Universidad de Concepción
Dirección de Postgrado
Facultad de Ingeniería - Programa de Magíster en Ciencias de la Ingeniería con mención en Ingeniería Eléctrica

Síntesis de alto nivel de arquitecturas para procesamiento de video

Tesis presentada a la Facultad de Ingeniería de la Universidad de Concepción para optar al grado de Magíster en Ciencias de la Ingeniería con mención en Ingeniería Eléctrica

POR: PABLO ALFREDO VERDUGO RUBILAR

Profesor Guía: Dr. Miguel Ernesto Figueroa Toro

abril, 2020
Concepción, Chile

Universidad de Concepción
Facultad de Ingeniería
Departamento de Ingeniería Eléctrica

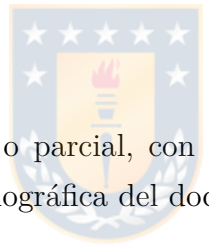
Profesor Patrocinante:
Dr. Miguel Figueroa T.

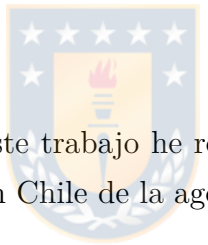
Síntesis de alto nivel de arquitecturas para procesamiento de video

Pablo Alfredo Verdugo Rubilar

Informe de tesis de grado para optar al grado de
"Magíster en Ciencias de la Ingeniería con mención en Ingeniería Eléctrica"

Concepción, Chile.
27 de abril de 2020





Durante las diferentes etapas de este trabajo he recibido el apoyo financiero del Programa de Becas para Estudios de Magíster en Chile de la agencia CONICYT del gobierno de Chile.

Resumen

El procesamiento de imágenes y video es una tarea intensiva que requiere una gran capacidad de cómputo para llevarse a cabo. En particular, el procesamiento de video en con restricciones de tiempo real requiere que este cómputo se realice en un tiempo determinado, esto es, el tiempo transcurrido entre la generación de cuadros. Esta restricción reduce las opciones al elegir una unidad de procesamiento para esta tarea. Dentro de las disponibles destacan los FPGAs. Estos ofrecen una gran flexibilidad, pero su programación requiere mayores tiempos de desarrollo, además de conocimientos de diseño de hardware.

En el afán por mitigar esta desventaja se han desarrollado herramientas que puedan hacer síntesis de alto nivel (HLS), esto es, generar arquitecturas hardware a partir de un lenguaje de alto nivel. Para esto se pueden usar lenguajes de programación para procesadores de propósito general o lenguajes de dominio específico (DSL). En el laboratorio de VLSI de la Universidad de Concepción se desarrolló un modelo de cómputo, que dio paso a un DSL para el dominio del procesamiento de imágenes y video.

En este trabajo se desarrolló una herramienta capaz de generar una descripción de hardware para FPGAs a partir del lenguaje antes mencionado, herramienta que lleva por nombre SALTT (SALTT is A Language Translation Tool). La herramienta cuenta con un analizador léxico y uno sintáctico cuyo fin es reconocer el texto estructurado de entrada y generar la salida. De esta forma, la herramienta desarrollada tiene la capacidad de transformar código desde el lenguaje de SALTT al HDL SystemVerilog.

La herramienta se probó transformando código para filtros de media, mediana y transformaciones de espacios de colores. Los códigos se sintetizaron e implementaron en una tarjeta de desarrollo Nexys Video de Digilent. La tarjeta se conectó a un monitor y se comprobó que las salidas correspondían a lo esperado, demostrando así el correcto funcionamiento del código generado. Además del código generado, se escribieron equivalentes de estos códigos en SystemVerilog y se compararon los resultados, observándose un similar uso de recursos, consumo de potencia y frecuencia máxima de operación en la gran mayoría de los casos. También se observó un reducción en el número de líneas de código requeridas.

Índice General

Índice de Figuras

Siglas

ASIC circuito integrado de aplicación específica (del inglés *Application-Specific Integrated Circuit)*

CPU unidad central de procesamiento (del inglés *Central Processing Unit)*

DSL lenguaje de domino específico (del inglés *Domain-Specific Language)*

DSP procesador de señales (del inglés *Digital Signal Processor)*

EDK kit de diseño electrónico (del inglés *Electronic Design Kit)*

FIFO primero en entrar, primero en salir (del inglés *First In, First Out)*

FPGA arreglo de compuertas programables (del inglés *Field Programmable Gate Array)*

GPU unidad de procesamiento gráfico (del inglés *Graphics Processor Unit)*

HDL lenguaje de descripción de hardware (del inglés *Hardware Description Language)*

HLS síntesis de alto nivel (del inglés *High Level Synthesis)*

IDE ambiente integrado de diseño (del inglés *Integrated Design Environment)*

IP propiedad intelectual (del inglés *Intellectual Property)*

SoC sistema en chip (del inglés *System on Chip)*

TCL lenguaje de comando para herramientas (del inglés *Tool Command Language)*

Capítulo 1: Introducción

1.1. Introdución general

Los sistemas embebidos son dispositivos de procesamiento que realizan tareas específicas. Debido a sus características, como velocidad de procesamiento, tamaño reducido y bajo consumo de potencia, son ampliamente utilizados, pudiéndose encontrar en la gran mayoría de las actividades realizadas por las personas. Una de sus principales ventajas es que son capaces de cumplir con restricciones de operación en tiempo real, esto es, que el tiempo de procesamiento esté dentro de los rangos permitidos, los que por lo general son cortos.

Un sistema embebido está compuesto por una unidad de procesamiento, la que puede ser un microcontrolador, un procesador de señales (del inglés *Digital Signal Processor*, DSP), una unidad de procesamiento gráfico (del inglés *Graphics Processor Unit*, GPU), un arreglo de compuertas programables (del inglés *Field Programmable Gate Array*, FPGA), un circuito integrado de aplicación específica (del inglés *Application-Specific Integrated Circuit*, ASIC) o una combinación de éstos. La elección de una de las opciones antes mencionadas depende de la aplicación, ya que factores como la velocidad de respuesta, los tiempos de desarrollo, el tipo de operaciones o los límites presupuestarios puede hacer que una opción sea más beneficiosa que el resto.

Una aplicación para los sistemas embebidos es el procesamiento de imágenes y video. Este procesamiento se caracteriza por aplicar una serie de operaciones (comúnmente sumas y multiplicaciones) de forma reiterada sobre porciones de un bloque de datos (imagen). Debido a lo común de este tipo de procesamiento y a lo recurrente de sus operaciones se diseñó una unidad de procesamiento dedicada a esta tarea, las GPU.

La arquitectura de las GPUs hace que procesamientos como filtros de imágenes tengan un gran desempeño, especialmente en su tiempo de ejecución. Sin embargo, el tiempo de ejecución aumenta al realizar procesamientos que involucren operaciones diferentes a sumas y multiplicaciones. Esto no es crítico al procesar imágenes individuales, como al sacar una fotografía, ya que el tiempo de ejecución no es crítico, pero al realizar procesamiento de video, el tiempo se convierte en una restricción, especialmente en el procesamiento en tiempo real, ya que se debe procesar cada cuadro antes que se genere el siguiente para mantener constante el flujo de datos.

Otra restricción que las GPUs tiene problemas en cumplir es el consumo de potencia. Se espera que un sistema embebido tenga un bajo consumo energético, de modo que este sea marginal con respecto al consumo total del dispositivo en el que se encuentran. Históricamente esto ha sido así, sin embargo, en el último tiempo se ha trabajado en GPUs de bajo consumo de potencia [1]. De todas formas, los FPGAs siguen teniendo un menor consumo energético, aunque la brecha se haya estrechado.

Los FPGA son unidades de procesamiento que permiten la reconfiguración a nivel de *hardware* de los elementos que lo componen, por lo tanto, todo sistema implementado en estas unidades corresponde a un diseño de hardware, lo que permite utilizar solamente los recursos necesarios, minimizando el tiempo de ejecución y uso de potencia. Por todo lo anterior los FPGAs son una tecnología atractiva para el desarrollo de sistemas embebidos para procesamiento de imágenes.

El problema de los FPGA es producto de sus ventajas, esto porque la libertad de diseño producto de su capacidad de configurarse a nivel de *hardware*, que permite cumplir con restricciones de consumo de potencia y tiempo de ejecución, implica tener la tener la capacidad de expresar una solución a nivel de *hardware*. Para facilitar este proceso existen lenguaje de descripción de hardware (del inglés *Hardware Description Language*, HDL), los que, como su nombre lo indica, describen el *hardware* de un diseño, a diferencia de los lenguajes de programación de *software*, que expresan los algoritmos de un diseño.

En el afán de disminuir la barrera de entrada al diseño de hardware se han propuestos sistemas que realicen síntesis de alto nivel (del inglés *High Level Synthesis*, HLS), esto es, hacer síntesis de hardware a partir de un lenguaje de nivel superior a los HDL. Se pueden encontrar programas que hacen HLS a partir de lenguajes de programación para procesadores de propósito general como C, C++, MATLAB o Python. Estos lenguajes, al ser comúnmente utilizados por programadores de *software*, disminuyen la barrera de entrada para los desarrolladores de *software* y permiten su migración al diseño de *hardware*. Sin embargo, esto significa pasar de una programación secuencial no consciente del reloj a programación paralela con la capacidad de manejar distintos dominios de reloj. Para lograrlo se pueden hacer suposiciones, lo que puede disminuir el desempeño, o analizar el código, lo que puede extender el tiempo de compilación.

En la búsqueda de elevar el nivel de abstracción y así simplificar el diseño de procesadores de video en FPGAs, Araneda [2] propuso un modelo de cómputo y lenguaje de programación, los que nacen a partir de observaciones realizadas a diversos trabajos de procesamiento de imágenes en el espectro infrarrojo. El lenguaje permite describir algoritmos a bajo nivel, resultando un intermedio entre un lenguaje de *software* y uno de *hardware*. Con esto se busca disminuir la barrera de entrada para programar diseños en FPGAs.

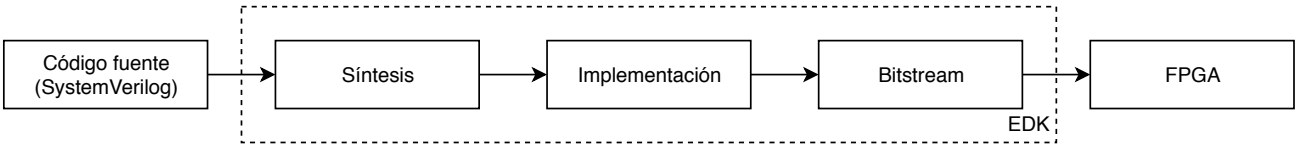


Fig. 1.1: Flujo de trabajo utilizando HDL.

Fig. 1.2: Flujo de trabajo propuesto.

Ahora, dicho trabajo sólo planteó el lenguaje de programación, faltando aún la evaluación de éste. Para llevar a cabo la evaluación, se requiere una plataforma que sea capaz de generar una descripción RTL a partir de código fuente en el lenguaje mencionado, el que se nombró SALTT (SALTT is A Language Translation Tool).

En este trabajo se construyó dicha plataforma, la que está constituida por un compilador, una serie de estructuras predefinidas y un set de instrucciones que sean capaces de crear un proyecto, agregarle bloques de propiedad intelectual (del inglés *Intellectual Property*, IP) y código fuente, e iniciar el proceso de síntesis, implementación y cargado a la tarjeta. Con esta plataforma, además de elevar el nivel de abstracción, se pretende cambiar el flujo de trabajo que se utiliza actualmente, que se muestra en la figura 1.1, simplificándolo para, además de abstraer al usuario del diseño *hardware*, abstraerlo del uso de un kit de diseño electrónico (del inglés *Electronic Design Kit*, EDK), cuyo manejo en ocasiones resulta engorroso. De esta forma, el nuevo flujo de trabajo utilizando la plataforma es el mostrado en la figura 1.2.

1.2. Trabajo previo

Como se mencionó anteriormente, el presente trabajo se basa en lo desarrollado por Araneda [2] en su tesis de magíster. En dicha tesis se propuso un modelo de cómputo y un lenguaje de programación para sintetizar lógica programable en un FPGA. Este lenguaje de programación se enfoca en el procesamiento de video infrarrojo, aunque no se limita a este tipo de imágenes.

En la tesis de Araneda, se analizaron una serie de trabajos desarrollados en el laboratorio de VLSI de la Universidad de Concepción, derivándose de éstos un modelo de cómputo a partir del cual se pudieran reproducir los resultados obtenidos en los trabajos analizados. Posteriormente, a partir del modelo de cómputo se planteó un lenguaje de programación que pudiera representar los trabajos estudiados. El lenguaje está compuesto por una serie de tipos de datos que representan

estructuras que se encontraron recurrentemente en todos los trabajos revisados. Finalmente se presentan propuestas de arquitecturas para algunas estructuras representadas por el lenguaje. La arquitectura presentada se centra en buses e interfaces.

Con respecto al trabajo futuro, Araneda menciona que el siguiente paso es desarrollar una herramienta que pueda transformar el lenguaje de programación a Verilog (SystemVerilog) o VHDL, para luego realizar una validación de los resultados obtenidos comparando factores como la utilización de recursos o consumo de potencia.

1.3. Estado del arte

Los FPGAs son dispositivos que permiten programar lógica a nivel de *hardware*. Esto hace que puedan cumplir con exigentes restricciones de tiempo de ejecución y consumo de potencia. Sin embargo, programar estas unidades no es una tarea trivial, debiendo poseerse conocimientos de diseño de *hardware*, lo que hace necesario dominar algún HDL, como VHDL o Verilog, aunque este último está quedando en desuso en favor de SystemVerilog, y algún EDK, como Vivado. Para simplificar la programación de los FPGAs, masificar su uso y aprovechar sus ventajas, se han propuesto diversos sistemas que realicen HLS, así como lenguaje de domino específico (del inglés *Domain-Specific Language*, DSL) con compiladores para FPGAs, con el fin de elevar el nivel de abstracción [3].

Actualmente hay diversos sistemas que realizan HLS. Algunos son propietarios mientras que otros son de código abierto. En [4] Daoud et. al. hace una revisión de diversos lenguajes y compiladores para HLS, separándolos entre HLS basados en programación C/C++, basados en lenguajes diferentes a los antes mencionados, basados en esquemáticos y basados en programación para GPU.

Por su parte, Nane et. al. [5] también hace una revisión y evaluación de herramientas de HLS para FPGA. Dentro de su trabajo clasifica las herramientas de 2 formas diferentes: Separándolas entre herramientas académicas y comerciales, y dependiendo del lenguaje de entrada, entre lenguajes genéricos y DSLs. De este trabajo, además de la revisión, es importante destacar que propone un sistema de evaluación de HLS, el que podría ser aplicado a lo que se está desarrollando para evaluar el desempeño final.

Además de lo presentado anteriormente, en este trabajo se investigó acerca de DSLs que operen sobre el dominio del procesamiento de imágenes y generen una salida para FPGAs.

A continuación se analizan las herramientas que se consideraron más relevantes para este trabajo, usando como principal criterio la similitud con lo que se pretende obtener. Posteriormente se analizan algunos DSLs.

1.3.1. Herramientas académicas

En el ámbito académico se ha hecho investigación y se han desarrollado herramientas de HLS, con el objetivo de elevar el nivel de abstracción y hacer más fácil el desarrollo de soluciones y modelos en hardware. Dentro de estas herramientas se analizaron las siguientes, siendo el principal criterio el operar en el dominio del procesamiento de imágenes y/o video, mismo dominio que se busca abordar. De todas formas se analizaron herramientas que trabajan en otros dominios.

La primera herramienta analizada es *Single-Assignment C* (SA-C) [6], desarrollada en la Universidad de Colorado. Esta herramienta acepta como entrada un subgrupo de instrucciones del lenguaje de programación C, y genera como salida código en el HDL VHDL, el que luego se sintetiza para ser implementado en un FPGA. Esta herramienta plantea una serie de restricciones al lenguaje C, pero a cambio permite la optimización y paralelización de algunas operaciones, como los ciclos *for*. Una de sus principales características es que las variables sólo se pueden asignar una vez. El énfasis de este lenguaje está en los arreglos y ciclos *for*, ya que se enfoca en procesamiento de imágenes y estas características son recurrentes en este tipo de procesamientos.

Otra herramienta analizada estudiada es *MATlab Compiler for Heterogeneous adaptive computing systems* (MATCH) [7], de la Universidad Northwestern. Ésta recibe como entrada código MATLAB y genera código VHDL o Verilog como salida. La herramienta fue diseñada para generar código de bajo nivel en sistemas heterogéneos. El proyecto fue transferido a la empresa AccelChip, que posteriormente fue comprada por Xilinx. El objetivo principal del proyecto era facilitar el desarrollo de sistemas heterogéneos, de modo que el programador sólo se enfocara en escribir una aplicación en lenguaje de alto nivel, en este caso MATLAB, despreocupándose de la programación a bajo nivel del FPGA.

También se analizó la herramienta *Riverside Optimizing Compiler for Configurable Circuits* (ROCCC) [8], de la Universidad de California. Esta herramienta utiliza un subgrupo de instrucciones de C como código de entrada y genera código VHDL como salida. El compilador no está diseñado para generar una arquitectura completa, sino para optimizar etapas críticas de un

sistema. Sus objetivos son mejorar el rendimiento, minimizar los accesos a memoria y minimizar el tamaño del circuito generado, razones que llevan a limitar las instrucciones de C permitidas. Este proyecto busca explotar el alto grado de paralelismo y la capacidad de implementar grandes *pipelines* a flujos de datos de los FPGAs, dejándole el control a un microprocesador.

Por último se analizó *LegUp* [9], de la Universidad de Toronto. Este proyecto recibe código en lenguaje C como entrada y genera código RTL como salida. La herramienta tiene la capacidad tanto de transformar un código completamente de C a hardware sintetizable, como de generar un sistema híbrido *software/hardware*. El compilador se basa en el *framework* LLVM. Una de sus ventajas es que no opera en un dominio específico, ni impone muchas restricciones al lenguaje de entrada. Sin embargo, sólo puede operar con FPGAs de la marca Intel (ex Altera), lo que limita su uso. Lo más destacable de esta herramienta es lo completa que está, aceptando programas en C sin restricciones, además de ser constantemente revisada, estando ya en su cuarto *release* público.

1.3.2. Herramientas comerciales

Dentro de las herramientas comerciales, destacan *Impulse CoDeveloper* [10] de Impulse Accelerated y *DK Design Suite* [11] de Mentor Graphics, ya que operan en el dominio de las imágenes y el *streaming*. *Impulse CoDeveloper* recibe como lenguaje de entrada Impulse-C, lenguaje basado en C. Aunque el lenguaje es de propósito general, por sus características resulta especialmente útil en el procesamiento de imágenes y *streaming*. *DK Design Suite* recibe como lenguaje de entrada Handel-C, un derivado de C con construcciones orientadas a diseño de hardware. La desventaja de esta herramienta está en que se necesita especificar detalles como dominios de tiempo y zonas paralelizables, lo que requiere conocimientos en diseño de hardware.

También se revisaron las herramientas ofrecidas por Cadence y Synopsys, *Stratus High-Level Synthesis* [12] y *Synphony Model Compiler* [13], ya que ambas son grandes compañías enfocadas en el diseño de hardware. La herramienta *Stratus High-Level Synthesis* cuenta con un ambiente integrado de diseño (del inglés *Integrated Design Environment*, IDE), recibe como entrada modelos en lenguaje C, C++ o SystemC, y genera implementaciones RTL. Además, se destaca que es posible usar esta herramienta para el diseño completo de un sistema en chip (del inglés *System on Chip*, SoC). Por su parte, *Synphony Model Compiler* se especializa en síntesis de alto nivel de DSPs, recibiendo como entrada diseños de MATLAB o Simulink. Posteriormente esta herramienta se puede conectar con las herramientas de síntesis de Synopsys como *Synplify*, lo que permite la implementación física del diseño. La mayor desventaja de estas

herramientas corresponde a los costos de las licencias, ya que las licencias de los productos de ambas empresas tiene muy elevados precios.

