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

Modelo y plataforma de cómputo heterogéneo para video infrarrojo

Tesis para optar al grado de Magíster en Ciencias de la Ingeniería con mención en Ingeniería Eléctrica

LUIS ALEJANDRO ARANEDA MENDEZ
CONCEPCIÓN-CHILE
2016

Profesor Guía: Dr. Miguel Figueroa T.
Comisión 1: Dr. Jorge Pezoa N.
Comisión 2: Dr. Marcos Diaz Q.
Dpto. de Ingeniería Eléctrica, Facultad de Ingeniería
Universidad de Concepción

Universidad de Concepción
Facultad de Ingeniería
Departamento de Ingeniería Eléctrica

Profesor Patrocinante:
Dr. Miguel Figueroa T.

Modelo y plataforma de cómputo heterogéneo para video infrarrojo

Luis Alejandro Araneda Mendez

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

Concepción, Chile.
Octubre de 2016

Agradecimientos

Durante el desarrollo de este trabajo viví muchas experiencias que recordaré por el resto de mi vida. Realicé un viaje de intercambio que me permitió ampliar la visión y perspectivas del mundo. Aprendí a manejar la frustración producida durante las jornadas en que no me resultaban las cosas, emoción que surge a menudo al realizar investigación. En resumen, fue una gran aventura en la que estoy feliz haberme embarcado. Por eso, le dedico las siguientes palabras a todas las personas que contribuyeron de alguna forma en este proceso.

Primero quiero agradecer a los integrantes de mi familia, mi padre, José; mi madre, María; y mi hermana, Viviana. Sin duda, ustedes han sido el soporte fundamental que me permitió terminar este trabajo. Gracias por el apoyo incondicional que han demostrado, incluso en esos ratos donde estaba frustrado y me ponía de mal humor, interrumpiendo esa tranquilidad característica de nuestro hogar. Siempre que pienso en ustedes agradezco el pertenecer a una familia tan unida y linda como la nuestra.

Agradezco a mis amigos, en especial a José, Javier, Alberto, Miguel, y Marcelo, por la preocupación y el ánimo que me dieron para terminar este trabajo, incluso en los momentos que pensaba abandonarlo; las invitaciones a comer al final de esos días de trabajo en los que terminaba frustrado; y esas conversaciones que fueron una mezcla agradable de cosas sin sentido, consejos, y experiencias personales.

Agradezco a mis compañeros del Laboratorio de VLSI que han sido excelentes acompañantes en este *viaje*. Javier, Wladimir, Javier, y a los ya graduados, con los que he creado muchas memorias, que serán recordadas con nostalgia en futuros encuentros. A los nuevos integrantes del laboratorio, Antonio, Pablo, Ignacio, y Silvana, que llegaron llenos de motivación por aprender cosas nuevas e irradian alegría por todo el laboratorio.

Quiero agradecer también a mi profesor patrocinante, el Dr. Miguel Figueroa Toro, por la gran paciencia y comprensión que tuvo durante el desarrollo de este trabajo. Agradezco además sus consejos, y esos momentos en donde necesitaba que alguien me presionara para mantener la motivación y sacar adelante el trabajo.

Por último, agradecer a todas las personas que han formado parte de mi vida en estos últimos años, y que contribuyeron en mi crecimiento personal y profesional.

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

Los sistemas embebidos con los que interactuamos día a día han ido creciendo a lo largo de los años, llegando a estar en dispositivos tan comunes como un reloj. A su vez, los dispositivos que ya incorporaban algún sistema embebido, como las cámaras digitales, han visto un drástico aumento en su capacidad de cómputo, permitiendo ampliar las funcionalidades ofrecidas a los usuarios. El aumento en las capacidades de cómputo se debe principalmente a la reducción en las tecnologías de manufactura de semiconductores, siendo ahora posible colocar más del doble de transistores que hace dos años.

