: std_logic;
SIGNAL id31_i: std_logic;
SIGNAL id31_i1: std_logic;
SIGNAL id31_o: std_logic;
SIGNAL id31_o1: std_logic;
SIGNAL id32_i: std_logic;
SIGNAL id32_i1: std_logic;
```

```
SIGNAL id32_i2: std_logic;
SIGNAL id32_o: std_logic;
SIGNAL id32_o1: std_logic;
SIGNAL id32_o2: std_logic;
SIGNAL id33_i: std_logic;
SIGNAL id33_i1: std_logic;
SIGNAL id33_i2: std_logic;
SIGNAL id33_o: std_logic;
SIGNAL id33_o1: std_logic;
SIGNAL id33_o2: std_logic;
SIGNAL id34_i: std_logic;
SIGNAL id34_i1: std_logic;
SIGNAL id34_o: std_logic;
SIGNAL id34_o1: std_logic;
SIGNAL id35_i: std_logic;
SIGNAL id35_i1: std_logic;
SIGNAL id35_o: std_logic;
SIGNAL id35_o1: std_logic;
SIGNAL id36_i: std_logic;
SIGNAL id36_i1: std_logic;
SIGNAL id36_i2: std_logic;
SIGNAL id36_o: std_logic;
SIGNAL id36_o1: std_logic;
SIGNAL id36_o2: std_logic;
SIGNAL id37_i: std_logic;
SIGNAL id37_i1: std_logic;
SIGNAL id37_i2: std_logic;
SIGNAL id37_o: std_logic;
SIGNAL id37_o1: std_logic;
SIGNAL id37_o2: std_logic;
SIGNAL id38_i: std_logic;
SIGNAL id38_i1: std_logic;
SIGNAL id38_o: std_logic;
SIGNAL id38_o1: std_logic;
SIGNAL id39_i: std_logic;
SIGNAL id39_i1: std_logic;
SIGNAL id39_i2: std_logic;
SIGNAL id39_o: std_logic;
SIGNAL id39_o1: std_logic;
SIGNAL id39_o2: std_logic;
SIGNAL id40_i: std_logic;
SIGNAL id40_i1: std_logic;
SIGNAL id40_o: std_logic;
SIGNAL id40_o1: std_logic;
SIGNAL id41_i: std_logic;
SIGNAL id41_i1: std_logic;
SIGNAL id41_i2: std_logic;
SIGNAL id41_o: std_logic;
SIGNAL id41_o1: std_logic;
SIGNAL id41_o2: std_logic;
SIGNAL id42_i: std_logic;
SIGNAL id42_i1: std_logic;
SIGNAL id42_o: std_logic;
SIGNAL id42_o1: std_logic;
SIGNAL id43_i: std_logic;
SIGNAL id43_i1: std_logic;
SIGNAL id43_o: std_logic;
SIGNAL id43_o1: std_logic;
SIGNAL id44_i: std_logic;
SIGNAL id44_i1: std_logic;
SIGNAL id44_i2: std_logic;
SIGNAL id44_o: std_logic;
SIGNAL id44_o1: std_logic;
SIGNAL id44_o2: std_logic;
SIGNAL id45_i: std_logic;
SIGNAL id45_i1: std_logic;
SIGNAL id45_o: std_logic;
SIGNAL id45_o1: std_logic;
SIGNAL id46_i: std_logic;
SIGNAL id46_i1: std_logic;
SIGNAL id46_i2: std_logic;
SIGNAL id46_o: std_logic;
SIGNAL id46_o1: std_logic;
SIGNAL id46_o2: std_logic;
SIGNAL id47_i: std_logic;
SIGNAL id47_i1: std_logic;
SIGNAL id47_o: std_logic;
```