MATLAB tiene 2 plataformas de HLS [14], System Generator [15] y HDL Coder [16], los que generan implementaciones para FPGAs a partir de código en MATLAB, modelos Simulink o gráficos Stateflow. El primero utiliza una serie de bloques prediseñados para FPGAs Xilinx que se conectan entre sí para generar una arquitectura. La primera limitación de esto es que se restringe su uso a FPGAs de Xilinx y la segunda es que no se pueden implementar cosas que no se encuentren en la librería de bloques prediseñados. Por su parte, HDL Coder genera código en VHDL o Verilog, el que puede ser implementado en FPGAs de Intel o Xilinx.

Finalmente se revisaron las herramientas ofrecidas por Xilinx e Intel. Es importante analizar estas herramientas ya que las empresas que las ofrecen son los principales productores de FPGA a nivel mundial, por lo que son las herramientas oficiales para realizar HLS en sus respectivas FPGAs.

Xilinx ofrece la herramienta *Vivado High-Level Synthesis* [17], que recibe como entrada código en lenguaje C, C++ o SystemC, generando como salida código RTL específico para dispositivos Xilinx. Esta herramienta se presenta como una mejora gratuita de su EDK Vivado. Dentro de las ventajas mencionadas están una serie de librerías para diferentes tareas, rápida verificación del modelo propuesto a través de simulación del código C/C+, además de generar automáticamente simulaciones para el modelo en VHDL o Verilog.

Xilinx además ofrece herramientas de la familia SDx, donde se encuentra SDAccel [18] y SDSoC [19]. SDAccel permite hacer usar FPGAs como aceleradores, generando *hardware* a partir de código en C, C++ u OpenCL. Esta herramienta utiliza bloques prediseñados que se conectan entre sí para generar el *hardware*, los que se pueden obtener de las librerías incluidas en el programa o agregarse bloques obtenidos de otras fuentes, lo que incluye diseños creados por el mismo programador. Por su parte, SDSoC permite generar sistemas heterogéneos *hardware-software* a partir de código en C, C++ u OpenCL. La herramienta permite seleccionar las funciones que se van a acelerar por *hardware*, y, al igual que en el caso de SDAccel, usa bloques predefinidos para generar la arquitectura que se sintetiza.

Intel ofrece la herramienta *Intel HLS Compiler* [20], que recibe como entrada código en el lenguaje de programación C++ y genera como salida código RTL optimizado para FPGAs de Intel. Entre sus ventajas se encuentran el permitir simulaciones en C++, lo que acelera los tiempos de prueba del sistema, y verificación del modelo RTL generado contra lo propuesto en C++.

Además de la herramienta mencionada anteriormente, Intel cuenta otra herramienta de HLS, *Intel FPGA SDK for OpenCL* [21]. Esta herramienta permite utilizar OpenCL para generar una implementación en un FPGA.

1.3.3. Lenguajes de dominio específico

Con el fin de elevar el nivel de abstracción sin tener que lidiar con lenguajes de programación de propósito general se han planteado DSLs, los que se enfocan en operar en un solo dominio, proveyendo al programador de instrucciones y operaciones que se adecuen de mejor forma a sus necesidades, reduciendo con ésto el número de líneas necesarias para realizar una misma tarea. Una ventaja de los DSLs es que se pueden traducir a diferentes lenguajes, permitiendo compilarlos para diferentes plataformas. Son de especial relevancia para este trabajo DSLs que apunten al dominio del procesamiento de imágenes y que generen código sintetizable en FPGAs.

Un lenguaje de dominio específico es *Darkroom* [22]. Este opera sobre líneas de video, construyendo *pipelines* para el procesamiento. La decisión de operar sobre líneas de video evita usar memoria externa al chip, lo que por lo general resulta problemático y puede aumentar el tiempo de ejecución de una aplicación. El lenguaje permite 2 tipos de operaciones: *pointwise*, que opera pixel a pixel y *stencil*, que opera sobre una ventana (vecindad) para producir un pixel de salida. Como salida *Darkroom* puede generar código en Verilog que posteriormente se puede sintetizar con algún EDK, o código para una unidad central de procesamiento (del inglés *Central Processing Unit*, CPU), que permite hacer validaciones. El compilador de este lenguaje está escrito en Terra.

Otro DSL es *CAPH just Aint Plain HDL* (CAPH), propuesto en [23]. Este lenguaje se enfoca en operar sobre *streams* de procesamiento. La metáfora usada por este lenguaje es que los programas son redes por los que los datos fluyen por un canal, típicamente en colas tipo primero en entrar, primero en salir (del inglés *First In, First Out*, FIFO). El flujo de datos es unidireccional. Cuenta con un compilador con la capacidad de generar una salida en SystemC o VHDL.

Rathlin Image Processing Language (RIPL) es un DSL que utiliza estructuras base (esqueletos) para realizar el procesamiento de imágenes [24, 25]. Los esqueletos corresponden a diferentes algoritmos típicos de procesamiento de imágenes, diseñados como *pipelines*. Estos esqueletos se conectan entre sí formando algoritmos de procesamiento más complejos capaces de abordar un problema completo. Los esqueletos pueden operar sobre filas, columnas y ventanas. El flujo de

compilación de este lenguaje es: RIPL se transforma a un *dataflow graph*, el que se expresa utilizando el lenguaje CAL. Luego se utiliza un compilador existente para convertir de CAL a Verilog. Finalmente el código Verilog se sintetiza e implementa en un FPGA.

También se analizó el DSL PolyMage [26]. Este lenguaje genera *pipelines* optimizados para filtros puntuales, que operen sobre una ventana, de submuestreo y de sobremuestreo. Una de las ventajas ofrecidas es que las implementaciones generadas ocupan la mayor área posible, para explotar al máximo el paralelismo permitido por los FPGAs. El flujo de compilación del lenguaje es: El código en PolyMage se traduce a C++, el que se compila utilizando la herramienta Vivado HLS, lo que produce una implementación para FPGA.

Por último se estudio el lenguaje *Heterogeneous Image Processing Acceleration* (HIPA^{cc}) [27, 28]. Este DSL en un principio se diseño para generar código para GPU (CUDA, OpenCL o Rendersctipt) y hacia allá están orientadas las optimizaciones que hace, sin embargo, se adaptó para generar implementaciones en FPGA. Al igual que PolyMage, el flujo de compilación transforma de HIPA^{cc} a C++, lo que luego se implementa en un FPGA utilizando Vivado HLS. El compilador de este DSL está basado en LLVM. El lenguaje también permite compilarse para CPU, ya que está basado en C++ y se puede compilar directamente utilizando un compilador para este lenguaje, con la desventaja que el código no estará optimizado.

1.4. Compiladores

Ya que se propone un lenguaje de programación nuevo, es necesario interpretar el código fuente escrito en este lenguaje. Para lograr esto se propone diseñar un compilador que pueda traducir el lenguaje propuesto en [2]. Para este trabajo se propone generar como salida código en el HDL SystemVerilog y en el lenguaje de programación C.

Un compilador se puede dividir en 7 etapas [29]:

1. Análisis léxico: Se lee el texto (código fuente) de entrada y se separa en *tokens*. Cada uno de estos *tokens* corresponde a un símbolo del lenguaje, el que puede ser un identificador, un número, un tipo de dato, etc.
2. Análisis sintáctico: Los *tokens* obtenidos en la etapa anterior se ordenan y estructuran en un árbol sintáctico. Este árbol representa la estructura jerárquica del programa. En esta etapa se encapsulan las diferentes estructuras definidas en un lenguaje, como inicializaciones, asignaciones, declaración de funciones, llamado de funciones, etc.

3. Revisión de tipos: En esta etapa se revisa que el código cumpla con las reglas impuestas por el lenguaje de modo que no hayan violaciones, ya sea de asignaciones de tipos de datos que no corresponde, asignaciones de valores mayores a los permitidos, etc. También se revisa si hay variables declaradas pero no utilizadas, entre otras cosas relacionadas.

4. Generación de código intermedio: El programa se traduce a un lenguaje intermedio independiente del equipo. Este lenguaje se compone de instrucciones atómicas que sean más fáciles de traducir al lenguaje de máquina.

5. Localización de registros: Las variables se traducen a números. Cada uno de estos números representa un registro en el equipo objetivo.

6. Generación de código de máquina: El lenguaje intermedio se traduce a lenguaje *assembly* específico para el equipo objetivo.

7. Ensamblado y linkeo: El código *assembly* se traduce a una representación binaria y se asocian direcciones a las variables, funciones, etc.

Las primeras 3 etapas se conocen como *frontend*, mientras que las últimas 3 corresponden al *backend*. Todo lo anterior hace referencia a un compilador para procesadores de propósito general.

Por razones que se discuten más adelante, es de especial interés el *frontend*. Por esto se recopiló más información y se profundizó más en este aspecto.

1.4.1. Análisis léxico

Como se mencionó anteriormente, la primera etapa de un compilador consiste en realizar un análisis léxico del código fuente. El analizador léxico, o *lexer*, tiene la función de dividir el texto en *tokens*. Un *token* es una etiqueta con la que se rotulan combinaciones de caracteres.

Entre los *lexers* disponibles uno que destaca es Lex [30, 31]. Éste fue propuesto en 1975 por Mike Lesk y Eric Schmidt, está escrito en el lenguaje de programación C y es la base de *lexers* más modernos. Lex genera un programa que reconoce expresiones regulares en un texto de entrada y, a partir de estas expresiones regulares, divide el texto. Además de reconocer patrones en un texto de entrada, Lex permite hacer operaciones inmediatamente después de reconocer un patrón. Una de las ventajas de Lex es que está diseñado para que su salida se conecte con la

entrada del analizador sintáctico Yacc. Esto facilita el flujo de datos entre el analizador léxico y el sintáctico.

Además de Lex, para este trabajo se estudiaron otros 2 *lexers* como alternativas, Flex [32, 33] y el módulo *lex* de la librería PLY de Python [34].

Flex es una implementación de Lex que puede operar junto a los analizadores sintácticos Bison y Berkeley Yacc. Corresponde es una versión mejorada de Lex, siendo la principal mejora su velocidad, de ahí su nombre "Fast Lexical Analyzer Generator". Flex es de código abierto, disponible bajo la licencia de Berkeley, lugar donde se diseñó.

PLY es una librería de Python de código abierto que implementa el *lexer* Lex y el *parser* Yacc, lo que origina su nombre, "Python Lex Yacc". Con esta librería se puede implementar el *lexer* Lex utilizando la sintaxis y el manejo de datos de Python, sin embargo, por la misma razón, la sintaxis del *lexer* no es igual a la sintaxis de Lex.

1.4.2. Análisis sintáctico

Una vez completado el análisis léxico y dividido el código de entrada en *tokens*, se procede a hacer un análisis sintáctico del código fuente. Esta tarea es hecha por un analizador sintáctico, o *parser*, y su función es encontrar, agrupar y encapsular estructuras dentro del texto de entrada. Estas estructuras están formadas tanto por *tokens* como por otras estructuras, de modo que se forma una jerarquía que finalmente lleva a encapsular todo el texto bajo una única estructura.

Dentro de los *parsers* disponibles, uno que sobresale es Yacc [31, 35]. Este *parser* fue diseñado entre 1975 y 1978 por Stephen Johnson en los laboratorios de Bell. Se convirtió en un *parser* muy popular ya que la base teórica en la que se basa la hacía una herramienta confiable. Sin embargo, como se distribuía bajo una licencia restrictiva de Unix, se usaba principalmente en el área académica y en Bell. Por lo anterior, su principal efecto fue servir de base para nuevas implementaciones. Al igual que Lex, Yacc es un programa que que se escribe en C y puede realizar operaciones durante los encapsulamientos, al terminar el encapsulamiento y al terminar de analizar todos los *tokens* que recibe como entrada. Como se mencionó en la sección anterior, los *tokens* generados por Lex sirven de entrada para Yacc.

Junto con Yacc se analizaron otras 3 alternativas, que corresponde a implementaciones posteriores de Yacc, éstas son Berkeley Yacc [36], Bison [33, 37] y el módulo *yacc* de la librería PLY de Python [34].

Berkeley Yacc es una variante de Yacc desarrollada en la Universidad de California, Berkeley, por Robert Corbett. Entre sus ventajas se encuentran el ser más rápido que Yacc, además de mantener la compatibilidad con éste. Este *parser* se popularizó ya que además de ser más rápido que Yacc por utilizar mejores algoritmos, se distribuye con una licencia flexible de Berkeley, lo que elimina una de las barreras de entrada de Yacc.

Bison, por su parte, también es una variante de Yacc, desarrollado por Charles Donnelly y Richard Stallman. Bison es una herramienta que se integrada en el proyecto GNU y adapta el trabajo realizado por Corbett en Berkeley Yacc. En lo que respecta al presente trabajo, ambas herramientas, Berkeley Yacc y Bison, son bastante similares entre sí.

1.4.3. Revisión de tipos

La etapa de revisión de tipo se realiza una vez que se identificaron todas las estructuras presentes en el texto con el *parser*. Esto lo puede hacer el mismo *parser*, ya que como permite realizar operaciones una vez reconocida una estructura, se pueden hacer revisiones de las reglas cuando corresponda. Todos los *parsers* analizados cuentan con esa capacidad.

1.5. Discusión

Se han presentado una serie de herramientas y programas enfocadas en HLS, dentro de las cuales hay diferentes opciones: Algunas han sido desarrollados dentro del ámbito académico, mientras otras han sido desarrollado en el ámbito comercial. Algunos proyectos han sido abandonados, mientras que otros se continúan perfeccionando. De todas las herramientas y proyectos, hay herramientas con grandes capacidades como las desarrolladas por Xilinx, Intel o Cadence, sin embargo, su mayor desventaja es el precio de sus licencias, pero además, en el caso de Xilinx e Intel tienen la desventaja de ser específicas para sus FPGAs. En cuanto a las herramientas académicas, hay casos como *LegUp* que tiene un buen desempeño, sin embargo, su uso está limitado a chips de Intel. Los otros proyectos académicos que se enfocaban en un dominio específico son antiguos y en su mayoría fueron abandonados. Actualmente la academia se encuentra enfocada en desarrollar herramientas de HLS de propósito general.

Algo que tienen en común la mayoría de las herramientas estudiadas es que la entrada para éstas es código en el lenguaje de programación C/C++ o algún subgrupo de instrucciones de estos lenguajes, esto para poder abordar la mayor cantidad de dominios y/o hacer las herramientas

más atractivas a los programadores de aplicaciones, a quienes apuntan estas herramientas, asumiendo que estos programadores están familiarizados al lenguaje C. Esto último, sin embargo, pone una mayor carga al compilador, ya que debe pasar de programación serial a programación paralela, debiendo buscar aquellas partes que son paralelizables. El lenguaje SALTT [2], sin embargo, pese a parecerse a C, tiene componentes propios que lo hacen más adecuado al dominio que aborda.

Además de las herramientas de HLS, hay DSLs que generan implementaciones para FPGA, elevando el nivel de abstracción para los programadores. Algunas de las estudiadas, además, generaban código en algún lenguaje de programación, lo que permite hacer validaciones. En su mayoría, los lenguajes estudiados tienen compiladores que traducen desde el DSL a algún HDL y luego usan algún EDK para la síntesis e implementación, similar a lo que se pretende hacer con SALTT.

En lo concerniente al compilador necesario para traducir de un lenguaje a otro, como se mencionó anteriormente, no es necesario que cuente con todas las etapas de un compilador para procesador de propósito general, siendo mayormente útil el *frontend*. Esto se debe a que se pretende que la salida sea código en SystemVerilog, código que debe ser sintetizado a hardware por otra herramienta como *Vivado*. Bajo la suposición que el EDK encargado de la síntesis hará una optimización del código fuente, se puede obviar una etapa de optimización por parte del compilador y, de esta forma, el paso a un lenguaje intermedio se hace innecesario. Del *backend*, se hace uso de la etapas de localización de registros y generación de código de máquina, aunque en este caso en lugar de lenguaje de máquina se trata de generación de un archivo con código en lenguaje SystemVerilog.

Para implementar las etapas necesarias del compilador se necesita un *lexer* y un *parser*, ya que, entre los dos, pueden llevar a cabo tanto el *frontend* como las etapas útiles del *backend*. Para hacer las pruebas iniciales se decidió utilizar la librería PLY de *Python*, porque el manejo de datos de este lenguaje lo hace atractivo para realizar procesamiento de texto. Además, la sintaxis es similar a Lex y Yacc, por lo que, en caso de ser necesario, se puede portar el trabajo desarrollado sin mucha dificultad.

Para llevar a cabo la síntesis e implementación en FPGA se decidió utilizar la herramienta Vivado. Esta herramienta, pese a que cuenta con un IDE, se basa en la ejecución de instrucciones en lenguaje de comando para herramientas (del inglés *Tool Command Language*, TCL), por lo que el IDE no es necesario, pudiendo usarse Vivado en modo *shell*. Junto a esto, la herramienta permite ejecutar instrucciones TCL cargadas en una archivo [38]. De esta forma, se puede diseñar un archivo que cree un proyecto, agregue el código fuente correspondiente junto a los

IPs necesarios y luego lleve a cabo la síntesis e implementación del sistema creado, evitando así que el usuario tenga que realizar el proceso y, de esta forma, alejarlo aún más del diseño de bajo nivel.

1.6. Hipótesis

Ejecutando un programa con algunos parámetros, se pueden generar arquitecturas *hardware* eficientes a partir de diseños planteados en el lenguaje SALTT, que posteriormente se puedan implementar y ejecutar en un FPGA.

1.7. Objetivos

1.7.1. Objetivos generales

El objetivo de este trabajo es diseñar una plataforma que genere una arquitectura *hardware* a partir de diseños planteados en el lenguaje SALTT. Además de esto, la plataforma debe ser capaz de, por un lado, generar código fuente en lenguaje C para evaluar de manera preliminar el diseño, y, por otro lado, implementar la arquitectura generada en un FPGA debiendo el usuario pasar por pocos pasos para lograrlo.

1.7.2. Objetivos específicos

1. Diseñar un analizador léxico y sintáctico para el lenguaje de SALTT.
2. Traducir código fuente en SALTT a C.
3. Traducir código fuente en SALTT a SystemVerilog.
4. Elegir estructuras del lenguaje para su estandarización en SystemVerilog.
5. Diseñar módulos en SystemVerilog de las estructuras elegidas.
6. Diseñar una plataforma que implemente un diseño en un FPGA a partir de código fuente en SALTT.
7. Evaluar el desempeño de la plataforma.

1.8. Temario

En el presente informe se propone un modelo de cómputo, un lenguaje de programación, una arquitectura hardware, y un flujo de trabajo recomendado. Los siguientes capítulos están organizados como sigue:

- Capítulo 2: Presenta las modificaciones hechas al lenguaje de programación propuesto por Araneda[2]. Se usa la misma estructura que la mostrada en el trabajo original.
- Capítulo 3: Muestra como se diseñaron los analizadores léxico y sintáctico.
- Capítulo 4: Muestra como se llevó a cabo la generación de código en SystemVerilog a partir de código en SALTT.
- Capítulo 5: Muestra los resultados obtenidos al generar código utilizando la herramienta desarrollada en este trabajo.
- Capítulo 6: Resume el trabajo realizado, presenta las conclusiones, y propone el trabajo a futuro.

Capítulo 2: SALTT

Antes de presentar el trabajo hecho se hace necesario presentar SALTT, lenguaje planteado por Araneda, por lo que todo lo que se presenta a continuación se encuentra en su tesis de magíster [2].