El aumento paulatino en la capacidad de cómputo ha impuesto un creciente desafío en el diseño de estos sistemas, ya que ahora hay una gran cantidad de recursos lógicos disponibles y se deben organizar de manera adecuada para terminar un producto y poderlos sacar al mercado en un plazo competitivo, debiendo aplicar diversas estrategias, como reutilizar partes de productos anteriores.

En este trabajo se propone un modelo de cómputo heterogéneo con el cual se puedan describir sistemas embebidos para procesamiento de video infrarrojo, incorporando la posibilidad de realizar co-procesamiento entre los elementos y un procesador de propósito general. Para complementar el modelo, se propone un lenguaje de programación de dominio específico. Con esto, se puede acelerar el desarrollo e investigación, lo que a la vez permite el ingreso de diseñadores que no están familiarizados con conceptos de bajo nivel.

Para diseñar el modelo de cómputo se analizan varios trabajos realizados sobre procesamiento embebido de video infrarrojo, extrayendo las características y elementos comunes entre ellos. Con esta información se define el modelo, que consiste en varias definiciones de elementos de procesamiento, y reglas que deben seguir las implementaciones. Además, el modelo define varios elementos de procesamiento básicos estandarizados que pueden ser usados por las implementaciones. El lenguaje de programación toma características de otros lenguajes, y añade algunas propias para generar un mayor grado de abstracción. Finalmente, se presenta una arquitectura de hardware base con detalles útiles para las posibles implementaciones.

Índice General

Índice de Figuras

Siglas

API interfaz de programación de aplicaciones (del inglés *Application Programming Interface)*

ARMA modelo autorregresivo de media móvil (del inglés *Autoregressive–Moving-Average Model)*

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

CMOS semiconductor complementario de óxido metálico (del inglés *Complementary Metal Oxide Semiconductor)*

DMA acceso directo a memoria (del inglés *Direct Memory Access)*

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

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

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

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

IRFPA arreglo infrarrojo de plano focal (del inglés *Infrared Focal Plane Array)*

LBP patrones binarios locales (del inglés *Local Binary Patterns)*

LDA Análisis Discriminante Lineal (del inglés *Linear discriminant analysis)*

NUC corrección de no uniformidad (del inglés *Non-Uniformity Correction)*

PL lógica programable (del inglés *Programmable Logic)*

PS sistema de procesamiento (del inglés *Processing System)*

RAM memoria de acceso aleatorio (del inglés *random-access memory)*

RGB rojo, verde y azul (del inglés *red, green, blue)*

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

SVM máquina de vector soporte (del inglés *support vector machine)*

TDMA acceso múltiple por división de tiempo (del inglés *Time Division Multiple Access)*

XML lenguaje de marcas extensible (del inglés *Extensible Markup Language)*

Capítulo 1: Introducción

1.1. Introdución general

Los sistemas embebidos son unidades de procesamiento diseñados para realizar tareas específicas, usualmente con restricciones de tiempo real. Se caracterizan por sus reducidas dimensiones y la especialización en las tareas que realizan. La Figura1.1 muestra algunos ejemplos de sistemas embebidos. Además, los podemos encontrar en muchos de los dispositivos con los que interactuamos día a día, como cargadores de baterías, teléfonos celulares inteligentes, y cámaras entre otros, y pueden ser tan simples como un procesador y un par de componentes extras, o tan complejos como tener múltiples procesadores ejecutando un sistema operativo completo en conjunto con varias unidades de procesamiento dedicadas.

Un sistema embebido usualmente incluye un procesador de propósito general encargado de orquestar uno o varios componentes de procesamiento dedicado, como procesadores de señales, microcontroladores, arreglo de compuertas programables (del inglés *Field Programmable Gate Array*, FPGA), y cualquier otro circuito integrado de aplicación específica (del inglés *Application-Specific Integrated Circuit*, ASIC) necesario.