```
SIGNAL id47_o1: std_logic;
SIGNAL id48_i: std_logic;
SIGNAL id48_i1: std_logic;
SIGNAL id48_i2: std_logic;
SIGNAL id48_o: std_logic;
SIGNAL id48_o1: std_logic;
SIGNAL id48_o2: std_logic;
SIGNAL id49_i: std_logic;
SIGNAL id49_i1: std_logic;
SIGNAL id49_o: std_logic;
SIGNAL id49_o1: std_logic;
SIGNAL id50_i: std_logic;
SIGNAL id50_i1: std_logic;
SIGNAL id50_i2: std_logic;
SIGNAL id50_o: std_logic;
SIGNAL id50_o1: std_logic;
SIGNAL id50_o2: std_logic;
SIGNAL id51_i: std_logic;
SIGNAL id51_i1: std_logic;
SIGNAL id51_o: std_logic;
SIGNAL id51_o1: std_logic;
SIGNAL id52_i: std_logic;
SIGNAL id52_i1: std_logic;
SIGNAL id52_i2: std_logic;
SIGNAL id52_o: std_logic;
SIGNAL id52_o1: std_logic;
SIGNAL id52_o2: std_logic;
SIGNAL id53_i: std_logic;
SIGNAL id53_i1: std_logic;
SIGNAL id53_o: std_logic;
SIGNAL id53_o1: std_logic;
SIGNAL id54_i: std_logic;
SIGNAL id54_i1: std_logic;
SIGNAL id54_i2: std_logic;
SIGNAL id54_o: std_logic;
SIGNAL id54_o1: std_logic;
SIGNAL id54_o2: std_logic;
SIGNAL id55_i: std_logic;
SIGNAL id55_i1: std_logic;
SIGNAL id55_o: std_logic;
SIGNAL id55_o1: std_logic;
SIGNAL id56_i: std_logic;
SIGNAL id56_i1: std_logic;
SIGNAL id56_o: std_logic;
SIGNAL id56_o1: std_logic;
SIGNAL id57_i: std_logic;
SIGNAL id57_i1: std_logic;
SIGNAL id57_i2: std_logic;
SIGNAL id57_o: std_logic;
SIGNAL id57_o1: std_logic;
SIGNAL id57_o2: std_logic;
SIGNAL id58_i: std_logic;
SIGNAL id58_i1: std_logic;
SIGNAL id58_o: std_logic;
SIGNAL id58_o1: std_logic;
SIGNAL id59_i: std_logic;
SIGNAL id59_i1: std_logic;
SIGNAL id59_i2: std_logic;
SIGNAL id59_o: std_logic;
SIGNAL id59_o1: std_logic;
SIGNAL id59_o2: std_logic;
SIGNAL id60_i: std_logic;
SIGNAL id60_i1: std_logic;
SIGNAL id60_o: std_logic;
SIGNAL id60_o1: std_logic;
SIGNAL id61_i: std_logic;
SIGNAL id61_i1: std_logic;
SIGNAL id61_i2: std_logic;
SIGNAL id61_o: std_logic;
SIGNAL id61_o1: std_logic;
SIGNAL id61_o2: std_logic;
SIGNAL id62_i: std_logic;
SIGNAL id62_i1: std_logic;
SIGNAL id62_o: std_logic;
SIGNAL id62_o1: std_logic;
SIGNAL id63_i: std_logic;
SIGNAL id63_i1: std_logic;
```