Este lenguaje nace como resultado de un modelo de cómputo que buscaba estandarizar la programación de FPGAs y, entre otras cosas, facilitar la reutilización de códigos. Para lograr esto se definieron elementos y sus interfaces de entrada/salida, los que posteriormente fueron la base de los tipos de datos del lenguaje.

El lenguaje cuenta con 8 tipos de datos, 2 elementos para agrupar instrucciones, las que se llamaron unidades de ejecución, y elementos de control de flujo. A continuación se presentarán brevemente estos elementos para luego mostrar un código ejemplo en el que se pueda ver el lenguaje en uso.

2.1. Tipos de datos

SALTT define 8 tipos de datos, de los cuales uno representa un elemento de video, dos representan elementos de memoria, cuatro representan números y parámetros y el último representa eventos.

2.1.1. pix_stream

Este tipo de dato representa un *stream* de video y cuenta con 6 propiedades y 2 métodos, los que se usan para describir las características del *stream*.

Las propiedades de este tipo de dato son:

- res_h: Resolución horizontal, en pixeles.
- res_v: Resolución vertical, en pixeles.
- depth: Arreglo que contiene el tamaño en bits de cada una de las dimensiones del *stream*.

- data: Arreglo que contiene los datos del pixel actual.
- dim: Número de dimensiones del *stream*.
- new_data: Evento que indica que los datos se han actualizado.

Los métodos de este tipo de datos son:

- set_format(): Método que permite establecer valores de los miembros res_h, res_v y depth.
- copy_format(): Función que permitirá copiar el formato de otro **pix_stream**.

Como se ve de lo anterior, el *stream* se representa por su resolución horizontal y vertical, y las características de un pixel, esto es, sus dimensiones y la profundidad en bits de estas dimensiones.

2.1.2. Elementos de memoria

El lenguaje define **memory** y **memory_port**, 2 tipos de datos para representar elementos de memoria.

El primer elemento **memory** representa una memoria, la que puede ser interna o externa. Dado el dominio en que se trabaja, la memoria se define como bidimensional, para así poder acomodar directamente un cuadro de video. Esto no quita que se pueda configurar como unidimensional definiendo una de las dimensiones en 1 al momento de configurar este tipo de dato. Cuenta con 4 propiedades:

- width: Ancho en bits de los elementos contenidos.
- len1: Longitud de la primera dimensión.
- len2: Longitud de la segunda dimensión.
- port: Arreglo que contiene los puertos de comunicación.

Por su parte **memory_port** representa un puerto de comunicación. Este tipo de dato se definió para facilitar el acceso a diferentes memorias utilizando un misma instrucciones, cambiando entre memorias en tiempo de ejecución.

2.1.3. Números

SALTT cuenta con 4 tipos de datos para representar números: **raw_bits**, **number**, **unumber** y **parameter**. De estos, los primeros 3 son de ancho en bits variable, el que se configura en tiempo de compilación y permanece constante en tiempo de ejecución.

El primero, como su nombre lo sugiere, corresponde a la representación de un arreglo de bits, por lo que es un número entero sin signo de un largo configurable, su valor puede cambiar en tiempo de ejecución, pero no así su largo.

Los siguientes 2, **number** y **unumber**, representan números fraccionales de punto fijo. Ambos permiten configurar el ancho en bits de su parte entera y fraccional. La única diferencia entre ambos es que el primero es un número con signo y el segundo no.

Por el último el tipo de datos **parameter**, como su nombre lo indica, corresponde a un parámetro, por lo que se espera que su valor sea definido en tiempo de compilación y permanezca invariable en tiempo de ejecución. No contempla configurar su ancho en bits.

2.1.4. event

Por último, el tipo de dato **event** se usa para notificar la ocurrencia de un evento, lo que desencadena que se ejecuten instrucciones que se encuentren dicho evento.

2.2. Unidades de ejecución

SALTT cuenta con 2 unidades de ejecución, cuya función es agrupar instrucciones y exponer un interfaz de entrada/salida para que puedan interactuar con otros elementos. Una de las unidades de ejecución se llamó **hardware-thread** y la otro se llamó **filter**.

2.2.1. hardware-thread

Es un elemento de cómputo encargado de realizar una tarea específica. Los elementos se definen usando una sintaxis similar a las funciones en C, esto es, un nombre, seguido por una lista de argumentos entre paréntesis y seguido de un conjunto de instrucciones entre paréntesis

de llave. Los argumentos pueden ser entradas o salidas y están limitados a los siguientes tipos de datos: **event**, **raw_bits**, **number**, **unumber**, **parameter**, **pix_stream** y **memory_port**.

La forma de llamar este elemento es también similar a las funciones en C: Se escribe el identificador seguido de una lista de argumentos entre paréntesis, siendo estos asignados por posición. La única diferencia es que tienen es que este elemento no se asigna ya que las salidas son parte de elementos.

2.2.2. filter

Este elemento corresponde a una versión restrictiva de **hardware-thread**, ya que sólo permite los siguientes tipos de datos como argumentos: **pix_stream**, **memory_port** y **parameter**. Esto se hace para fomentar su reutilización. Además, impone que reciba como entrad y como salida un argumento de tipo **pix_stream**. En cuanto a su definición, la única diferencia con **hardware-thread** es que se debe anteponer la palabra clave **filter** al identificador, el resto, tanto la definición de argumentos como de las instrucciones que componen esta unidad de ejecución son iguales. El llamado también es igual.

Entre los **filter** es importante destacar aquel cuyo identificador es **top**, ya que este siempre debe estar presente y corresponde al **filter** que se comunicará con el exterior.

2.3. Control de flujo

Para el control de flujo el lenguaje define 3 sentencias, una sentencia condicional, una sentencia para la ejecución de un conjunto de instrucciones un número determinado de personas y una sentencia que se ejecuta al recibir un evento.

La sentencia condicional es de tipo *if - else if - else* y su sintaxis es similar al lenguaje C. La ejecución repetitiva es de tipo *for* y, al igual que el caso anterior, su sintaxis es similar al lenguaje C.

Por último, se definió una sentencia propia, **on_event**, que ejecuta un conjunto de instrucciones cada vez que se detecte un evento. Su sintaxis es la palabra clave **on_event** seguido por el identificador de un evento entre paréntesis y finalmente un conjunto de instrucciones entre paréntesis de llave. El evento dado como argumento es el que determina la ejecución de esta sentencia.

2.4. Ejemplo

El código 2.4 genera un patrón de 16 líneas horizontales cuyo valor está representado en escala de grises utilizando 8 bits y aumenta de forma constante entre líneas yendo del negro al blanco. En este ejemplo se muestran las principales características del lenguaje: La declaración de un **filter**, la declaración y asignación de variables, la configuración de variables de tipo **pix_stream** y el uso de las sentencia **on_event** y **set_event**.

Entre los elementos presentes destaca el **filter top**, cuya entrada es el **pix_stream** *input* y cuya salida es el textbfpix_stream *input*, con lo que se cumple el mínimo para un **filter** exijido por el lenguaje. También destaca el uso de la sentencia **on_event** que se ejecuta cada vez que el *stream* representado por *input* tiene un nuevo dato disponible.

```
1  filter top(pix_stream in_pix, pix_stream out_pix)
2  {
3      in_pix.set_format(res_h = 1280, res_v = 720, depth = {8, 8, 8});
4      out_pix.copy_format(in_pix);
5
6      raw_bits(8) value = 0;
7      raw_bits(6) count = 0;
8
9      on_event(in_pix.new_data){
10         out_pix.data[0] = value;
11         out_pix.data[1] = value;
12         out_pix.data[2] = value;
13
14         if(in_pix.pos_h == 0) {
15             if(count == 45) {
16                 value = value + 15;
17                 count = 0;
18             }
19             else
20                 count = count + 1;
21         }
22
23         set_event(out_pix.new_data);
24     }
25 }
```

Capítulo 3: Modificaciones al lenguaje SALTT

En este capítulo se muestran los cambios hechos a SALTT. Junto a los cambios se explican las razones que motivaron dichos cambios, los que pueden ser agregar o modificar un elemento.

La sintaxis de los cambios está descrita usando la notación Backus-Naur Form (BNF), tal como hizo Araneda[2].

3.1. Modificaciones a tipo de dato pix_stream

Pese a que este tipo de dato permite configurar la resolución horizontal y vertical del *stream* de datos que representa, no contemplaba miembros que permitieran determinar la ubicación espacial del pixel en un momento determinado. Para poder conocer esta información se definieron los siguientes miembros, los que se agregan a los ya definidos:

- `pos_h`: Posición horizontal, en pixeles. De tipo **raw_bits**, sólo lectura.
- `pos_v`: Posición vertical, en pixeles. De tipo **raw_bits**, sólo lectura.

Durante el desarrollo de la transformación desde SALTT a SystemVerilog, se notó que el lenguaje no disponía de un método para determinar el pixel inicial de un *stream* creado dentro de una unidad de ejecución. Para indicar esto se agregó la función **set_stream_start**, definida como:

set_stream_start_call ::= set_stream_start_keyword **(** pix_stream_identifier **) ;**
set_stream_start_keyword ::= **set_stream_start**

3.2. Modificación a tipo de dato memory

Este tipo de dato no explicitaba el tipo de memoria que representado, debiendo la herramienta elegir entre realizar el almacenamiento localmente utilizando *block RAMs* o almacenar

los datos externamente utilizando *DRAM*.

Dado que esto implica hacer análisis a las memorias como la cantidad de datos requeridos, la cantidad total de memoria interna y externa, la inmediatez de la lectura de un dato, entre otras cosas, se decidió traspasar esta decisión al programador, debiendo especificar si la memoria es local (**BRAM**) o externa (**DRAM**).

La sintaxis para definir una variable de tipo **memory**, luego de aplicado el cambio, queda:

```
memory_declaration ::= memory_type memory_ident_list ;
memory_type ::= memory_keyword ( mem_width , mem_len1 , mem_len2 , memory_type_keyword )
memory_ident_list ::= memory_identifier { , memory_identifier }
mem_width ::= non_zero_unsigned_number
mem_len1 ::= non_zero_unsigned_number
mem_len2 ::= unsigned_number
memory_identifier ::= single_identifier
memory_type_keyword ::=
          BRAM
        | DRAM
memory_keyword ::= memory
```

3.3. Modificación a tipo de dato memory_port

Dado que la sintaxis propuesta para acceder a una dirección de memoria era ambigua desde el punto de vista del analizador sintáctico, pudiéndose reconocer el acceso a un dato en memoria como un llamado a una unidad de ejecución, se decidió reemplazar los paréntesis por corchetes para indicar el elemento de la memoria al que se desea acceder. Con esto se evita todo tipo de ambigüedad, ya que ningún otro elemento utiliza dicha sintaxis.

Considerando el cambio anterior, el acceso al contenido de una memoria se realiza como:

```
memory_access ::= memory_port_identifier [ mem_idx1 , mem_idx2 ]
mem_idx1 ::=
          unsigned_number
        | unumber_identifier
mem_idx2 ::=
          unsigned_number
        | unumber_identifier
```

3.4. Modificación a unidades de ejecución

En la descripción original de SALTT, la dirección de los argumentos de una unidad de ejecución es inferida por la herramienta. Durante el desarrollo de la transformación de las unidades de ejecución, se notó que inferir la dirección no sería una tarea trivial, ya que ésta dependía de distintos factores.

Para evitar lidiar con estos problemas en una primera etapa de desarrollo se definieron dos nuevas palabras claves, **input** y **output**. Antes de escribir el tipo de dato de un argumento se debe escribir una de estas dos palabras. Con esto se traspasa la tarea de decidir la dirección de un argumento de la herramienta al programador.

Luego de aplicada la modificación planteada, la sintaxis de la definición de un **hardware-thread** queda como:

```
hw_thread_def ::= hw_thread_identifier ( hw_thread_def_arg_list ) compound_statement
hw_thread_def_arg_list ::= hw_thread_def_arg { , hw_thread_def_arg }
hw_thread_def_direction_arg ::= hw_thread_def_direction hw_thread_def_arg
hw_thread_def_direction ::=
          input
        | output
hw_thread_def_arg ::=
          event_type event_identifier
        | raw_bits_type raw_bits_identifier
        | number_type number_identifier
        | unumber_type unumber_identifier
        | parameter_type parameter_identifier
        | pix_stream_type pix_stream_identifier
        | memory_port_type memory_port_identifier
hw_thread_identifier ::= single_identifier
```

Por su parte, la nueva sintaxis de la definición de un **filter** es:

```
filter_def ::= filter filter_identifier ( filter_def_arg_list ) compound_statement
filter_def_arg_list ::= filter_def_direction_arg { , filter_def_direction_arg }
filter_def_direction_arg ::= filter_def_direction filter_def_arg
filter_def_direction ::=
          input
        | output
filter_def_arg ::=
          pix_stream_type pix_stream_identifier
        | memory_port_type memory_port_identifier
        | parameter_type parameter_identifier
filter_identifier ::= single_identifier
```

Pese a todo lo anterior no se descarta eliminar estas dos palabras claves en el futuro y poner la carga de determinar la dirección de un argumento en la herramienta, como se definió originalmente en la descripción de SALTT.

3.5. Modificación a elementos incorporados

A los elementos incorporados definidos inicialmente se agregó uno nuevo de tipo **filter**:

```
flt_dim_split(input pix_stream src, output pix_stream dst_1, ...,
    output pix_stream dst_n)
```

Este filtro divide un **pix_stream** de múltiples dimensiones en múltiples **pix_stream** de una dimensión. El número de argumentos dados como destino debe ser igual al número de dimensiones del origen y no deben haber sido configurados previamente, ya que la herramienta se encarga de eso. Sus argumentos son:

- `src`: Stream de entrada.
- `dst_1, ..., dst_`n: Streams de salida.

3.6. Sentencia init

SALTT permite asignarles un valor inicial a una variable inicializándola en su declaración, lo que se ilustra en el código 3.1:

Código 3.1: Ejemplo de inicialización.

```
raw_bits(8) foo = 5;
```

Sin embargo, esto no permite indicar el valor inicial de una variable que forme parte de los argumentos de una unidad de ejecución, lo que puede resultar necesario en argumentos de salida.

Para agregar esta funcionalidad se definió una nueva sentencia que le indica a la herramienta el valor inicial que debe asumir una variable dada. Su sintaxis es:

```
init_statement ::= init assignment_statement
```

El ejemplo que sigue muestra esta sentencia utilizada para asignar un valor a un argumento de salida de un **hardware-thread**:

```
dummy_ht(input number(3,4) foo_in, output raw_bits(6) foo_out)
{
    init foo_out = 26;
    ...
}
```

Capítulo 4: Analizador léxico y sintáctico

En este capítulo se presenta el diseño de los analizadores léxico y sintáctico. El desarrollo de ambos se llevó a cabo utilizando las herramientas de análisis sintáctico y léxico de la librería PLY de Python, presentada previamente.

4.1. Lexer

De la investigación realizada se tiene que el *lexer* tiene la función de dividir y etiquetar un flujo de caracteres dados como entrada. El analizador se diseñó de tal manera que sea capaz de reconocer las siguientes etiquetas:

- **ID**: Corresponde a un identificador. Puede estar compuesto por letras, números y o el carácter guion bajo (_). El primer carácter debe ser un letra, mayúsculas o minúscula.
- **DEC_NUMB**, **HEX_NUMB**, **BIN_NUMB**, **FRAC_NUMB**: Corresponden a números en formato decimal, hexadecimal, binario y fraccional, respectivamente.
- **Símbolos**: Corresponden a caracteres que no son letras ni números pero están permitidos por el lenguaje. En su mayoría corresponden a operadores, sin embargo algunos son delimitadores. Son individualizados ya que múltiples estructuras del analizador sintáctico dependen de su presencia. El *lexer* se diseñó para reconocer los siguientes símbolos:

&	==	[	>>
&&	!=	]	<<
|	>	(	++
||	>=	)	−−
;	<	−	=
,	<=	+	~
:	{	*	!
.	}	/	?

- **Palabras reservadas**: Corresponden a combinaciones de letras que pueden representar tipos de datos o instrucciones del lenguaje. Se individualizan para evitar que sean etiquetadas como identificadores, lo que simplifica el encapsulamiento de estructuras durante el análisis sintáctico, además de prevenir su uso como identificadores. El *lexer* se diseñó para reconocer las palabras reservadas:

and	**memory**	**pix_stream**
cast	**memory_port**	**raw_bits**
else	**memory_port_mux**	**round**
event	**number**	**set_event**
filter	**num_approx**	**set_stream_start**
for	**on_event**	**top**
if	**or**	**unumber**
init	**output**	**unum_approx**
input	**parameter**	

4.2. Parser

Una vez definidas las etiquetas reconocibles por el *lexer*, se procede a diseñar el analizador sintáctico. Éste recibe las etiquetas definidas previamente y la encapsula en estructuras, las que pueden estar formadas por etiquetas, otras estructuras o una combinación de ambas opciones. Las estructuras definidas en el *parser* darán paso a la generación de código en SystemVerilog, etapa final de la herramienta.

El analizador sintáctico se diseñó de tal forma que genera un árbol jerárquico, el que se muestra en la figura 4.1. Por simplicidad, sólo se muestran las estructuras principales.

A continuación, se muestran las estructuras definidas en el *parser* y que finalmente forman el árbol jerárquico del analizador sintáctico del lenguaje. Estas estructuras se basan en lo definido por Araneda[2].

Para mostrar la conformación de las estructuras se utiliza la siguiente nomenclatura:

- A la izquierda se muestra el nombre de la estructura. Empieza con una letra mayúscula.
- Las palabras en negrita y en mayúsculas a la derecha corresponden a etiquetas.

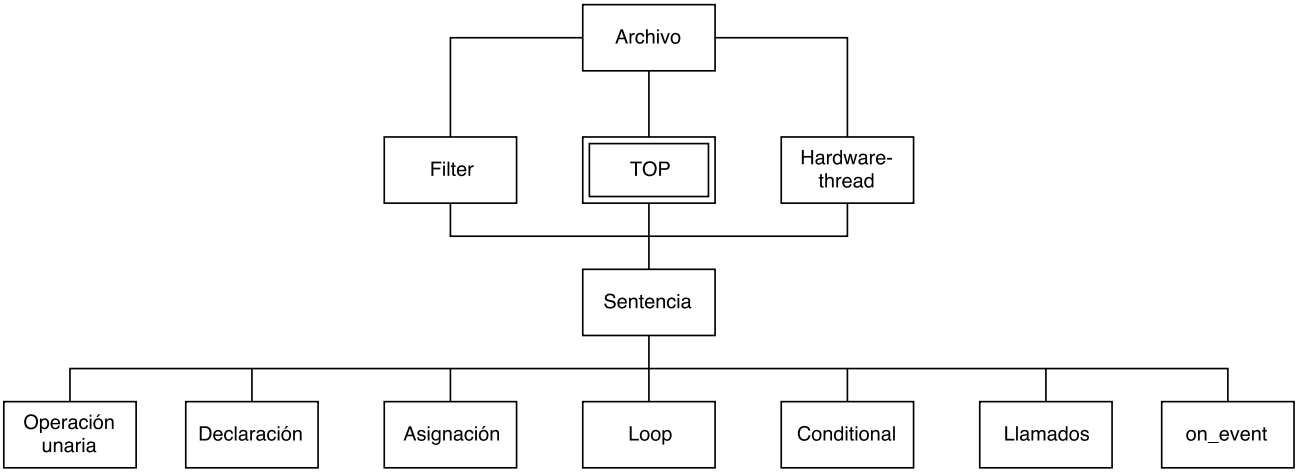


Fig. 4.1: Árbol jerárquico del lenguaje.