En general, la tarea que más recursos y tiempo consume es el diseño e interacción de los componentes que procesan los datos, tratando de reducir toda la funcionalidad a un solo circuito integrado, lo que reduce costos y posibles fallas. Debido a esto, usualmente se utiliza un sistema en un chip (del inglés *System on a Chip*, SoC), circuito que incorpora una variada cantidad de

(a) Sintonizador de televisión digital

(b) Medidor de glucosa

(c) Placa de desarrollo

Fig. 1.1: Ejemplos de sistemas embebidos

subsistemas comunicados entre sí, capaces de ejecutar el software de control y el procesamiento de los datos.

Con el aumento en la densidad de los circuitos integrados, algunos FPGAs ahora incorporan procesadores de propósito general junto con una variedad de periféricos asociados, lo que los transforma en un SoC capaz de implementar un sistema embebido completo con aún menos componentes, [1, 2]. Debido a que el procesador y la lógica programable están en el mismo chip, el ancho de banda entre ellos aumenta drásticamente, dando lugar a aplicaciones que antes no eran posibles. Una de ellas es el trabajo en conjunto de componentes de hardware y software, co-procesando datos e intercambiando información entre ambos [3, 4]. Esto lograría combinar las fortalezas de un procesador, como operación a altas frecuencias, operaciones aritméticas, punto flotante, y facilidad de programación; con las de un FPGA, como el paralelismo y la especialización alcanzada, para lograr implementar sistemas de cómputo heterogéneos. Si se le añade además la capacidad de reconfiguración parcial en tiempo de funcionamiento, una de las características únicas que poseen algunos FPGAs, el sistema se hace aún más denso, pudiendo incluir una mayor cantidad de aplicaciones en el mismo espacio.

El diseño de aplicaciones se hace con diversas herramientas, como compiladores de software para el procesador, y herramientas para síntesis de lógica en el caso de los FPGAs. Si bien existen herramientas que simplifican estos procesos para que puedan ser usados por diseñadores no especialistas, resultan en sistemas poco eficientes en el uso de recursos del dispositivo y con un pobre rendimiento [5, 6]. Esto se debe a la gran cantidad de grados de libertad disponibles y las abstracciones necesarias para simplificar el diseño.

En este trabajo, son de especial interés los sistemas embebidos que capturan y procesan video infrarrojo. Dentro de las aplicaciones típicas se incluyen biomédicas, como detección de tumores de forma no invasiva; industriales, como detección temprana de fallas en piezas mecánicas o sistemas eléctricos; y tele-vigilancia, como visión nocturna y detección de personas en zonas de rescate, [7, 8, 9, 10, 11]. La entrada a estos sistemas, radiación infrarroja, es un tipo de onda electromagnética que abarca el rango de frecuencias mayores a las microondas y menores a la luz visible. Usualmente se captura usando un arreglo infrarrojo de plano focal (del inglés *Infrared Focal Plane Array*, IRFPA). Una vez capturada la imagen, y antes de que esté disponible para ser utilizada por algoritmos de procesamiento, se deben corregir los problemas intrínsecos del IRFPA y los circuitos de lectura, realizando correcciones de no uniformidad, reemplazo de píxeles muertos y estabilización de temperatura, entre otros. Luego, se pueden aplicar diversos filtros, como estabilización de movimiento y súper-resolución, que facilitan el uso de las imágenes al resto de los algoritmos en la cadena de procesamiento. Finalmente, los

fabricantes de cámaras infrarrojas incluyen procesadores de señales e incluso SoCs programables con FPGAs incorporados, pero típicamente los usuarios sólo pueden activar o desactivar ciertos elementos de la cadena de procesamiento, sin la posibilidad de agregar los suyos.

Los sistemas de procesamiento de video embebido contienen usualmente un pipeline donde se van procesando las imágenes a medida que son adquiridas, adicionando algunos elementos externos, como memorias, en caso de ser necesarios para el procesamiento. Por el hecho de procesar video, que consiste en una gran cantidad de imágenes por segundo, estos sistemas tienen altos requerimientos de ancho de banda para el procesamiento, lo que los hace buenos candidatos para el co-procesamiento de datos en una arquitectura heterogénea como la recién mencionada. Sin embargo, todavía existe la brecha del conocimiento previo para el diseño.