```
SIGNAL id63_i2: std_logic;
SIGNAL id63_o: std_logic;
SIGNAL id63_o1: std_logic;
SIGNAL id63_o2: std_logic;
SIGNAL id64_i: std_logic;
SIGNAL id64_i1: std_logic;
SIGNAL id64_o: std_logic;
SIGNAL id64_o1: std_logic;
SIGNAL id65_i: std_logic;
SIGNAL id65_i1: std_logic;
SIGNAL id65_i2: std_logic;
SIGNAL id65_o: std_logic;
SIGNAL id65_o1: std_logic;
SIGNAL id65_o2: std_logic;
SIGNAL id66_i: std_logic;
SIGNAL id66_i1: std_logic;
SIGNAL id66_o: std_logic;
SIGNAL id66_o1: std_logic;
SIGNAL id67_i: std_logic;
SIGNAL id67_i1: std_logic;
SIGNAL id67_i2: std_logic;
SIGNAL id67_o: std_logic;
SIGNAL id67_o1: std_logic;
SIGNAL id67_o2: std_logic;
SIGNAL id68_i: std_logic;
SIGNAL id68_i1: std_logic;
SIGNAL id68_o: std_logic;
SIGNAL id68_o1: std_logic;
SIGNAL id69_i: std_logic;
SIGNAL id69_i1: std_logic;
SIGNAL id69_i2: std_logic;
SIGNAL id69_o: std_logic;
SIGNAL id69_o1: std_logic;
SIGNAL id69_o2: std_logic;
SIGNAL id70_i: std_logic;
SIGNAL id70_i1: std_logic;
SIGNAL id70_i2: std_logic;
SIGNAL id70_o: std_logic;
SIGNAL id70_o1: std_logic;
SIGNAL id70_o2: std_logic;
SIGNAL id71_i: std_logic;
SIGNAL id71_i1: std_logic;
SIGNAL id71_i2: std_logic;
SIGNAL id71_o: std_logic;
SIGNAL id71_o1: std_logic;
SIGNAL id71_o2: std_logic;
SIGNAL id72_i: std_logic;
SIGNAL id72_i1: std_logic;
SIGNAL id72_o: std_logic;
SIGNAL id72_o1: std_logic;
SIGNAL id73_i: std_logic;
SIGNAL id73_i1: std_logic;
SIGNAL id73_o: std_logic;
SIGNAL id73_o1: std_logic;
SIGNAL id74_i: std_logic;
SIGNAL id74_i1: std_logic;
SIGNAL id74_i2: std_logic;
SIGNAL id74_o: std_logic;
SIGNAL id74_o1: std_logic;
SIGNAL id74_o2: std_logic;
SIGNAL id75_i: std_logic;
SIGNAL id75_i1: std_logic;
SIGNAL id75_o: std_logic;
SIGNAL id75_o1: std_logic;
SIGNAL id76_i: std_logic;
SIGNAL id76_i1: std_logic;
SIGNAL id76_i2: std_logic;
SIGNAL id76_o: std_logic;
SIGNAL id76_o1: std_logic;
SIGNAL id76_o2: std_logic;
SIGNAL id77_i: std_logic;
SIGNAL id77_i1: std_logic;
SIGNAL id77_o: std_logic;
SIGNAL id77_o1: std_logic;
SIGNAL id78_i: std_logic;
SIGNAL id78_i1: std_logic;
SIGNAL id78_i2: std_logic;
```

```
SIGNAL id78_o: std_logic;
SIGNAL id78_o1: std_logic;
SIGNAL id78_o2: std_logic;
SIGNAL id79_i: std_logic;
SIGNAL id79_i1: std_logic;
SIGNAL id79_i2: std_logic;
SIGNAL id79_o: std_logic;
SIGNAL id79_o1: std_logic;
SIGNAL id79_o2: std_logic;

BEGIN
id0: basic_block PORT MAP (clock, "01100010", charIn, id0_i, id0_o);

id1: basic_block PORT MAP (clock, "01100001", charIn, id1_i, id1_o);

id2: andBlock PORT MAP(id2_i, id2_i1, id2_i2, id2_o, id2_o1, id2_o2);



PROCESS(id1_o, id2_o2)
BEGIN
id2_i2 <= id1_o;
id1_i <= id2_o2;
END PROCESS;


PROCESS(id0_o, id2_o1)
BEGIN
id2_i1 <= id0_o;
id0_i <= id2_o1;
END PROCESS;
id3: basic_block PORT MAP (clock, "01100010", charIn, id3_i, id3_o);

id4: basic_block PORT MAP (clock, "01100101", charIn, id4_i, id4_o);

id5: andBlock PORT MAP(id5_i, id5_i1, id5_i2, id5_o, id5_o1, id5_o2);



PROCESS(id4_o, id5_o2)
BEGIN
id5_i2 <= id4_o;
id4_i <= id5_o2;
END PROCESS;


PROCESS(id3_o, id5_o1)
BEGIN
id5_i1 <= id3_o;
id3_i <= id5_o1;
END PROCESS;
id6: orBlock PORT MAP(id6_i, id6_i1, id6_i2, id6_o, id6_o1, id6_o2);



PROCESS(id5_o, id6_o2)
BEGIN
id6_i2 <= id5_o;
id5_i <= id6_o2;
END PROCESS;


PROCESS(id2_o, id6_o1)
BEGIN
id6_i1 <= id2_o;
id2_i <= id6_o1;
END PROCESS;
id7: basic_block PORT MAP (clock, "01100010", charIn, id7_i, id7_o);

id8: basic_block PORT MAP (clock, "01101001", charIn, id8_i, id8_o);

id9: andBlock PORT MAP(id9_i, id9_i1, id9_i2, id9_o, id9_o1, id9_o2);



PROCESS(id8_o, id9_o2)
BEGIN
```

```
id9_i2 <= id8_o;
id8_i <= id9_o2;
END PROCESS;