- Las palabras en negrita a la derecha iniciadas con una letra mayúscula corresponden a otras estructuras.
- Las palabras en negrita a la derecha iniciadas con una letra minúscula corresponden a palabras clave.
- Los símbolos se representan a si mismos.
- Las palabras en cursiva corresponden a un uso recursivo de la estructura que se está definiendo.

4.2.1. Número

Para agrupar las diferentes formas en que se puede presentar un número se define la siguiente estructura:

Número:	**DEC_NUMB**
	HEX_NUMB
	BIN_NUMB
	FRAC_NUMB

4.2.2. Variable

Agrupa las diferentes formas en que una variable se puede usar. De acuerdo a la definición de SALTT, una variable puede ser llamada por su nombre, se puede obtener una *slice* de ésta

acompañándola de corchetes, o se puede obtener una propiedad o llamar un método utilizando el símbolo "**.**" si la variable es de tipo **pix_stream** o **memory**. Se define como:

Variable:	**ID**
	ID + [+ **DEC_NUMB** +]
	ID + [+ **DEC_NUMB** + **:** + **DEC_NUMB** +]
	ID + [+ **DEC_NUMB** +] + [+ **DEC_NUMB** + **:** + **DEC_NUMB** +]
	ID + **.** + *Variable*

4.2.3. Tipo de dato

Esta estructura agrupa todas las formas en que se puede hacer referencia a un tipo de datos. Puede estar compuesto sólo de la palabra clave que representa el tipo de dato o estar acompañada de alguna característica del tipo de dato correspondiente. La estructura se define como:

Tipo de dato:	**event**
	raw_bits + (+ **DEC_NUMB** +)
	unumber + (+ **DEC_NUMB** + , + **DEC_NUMB** +)
	number + (+ **DEC_NUMB** + , + **DEC_NUMB** +)
	parameter
	memory + (+ **DEC_NUMB** + , + **DEC_NUMB** + , + **DEC_NUMB** + , + **ID** +)
	memory_port
	pix_stream

4.2.4. Declaración e inicialización

Una declaración corresponde a la primer mención que se hace de una variable en un código. Si la variable recibe un valor inicial la declaración pasa a ser una inicialización.

Tanto la declaración como la inicialización de variables se encapsulan como una misma estructura, ya que una inicialización implica una previa declaración y además ambos eventos pueden ocurrir en una misma línea de código. Por simplicidad se nombró a la estructura **Declaración de variable**. Se define como:

Declaración de variable:	**Tipo de dato** + **ID**
	Tipo de dato + **ID** + [+ **DEC_NUMB** +]
	Tipo de dato + **ID** + = + **Variable**
	Declaración de variable + , + **ID**
	Declaración de variable + , + **ID** + [+ **DEC_NUMB** +]
	Declaración de variable + , + **ID** + = + **Variable**

Notar que se incluye la declaración de arreglos, pero no su inicialización ya que el lenguaje no permite que éstos sean inicializados.

4.2.5. Conversión de tipo de datos

Función que modifica la interpretación de una variable, o bien, fuerza a que un número se interprete como un tipo determinado de dato. Por simplicidad se usó la estructura **Tipo de dato**, aunque sólo se permiten los tipos **raw_bits**, **number** y **unumber**. Se define como:

Conversión de tipo:	**cast** + (+ **Tipo de dato** + , + **Variable** +)
	cast + (+ **Tipo de dato** + , + **Número** +)

4.2.6. Aproximación

Funciones que expresan un número cualquiera como un número de punto fijo. El resultado se interpreta como una variable de tipo **number** o **unumber** dependiendo de la función utilizada. Se define como:

Aproximación:	**num_approx** + (+ **Número** + , + **DEC_NUMB** + , + **DEC_NUMB** +)
	num_approx + (+ **Variable** + , + **DEC_NUMB** + , + **DEC_NUMB** +)
	unum_approx + (+ **Número** + , + **DEC_NUMB** + , + **DEC_NUMB** +)
	unum_approx + (+ **Variable** + , + **DEC_NUMB** + , + **DEC_NUMB** +)

4.2.7. Redondeo

Función que aproxima el número de una variable de tipo **number** o **unumber** al entero más cercano. Se define como:

Redondeo:	**round** + (+ **ID** +)

4.2.8. Función de conversión

Esta estructura agrupa las funciones utilizadas para llevar a cabo las conversiones de tipo en el lenguaje. Se define como:

Función de conversión:	**Conversión de tipo**
	Aproximación
	Redondeo

4.2.9. Expresión unaria

Esta estructura agrupa las expresiones unarias permitidas por el lenguaje. Se define como:

Expresión unaria:	$\sim$	+	**Variable**
	!	+	**Variable**
	$+$	+	**Variable**
	$-$	+	**Variable**
	$++$	+	**Variable**
	$--$	+	**Variable**
	Variable	+	$++$
	Variable	+	$--$

4.2.10. Expresión binaria

Esta estructura agrupa las expresiones binarias permitidas por el lenguaje. Para la definición se hace uso de la estructura **Expresión**, cuya definición se mostrará más adelante.

Expresión binaria:	**Expresión**	+	+	**Expresión**
	Expresión	−	+	**Expresión**
	Expresión	*	+	**Expresión**
	Expresión	/	+	**Expresión**
	Expresión	&	+	**Expresión**
	Expresión	|	+	**Expresión**
	Expresión	&&	+	**Expresión**
	Expresión	||	+	**Expresión**
	Expresión	==	+	**Expresión**
	Expresión	!=	+	**Expresión**
	Expresión	>	+	**Expresión**
	Expresión	>=	+	**Expresión**
	Expresión	<	+	**Expresión**
	Expresión	<=	+	**Expresión**
	Expresión	>>	+	**Expresión**
	Expresión	<<	+	**Expresión**

4.2.11. Expresión

Esta estructura agrupa las diferentes expresiones que el lenguaje permite que se asignen a una variable. Se define como:

Expresión:	**Variable**
	Número
	Expresión unaria
	Expresión binaria
	Función de conversión

4.2.12. Asignación

Las asignaciones corresponden a cambios al valor de una variable. Esta estructura se define como:

Asignación:	**Variable**	+ = +	**Expresión**

4.2.13. Loop

Una sentencia **Loop** itera sobre una o un conjunto de instrucciones un número determinado de veces. Se utiliza la palabra clave **for** para representar un iterador. A diferencia de otros lenguajes, SALTT sólo establece un tipo de iterador. Se define como:

Loop:	**for** + (+ **Asignación** + **;** + **Expresión** + **;** + **Asignación** +) + **Sentencia**
	for + (+ **Asignación** + **;** + **Expresión** + **;** + **Expresión unaria** +) + **Sentencia**

4.2.14. Condicional

Una sentencia **Condicional** permite ejecutar un conjunto de instrucciones si se cumple una condición. El lenguaje soporta las sentencias **if** - **else**, siendo la última opcional. Se define como:

Condicional:	**if** + (+ **Expresión** +) + **Sentencia**
	Condicional + **else** + **Sentencia**

4.2.15. Llamados

El lenguaje posee palabras reservadas que son utilizadas para identificar funciones propias del lenguaje. Además, se pueden llamar **Hardware-threads** y **Filters** que hayan sido definidos con anterioridad, siendo la sintaxis de este llamado similar al de las funciones de sistema. Considerando lo anterior un llamado se define como:

Llamado:	**Función del lenguaje** + (+ **Argumentos de llamado** +)
	ID + (+ **Argumentos de llamado** +)

donde,

Función del lenguaje:	**set_event**
	ID.set_format
	ID.copy_format
	memory_port_mux
	set_stream_start
Argumentos de llamado:	**Expresión** + **,** + **...** + **,** + **Expresión**

Notar que un argumento puede ser uno o un conjunto de expresiones, lo que está limitado por las cantidad de argumentos permitidos por la función o unidad de ejecución que se está llamando.

4.2.16. on_event

El lenguaje permite ejecutar ciertas acciones solamente después de ocurrido algún evento. Esto se realiza utilizando la sentencia **on_event**, cuya estructura se define como:

on_event:	**on_event** + (+ ID +) + **Sentencia**
	on_event + (+ ID + **and** + ID +) + **Sentencia**
	on_event + (+ ID + **or** + ID +) + **Sentencia**

4.2.17. init

El lenguaje dispone de una sentencia que permite asignar un valor inicial a una variable de forma posterior a su declaración, la que se puede usar cuando no es posible o no se desea hacer esta acción utilizando una inicialización. La estructura que encapsula esta sentencia se define como:

init:	**init** + **Asignación**

4.2.18. Sentencia

Una sentencia corresponde a una o un conjunto de las estructuras vistas anteriormente. Su principal función es agrupar otras estructuras. Es importante destacar que, a excepción de **Bloque de sentencias**, todas las sentencias deben terminar con "**;**", ya que este símbolo delimita el fin de una sentencia. Se define como:

Sentencia:	**Declaración de variable** + **;**
	Expresión unaria + **;**
	Asignación
	Loop + **;**
	Condicional + **;**
	Llamados + **;**
	on_event + **;**
	initial + **;**
	{ + **Bloque de sentencias** + **}**

4.2.19. Bloque de sentencias

En la mayoría de los casos se escribirán múltiples sentencias seguidas que deben ejecutarse en conjunto. Para encapsular este comportamiento se define la siguiente estructura:

Bloque de sentencias:	**Sentencia**
	Bloque de sentencia + **Sentencia**

4.2.20. Filter y Hardware-thread

El lenguaje no permite definir funciones, sin embargo, permite la definición de unidades de ejecución, las que pueden ser **filters** o **hardware-threads**. La sintaxis de ambas es similar, siendo la única diferencia la palabra clave **filter** que se antepone al definir una unidad de ejecución **filter**. En términos de operación, se diferencian en el tipo de datos permitidos como argumentos: Los **filters** permiten un subconjunto de los tipo de datos permitidos por **hardware-threads**. Considerando lo anterior, la estructura para cada uno se define como:

Filter:	**filter** + ID + **(** + **Argumentos de definición** + **)** + **Sentencia**
Hardware-thread:	ID + **(** + **Argumentos de definición** + **)** + **Sentencia**

donde,

Argumentos de definición:	**Tipo de dato** + ID + **,** + **...** + **,** + **Tipo de dato** + ID

4.2.21. Unidades de ejecución

Esta estructura agrupa la definición de **filters** y **hardware-threads**. Se define como:

Unidad de ejecución:	**Filter**
	Hardware-thread

4.2.22. Top

El lenguaje establece un filtro especial, **top**, que siempre debe estar presente. Por esta razón **top** se considera una palabra clave y tiene una etiqueta propia. Por lo mismo, se encapsula de forma separada de otros **filters**. Se define como:

Top:	**filter** + **top** + (+ **Argumentos de definición** +) + **Sentencia**

4.2.23. Archivo

Finalmente, la estructura **top**, junto con la definición opcional de **filters** y **hardware-threads**, componen el archivo completo. De esta forma, la estructura de mayor jerarquía del *parser* se define como:

Archivo:	**Top**
	Unidad de ejecución + **Top**

Para que el *parser* termine correctamente, al recibir el último *token* del *lexer* debe ser capaz de encapsular todo como un **Archivo**.

Capítulo 5: Conversión

Una vez definidas todas las estructuras reconocibles durante el análisis sintáctico, corresponde el proceso de revisión de tipos, donde se comprueba que el código cumpla con las reglas definidas por el lenguaje. Sin embargo esta etapa se omitió asumiendo que el código a procesar cumple con las reglas impuestas por el lenguaje, por lo tanto, se procedió a desarrollar la generación de código en SystemVerilog a partir del código fuente en SALTT, a partir de lo que se pueden hacer análisis del lenguaje y la herramienta.

Como ya se mencionó, la librería PLY de Python permite realizar acciones cada vez que se encapsula una estructura. Se hace uso de esta capacidad de la librería para generar el código en SystemVerilog o guardar información importante a medida que se van encapsulando las estructuras.

Para llevar a cabo la conversión, se definieron dos variables globales y una librería, como se muestra en la figura 5.1. La primera variable global corresponde a un diccionario llamado *identifiers*, que almacena todas las variables definidas dentro de una unidad de ejecución; la segunda variable, la lista *module*, almacena el código en SystemVerilog generado separado en secciones.

Para que el código generado sea ordenado, y ya que SystemVerilog lo permite, cada unidad de ejecución se transforma a un módulo en SystemVerilog, el que se escribe en un archivo. Esto implica que las variables globales *identifiers* y *module* deben restablecer sus valores cada vez que se inicia el procesamiento de una unidad de ejecución, lo que a su vez simplifica la verificación de identificadores.

Para explicar como se llevó a cabo la conversión a SystemVerilog se muestra el modo en que se agrega un elemento al diccionario *identifiers* y, luego, como se llenan cada una de las secciones que forman la variable *module*, las que se listan a continuación:

- Declaración del módulo.
- Declaración de parámetros locales.
- Declaración de variables internas.
- Instanciación de otro módulos.

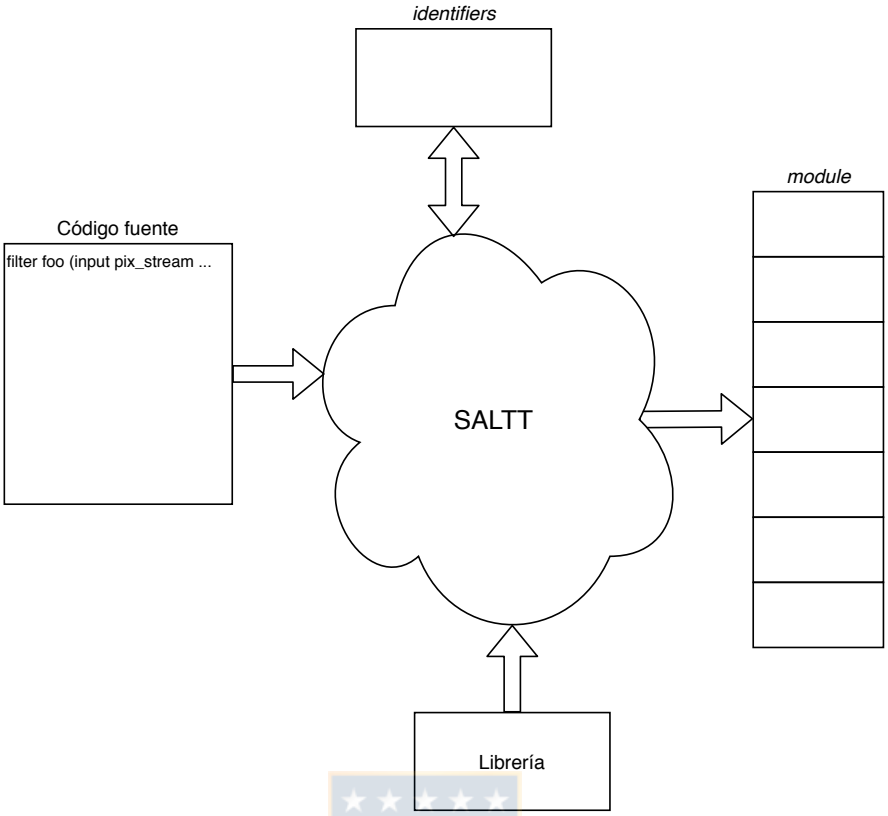


Fig. 5.1: Diagrama de funcionamiento de herramienta

- *always_comb*.
- Resets.
- Procesamiento.

5.1. identifiers

Mantener un registro de los identificadores presentes en el código que se procesa resulta necesario, ya que estos están asociados a características a las que se debe ser capaz de acceder al momento de utilizar dicho identificador. El lenguaje cuenta con dos mecanismos para agregar un nuevo identificador: Mediante la declaración de una variable o la declaración de una unidad de ejecución, en esto útlimo agregando tanto la unidad de ejecución misma como sus argumentos.

Para estandarizar la información disponible en el diccionario se creó una clase que represente un identificador, la que dispone de los siguientes campos para almacenar datos:

- Tipo del identificador: Este puede ser un tipo de dato o una unidad de ejecución.

- Entrada o salida: En caso de que el identificador sea el argumento de una unidad de ejecución, este campo indica si este argumento es de entrada o salida. En el resto de los casos este campo permanece vacío.
- Cuenta: Este campo indica cuantas veces se ha asignado el identificador previamente.
- Valor inicial: Este campo indica el valor inicial dado al identificador.
- Parámetros: En caso que el identificador tenga asociada alguna características, como su largo, ésta es almacenada en este campo.

5.2. module

Para poder editar múltiples partes del código generado de forma consecutiva se creó una lista que unida resulta en un módulo en SystemVerilog equivalente al código fuente en SALTT. Con el fin de mostrar como se generan cada una de las secciones definidas se utilizará el código del anexo A.

5.2.1. Declaración de módulo

Al encapsular el encabezado de la definición de una unidad de ejecución (Líneas 1 y 2 del código A.1), se guardan los argumentos de esta unidad de ejecución en el diccionario *identifiers*, indicando en el campo correspondiente si el argumento corresponde a una entrada o una salida.

Al encapsularse una unidad de ejecución completa, lo que ocurre al terminar de analizar la línea 29, se transforman todos los elementos presentes en el diccionario *identifiers*, como se muestra en esta sección y las 3 siguientes. La transformación se hace de acuerdo al tipo, parámetros, valor y dirección asociados a cada identificador. En esta sección sólo se transforman los identificadores que tengan una dirección asignada.

A continuación se muestran tanto las líneas que se usaron del código fuente en SALTT como el resultado generado:

Código 5.1: Declaración de módulo en SALTT

```
filter top ( input pix_stream in1, input raw_bits(8) in2, parameter par1,
    output pix_stream out1, output raw_bits(4) out2 ) {
```

Código 5.2: Declaración de módulo en SystemVerilog

```
module example
#(
    parameter par1 = 0
) (
    input logic in1___new_data,
    input logic in1___reset,
    input logic in1___last,
    input logic [7:0] in1___data0,
    input logic [7:0] in1___data1,
    input logic [3:0] in1___data2,
    input logic [7:0] in2,
    output logic out1___new_data,
    output logic out1___reset,
    output logic out1___last,
    output logic [7:0] out1___data0,
    output logic [7:0] out1___data1,
    output logic [3:0] out1___data2,
    output logic [3:0] out2,
    input logic clock,
    input logic reset
);
```

En el código generado destaca la transformación de “in1” y “out1”, ya que a partir de un solo identificador se generan múltiples señales. Esto se debe a que ambos identificadores son de tipo **pix_stream**, tipo asociado a señales de control (new_data, reset y last), además de una señal de datos por cada canal. Esta última información se extrae de la línea 11 y 12 del código A.1.

Al definir un parámetro, este se inicializa en 0, ya que SystemVerilog exige que los parámetros estén inicializados. Se eligió el valor 0 puesto que frecuentemente se espera que un parámetro tenga un valor distinto de 0, por lo que se puede considerar un valor inválido.

Al final se agregan las señales “clock” y “reset”, las que no estaban presentes en la declaración en SALTT. La señal “clock” se agrega bajo la asunción que el sistema generado es secuencial, razón por la que necesita un reloj para operar. La señal “reset” se agrega para llevar los valores

a un valor inicial conocido cuando sea necesario.

Por último, se puede notar que pese a que la unidad de ejecución tiene como identificador "top", el nombre del módulo es "example". Esto se debe a que se reserva la palabra "top" para el módulo que interactuará con el exterior (el módulo con la más alta jerarquía al sintetizar). El nombre "example" que se utiliza para nombrar el módulo generado de mayor jerarquía corresponde al nombre del archivo que contiene el código fuente. Esto se aplica sólo a la unidad de ejecución "top", las otras unidades de ejecución respetan el nombre asignado por el usuario.