Considerando lo anteriormente mencionado, este trabajo propone el diseño de un modelo de cómputo para video infrarrojo que sea capaz de implementarse en un sistema de procesamiento heterogéneo de manera eficiente, y que a la vez logre abstraer varios conceptos esenciales propios del diseño de circuitos digitales, como buses de comunicación y sincronización entre unidades de procesamiento. Incluyendo además una arquitectura de referencia y un flujo de trabajo para desarrollar aplicaciones. Con lo que se logra simplificar el proceso de diseño y atraer más diseñadores a él.

1.2. Estado del arte

A continuación se presentan una serie de trabajos organizados en los distintos temas de interés para este trabajo, como lo son el procesamiento de video infrarrojo, destacando los problemas que se presentan al momento de su diseño; los lenguajes de dominio específico, que permiten simplificar el diseño de sistemas; la reconfiguración parcial de FPGAs, que permite compartir recursos y funciones dentro del sistema; finalizando en el procesamiento de video en sistemas embebidos.

1.2.1. Procesamiento de video infrarrojo

Una característica que comparten los IRFPAs, y que es atribuida principalmente a imperfecciones en la fabricación, es la respuesta no uniforme de sus sensores, tanto espacial como temporal [12]. Para corregirlo se realiza un proceso conocido como corrección de no uniformidad (del inglés *Non-Uniformity Correction*, NUC), que debe estar presente incluso en el más

básico de los sistemas. Este proceso es usualmente realizado por algoritmos de procesamiento de señales, como filtros de Kalman, [13]; filtros espaciales y temporales, [14, 15]; e incluso redes neuronales artificiales, [7]. Sin importar cual sea la forma de realizar la corrección, los algoritmos usados son computacionalmente intensivos con características de procesamiento paralelo, lo que los hace adecuados para implementar en arquitecturas dedicadas de hardware.

Otro proceso usualmente realizado antes de tener imágenes útiles es el reemplazo de píxeles defectuosos, encargado de remover y reemplazar de la imagen los píxeles que no cumplen con cierto criterio, como que su ganancia sea muy alta o baja con respecto al resto. Típicas implementaciones realizan el promedio de los píxeles, [16], o una suma ponderada, [17]. Al igual que NUC, este tipo de algoritmos cuenta con características que permiten su eficiente implementación en arquitecturas de hardware dedicadas.

Una vez corregida la imagen, puede ser procesada por uno o más algoritmos de procesamiento de imagen, como filtros digitales, registro de imagen, [8], mejoramiento de contraste, [18], entre otros. Estos algoritmos pueden ser iguales o muy similares a los ocupados para el espectro visible, lo que permite acelerar la investigación y desarrollo al usar algoritmos ya probados, realizando sólo pequeñas modificaciones.

Finalmente, se pueden aplicar algoritmos de visión por computador, considerados de más alto nivel, como detección de rostros [9, 10], o detección de fallas [19], los que al igual que la mayoría de los algoritmos mencionados, pueden ser implementados de forma eficiente en una arquitectura de hardware personalizada.

1.2.2. Lenguajes de dominio específico

Un lenguaje de domino específico (del inglés *Domain-Specific Language*, DSL) es un lenguaje de programación desarrollado para resolver un problema particular, pero extensible a una clase de problemas con características similares. Los problemas y sus soluciones son expresados de forma más clara y simple que ocupando un lenguaje de programación de propósito general. Específicamente, en el procesamiento de video embebido, apuntan a simplificar conceptos del diseño de hardware, logrando acelerar el proceso de diseño.