PROCESS(id7_o, id9_o1)
BEGIN
id9_i1 <= id7_o;
id7_i <= id9_o1;
END PROCESS;
id10: orBlock PORT MAP(id10_i, id10_i1, id10_i2, id10_o, id10_o1, id10_o2);



PROCESS(id9_o, id10_o2)
BEGIN
id10_i2 <= id9_o;
id9_i <= id10_o2;
END PROCESS;


PROCESS(id6_o, id10_o1)
BEGIN
id10_i1 <= id6_o;
id6_i <= id10_o1;
END PROCESS;
id11: basic_block PORT MAP (clock, "01100010", charIn, id11_i, id11_o);

id12: basic_block PORT MAP (clock, "01101111", charIn, id12_i, id12_o);

id13: andBlock PORT MAP(id13_i, id13_i1, id13_i2, id13_o, id13_o1, id13_o2);



PROCESS(id12_o, id13_o2)
BEGIN
id13_i2 <= id12_o;
id12_i <= id13_o2;
END PROCESS;


PROCESS(id11_o, id13_o1)
BEGIN
id13_i1 <= id11_o;
id11_i <= id13_o1;
END PROCESS;
id14: orBlock PORT MAP(id14_i, id14_i1, id14_i2, id14_o, id14_o1, id14_o2);



PROCESS(id13_o, id14_o2)
BEGIN
id14_i2 <= id13_o;
id13_i <= id14_o2;
END PROCESS;


PROCESS(id10_o, id14_o1)
BEGIN
id14_i1 <= id10_o;
id10_i <= id14_o1;
END PROCESS;
id15: basic_block PORT MAP (clock, "01100010", charIn, id15_i, id15_o);

id16: basic_block PORT MAP (clock, "01110101", charIn, id16_i, id16_o);

id17: andBlock PORT MAP(id17_i, id17_i1, id17_i2, id17_o, id17_o1, id17_o2);



PROCESS(id16_o, id17_o2)
BEGIN
id17_i2 <= id16_o;
id16_i <= id17_o2;
END PROCESS;
```

```
PROCESS(id15_o, id17_o1)
BEGIN
id17_i1 <= id15_o;
id15_i <= id17_o1;
END PROCESS;
id18: orBlock PORT MAP(id18_i, id18_i1, id18_i2, id18_o, id18_o1, id18_o2);



PROCESS(id17_o, id18_o2)
BEGIN
id18_i2 <= id17_o;
id17_i <= id18_o2;
END PROCESS;


PROCESS(id14_o, id18_o1)
BEGIN
id18_i1 <= id14_o;
id14_i <= id18_o1;
END PROCESS;
id19: basic_block PORT MAP (clock, "01100010", charIn, id19_i, id19_o);

id20: basic_block PORT MAP (clock, "01100001", charIn, id20_i, id20_o);

id21: andBlock PORT MAP(id21_i, id21_i1, id21_i2, id21_o, id21_o1, id21_o2);



PROCESS(id20_o, id21_o2)
BEGIN
id21_i2 <= id20_o;
id20_i <= id21_o2;
END PROCESS;


PROCESS(id19_o, id21_o1)
BEGIN
id21_i1 <= id19_o;
id19_i <= id21_o1;
END PROCESS;
id22: basic_block PORT MAP (clock, "01100010", charIn, id22_i, id22_o);

id23: basic_block PORT MAP (clock, "01100101", charIn, id23_i, id23_o);

id24: andBlock PORT MAP(id24_i, id24_i1, id24_i2, id24_o, id24_o1, id24_o2);



PROCESS(id23_o, id24_o2)
BEGIN
id24_i2 <= id23_o;
id23_i <= id24_o2;
END PROCESS;


PROCESS(id22_o, id24_o1)
BEGIN
id24_i1 <= id22_o;
id22_i <= id24_o1;
END PROCESS;
id25: orBlock PORT MAP(id25_i, id25_i1, id25_i2, id25_o, id25_o1, id25_o2);



PROCESS(id24_o, id25_o2)
BEGIN
id25_i2 <= id24_o;
id24_i <= id25_o2;
END PROCESS;


PROCESS(id21_o, id25_o1)
BEGIN
id25_i1 <= id21_o;
id21_i <= id25_o1;
END PROCESS;
```