5.2.2. Declaración de parámetros locales

En caso que un parámetro sea definido dentro de una unidad de ejecución, éste se traduce como un "localparam" de SystemVerilog. Además de los parámetros, en esta sección se encuentran las dimensiones horizontal y vertical de un **pix_stream**, las que de acuerdo a la definición del lenguaje no pueden ser modificadas en tiempo de ejecución. Pese a que las dimensiones pudieron haberse transformado como parámetros, considerando que "in1" y "out1" son argumentos de entrada y salida, se decidió que esta información se manejaría de manera interna por simplicidad. Por esta razón, cada vez que se declara un **pix_stream** se debe configurar, a menos que alguna función del sistema o incorporada lo haga.

Esta sección se genera a partir de la información recopilada de las líneas 5, 11 y 12 del código A.1, las que se muestran en el código 5.3. La salida generada en SystemVerilog se muestra en el código 5.4:

Código 5.3: Parámetros locales en SALTT

```
parameter local_par = 35;
 ...
in1.set_format(res_h = 1280, res_v = 720, depth = {8, 8, 4});
out1.copy_format(in1);
```

Código 5.4: Transformación de parámetros locales

```
localparam local_par = 35;
localparam in1___res_h = 1279;
localparam in1___res_v = 719;
localparam out1___res_h = 1279;
localparam out1___res_v = 719;
```

La función **set_format**, llamada en la línea 11, guarda la información entregada como argumento en la entrada correspondiente del diccionario para poder ser transformada correctamente.

5.2.3. Variables internas

Las variables declaradas dentro de una unidad de ejecución se transforman de forma similar que los argumentos, con la salvedad que en este caso no se tiene una dirección. Esto no incluye los parámetros, que fueron cubiertos anteriormente como parámetros locales.

A partir de la información presente en las líneas 1 a la 4, 6, 7, 11 y 12 del código A.1, mostradas en el código 5.5, se genera lo mostrado en el código 5.6:

Código 5.5: Declaración de variables en SALTT

```
filter  top ( input pix_stream in1, input raw_bits(8) in2, parameter par1,
    output pix_stream out1, output raw_bits(4) out2 ) {
raw_bits(8) var1 = 10, var2, var3;
unumber(3,2) var4;
 ...
memory(8, 16, 1, BRAM) mem;
memory_port mem_port1 = mem.port[0];
 ...
in1.set_format(res_h = 1280, res_v = 720, depth = {8, 8, 4});
out1.copy_format(in1);
```

Código 5.6: Transformación de variables locales

```
logic [10:0] in1___pos_h;
logic [9:0] in1___pos_v;
logic [10:0] out1___pos_h;
logic [9:0] out1___pos_v;
logic [7:0] _out1___data0_0;
logic [7:0] _out1___data1_0;
logic [7:0] _out1___data2_0;
logic [3:0] _out2_0;
logic [7:0] var1, _var1_0;
logic [7:0] var2;
logic [4:0] var3, _var3_0;
logic [4:0] var4;
logic mem___wea;
logic [3:0] mem___addra;
logic [7:0] mem___dina, mem___douta;
```

Pese a que internamente no se declaró ninguna variable de tipo **pix_stream**, se generan señales internas asociadas a identificadores de tipo **pix_stream**. Estas señales, "pos_h" y "pos_v", se encarga de indicar la posición actual del pixel dentro del cuadro representado por **pix_stream**. Esto se hace de forma interna ya que de otra forma la dirección que deben estar estas variables dependería de donde se configure el formato, debiendo determinarse la dirección para cada caso. El inconveniente de esta determinación es que se va a replicar lógica puesto que la posición se va a calcular en cada módulo de forma propia. La información del ancho de ambas señales se extrae de la información presente en las líneas 11 y 12.

La transformación del tipo **memory** consiste en generar variables tal que cada señal de la interfaz de memoria tenga una variable asignada, independiente de si esa señal se utiliza o no. La última letra de las variables generadas a partir de memoria, en el código 5.6 la letra "a", corresponde al puerto correspondiente de la interfaz. Por cada **memory_port** que se asocia a una variable **memory** se agrega un nuevo puerto. En el caso de la memoria BRAM, el número de puertos está limitado por la arquitectura del FPGA.

También destaca del código generado que en una misma línea se declaran variables con nombres similares. Las variables que inician con un guion bajo son variables auxiliares que se usan para asignar expresiones. Esto se profundiza más adelante. Las variables auxiliares siempre

son internas.

5.2.4. Instanciación de otros módulos

Al igual que los módulos de SystemVerilog, las unidades de procesamiento no contemplan valores de retorno, por lo que la transformación de los llamados de unidades de ejecución se hace de forma directa, instanciando un módulo cada vez que se encapsula un llamado a una unidad de ejecución.

Por otro lado, las memorias en SystemVerilog corresponderán a IPs, razón por la cual cada vez que se declare una se debe crear una instancia del IP correspondiente para poder utilizarla.

Esta sección se genera a partir de la información recopilada en las líneas 6 y 16 del código A.1, las que se muestran en el código 5.7. La salida generada en SystemVerilog se muestra en el código 5.8:

Código 5.7: Llamados en SALTT

```
memory(8, 16, 1, BRAM) mem;
 ...
my_ht( in1, var4 );
```

Código 5.8: Transformación de instanciación de módulos

```
BRAM_8_16_1 mem(
    .clka(clock),
    .wea(mem___wea),
    .addra(mem___addra),
    .dina(mem___dina),
    .douta(mem___douta),
);

my_ht my_ht0 (in1, var4, clock, reset);
```

Para la instanciación de la memoria, como se mencionó anteriormente, se conecta cada señal a una variable interna. El nombre dado al IP está determinado por las configuración dada a la variable **memory**, lo que permite reutilizar el mismo IP.

La instanciación de las unidades de procesamiento se hace conectando las señales por orden de declaración. Esto facilita el pasó de señales ya que éstas se pasan directamente, principalmente porque SALTT sólo permite el paso de argumentos por posición.

5.2.5. always_comb

Con todo lo anterior el módulo posee todas las variables y señales necesarias para realizar procesamiento. En esta sección se expresan las operaciones que el módulo debe llevar a cabo.

Todas las asignaciones expresadas en SALTT tienen un equivalente combinacional, sin embargo, no todas tiene un equivalente secuencial, dependiendo esto de la ubicación de la asignación dentro del código. Las asignaciones ubicadas dentro de una sentencia **on_event** tienen un equivalente combinacional y secuencial, mientras que las que no cumplan esta condición sólo tendrán un equivalente combinacional.

La transformación de las asignaciones que ocurren dentro de una sentencia **on_event** se planteó de tal forma que las expresiones a continuación del símbolo "=" en una asignación se expresan de forma combinacional y se guardan en una variable temporal. Estos valores son almacenados posteriormente de forma secuencial, como se verá más adelante.

A partir de la información presente en las líneas 14 y de la 18 a la 28 del código A.1, mostradas en el código 5.9, se genera lo mostrado en el código 5.10:

Código 5.9: Instrucciones en SALTT

```
var2 = var1 + 1;
 ...
on_event(in1.new_data) {
    out1.data[0] = var1;
    out1.data[1] = var3;
    out1.data[2] = out2;

    var1 = var2;
    var3 = var3 + 1;
    out2 = out2 + 3;

    set_event(out1);
}
```

Código 5.10: Transformación de lógica combinacional

```
always_comb begin
  var2 = var1 + 8'd1;
  _out1___data0_0 = var1;
  _out1___data1_0 = var3;
  _out1___data2_0 = out2;
  _var1_0 = var2;
  _var3_0 = var3 + 8'd1;
  _out2_0 = out2 + 4'd3;
end
```

El formato usado para las variables temporales, presentado anteriormente, consiste en escribir el símbolo "__", seguido por el identificador de la variable, otro símbolo "__" y finalmente un número, mayor o igual a 0, usado para diferenciar múltiples asignaciones de la misma variable. Este número aumenta en 1 cada vez que se hace una nueva asignación a la misma variable.

Este formato no se usa para las asignaciones que están fuera un bloque **on_event**, ya que se determinó que el lenguaje no permita una nueva asignación a la misma variable si se da el caso.

Al usarse un número como parte de una asignación, este se ajusta de acuerdo al tamaño máximo permitido por las variables involucradas en la asignación.

5.2.6. Resets

Dado que SALTT permite la inicialización de variables, es necesario que el código SystemVerilog equivalente tengan la capacidad de llevar las variables a un estado conocido cuando se le de aviso al módulo que debe comenzar a operar. Para hacer esto se usa una señal reset, que lleva todas las variables al valor con el que fueron inicializadas. Junto con esto, la señal reset lleva los contadores que indican la posición de un pixel dentro de un cuadro de video a 0.

Esta sección se genera a partir de la información presentes de las líneas de la 1 a la 3, 9, 11 y 12 del código A.1, las que se muestran en el código 5.11. La salida generada en SystemVerilog se muestra en el código 5.12:

Código 5.11: Inicialización en SALTT

```
filter top ( input pix_stream in1, input raw_bits(8) in2, parameter par1,
  output pix_stream out1, output raw_bits(4) out2 ) {
    raw_bits(8) var1 = 10, var2, var3;
    ...
    initial  out2 = 5;
    ...
    in1.set_format(res_h = 1280, res_v = 720, depth = {8, 8, 4});
    out1.copy_format(in1);
```

Código 5.12: Transformación de valores iniciales

```
always_ff @ (posedge clock) begin
  if(reset) begin
    in1___pos_h <= 11'd0;
    in1___pos_v <= 10'd0;
    out1___pos_h <= 11'd0;
    out1___pos_v <= 10'd0;
    var1 <= 8'd10;
    out2 <= 4'd5;
  end
```

Se ve que la primera línea de la transformación corresponde a un llamado a "always_ff" que inicia pero no termina. Esto ocurre porque el bloque secuencial se dividió en 2 partes, una parte de reset y una de operaciones. Esto permite agregar nuevos elementos a la etapa de reset aunque se hayan declarado instrucciones secuenciales antes.

En cuanto a las variables asignadas en esta sección y sus valores, éstos pueden proceder de 3 fuentes: Variables que al declararlas fueron inicializadas, sentencias **init** y variables **pix_stream** con resolución horizontal o vertical definida. Esto último genera una asignación que lleva la posición actual de un pixel a la posición inicial (0,0), antes de comenzar a procesar datos.

5.2.7. Procesamiento

Ésta corresponde a la última sección y es la encargada de representar el procesamiento descrito en el código fuente en SALTT dentro de las sentencias **on_event**. Todas las asignaciones declaradas dentro de estas sentencias son guardadas en registros y las funciones permitidas dentro de este tipo de sentencias producen registros donde se guarda el valor correspondiente dependiendo de la función llamada. Las sentencias condicionales son transformadas en esta sección, de modo que el valor almacenado corresponda al definido por el usuario al momento de expresar el la sentencia.

Junto a lo anterior, en esta sección se lleva a cabo el cálculo de la posición actual de un **pix_stream**, lo que se hace aumentando los contadores horizontal y vertical según corresponda.

A partir de la información presente en las líneas desde la 18 a la 28 del código A.1, mostradas en el código 5.13, se genera lo mostrado en el código 5.14:

Código 5.13: Instrucciones en SALTT

```
on_event(in1.new_data) {
    out1.data[0] = var1;
    out1.data[1] = var3;
    out1.data[2] = out2;

    var1 = var2;
    var3 = var3 + 1;
    out2 = out2 + 3;

    set_event(out1);
}
```

Código 5.14: Transformación de lógica secuencial

```
    else begin
        if (in1___new_data) begin
            out1___data0 <= _out1___data0_0;
            out1___data1 <= _out1___data1_0;
            out1___data2 <= _out1___data2_0;
            var1 <= _var2_0;
            var3 <= _var3_0;
            out2 <= _out2_0;

            out1___new_data <= 1'b1;
        end
        else begin
            out1___new_data <= 1'b0;
        end

        if (in1___reset) begin
            in1___pos_h <= 11'd0;
            in1___pos_v <= 10'd0;
        end
        else if (in1___new_data) begin
            if (in1___pos_h == in1___res_h) begin
                in1___pos_h <= 11'd0;

                if (in1___pos_v == in1___res_v)
                    in1___pos_v <= 10'd0;
                else
                    in1___pos_v <= in1___pos_v + 1'b1;
            end
            else
                in1___pos_h <= in1___pos_h + 1'b1;
        end
    end
end

endmodule
```

De manera similar a la sección anterior, se ve que se cierra un bloque con la palabra clave “end” que no se ve iniciado, lo que ocurre ya que se está cerrando el bloque “always_ff” que fue iniciado anteriormente. También se ve la palabra clave “endmodule” que cierra el módulo inicializado en la primera sección.

En cuanto al almacenamiento de registros, se almacenan las variables temporales asignadas previamente de forma combinacional. De esta forma cada variable se guarda sólo una vez, evitando con esto conflictos por intentar realizar múltiples conexiones simultáneas a una misma variable.

Finalmente, en lo respectivo al manejo de la posición de una variable **pix_stream**, éste se puede dividir en 3 etapas: Una etapa de *reset*, una de aumento de la cuenta horizontal y una de aumento de la cuenta vertical. Notar que la última etapa depende de la definición de la resolución vertical de **pix_stream**, por lo que no se declara si la definición no fue hecha. Los contadores se aumentarán cada vez que llegue un nuevo dato, indicado por la variable **ID** + “___new_data”, y volverá a 0 cada vez que el contador sea igual a la resolución definida, horizontal o vertical dependiendo del caso.

Capítulo 6: Resultados

Este capítulo presenta los resultados obtenidos al compilar códigos en SALTT utilizando la herramienta desarrollada en este trabajo. Los códigos generados fueron luego sintetizados utilizando el EDK Vivado de Xilinx.

Para poder sintetizar se escribió un archivo en SystemVerilog que maneje la entrada y salida HDMI, y el reloj. Además, se crearon y agregaron bloques IP para la conversión HDMI/DVI a RGB y viceversa, manejo de reloj y de *block RAM*, cuando correspondió, que resultaron necesarios para llevar a cabo las pruebas. Lo anterior se ilustra en la figura 6.1, donde "Procesador" son los códigos generados, "Interfaz Entrada" y "Interfaz Salida" representan a los manejadores se entrada y salida, los que pueden ser puertos HDMI, botones, etc; y "top" es el módulo de mayor jerarquía.

Para las pruebas se utilizó una tarjeta de desarrollo Nexys Video de Digilent, la que cuenta con un FPGA XC7A200T-1SBG484C de la serie Artix-7 de Xilinx. Se eligió esta tarjeta ya que contaba con un conector HDMI de entrada y otro de salida, con lo que se podía probar el buen funcionamiento de la entrada y salida de video, además de poder sintetizarse con la licencia gratuita ofrecida por Xilinx.

Pese a lo anterior, el código en SystemVerilog generado tiene la capacidad de portarse a otras tarjetas de la serie 7 ya que las interfaces de los IP utilizados son comunes dentro de la serie. Esto se comprobó sintetizando e implementando algunos ejemplos en la tarjeta de desarrollo Genesys 2 de Digilent, la que cuenta con un FPGA Kintex-7.

La tabla 6.1 muestra los recursos disponibles en la tarjeta Nexys Video.

A continuación se muestran los resultados obtenidos al compilar 6 códigos en SALTT utilizando la herramienta desarrollada en este trabajo y luego sintetizando el código para la tarjeta

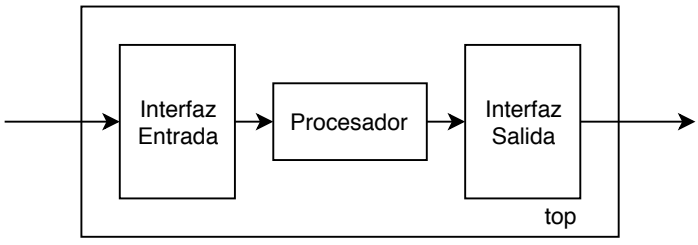


Fig. 6.1: Arquitectura general de prueba

Recurso	Disponible	Unidad
Slices lógicos	33650	
BRAM	13	Mbits
DSP *slices*	740	
DRAM	512	MB
Reloj de sistema	100	kHz
Frecuencia máxima de operación	450	kHz
HDMI	Entrada y salida	

Tabla 6.1: Recursos tarjeta Nexys Video

mencionada anteriormente.

6.1. Código degrade con memoria

El primer código escrito genera un patrón de 16 líneas horizontales cuyo valor está representado en escala de grises utilizando 8 bits y aumenta de forma constante entre líneas yendo del negro al blanco. El valor de cada una de las líneas se guarda en *block RAM*, la que tiene valores precargados al momento de la síntesis, y puede ser modificado en tiempo de ejecución utilizando los switches y botones presentes en la tarjeta de desarrollo. El código fuente y el SystemVerilog generado con la herramienta se encuentran en el anexo B. Este es un código sencillo cuyo principal objetivo es demostrar la capacidad de la herramienta de generar código en SystemVerilog, en especial de los elementos centrales de SALTT, la interfaz de video y la memoria.

La tabla 6.2 muestra los recursos utilizado al sintetizar el código generado. Se muestran los recursos utilizados por todo el sistema y los recursos utilizados sólo por el código generado.

La tabla 6.3 muestra el consumo en potencia de todo el sistema y sólo del código generado automáticamente. El consumo estático del sistema son 140 mW y el consumo total son 491 mW.

En cuanto al número de líneas, el código fuente está compuesto por 37 líneas, mientras que el código generado consta de 142 líneas. De éstas, 22 corresponden a operaciones en el caso del código en SALTT, mientras que el código generado tiene 52 líneas correspondientes a operaciones.

En la figura 6.2 se muestra el sistema funcionando. Al igual que en el código anterior, en

Recurso	Disponible	Utilizado sistema	Utilizado módulo
LUT	133800	580	31
LUTRAM	46200	24	0
FF	267600	653	42
BRAM	365	0.5	0.5
IO	285	31	0
PLL	10	2	0

Tabla 6.2: Uso de recursos código degrade con memoria

Recurso	Potencia sistema [mW]	Potencia módulo [mW]
Relojes	6	<1
Señales	3	<1
Lógica	3	<1
BRAM	<1	<1
MMCM	168	<1
I/O	171	<1
Total	351	<1

Tabla 6.3: Consumo de potencia código degrade con memoria

pantalla se ven las 16 líneas que van de negro en la parte superior a negro.

Como ya se mencionó, el objetivo principal de este código era demostrar el funcionamiento de la herramienta, para lo cual se planteó un código sencillo. Por esta razón, tanto el uso de recursos como el de potencia es bajo, estando principalmente concentrado en las conexiones con el exterior.

6.2. Código umbralización

Este código transforma los pixeles de una imagen de entrada RGB de 24 bits a escala de grises de 8 bits (por simplicidad, se sumaron los 3 canales y se dividieron por 3, utilizando una multiplicación y un desplazamiento de bits del resultado de la multiplicación anterior), y entrega un 1 o un 0 si el pixel en escala de grises supera o no un umbral. El código fuente se encuentran en el anexo C.