Un ejemplo de lenguaje DSL es Darkroom, propuesto en [20]. Opera construyendo pipelines de procesamiento de video. Cada uno de los filtros de procesamiento opera a nivel de líneas de video, eliminando la necesidad de acceder a memoria fuera del chip. Cuenta con dos tipos básicos de operaciones: Pointwise, que operan píxel a píxel, por lo que no requieren buffers; y Stencils,

los que usan una ventana de entrada (vecindad) para producir un píxel de salida, por lo que son necesarios buffers de línea. Una de las ventajas en las herramientas desarrolladas para Darkroom es que pueden generar códigos en Verilog sintetizable para FPGAs y ASICs, o código ejecutable en un procesador para propósitos de pruebas y validación. Sin embargo, está desarrollado en un lenguaje de programación de los mismos desarrolladores pero poco masificado, llamado Terra, lo que puede limitar el uso y soporte en el futuro. Finalmente el proceso de compilación del lenguaje se hace en varias etapas, entre las que destaca la minimización de los buffers de línea, que es resuelto por una herramienta externa como un problema de optimización de scheduling.

Otro ejemplo es el lenguaje llamado CAPH propuesto en [21]. Está enfocado en implementar arquitecturas de procesamiento stream específicas para FPGAs. Fue creado por la necesidad de contar con un lenguaje de programación específico para procesamiento de imágenes, pues las soluciones actuales, como el uso del lenguaje C para describir hardware, logran las abstracciones usando directivas al compilador, lo que si bien logra su cometido, no lo hace de una forma limpia y clara. Los programas son descritos como redes en las que los datos fluyen por un canal, típicamente en colas de tipo "primero en entrar, primero en salir" (del inglés *First In, First Out*, FIFO), en una sola dirección entre los nodos. La ejecución ocurre de acuerdo a un set de reglas, que definen el comportamiento de cada actor de manera independiente al resto. Finalmente, el lenguaje cuenta con un intérprete de referencia y un compilador cuya salida puede ser en SystemC o VHDL sintetizable.

1.2.3. Reconfiguración parcial de FPGAs

La reconfiguración parcial permite poner en un mismo lugar del FPGA unidades funcionales necesarias en distintos momentos del funcionamiento, lo que otorga una reducción de los recursos totales usados. Al reducir el nivel de ocupación y aumentar la funcionalidad, se obtiene una mayor densidad de funciones en el chip. Para un sistema embebido de procesamiento de video, esto significa que puede albergar más filtros de procesamiento en el mismo espacio físico. También permite la coexistencia de características que ocupan una gran cantidad de recursos del chip y que ahora pueden ser activadas sólo cuando son necesarias.

Si bien esta característica es útil, al momento de escribir este informe, no se ha encontrado un flujo de diseño que sirva para varios fabricantes de FPGAs, pues cada uno la implementa de una forma específica según sus productos disponibles. Aunque no haya un flujo de trabajo común, sí hay ciertos elementos relacionados.