```
id26: basic_block PORT MAP (clock, "01100010", charIn, id26_i, id26_o);

id27: basic_block PORT MAP (clock, "01101001", charIn, id27_i, id27_o);

id28: andBlock PORT MAP(id28_i, id28_i1, id28_i2, id28_o, id28_o1, id28_o2);



PROCESS(id27_o, id28_o2)
BEGIN
id28_i2 <= id27_o;
id27_i <= id28_o2;
END PROCESS;


PROCESS(id26_o, id28_o1)
BEGIN
id28_i1 <= id26_o;
id26_i <= id28_o1;
END PROCESS;
id29: orBlock PORT MAP(id29_i, id29_i1, id29_i2, id29_o, id29_o1, id29_o2);



PROCESS(id28_o, id29_o2)
BEGIN
id29_i2 <= id28_o;
id28_i <= id29_o2;
END PROCESS;


PROCESS(id25_o, id29_o1)
BEGIN
id29_i1 <= id25_o;
id25_i <= id29_o1;
END PROCESS;
id30: basic_block PORT MAP (clock, "01100010", charIn, id30_i, id30_o);

id31: basic_block PORT MAP (clock, "01101111", charIn, id31_i, id31_o);

id32: andBlock PORT MAP(id32_i, id32_i1, id32_i2, id32_o, id32_o1, id32_o2);



PROCESS(id31_o, id32_o2)
BEGIN
id32_i2 <= id31_o;
id31_i <= id32_o2;
END PROCESS;


PROCESS(id30_o, id32_o1)
BEGIN
id32_i1 <= id30_o;
id30_i <= id32_o1;
END PROCESS;
id33: orBlock PORT MAP(id33_i, id33_i1, id33_i2, id33_o, id33_o1, id33_o2);



PROCESS(id32_o, id33_o2)
BEGIN
id33_i2 <= id32_o;
id32_i <= id33_o2;
END PROCESS;


PROCESS(id29_o, id33_o1)
BEGIN
id33_i1 <= id29_o;
id29_i <= id33_o1;
END PROCESS;
id34: basic_block PORT MAP (clock, "01100010", charIn, id34_i, id34_o);

id35: basic_block PORT MAP (clock, "01110101", charIn, id35_i, id35_o);

id36: andBlock PORT MAP(id36_i, id36_i1, id36_i2, id36_o, id36_o1, id36_o2);
```

```
PROCESS(id35_o, id36_o2)
BEGIN
id36_i2 <= id35_o;
id35_i <= id36_o2;
END PROCESS;


PROCESS(id34_o, id36_o1)
BEGIN
id36_i1 <= id34_o;
id34_i <= id36_o1;
END PROCESS;
id37: orBlock PORT MAP(id37_i, id37_i1, id37_i2, id37_o, id37_o1, id37_o2);



PROCESS(id36_o, id37_o2)
BEGIN
id37_i2 <= id36_o;
id36_i <= id37_o2;
END PROCESS;


PROCESS(id33_o, id37_o1)
BEGIN
id37_i1 <= id33_o;
id33_i <= id37_o1;
END PROCESS;
id38: starBlock PORT MAP(id38_i, id38_i1, id38_o, id38_o1);



PROCESS(id37_o, id38_o1)
BEGIN
id38_i1 <= id37_o;
id37_i <= id38_o1;
END PROCESS;
id39: andBlock PORT MAP(id39_i, id39_i1, id39_i2, id39_o, id39_o1, id39_o2);



PROCESS(id38_o, id39_o2)
BEGIN
id39_i2 <= id38_o;
id38_i <= id39_o2;
END PROCESS;


PROCESS(id18_o, id39_o1)
BEGIN
id39_i1 <= id18_o;
id18_i <= id39_o1;
END PROCESS;
id40: basic_block PORT MAP (clock, "01000000", charIn, id40_i, id40_o);

id41: andBlock PORT MAP(id41_i, id41_i1, id41_i2, id41_o, id41_o1, id41_o2);



PROCESS(id40_o, id41_o2)
BEGIN
id41_i2 <= id40_o;
id40_i <= id41_o2;
END PROCESS;