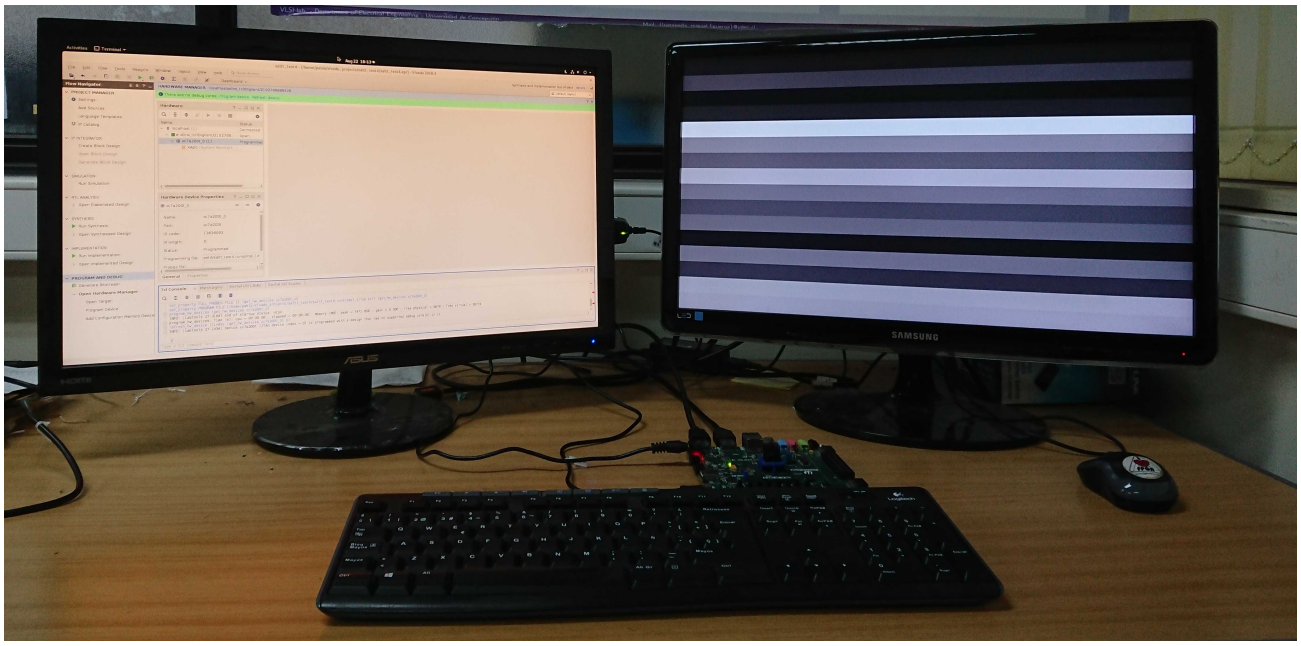


Fig. 6.2: *Setup* de código degrade con memoria

Recurso	Disponible	Utilizado sistema	Utilizado módulo
LUT	133800	545	11
LUTRAM	46200	25	1
FF	267600	640	22
IO	285	22	0
MMCM	10	2	0

Tabla 6.4: Uso de recursos código umbral

La tabla 6.4 muestra los recursos utilizado al sintetizar el código generado. Se muestran los recursos utilizados por todo el sistema y los recursos utilizados sólo por el código generado.

La tabla 6.5 muestra el consumo en potencia de todo el sistema y sólo del código generado automáticamente. El consumo estático del sistema son 140 mW y el consumo total son 498 mW.

En cuanto al número de líneas, el código fuente está compuesto por 45 líneas, mientras que la sumatoria de los códigos generados es de 202 líneas. De éstas, 20 corresponden a operaciones para el código en SALTT, mientras que 61 corresponden a operaciones en el código generado.

En la figura 6.3 se muestra el sistema funcionando. En el monitor de la izquierda se ve la imagen original y en el de la derecha la imagen umbralizada.

Pese a que este código generó más líneas de código que el anterior, además de procesar la

Recurso	Potencia sistema [mW]	Potencia módulo [mW]
Relojes	6	<1
Señales	6	<1
Lógica	7	<1
MMCM	168	<1
I/O	172	<1
Total	358	<1

Tabla 6.5: Consumo de potencia código umbral

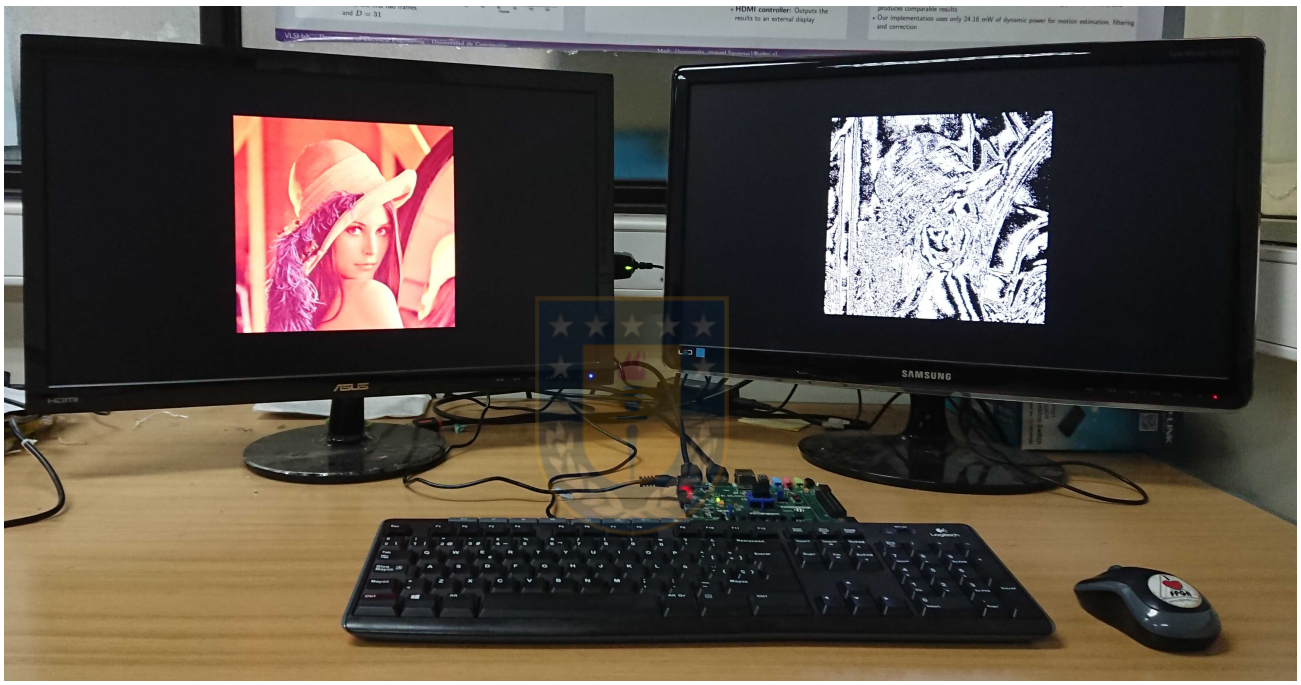


Fig. 6.3: *Setup* de código umbral

señal de entrada, su uso de recursos y consumo de potencia es menor. Esto ocurre ya que, este procesamiento es pixel a pixel y no depende de la posición horizontal y vertical, razón por la cual Vivado elimina los contadores de posición por no utilizarse, que era algo esperado y la razón que motivó la omisión de la etapa de optimización en la herramienta.

6.3. Código conversión RGB a HSL

Este código transforma los píxeles de entrada del espacio de colores RGB al espacio de colores HSL, lo que se realiza pixel a pixel. Tanto el código fuente en SALTT como el equivalente en

Recurso	Utilizado módulo	Utilizado custom
LUT	141	117
LUTRAM	10	0
FF	65	45
BRAM	1	1
DSP	2	2

Tabla 6.6: Uso de recursos código RGB a HSL

Recurso	Potencia módulo [mW]	Potencia custom [mW]
Relojes	1	<1
Señales	<1	<1
Lógica	<1	<1
BRAM	4	4
MMCM	<1	<1
I/O	<1	<1
Total	4	4

Tabla 6.7: Consumo de potencia código RGB a HSL

SystemVerilog se muestran en el anexo D.

A partir de este ejemplo, además del código en SALTT se escribió un código que realiza el mismo algoritmo en SystemVerilog. Las tablas de resultados y las comparaciones se harán entre lo generado por la herramienta a partir de código fuente en SALTT y el código escrito en SystemVerilog.

La tabla 6.6 muestra el uso de recursos obtenidos de la síntesis. Como se adelantó, se muestran los resultados del código generado a partir de SALTT y del escrito en SystemVerilog.

La tabla 6.7 muestra el consumo en potencia para el código generado por la herramienta y el escrito en SystemVerilog.

El código escrito en SALTT cuenta con 68 líneas de código correspondientes a operaciones, mientras que el código en SystemVerilog tiene 51 líneas de operaciones. Por último, el código generado puede operar a una frecuencia máxima de 111 MHz, mientras que el código en SystemVerilog puede operar a 107 MHz máximo. Para obtener esta frecuencia se iteraron implementaciones modificando la frecuencia de operación del pixel (*pixel clock*) hasta que Vivado

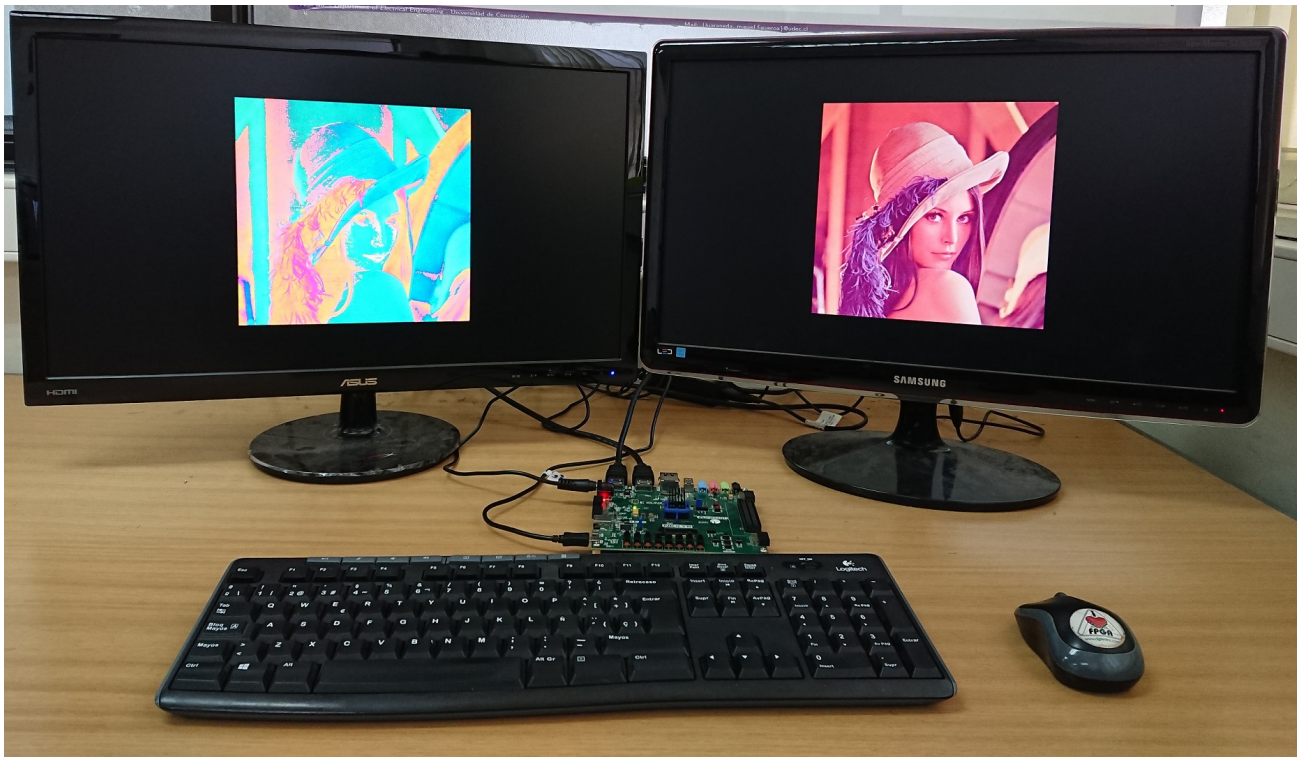


Fig. 6.4: *Setup* de código RGB a HSL

informó que no se cumplían las restricciones de tiempo luego de terminar la implementación. Para asegurar la consistencia del resultado, se repitió múltiples veces la implementación usando el mismo tiempo, cumpliendo en todos los casos con las restricciones de tiempo. Es importante mencionar que se usó la misma configuración de implementación durante la implementación del código generado a partir de SALTT y del código escrito directamente en SystemVerilog, de modo que esto no afectara la comparación. Este mismo procedimiento se utilizó en los códigos.

En la figura 6.4 se muestra el sistema funcionando. En el monitor de la derecha se ve la imagen original y en el de la izquierda la imagen en el espacio HSL, con el canal *Hue* en la posición del color rojo RGB, el canal *Saturation* en la posición del verde y el canal *Luminance* en la posición del azul, esto para poder desplegarlo en el monitor.

Se ve que el uso de recursos de la versión en SystemVerilog es menor que para el código generado a partir de SALTT, sin embargo, la diferencia es pequeña. De esto se puede decir que SALTT puede competir con códigos escritos directamente en SystemVerilog. El uso de potencia es igual en ambos casos y las frecuencias máximas de operación son similares, lo que reafirma lo anterior.

Por otro lado, se ve que el código en SALTT tiene más líneas que en SystemVerilog, ya que fue necesario escribir líneas que correspondían a configuraciones que no se podían obviar

Recurso	Utilizado módulo	Utilizado custom
LUT	286	176
FF	167	163
BRAM	3	3
DSP	3	3

Tabla 6.8: Uso de recursos código filtro de media

Recurso	Potencia módulo [mW]	Potencia custom [mW]
Relojes	1	1
Señales	<1	<1
Lógica	<1	<1
BRAM	<1	<1
MMCM	<1	<1
I/O	<1	<1
Total	1	1

Tabla 6.9: Consumo de potencia código filtro de media

porque el lenguaje así lo determina, pero que no tienen un equivalente en SystemVerilog por no ser necesario. Esto se debe principalmente a que se decidió dividir el código SALTT en múltiples hardware_threads, por lo que algunos recursos no se pudieron compartir, entre ellos, las configuraciones de los pix_stream. En el caso de la versión en SystemVerilog, se escribió todo en un archivo y se compartieron todos los recursos que se pudieron.

6.4. Filtro de media

Este código calcula el promedio de los pixeles en una vecindad de 3x3 y reemplaza su valor por este resultado, lo que se hace para los 3 canales RGB de un *stream* de entrada. El código fuente en SALTT se puede puede ver en el anexo E.

La tabla 6.8 muestra los recursos utilizado al sintetizar tanto el código generado a partir de SALTT como el del código en SystemVerilog.

La tabla 6.9 muestra el consumo en potencia para el código generado por la herramienta y el escrito en SystemVerilog.

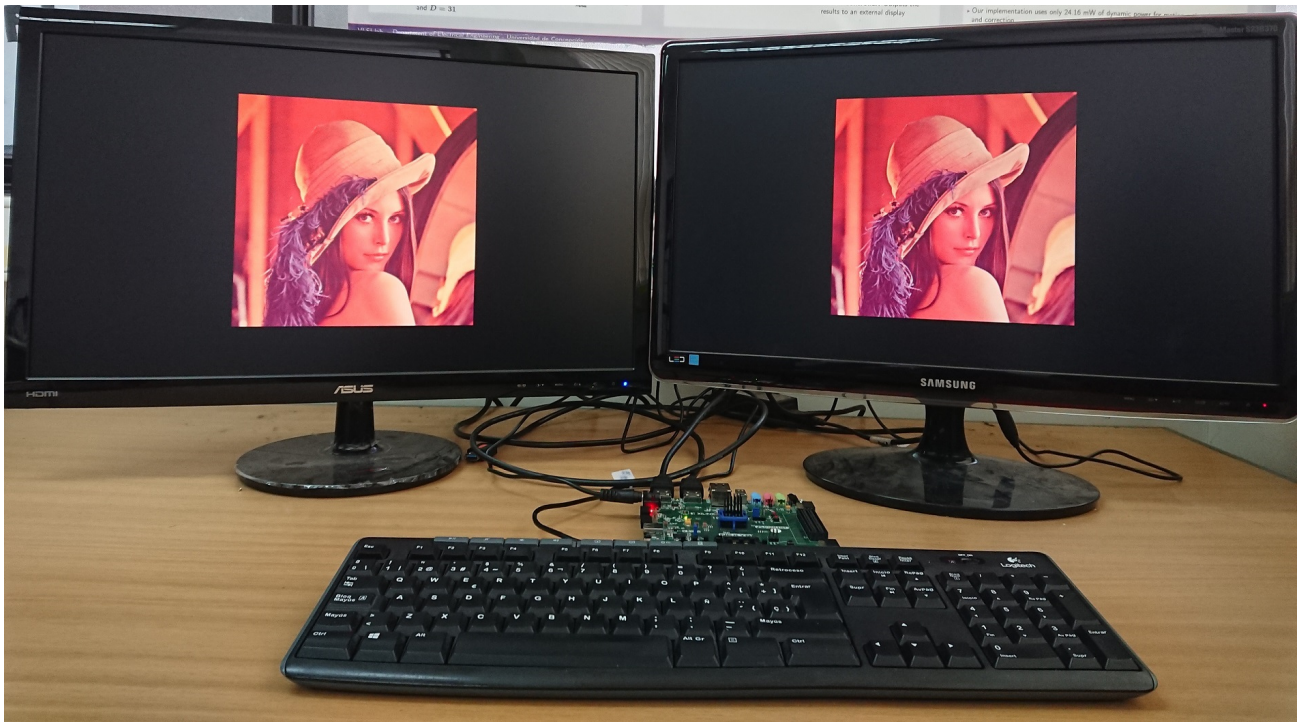


Fig. 6.5: *Setup* de código filtro de media

El código escrito en SALTT cuenta con 17 líneas de código correspondientes a operaciones, mientras que el código en SystemVerilog tiene 31 líneas de operaciones. La frecuencia máxima para el código generado es de 85 MHz, mientras que la frecuencia máxima para el código en SystemVerilog es de 82 MHz.

En la figura 6.5 se muestra el sistema funcionando. En el monitor de la derecha se ve la imagen original y en el de la izquierda la imagen procesada.

En este caso se repite el comportamiento anterior teniéndose un menor uso de recursos en el caso custom que el código generado. La principal causa de esto es que la versión en SystemVerilog obvia lógica de control para la ventana que no es necesario, además de hacer el control de posición horizontal utilizando las señales de sincronismo y no llevando un registro de la posición vertical, como si lo hace el código generado. Como el circuito diseñado resultante en ambos casos tiene un consumo tan bajo de potencia, no se ven diferencias. Sin embargo, se nota que la frecuencia máxima de operación del código generado es mayor que la del código en SystemVerilog, lo que se puede deber a que en el afán de reducir recursos resulto un camino crítico más largo. A pesar de esto, la diferencia de tiempo es pequeña, lo que demuestra que al diseñar un mismo algoritmo tanto en SystemVerilog como en SALTT se obtienen resultados similares.

Recurso	Utilizado módulo	Utilizado custom
LUT	152	181
FF	73	71
BRAM	1	1
DSP	1	1

Tabla 6.10: Uso de recursos código operador Sobel

Recurso	Potencia módulo [mW]	Potencia custom [mW]
Relojes	1	<1
Señales	2	1
Lógica	2	1
BRAM	2	2
MMCM	<1	<1
I/O	<1	<1
Total	7	4

Tabla 6.11: Consumo de potencia código operador Sobel

6.5. Operador Sobel

Este código calcula el operador Sobel para una imagen de entrada en RGB. Para esto, se transforma la imagen a escala de grises y en este espacio de colores se aplica el operador. El código fuente en SALTT se puede puede ver en el anexo F.

La tabla 6.10 muestra los recursos utilizado al sintetizar tanto el código generado a partir de SALTT como el del código en SystemVerilog.

La tabla 6.11 muestra el consumo en potencia para el código generado por la herramienta y el escrito en SystemVerilog.

El código escrito en SALTT cuenta con 19 líneas de código correspondientes a operaciones, mientras que el código en SystemVerilog tiene 29 líneas de operaciones. La frecuencia máxima para el código generado es de 74 MHz, mientras que la frecuencia máxima para el código en SystemVerilog es de 75 MHz.