Dos de los fabricantes de FPGAs más conocidos en este momento, Altera y Xilinx, presentan guías sobre como realizar este proceso en [22, capítulo 4] y [23] respectivamente. En general, el proceso puede ser divido en dos tareas: Síntesis de archivos, donde se generan varios archivos para reconfigurar el FPGA; y reconfiguración de la lógica, donde se reconfigura el FPGA dinámicamente durante el funcionamiento de acuerdo a un criterio establecido. La primera tarea, para FPGAs Xilinx, utiliza la herramienta Vivado en modo non-project a través de comandos de configuración/ejecución. Son varias las etapas que la componen, debiendo sintetizar por separado los elementos reconfigurables para luego agregarlos uno a uno al diseño final. El resultado de esta tarea es un archivo de configuración con todos los elementos por defecto, y varios archivos más pequeños con las variaciones de cada uno de los elementos reconfigurables. La segunda tarea es la más compleja, pues para un sistema embebido requiere que se reconfigure a sí mismo. Para esto hay diversas soluciones, como utilizar una parte de la lógica estática, llamados controladores de reconfiguración parcial [24, 25], o a través de un sistema externo como un microcontrolador, [26]. Sin embargo, últimamente han aparecido en el mercado FPGAs con procesadores de propósito general incorporado, simplificando todo el esquema de reconfiguración parcial en funcionamiento, pues ya no se necesita de un sistema externo o lógica dedicada para realizar la reconfiguración. Un ejemplo de esto, para FPGAs de la serie Zynq de Xilinx, se muestra en [27], donde se diseña un sistema de procesamiento de video, cuyos filtros son configurados por el procesador mientras funciona el dispositivo. En este contexto, un buen análisis sobre como realizar la reconfiguración parcial se realiza en [28], mostrando tres enfoques distintos. El primero de ellos es la reconfiguración desde el sistema de procesamiento (del inglés *Processing System*, PS), utilizando un conjunto de funciones dentro de una interfaz de programación de aplicaciones (del inglés *Application Programming Interface*, API) llamada "Processor configuration access port" (PCAP). Este enfoque tiene la ventaja de no requerir ningún recurso lógico, por lo que es posible usar completamente el área asignada a lógica programable (del inglés *Programmable Logic*, PL). La función que realiza la reconfiguración tiene la desventaja de ser una llamada bloqueante, esto significa que el procesador queda bloqueado hasta que el proceso termine. El segundo método de reconfiguración analizado es un IP-core de Xilinx llamado "internal configuration access port" (ICAP), e incluye drivers para realizar un llamado desde el procesador, que también resulta ser bloqueante. Este enfoque, a pesar ser más lento y ocupar parte del PL, tiene la ventaja de poder liberar al procesador si es que se controla totalmente del PL, desacoplando completamente los sistemas. El tercer método es el propuesto por los autores del trabajo, [28], y consiste en el diseño de un IP-core de código abierto llamado ZyCAP, que cuenta con drivers para el procesador y una interfaz para la lógica. Además, cuando es utilizado desde el procesador el llamado no es bloqueante, siendo el proceso controlado mediante interrupciones. Como última adición implementaron acceso directo a

memoria (del inglés *Direct Memory Access*, DMA), lo que resulta en la alternativa con mayor velocidad de las opciones analizadas.

1.2.4. Procesamiento de video en sistemas embebidos

Un tópico que ha cobrado relevancia actualmente son las cámaras inteligentes y el procesamiento que se puede realizar con ellas. En este contexto, se presentan una serie de trabajos de un grupo de autores correspondientes a los distintos subsistemas de una red compuesta por cámaras inteligentes. Tener una red de cámaras permite hacer procesamiento descentralizado, donde cada nodo procesa parte del algoritmo. En [29] se describen los trabajos iniciales de la plataforma de cámaras propuesta. Cada cámara, o nodo de procesamiento, está compuesta por un sensor de imagen con tecnología semiconductor complementario de óxido metálico (del inglés *Complementary Metal Oxide Semiconductor*, CMOS), un FPGA con lógica de control y procesamiento, seis memorias externas, y conectores Ethernet para comunicación, todo funcionando a una resolución de 1240 x 1080 píxeles. La infraestructura de red implementada consiste en dos nodos de procesamiento, un switch Ethernet para interconectarlos, y un computador para interceptar el tráfico y realizar validaciones. Los nodos realizan el procesamiento de un algoritmo de forma distribuida utilizando la red Ethernet como un enlace de comunicación de alta velocidad. Una vista general de la plataforma es presentada por [30], introduciendo las capacidades dinámicas de los nodos al ser configurados a través de la red. Además muestra esbozos del diseño interno de la lógica programable y sus capacidades de procesar distintos algoritmos. Este último concepto es ampliado en [26], donde además se muestra la arquitectura completa de la lógica programable. El hardware ha sido levemente modificado añadiendo un microcontrolador para las tareas de manejo de red y reconfiguración del FPGA, además de un transceptor para comunicación inalámbrica.