PROCESS(id39_o, id41_o1)
BEGIN
id41_i1 <= id39_o;
id39_i <= id41_o1;
END PROCESS;
id42: basic_block PORT MAP (clock, "01100101", charIn, id42_i, id42_o);

id43: basic_block PORT MAP (clock, "01110010", charIn, id43_i, id43_o);
```

```
id44: andBlock PORT MAP(id44_i, id44_i1, id44_i2, id44_o, id44_o1, id44_o2);


PROCESS(id43_o, id44_o2)
BEGIN
id44_i2 <= id43_o;
id43_i <= id44_o2;
END PROCESS;


PROCESS(id42_o, id44_o1)
BEGIN
id44_i1 <= id42_o;
id42_i <= id44_o1;
END PROCESS;
id45: basic_block PORT MAP (clock, "01101001", charIn, id45_i, id45_o);

id46: andBlock PORT MAP(id46_i, id46_i1, id46_i2, id46_o, id46_o1, id46_o2);


PROCESS(id45_o, id46_o2)
BEGIN
id46_i2 <= id45_o;
id45_i <= id46_o2;
END PROCESS;


PROCESS(id44_o, id46_o1)
BEGIN
id46_i1 <= id44_o;
id44_i <= id46_o1;
END PROCESS;
id47: basic_block PORT MAP (clock, "01110100", charIn, id47_i, id47_o);

id48: andBlock PORT MAP(id48_i, id48_i1, id48_i2, id48_o, id48_o1, id48_o2);


PROCESS(id47_o, id48_o2)
BEGIN
id48_i2 <= id47_o;
id47_i <= id48_o2;
END PROCESS;


PROCESS(id46_o, id48_o1)
BEGIN
id48_i1 <= id46_o;
id46_i <= id48_o1;
END PROCESS;
id49: basic_block PORT MAP (clock, "01100001", charIn, id49_i, id49_o);

id50: andBlock PORT MAP(id50_i, id50_i1, id50_i2, id50_o, id50_o1, id50_o2);


PROCESS(id49_o, id50_o2)
BEGIN
id50_i2 <= id49_o;
id49_i <= id50_o2;
END PROCESS;


PROCESS(id48_o, id50_o1)
BEGIN
id50_i1 <= id48_o;
id48_i <= id50_o1;
END PROCESS;
id51: basic_block PORT MAP (clock, "01101100", charIn, id51_i, id51_o);

id52: andBlock PORT MAP(id52_i, id52_i1, id52_i2, id52_o, id52_o1, id52_o2);
```

```
PROCESS(id51_o, id52_o2)
BEGIN
id52_i2 <= id51_o;
id51_i <= id52_o2;
END PROCESS;


PROCESS(id50_o, id52_o1)
BEGIN
id52_i1 <= id50_o;
id50_i <= id52_o1;
END PROCESS;
id53: basic_block PORT MAP (clock, "01101011", charIn, id53_i, id53_o);

id54: andBlock PORT MAP(id54_i, id54_i1, id54_i2, id54_o, id54_o1, id54_o2);



PROCESS(id53_o, id54_o2)
BEGIN
id54_i2 <= id53_o;
id53_i <= id54_o2;
END PROCESS;


PROCESS(id52_o, id54_o1)
BEGIN
id54_i1 <= id52_o;
id52_i <= id54_o1;
END PROCESS;
id55: basic_block PORT MAP (clock, "01100101", charIn, id55_i, id55_o);

id56: basic_block PORT MAP (clock, "01110010", charIn, id56_i, id56_o);

id57: andBlock PORT MAP(id57_i, id57_i1, id57_i2, id57_o, id57_o1, id57_o2);



PROCESS(id56_o, id57_o2)
BEGIN
id57_i2 <= id56_o;
id56_i <= id57_o2;
END PROCESS;


PROCESS(id55_o, id57_o1)
BEGIN
id57_i1 <= id55_o;
id55_i <= id57_o1;
END PROCESS;
id58: basic_block PORT MAP (clock, "01101001", charIn, id58_i, id58_o);

id59: andBlock PORT MAP(id59_i, id59_i1, id59_i2, id59_o, id59_o1, id59_o2);



PROCESS(id58_o, id59_o2)
BEGIN
id59_i2 <= id58_o;
id58_i <= id59_o2;
END PROCESS;


PROCESS(id57_o, id59_o1)
BEGIN
id59_i1 <= id57_o;
id57_i <= id59_o1;
END PROCESS;
id60: basic_block PORT MAP (clock, "01100011", charIn, id60_i, id60_o);

id61: andBlock PORT MAP(id61_i, id61_i1, id61_i2, id61_o, id61_o1, id61_o2);