En la figura 6.6 se muestra el sistema funcionando. En el monitor de la derecha se ve la

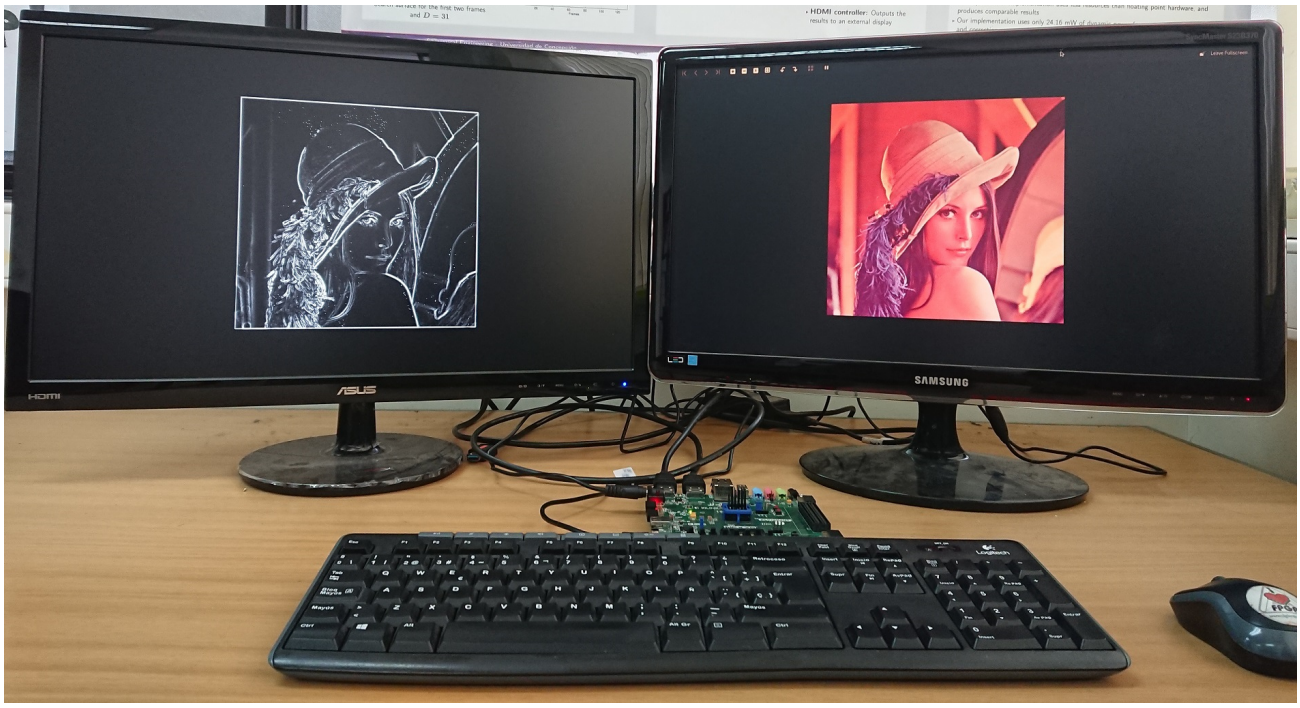


Fig. 6.6: *Setup* de código operador Sobel

imagen original y en el de la izquierda la imagen procesada.

En este caso se obtuvo un menor uso de recursos en el código generado a partir de SALTT. Con esto se ve que en algunos casos SALTT puede presentar una mejor solución que un código escrito en SystemVerilog directamente. Por su parte, se ve una reducción en el número de líneas, pero no es tan significativa ya que se reducen los 3 canales de entrada a uno y las operaciones en cantidad de líneas son similares.

6.6. Estabilización de video

Este código aplica un algoritmo de estabilización de video, el que fue implementado por Araneda [39]. El código se basa en el desarrollado también por Araneda [2]. El código fuente en SALTT se puede puede ver en el anexo F. El sistema opera a una resolución de 800x600@60.

La tabla 6.12 muestra los recursos utilizado al sintetizar tanto el código generado a partir de SALTT como el del código en SystemVerilog.

La tabla 6.13 muestra el consumo en potencia para el código generado por la herramienta y el escrito en SystemVerilog.

Recurso	Utilizado módulo	Utilizado custom
LUT	1431	1427
FF	306	315
BRAM	237	237
DSP	4	1

Tabla 6.12: Uso de recursos código estabilización de video

Recurso	Potencia módulo [mW]	Potencia custom [mW]
Relojes	8	9
Señales	32	<1
Lógica	8	<1
BRAM	10	7
MMCM	<1	<1
I/O	<1	<1
Total	59	17

Tabla 6.13: Consumo de potencia código estabilización de video

El código escrito en SALTT cuenta con 250 líneas de código correspondientes a operaciones, mientras que el código en SystemVerilog tiene 393 líneas de operaciones. La frecuencia máxima para el código generado es de 100 MHz, mientras que la frecuencia máxima para el código en SystemVerilog es de 103 MHz.

En cuanto al uso de recursos se nota que estos son similares entre ambos. En este caso se nota un aumento considerable en el consumo de potencia. Al analizar los datos entregados por Vivado, se tiene que este aumento se encuentra principalmente motivado por el calculo de la dirección corregida, la que debe transformarse de una dirección bidireccional a una unidireccional en SALTT, mientras que en SystemVerilog se utilizó otro enfoque menos costoso computacionalmente para obtener el mismo resultado. También se nota que la frecuencia máxima es similar, aunque es necesario señalar que ambos casos operan más rápido que los 40 MHz necesarios para operar a la resolución dada. La mayor diferencia se nota en el número de líneas, siendo el código en SALTT un 63 % más corto que el equivalente en SystemVerilog.

Capítulo 7: Conclusiones

En este trabajo se presentó una herramienta para compilar el lenguaje de programación de dominio específico propuesto por Araneda[2]. Esta herramienta transforma código fuente en SALTT a código en el HDL SystemVerilog.

La herramienta cuenta con una etapa de análisis léxico, seguido de un análisis sintáctico y una etapa de generación de código en SystemVerilog, sin embargo no cuenta con una etapa de revisión de tipos para comprobar el cumplimiento de las reglas del lenguaje.

Para el desarrollo de la herramienta se escribieron códigos en SALTT y equivalentes esperados en SystemVerilog. Con esto, se diseñaron las transformaciones necesarias para generar los códigos esperados, haciendo las generalizaciones necesarias y completando con la información presente en el código fuente.

Se estableció la forma en que se maneja la parte combinacional y secuencial del código generado. Se hicieron simplificaciones a la arquitectura propuesta originalmente y se crearon nuevas funciones para admitir nuevas funcionalidades no contempladas. Se definió el mecanismo para determinar la ubicación espacial de un pixel representado por la variable **pix_stream**.

Se probó la herramienta desarrollada compilando diversos códigos, cuyo propósito era demostrar la capacidad del lenguaje de transformar las instrucciones básicas del lenguaje, entre las que se encuentran la declaración de variables, destacando las variables de manejo de pixeles (**pix_stream**) y las de manejo de memoria (**memory**), operaciones aritméticas y el manejo de eventos (**on_event**).

Se comprobó que el código generado no tenía errores de escritura sintetizándolos con el EDK Vivado. También se comprobó que ejecutaran los procesamientos esperados implementando los códigos generados en un FPGA y viendo la salida de video desplegada en una pantalla. Para esto se utilizaron las interfaces HDMI de entrada y salida presentes en la tarjeta de prueba.

Se comprobó que es posible generar arquitecturas *hardware* a partir del DSL SALTT, pudiendo estos diseños ser implementados en un FPGA luego de escribir un módulo que maneje las interfaces de entrada y salida, y de agregar los módulos IP necesarios.

Se logró el objetivo de diseñar una herramienta que transforme código fuente escrito en

SALTT en una arquitectura *hardware*, representada mediante un código en SystemVerilog. Sin embargo, todavía hay que pasar por una etapa de escritura de un módulo de manejo de interfaces y se deben agregar manualmente los bloques IP.

Con las observaciones mencionadas anteriormente, se pudo implementar la arquitectura diseñada en un FPGA, para lo que se creó manualmente en Vivado un proyecto, que luego se sintetizó e implementó.

Por razones de tiempo, no se pudo lograr el objetivo de transformar código fuente al lenguaje de programación C, aunque se hicieron algunos avances en este ámbito. Además, se estableció un flujo de trabajo para desarrollar la transformación, mismo utilizado durante el diseño de la transformación a SystemVerilog: Escribir códigos sencillos en SALTT y equivalentes en el lenguaje objetivo, C en este caso; traspasar la información necesaria entre estructuras durante el análisis sintáctico, y, con todo esto, generar un código similar al propuesto.

Por la misma razón tampoco se logró el objetivo de elegir y diseñar módulos en SystemVerilog para las estructuras más comunes, no se automatizó el proceso de creación de proyecto en Vivado, síntesis e implementación.

Finalmente, se lograron obtener resultados de desempeño al sintetizar algunos códigos, sin embargo, estos resultados no permiten evaluar de forma fidedigna el desempeño ya que son pequeños, siendo su principal función demostrar la capacidad de la herramienta de generar una arquitectura *hardware* y que ésta se pueda implementar en un FPGA.

Por lo anterior, el primero de los trabajos futuros propuestos es hacer pruebas del lenguaje con procesamientos más complejos, comparando el despeño obtenido con códigos escritos directamente en un HDL y con otros generados por una herramienta de HLS.

También se puede continuar el desarrollo de la herramienta agregando funciones como el manejo de memoria DRAM o la comprobación del cumplimiento de las reglas del lenguaje. Junto con esto se puede hacer una nueva revisión del lenguaje y la herramienta.

Otro aspecto en el que se puede continuar trabajando es en el diseño de módulos en SystemVerilog que realicen tareas frecuentes, de modo de explotar al máximo las capacidades del diseño de *hardware* y mejorar con esto el rendimiento de los sistemas diseñados.

Por último, se puede trabajar en la automatización de la etapa correspondiente a Vivado, creando o generando un *script* que cree un proyecto, agregue los códigos fuentes en SystemVerilog, los bloques IP y finalmente sintetice e implementé en un FPGA.

Bibliografía

[1] D. Franklin, "NVIDIA Jetson TX2 delivers twice the intelligence to the edge," 2017, accessed: 2018-06-06. [Online]. Available: https://devblogs.nvidia.com/jetson-tx2-delivers-twice-intelligence-edge/

[2] L. Araneda, "Modelo y plataforma de cómputo heterogéneo para video infrarrojo," Tesis de magíster, Universidad de Concepción, Septiembre 2016.

[3] D. Bacon, R. Rabbah, and S. Shukla, "FPGA programming for the masses," *Queue*, vol. 11, no. 2, pp. 40:40–40:52, Feb. 2013. [Online]. Available: http://doi.acm.org/10.1145/2436696.2443836

[4] L. Daoud, D. Zydek, and H. Selvaraj, *A Survey of High Level Synthesis Languages, Tools, and Compilers for Reconfigurable High Performance Computing.* Cham: Springer International Publishing, 2014, pp. 483–492. [Online]. Available: https://doi.org/10.1007/978-3-319-01857-7_47

[5] R. Nane, V. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of fpga high-level synthesis tools," *IEEE Transactions on Computer - Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, 2016.

[6] W. A. Najjar, W. Bohm, B. A. Draper, J. Hammes, R. Rinker, J. R. Beveridge, M. Chawathe, and C. Ross, "High-level language abstraction for reconfigurable computing," *Computer*, vol. 36, no. 8, pp. 63–69, Aug 2003.

[7] N. University, "MATCH compiler," accessed: 2018-03-29. [Online]. Available: http://www.ece.northwestern.edu/cpdc/Match/

[8] J. Villarreal, A. Park, W. Najjar, and R. Halstead, "Designing modular hardware accelerators in c with roccc 2.0," in *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2010, pp. 127–134.

[9] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: High-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field*

Programmable Gate Arrays, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 33–36. [Online]. Available: http://doi.acm.org/10.1145/1950413.1950423

[10] I. A. Technologies, "Tools - impulse accelerated technologies," 2015, accessed: 2018-03-22. [Online]. Available: http://www.impulseaccelerated.com/tools.html

[11] M. Graphics, "DK design suite," 2010, accessed: 2018-03-22. [Online]. Available: http://s3.mentor.com/public_documents/datasheet/products/fpga/handel-c/dk-design-suite/dk-ds.pdf

[12] Cadence, "Stratus high-level synthesis," 2015, accessed: 2018-03-21. [Online]. Available: https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/digital-design-signoff/stratus-ds.pdf

[13] Synposys, "Synphony model compiler," 2014, accessed: 2018-03-21. [Online]. Available: https://www.synopsys.com/content/dam/synopsys/implementation&signoff/datasheets/synphony-model-comp-ds.pdf

[14] MATLAB, "Xilinx FPGAs and Zynq SoCs," accessed: 2018-06-06. [Online]. Available: https://www.mathworks.com/solutions/fpga-design/simulink-with-xilinx-system-generator-for-dsp.html

[15] ——, "Xilinx system generator for DSP," accessed: 2018-06-06. [Online]. Available: https://www.mathworks.com/products/connections/product_detail/xilinx-system-generator-for-dsp.html

[16] ——, "Generate VHDL and verilog code for FPGA and ASIC designs," accessed: 2018-06-06. [Online]. Available: https://www.mathworks.com/products/hdl-coder.html

[17] Xilinx, "Vivado high-level synthesis," accessed: 2017-11-21. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[18] ——, "The Xilinx SDAccel development environment," accessed: 2018-06-06. [Online]. Available: https://www.xilinx.com/support/documentation/backgrounders/sdaccel-backgrounder.pdf

[19] ——, "SDSoC development environment," accessed: 2018-06-06. [Online]. Available: https://www.xilinx.com/support/documentation/backgrounders/sdsoc-development-environment-backgrounder.pdf

[20] Intel, "Intel® HLS compiler," accessed: 2018-03-25. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/products/hls/hls-production-brief.pdf

[21] ——, "Intel® FPGA SDK for OpenCL™," accessed: 2018-05-09. [Online]. Available: https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.tablet.html

[22] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan, "Darkroom: Compiling high-level image processing code into hardware pipelines," *ACM Trans. Graph.*, vol. 33, no. 4, pp. 144:1–144:11, Jul. 2014. [Online]. Available: http://doi.acm.org/10.1145/2601097.2601174

[23] J. Serot, F. Berry, and S. Ahmed, "Implementing stream-processing applications on FPGAs: A DSL-Based approach," in *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, Sept 2011, pp. 130–137.

[24] R. J. Stewart, D. Bhowmik, G. Michaelson, and A. M. Wallace, "RIPL: an efficient image processing DSL for fpgas," *CoRR*, vol. abs/1508.07136, 2015. [Online]. Available: http://arxiv.org/abs/1508.07136

[25] R. Stewart, K. Duncan, G. Michaelson, P. Garcia, D. Bhowmik, and A. Wallace, "RIPL: A Parallel Image Processing Language for FPGAs," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 11, no. 1, pp. 7:1–7:24, March 2018. [Online]. Available: http://doi.acm.org/10.1145/3180481

[26] N. Chugh, V. Vasista, S. Purini, and U. Bondhugula, "A dsl compiler for accelerating image processing pipelines on fpgas," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, ser. PACT '16. New York, NY, USA: ACM, 2016, pp. 327–338. [Online]. Available: http://doi.acm.org/10.1145/2967938.2967969

[27] R. Membarth, O. Reiche, F. Hannig, J. Teich, M. Körner, and W. Eckert, "Hipacc: A domain-specific language and compiler for image processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 1, pp. 210–224, Jan 2016.

[28] O. Reiche, M. Schmid, F. Hannig, R. Membarth, and J. Teich, "Code generation from a domain-specific language for c-based hls of hardware accelerators," in *2014 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Oct 2014, pp. 1–10.

[29] T. Ægidius Mogensen, *Basics of Compiler Design*, anniversary ed. lulu.com, Agosto 2010.

[30] M. E. Lesk and E. Schmidt, *Lex, a Lexical Analyzer Generator*, ser. Computing science technical report. Bell Laboratories, 1975.

[31] J. R. Levine, T. Mason, and D. Brown, *Lex & Yacc (2Nd Ed.)*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1992.

[32] W. E. Vern Paxson and J. Millaway, "Lexical analysis with flex," 2017, accessed: 2018-03-05. [Online]. Available: https://github.com/westes/flex

[33] J. Levine and L. John, *Flex & Bison*, 1st ed. O'Reilly Media, Inc., 2009.

[34] D. M. Beazley, "PLY (Python Lex-Yacc)," 2018, accessed: 2018-03-05. [Online]. Available: http://www.dabeaz.com/ply

[35] S. C. Johnson, "Yacc: Yet another compiler-compiler," Bell Laboratories, Tech. Rep., 1979.

[36] T. E. Dickey, "BYACC - berkeley yacc - generate LALR(1) parsers," 2018, accessed: 2018-03-31. [Online]. Available: https://invisible-island.net/byacc/byacc.html

[37] C. Donnelly and R. Stallman, "Bison," 2015, accessed: 2018-03-31. [Online]. Available: https://www.gnu.org/software/bison/manual/bison.pdf

[38] Xilinx, "Vivado design suite tcl command reference guide," 2017, accessed: 2018-04-01. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug835-vivado-tcl-commands.pdf

[39] L. Araneda and M. Figueroa, "A compact hardware architecture for digital image stabilization using integral projections," *Microprocessors and Microsystems*, 2015.