PROCESS(id60_o, id61_o2)
BEGIN
id61_i2 <= id60_o;
```

```
id60_i <= id61_o2;
END PROCESS;


PROCESS(id59_o, id61_o1)
BEGIN
id61_i1 <= id59_o;
id59_i <= id61_o1;
END PROCESS;
id62: basic_block PORT MAP (clock, "01110011", charIn, id62_i, id62_o);

id63: andBlock PORT MAP(id63_i, id63_i1, id63_i2, id63_o, id63_o1, id63_o2);



PROCESS(id62_o, id63_o2)
BEGIN
id63_i2 <= id62_o;
id62_i <= id63_o2;
END PROCESS;


PROCESS(id61_o, id63_o1)
BEGIN
id63_i1 <= id61_o;
id61_i <= id63_o1;
END PROCESS;
id64: basic_block PORT MAP (clock, "01110011", charIn, id64_i, id64_o);

id65: andBlock PORT MAP(id65_i, id65_i1, id65_i2, id65_o, id65_o1, id65_o2);



PROCESS(id64_o, id65_o2)
BEGIN
id65_i2 <= id64_o;
id64_i <= id65_o2;
END PROCESS;


PROCESS(id63_o, id65_o1)
BEGIN
id65_i1 <= id63_o;
id63_i <= id65_o1;
END PROCESS;
id66: basic_block PORT MAP (clock, "01101111", charIn, id66_i, id66_o);

id67: andBlock PORT MAP(id67_i, id67_i1, id67_i2, id67_o, id67_o1, id67_o2);



PROCESS(id66_o, id67_o2)
BEGIN
id67_i2 <= id66_o;
id66_i <= id67_o2;
END PROCESS;


PROCESS(id65_o, id67_o1)
BEGIN
id67_i1 <= id65_o;
id65_i <= id67_o1;
END PROCESS;
id68: basic_block PORT MAP (clock, "01101110", charIn, id68_i, id68_o);

id69: andBlock PORT MAP(id69_i, id69_i1, id69_i2, id69_o, id69_o1, id69_o2);



PROCESS(id68_o, id69_o2)
BEGIN
id69_i2 <= id68_o;
id68_i <= id69_o2;
END PROCESS;


PROCESS(id67_o, id69_o1)
```

```
BEGIN
id69_i1 <= id67_o;
id67_i <= id69_o1;
END PROCESS;
id70: orBlock PORT MAP(id70_i, id70_i1, id70_i2, id70_o, id70_o1, id70_o2);



PROCESS(id69_o, id70_o2)
BEGIN
id70_i2 <= id69_o;
id69_i <= id70_o2;
END PROCESS;


PROCESS(id54_o, id70_o1)
BEGIN
id70_i1 <= id54_o;
id54_i <= id70_o1;
END PROCESS;
id71: andBlock PORT MAP(id71_i, id71_i1, id71_i2, id71_o, id71_o1, id71_o2);



PROCESS(id70_o, id71_o2)
BEGIN
id71_i2 <= id70_o;
id70_i <= id71_o2;
END PROCESS;


PROCESS(id41_o, id71_o1)
BEGIN
id71_i1 <= id41_o;
id41_i <= id71_o1;
END PROCESS;
id72: basic_block PORT MAP (clock, "00101110", charIn, id72_i, id72_o);

id73: basic_block PORT MAP (clock, "01100011", charIn, id73_i, id73_o);

id74: andBlock PORT MAP(id74_i, id74_i1, id74_i2, id74_o, id74_o1, id74_o2);



PROCESS(id73_o, id74_o2)
BEGIN
id74_i2 <= id73_o;
id73_i <= id74_o2;
END PROCESS;


PROCESS(id72_o, id74_o1)
BEGIN
id74_i1 <= id72_o;
id72_i <= id74_o1;
END PROCESS;
id75: basic_block PORT MAP (clock, "01101111", charIn, id75_i, id75_o);

id76: andBlock PORT MAP(id76_i, id76_i1, id76_i2, id76_o, id76_o1, id76_o2);



PROCESS(id75_o, id76_o2)
BEGIN
id76_i2 <= id75_o;
id75_i <= id76_o2;
END PROCESS;


PROCESS(id74_o, id76_o1)
BEGIN
id76_i1 <= id74_o;
id74_i <= id76_o1;
END PROCESS;
id77: basic_block PORT MAP (clock, "01101101", charIn, id77_i, id77_o);

id78: andBlock PORT MAP(id78_i, id78_i1, id78_i2, id78_o, id78_o1, id78_o2);
```

```
PROCESS(id77_o, id78_o2)
BEGIN
id78_i2 <= id77_o;
id77_i <= id78_o2;
END PROCESS;


PROCESS(id76_o, id78_o1)
BEGIN
id78_i1 <= id76_o;
id76_i <= id78_o1;
END PROCESS;
id79: andBlock PORT MAP(id79_i, id79_i1, id79_i2, id79_o, id79_o1, id79_o2);



PROCESS(id78_o, id79_o2)
BEGIN
id79_i2 <= id78_o;
id78_i <= id79_o2;
END PROCESS;


PROCESS(id71_o, id79_o1)
BEGIN
id79_i1 <= id71_o;
id71_i <= id79_o1;
END PROCESS;

PROCESS(id79_o)
BEGIN
id79_i <= '1';
output <= id79_o;
END PROCESS;
END structuralEngine;


library ieee;
use ieee.std_logic_1164.all;


ENTITY regExpMatching IS
PORT(clock: IN std_logic; charIn: IN std_logic_vector(7 downto 0); output: OUT std_logic);
END;

ARCHITECTURE structuralRegExpMatching OF regExpMatching IS
COMPONENT engine0
PORT(clock: IN std_logic; charIn: IN std_logic_vector(7 downto 0); output: OUT std_logic);
END COMPONENT;
SIGNAL s0: std_logic;

BEGIN
e0: engine0 PORT MAP(clock, charIn, s0);

PROCESS(s0)
BEGIN
output <= s0;
END PROCESS;
END structuralRegExpMatching;
```

BILIOGRAFÍA

Capítulo 1: Introducción

Hardware reconfigurable: FPGAs

- [1] http://www.xilinx.com
- [2] http://www.altera.com
- [3] http://www.latticesemi.com
- [4] http://www.quicklogic.com
- [5] http://www.actel.com
- [6] http://www.atmel.com
- [7] http://www.achronix.com
- [8] http://proxacutor.free.fr/index.htm
- [9] http://www.tutorial-reports.com/computer-science/fpga
- [10] http://www.tutorial-reports.com/book/print/260

Introducción a las expresiones regulares

- [11] “Introduction to languages and the theory of computation”. John C. Martin. Mc Graw Hill, 1991.
- [12] http://en.wikipedia.org/wiki/Regular_expression

Importancia del reconocimiento de ERs en las Telecomunicaciones

- [13] Badii, A. Adetoye, D. Patel, K. Hameed “*Efficient FPGA-Based Regular Expresión Pattern Matching*” (EMCIS2008)
- [14] B.C. Brodie, R.K. Cytron, D.E. Taylor “*A Scalable Architecture For High-Throughput Regular-Expression Pattern Matching*” (ISCA’06)

Capítulo 2: Trabajo Relacionado

- [15] http://www.snort.org
- [16] http://www.netlogicmicro.com
- [17] http://www.proceranetworks.com
- [18] Bonesana, M. Paolieri y M. D. Santambrogio "*An adaptable FPGA-based System for Regular Expression Matching*", (DATE'08)
- [19] J. Bispo, I. Sourdis, J. Cardoso y S. Vassiliadis "*Regular Expression Matching for Reconfigurable Packet Inspection*", (FPT 2006)
- [20] P. Sutton "*Partial Character Decoding for Improved Regular Expression Matching in FPGAs*", (FPT 2004)

Capítulo 3: Reconocedor de Expresiones Regulares

- [21] R. Sidhu, V.K. Prasanna "*Fast Regular Expression Matching Using FPGAs*" (FCCM'01)
- [22] J. Divashree, H. Rajashekar , Kuruvilla Varghese "*Dynamically Reconfigurable Regular Expression Matching Architecture*", 2008 International Conference on Application-Specific Systems, Architectures and Processors
- [23] B.C. Brodie, R.K. Cytron, D.E. Taylor "*A Scalable Architecture For High-Throughput Regular-Expression Pattern Matching*" (ISCA'06)