Anexo A: Código ejemplo

El código fuente utilizado para ejemplificar el funcionamiento de la herramienta se presenta a continuación:

Código A.1: Código ejemplo transformación

```
1  filter top ( input pix_stream in1, input raw_bits(8) in2, parameter par1,
2      output pix_stream out1, output raw_bits(4) out2 ) {
3      raw_bits(8) var1 = 10, var2, var3;
4      unumber(3,2) var4;
5      parameter local_par = 35;
6      memory(8, 16, 1, BRAM) mem;
7      memory_port mem_port1 = mem.port[0];
8
9      initial out2 = 5;
10
11     in1.set_format(res_h = 1280, res_v = 720, depth = {8, 8, 4});
12     out1.copy_format(in1);
13
14     var2 = var1 + 1;
15
16     my_ht( in1, var4 );
17
18     on_event(in1.new_data) {
19         out1.data[0] = var1;
20         out1.data[1] = var3;
21         out1.data[2] = out2;
22
23         var1 = var2;
24         var3 = var3 + 1;
25         out2 = out2 + 3;
26
27         set_event(out1);
28     }
29 }
30
31 my_ht (output pix_stream in1, input unumber(3,2) out1) {
32     ...
33 }
```

Anexo B: Código generación degrade con BRAM

B.1. Código SALTT

El código fuente completo de la aplicación se presenta a continuación:

```
filter top(input pix_stream in_pix, input raw_bits(8) sw,
  input event btn, output pix_stream out_pix)
{
    in_pix.set_format(res_h = 1280, res_v = 720, depth = {8, 8, 8});
    out_pix.copy_format(in_pix);
    raw_bits(4) rd_idx = 15, wr_idx = 0;
    raw_bits(6) count = 44;
    memory(8, 16, 1, BRAM) mem;
    memory_port mem_port1 = mem.port[0], mem_port2;;

    mem_port2 = mem.port[1];

    out_pix.data[0] = mem_port1[rd_idx, 1];
    out_pix.data[1] = mem_port1[rd_idx, 1];
    out_pix.data[2] = mem_port1[rd_idx, 1];

    on_event(in_pix.new_data) {

        if (in_pix.pos_h == 0) {
          if(count == 44) {
            rd_idx = rd_idx + 1;
            count = 0;
          }
          else
            count = count + 1;
        }

        set_event(out_pix.new_data);
        if (in_pix.pos_h == in_pix.res_h && in_pix.pos_v == in_pix.res_v)
            set_pix_reset(out_pix);
    }
```

```
32
33      on_event(btn) {
34          mem_port2[wr_idx, 1] = sw;
35          wr_idx = wr_idx + 1;
36      }
37  }
```

B.2. Código generado en SystemVerilog

El código generado por la herramienta se presenta a continuación:

```
1  // [0]
2  module demo4 (
3      in_pix__data,
4      in_pix__new_data,
5      in_pix__reset,
6      in_pix__last,
7      sw,
8      btn,
9      out_pix__data,
10     out_pix__new_data,
11     out_pix__reset,
12     out_pix__last,
13     clock,
14     reset
15 );
16
17 // [1]
18 localparam in_pix__res_h = 1279;
19 localparam in_pix__res_v = 719;
20 localparam MSB_in_pix = 23;
21 localparam out_pix__res_h = 1279;
22 localparam out_pix__res_v = 719;
23 localparam MSB_out_pix = 23;
24
25 // [2]
26 input logic in_pix__new_data;
27 input logic [MSB_in_pix : 0] in_pix__data;
28 input logic in_pix__reset;
29 input logic in_pix__last;
30 input logic [7:0] sw;
31 input logic btn;
32 output logic out_pix__new_data;
33 output logic [MSB_out_pix : 0] out_pix__data;
34 output logic out_pix__reset;
35 output logic out_pix__last;
36 input logic clock;
37 input logic reset;
38
39 // [3]
```

```
40 logic [10:0] in_pix__pos_h;
41 logic [9:0] in_pix__pos_v;
42 logic [7:0] in_pix__data0;
43 logic [7:0] in_pix__data1;
44 logic [7:0] in_pix__data2;
45 logic [10:0] out_pix__pos_h;
46 logic [9:0] out_pix__pos_v;
47 logic [7:0] out_pix__data0;
48 logic [7:0] out_pix__data1;
49 logic [7:0] out_pix__data2;
50 logic [3:0] rd_idx, _rd_idx_0;
51 logic [3:0] wr_idx, _wr_idx_0;
52 logic [5:0] count, _count_0, _count_1;
53
54 // [4]
55 logic mem__wea, mem__web;
56 logic [3:0] mem__addra, mem__addrb;
57 logic [7:0] mem__dina, mem__douta, mem__dinb, mem__doutb;
58
59 BRAM_8_16_1 mem(
60     .clka(clock),
61     .wea(mem__wea),
62     .addra(mem__addra),
63     .dina(mem__dina),
64     .douta(mem__douta),
65     .clkb(clock),
66     .web(mem__web),
67     .addrb(mem__addrb),
68     .dinb(mem__dinb),
69     .doutb(mem__doutb)
70 );
71
72
73 // [5]
74 assign {in_pix__data0, in_pix__data1, in_pix__data2} = in_pix__data;
75 assign out_pix__data = {out_pix__data0, out_pix__data1, out_pix__data2};
76
77 // [6]
78 always_comb begin
79     mem__addra = 4'd0 + rd_idx;
80     out_pix__data0 = mem__douta;
81     out_pix__data1 = mem__douta;
82     out_pix__data2 = mem__douta;
```

```
83      _rd_idx_0 = rd_idx + 1'd1;
84      _count_0 = 6'd0;
85      _count_1 = count + 1'd1;
86      _wr_idx_0 = wr_idx + 1'd1;
87  end
88
89  // [7]
90  always_ff @ (posedge clock) begin
91      if(reset) begin
92          in_pix__pos_h <= 11'd0;
93          in_pix__pos_v <= 10'd0;
94          out_pix__pos_h <= 11'd0;
95          out_pix__pos_v <= 10'd0;
96          rd_idx <= 4'd15;
97          wr_idx <= 4'd0;
98          count <= 6'd44;
99      end
100
101 // [8]
102     else begin
103         if (in_pix__new_data) begin
104
105             if (in_pix__pos_h == 11'd0) begin
106
107                 if (count == 6'd44) begin
108                     rd_idx <= _rd_idx_0;
109                     count <= _count_0;
110                 end
111                 else begin
112                     count <= _count_1;
113                 end
114
115             end
116
117             out_pix__new_data <= 1'b1;
118
119             if (in_pix__pos_h == in_pix__res_h &&
120               in_pix__pos_v == in_pix__res_v) begin
121                 out_pix__reset <= 1'b1;
122             end
123             else begin
124                 out_pix__reset <= 1'b0;
125             end
```

```
126
127             end
128             else begin
129                 out_pix__new_data <= 1'b0;
130                 out_pix__reset <= 1'b0;
131             end
132
133             if (btn) begin
134                 mem__addrb <= 4'd0 + wr_idx;
135                 mem__dinb <= sw;
136                 mem__web <= 1'b1;
137                 wr_idx <= _wr_idx_0;
138             end
139             else begin
140                 mem__web <= 1'b0;
141             end
142
143
144             if (in_pix__reset) begin
145                 in_pix__pos_h <= 11'd0;
146                 in_pix__pos_v <= 10'd0;
147             end
148             else if (in_pix__new_data) begin
149                 if (in_pix__pos_h == in_pix__res_h) begin
150                     in_pix__pos_h <= 11'd0;
151
152                     if (in_pix__pos_v == in_pix__res_v)
153                         in_pix__pos_v <= 10'd0;
154                     else
155                         in_pix__pos_v <= in_pix__pos_v + 1'b1;
156                 end
157                 else
158                     in_pix__pos_h <= in_pix__pos_h + 1'b1;
159             end
160
161             if (out_pix__reset) begin
162                 out_pix__pos_h <= 11'd0;
163                 out_pix__pos_v <= 10'd0;
164             end
165             else if (out_pix__new_data) begin
166                 if (out_pix__pos_h == out_pix__res_h) begin
167                     out_pix__pos_h <= 11'd0;
168
```

```
169                     if (out_pix__pos_v == out_pix__res_v)
170                         out_pix__pos_v <= 10'd0;
171                     else
172                         out_pix__pos_v <= out_pix__pos_v + 1'b1;
173                 end
174                 else
175                     out_pix__pos_h <= out_pix__pos_h + 1'b1;
176             end
177         end
178 end
179
180 endmodule
```

Anexo C: Código umbralización

El código fuente completo de la aplicación se presenta a continuación:

```
filter rgb24_to_grayscale( input pix_stream in_pix, output pix_stream out_pix ) {
    // Set (require) input format
    in_pix.set_format( depth={8, 8, 8} );
    // Set output format
    out_pix.set_format( depth={8} );
    // Declare temporal variables
    raw_bits(18) gray_conv;

    out_pix.data[0] = gray_conv[17:10];

    on_event( in_pix.new_data ) {
        gray_conv = (in_pix.data[0] + in_pix.data[1] + in_pix.data[2]) * 341;
        set_event( out_pix.new_data );
    }
}

filter umbralize( input pix_stream in_pix, output pix_stream out_pix ) {
    // Set (require) input format
    in_pix.set_format( depth={8} );
    // Set output format
    out_pix.set_format( depth={1} );
    // Declare variables
    unumber(8,0) threshold = 127;
    on_event( in_pix.new_data ) {
        if( cast(unumber(8,0),in_pix.data[0]) > threshold ) {
            out_pix.data[0] = cast( raw_bits(1), 1 );
        }
        else {
            out_pix.data[0] = cast( raw_bits(1), 0 );
        }
        set_event( out_pix.new_data );
    }
}

filter top( input pix_stream in_pix, output pix_stream out_pix ) {
    // Set (require) input format
    in_pix.set_format( res_h=1280, res_v=720, depth={8, 8, 8} );
```

```
38      // Set output format
39      out_pix.set_format( res_h=1280, res_v=720, depth={1} );
40      // Pipeline
41      pix_stream tmp;
42      tmp.set_format( depth={8} );
43      rgb24_to_grayscale( in_pix, tmp );
44      umbralize( tmp, out_pix );
45  }
```

Anexo D: Código RGB a HSL

D.1. Código SALTT

El código fuente completo de la aplicación se presenta a continuación:

```
1  maxmin (input pix_stream rgb_in, output raw_bits(9) add, output raw_bits(8) dif, output raw
2  {
3      rgb_in.set_format(depth = {8, 8, 8});
4      rgb_out.copy_format(rgb_in);
5
6      on_event(rgb_in.new_data) {
7          if (rgb_in.data[1] > rgb_in.data[2]) {
8              if (rgb_in.data[1] > rgb_in.data[0]) {
9                  sel = 1;
10                 if (rgb_in.data[2] < rgb_in.data[0]) {
11                     add = rgb_in.data[1] + rgb_in.data[2];
12                     dif = rgb_in.data[1] - rgb_in.data[2];
13                 }
14                 else {
15                     add = rgb_in.data[1] + rgb_in.data[0];
16                     dif = rgb_in.data[1] - rgb_in.data[0];
17                 }
18             }
19             else {
20                 sel = 0;
21                 add = rgb_in.data[0] + rgb_in.data[2];
22                 dif = rgb_in.data[0] - rgb_in.data[2];
23             }
24         }
25         else {
26             if (rgb_in.data[2] > rgb_in.data[0]) {
27                 sel = 2;
28                 if (rgb_in.data[1] < rgb_in.data[0]) {
29                     add = rgb_in.data[2] + rgb_in.data[1];
30                     dif = rgb_in.data[2] - rgb_in.data[1];
31                 }
32                 else {
33                     add = rgb_in.data[2] + rgb_in.data[0];
```

```
34                      dif = rgb_in.data[2] - rgb_in.data[0];
35                  }
36              }
37              else {
38                  sel = 0;
39                  add = rgb_in.data[0] + rgb_in.data[1];
40                  dif = rgb_in.data[0] - rgb_in.data[1];
41              }
42          }
43
44          rgb_out.data[0] = rgb_in.data[0];
45          rgb_out.data[1] = rgb_in.data[1];
46          rgb_out.data[2] = rgb_in.data[2];
47          set_event(rgb_out.new_data);
48      }
49  }
50
51  calcL (input raw_bits(9) add, input event new_data, output raw_bits(8) L)
52  {
53      raw_bits(8) tempL[2];
54      raw_bits(1) idx;
55
56
57      on_event (new_data) {
58          tempL[idx] = add[8:1];
59          L = tempL[~idx];
60          idx = ~idx;
61      }
62  }
63
64  calcS (input raw_bits(9) add, input raw_bits(8) dif, input event new_data, output raw_bits(
65  {
66      memory(16, 256, 1, BRAM) mult;
67      memory_port multPort = mult.port[0];
68
69      // memory(255,4)
70      raw_bits(8) idx;
71      raw_bits(16) currentMult;
72
73      raw_bits(8) difTemp;
74      raw_bits(16) sTemp;
75
76      idx = add[8] ? 510 - add : add;
```

```
77      currentMult = multPort[idx, 1];
78      S = sTemp[15:8];
79
80      on_event(new_data) {
81          difTemp = dif;
82          sTemp = difTemp * currentMult;
83      }
84  }
85
86  calcH (input pix_stream rgb, input raw_bits(8) dif, input raw_bits(2) sel, output raw_bits(
87  {
88      memory(16, 256, 1, BRAM) mult;
89      memory_port multPort = mult.port[0];
90
91      number(17,0) currentMult;
92      number(9,0) dividend;
93
94      raw_bits(18) quotient;
95      // raw_bits(8) quotient;
96      raw_bits(2) selTemp[2];
97      raw_bits(1) idx;
98      event pipeEv;
99
100     rgb.set_format(depth = {8, 8, 8});
101
102     currentMult = multPort[dif, 1];
103     quotient = dividend * currentMult;
104
105     on_event(rgb.new_data) {
106         if (sel == 0)
107             dividend = rgb.data[1] - rgb.data[2];
108         else if (sel == 1)
109             dividend = rgb.data[2] - rgb.data[0];
110         else if (sel == 2)
111             dividend = rgb.data[0] - rgb.data[1];
112         else
113             dividend = 0;
114
115         selTemp[~idx] = sel;
116         idx = ~idx;
117
118         set_event(pipeEv);
119     }
```

```
120
121     on_event(pipeEv) {
122         if (selTemp[idx] == 0)
123             H = quotient[17] ? quotient[17:10] + 255 : quotient[17:10];
124         else if (selTemp[idx] == 1)
125             H = quotient[17:10] + 85;
126         else if (selTemp[idx] == 2)
127             H = quotient[17:10] + 170;
128         else
129             H = 0;
130
131         set_event(valid);
132     }
133 }
134
135 filter top(input pix_stream rgb, output pix_stream hsl)
136 {
137     rgb.set_format(depth = {8, 8, 8});
138     hsl.set_format(depth = {8, 8, 8});
139
140     pix_stream rgbTemp;
141     rgbTemp.copy_format(rgb);
142
143     raw_bits(9) add;
144     raw_bits(8) dif;
145     raw_bits(2) sel;
146
147     raw_bits(8) H, L, S;
148
149     // Maximum and minimum RGB calculation
150     maxmin(rgb, add, dif, sel, rgbTemp);
151     calcL(add, rgbTemp.new_data, L);
152     calcS(add, dif, rgbTemp.new_data, S);
153     calcH(rgbTemp, dif, sel, H, hsl.new_data);
154
155     hsl.data[0] = H;
156     hsl.data[1] = S;
157     hsl.data[2] = L;
158 }
```

D.2. Códigos generado en SystemVerilog

El código escrito en SystemVerilo se presenta a continuación:

```
1  module rgb2hslCustom (
2      input logic [7:0] R,
3      input logic [7:0] G,
4      input logic [7:0] B,
5
6      input logic clk,
7
8      output logic [7:0] H,
9      output logic [7:0] S,
10     output logic [7:0] L
11 );
12
13 // Maximum and minimum
14
15 logic [8:0] add;
16 logic [7:0] dif;
17 logic [1:0] max;
18 logic signed [8:0] dividendH;
19
20 always_ff @( posedge clk ) begin
21     if (G > B) begin
22         if (G > R) begin
23             max <= 2'd1;
24             dividendH <= B - R;
25             add <= G + ((R > B) ? B : R);
26             dif <= G - ((R > B) ? B : R);
27         end
28         else begin
29             max <= 2'd0;
30             dividendH <= G - B;
31             add <= R + B;
32             dif <= R - B;
33         end
34     end
35     else begin
36         if ( B > R ) begin
37             max <= 2'd2;
38             dividendH <= R - G;
39             add <= B + ((G > R) ? R : G);
```

```
40              dif <= B - ((G > R) ? R : G);
41          end else begin
42              max <= 2'd0;
43              dividendH <= G - B;
44              add <= R + G;
45              dif <= R - G;
46          end
47      end
48  end
49
50  // L
51
52  logic [7:0] L1;
53
54  always_ff @( posedge clk ) begin
55      L1 <= add[8:1];
56      L <= L1;
57  end
58
59  // S
60
61  logic [7:0] dif1;
62  logic [7:0] idxS;
63  logic [15:0] multS, sTemp;
64
65  LUT_S lutS (
66      .clka(clk),
67      .addra(idxS),
68      .douta(multS)
69  );
70
71  always_comb begin
72      idxS = add[8] ? 9'd510 - add : add;
73      sTemp = dif1 * multS;
74  end
75
76  always_ff @( posedge clk ) begin
77      dif1 <= dif;
78      S <= sTemp[15:8];
79  end
80
81  // H
82
```

```
83  logic [1:0] max1;
84  logic signed [8:0] dividend;
85  logic [7:0] idxH;
86  logic signed [16:0] multH;
87  logic signed [17:0] hTemp;
88
89  LUT_H lutH (
90      .clka(clk),
91      .addra(idxH),
92      .douta(multH[15:0])
93  );
94
95  always_comb begin
96      idxH = dif;
97      multH[16] = 1'b0;
98      hTemp = dividend * multH;
99  end
100
101 always_ff @( posedge clk ) begin
102     dividend <= dividendH;
103
104     max1 <= max;
105
106     unique case(max1)
107         2'd0: H <= hTemp[17] ? 8'd255 + hTemp[17:10] : hTemp[17:10];
108         2'd1: H <= hTemp[17:10] + 7'd85;
109         2'd2: H <= hTemp[17:10] + 8'd170;
110     endcase
111 end
112
113 endmodule
```

Anexo E: Código filtro de media

El código fuente completo de la aplicación se presenta a continuación:

```
filter top (input pix_stream in_pix, output pix_stream out_pix)
{
    in_pix.set_format(res_h = 1920, res_v = 1080, depth = {8, 8, 8});
    out_pix.copy_format(in_pix);

    pix_stream R, G, B;
    pix_stream R_window, G_window, B_window;

    raw_bits(13) R_accum, G_accum, B_accum;
    raw_bits(18) R_proc, G_proc, B_proc;

    flt_dim_split(in_pix, R, G, B);

    R_window.set_format(depth = {8, 8, 8, 8, 8, 8, 8, 8, 8});
    G_window.copy_format(R_window);
    B_window.copy_format(R_window);

    flt_slide_window(R, R_window, 3, 0);
    flt_slide_window(G, G_window, 3, 0);
    flt_slide_window(B, B_window, 3, 0);

    R_accum = R_window.data[0] + R_window.data[1] + R_window.data[2] + R_window.data[3] + R
    G_accum = G_window.data[0] + G_window.data[1] + G_window.data[2] + G_window.data[3] + G
    B_accum = B_window.data[0] + B_window.data[1] + B_window.data[2] + B_window.data[3] + B

    out_pix.data[0] = R_proc[17:10];
    out_pix.data[1] = G_proc[17:10];
    out_pix.data[2] = B_proc[17:10];

    on_event(in_pix.new_data) {
        R_proc = R_accum * 114;
        G_proc = G_accum * 114;
        B_proc = B_accum * 114;

        set_event(out_pix.new_data);
    }
}
```

Anexo F: Código operador sobel

El código fuente completo de la aplicación se presenta a continuación:

```
1  filter rgb24_to_grayscale( input pix_stream in_pix, output pix_stream out_pix ) {
2      // Set (require) input format
3      in_pix.set_format( depth={8, 8, 8} );
4      // Set output format
5      out_pix.set_format( depth={8} );
6      // Declare temporal variables
7      raw_bits(18) gray_conv;
8
9      out_pix.data[0] = gray_conv[17:10];
10
11     on_event( in_pix.new_data ) {
12         gray_conv = (in_pix.data[0] + in_pix.data[1] + in_pix.data[2]) * 341;
13         set_event( out_pix.new_data );
14     }
15 }
16
17 filter top ( input pix_stream in_pix, output pix_stream out_pix ) {
18     in_pix.set_format( res_h = 1920, res_v = 1080, depth = {8, 8, 8} );
19     out_pix.copy_format(res_h = 1920, res_v = 1080, depth = {8} );
20
21     pix_stream gray, gray_window;
22     number(11,0) Gx, Gy;
23     number(10, 0) Gx_mod, Gy_mod;
24     raw_bits(11) G;
25
26     gray.set_format( res_h = 1920, res_v = 1080, depth = {8} );
27     gray_window.set_format(depth = {8, 8, 8, 8, 8, 8, 8, 8, 8});
28
29     rgb24_to_grayscale( in_pix, gray );
30     flt_slide_window(gray, gray_window, 3, 0);
31
32     Gx = gray_window.data[2] + gray_window.data[8] + (gray_window.data[5] << 1) - gray_wind
33     Gy = gray_window.data[0] + gray_window.data[2] + (gray_window.data[1] << 1) - gray_wind
34
35     Gx_mod = Gx[10] ? -Gx : Gx;
36     Gy_mod = Gy[10] ? -Gy : Gy;
37
```

```
38      G = Gx_mod + Gy_mod;
39
40      on_event(in_pix.new_data) {
41          out_pix.data[0] = G > 255 ? 255 : G;
42          set_event(out_pix.new_data);
43      }
44  }
```