La arquitectura de la lógica programable tiene la forma de un pipeline de procesamiento de imágenes, y está constituida por dos elementos principales. El primero de ellos es llamado “RouteMatrix”, y es el encargado de transmitir los datos entre los módulos de procesamiento, pudiendo tomar distintos caminos según los registros de configuración que posee, lo que le brinda la capacidad de cambiar durante el funcionamiento los algoritmos de procesamiento por opciones programadas previamente en la lógica. El segundo es llamado “Elab”, e implementa distintos algoritmos orientados al procesamiento de video, como extracción de fondo, corrección de gama, etc. Todos los módulos “Elab” son almacenados en una biblioteca propia, que se ocupa en el proceso de diseño del pipeline de procesamiento. Cada elemento de la biblioteca tiene asociadas algunas propiedades, como su latencia, lo que permite utilizar software de

automatización para el diseño. El ejemplo mostrado procesa video a una resolución de 320 x 240 píxeles (Q-VGA) trabajando a 50 MHz, e incluye un módulo de adquisición para el sensor CMOS, tres módulos "RouteMatrix" y cuatro "Elab", que hacen cálculos de histograma y filtros de gradiente espacial. El uso de recursos del FPGA es cercano al 40 %, con el módulo de histograma consumiendo una gran cantidad de memoria debido a los datos que debe almacenar. El último trabajo en la serie, [31], describe el rol del sistema operativo en cada nodo, siendo el encargado de las comunicaciones, control y configuración del FPGA durante su funcionamiento. El sistema operativo usado es ERIKA OS, de tiempo real y con un bajo consumo de memoria flash (entre 1 y 4Kb), característica clave al ejecutarlo en un microcontrolador. La comunicación se hace con un transceptor inalámbrico IEEE 802.15.4, y los datos que circulan por la red son datos procesados, de bajo ancho de banda, lo que permite bajo consumo y bajo costo. El ejemplo mostrado implementa un algoritmo para detección, descrito en lenguaje CAPH [21], de otros vehículos en el campo de visión de la cámara usando histograma de gradientes orientados. Procesa video a una resolución de 1280 x 960 píxeles, con un uso de recursos cercano al 20 % en un FPGA Altera Cyclone III EP3C120. Si bien los sistemas presentados por los autores forman una solución completa, hay dos características que se podrían agregar para hacerlo aún más completo. La primera de ellas es la migración del microcontrolador usado para control y manejo a un procesador interno del SoC, eliminando un componente externo. La segunda es un cambio más profundo, e implica agregar una forma de realizar co-procesamiento con procesadores de propósito general, pues la solución presentada sólo incluye el procesamiento realizado por lógica programable.

Xilinx plantea formas de diseñar pipelines de procesamiento de video en dos notas de aplicación. La primera, [32], muestra la versión 2.1 del motor de video en tiempo real, que añade soporte para la serie Zynq de FPGAs, liberando los recursos ocupados en versiones anteriores por el procesador MicroBlaze. El sistema soporta hasta ocho streams de entrada de video y dos de salida, con hasta seis pipelines para el FPGA más grande de la serie Zynq. El procesador provee una plataforma de control usando GNU/Linux 3.3 y el entorno gráfico Qt, en el que se pueden desarrollar aplicaciones de usuario. Los datos son procesados por la lógica programable utilizando distintas versiones del bus AXI para control, acceso a memoria externa, y transferencia entre unidades funcionales del pipeline. La gran variedad de filtros disponibles están basados en la plataforma propietaria *OmniTek Scalable Video Processor* (OSVP), los que son capaces de procesar video a una resolución de 1920 × 1080 píxeles a una tasa de 60 cuadros por segundo. Para configurar los filtros en tiempo de ejecución, se requiere la carga de módulos propietarios en el Kernel, que junto con una serie de APIs forman una plataforma robusta, pues permite la migración hacia otros sistemas operativos y tipos de conexiones sin cambiar las aplicaciones

de usuario escritas. La implementación del sistema se hace sobre la tarjeta OmniTek OZ745, que debe ser comprada junto con licencias para acceder al código fuente o archivos compilados. La segunda nota de aplicación, [33], muestra un sistema bastante similar a la primera, pero utiliza sólo filtros pertenecientes a Xilinx y en una tarjeta de desarrollo más asequible. Si bien los resultados no son directamente comparables, este sistema no cuenta con la versatilidad del primero, pero tiene la ventaja de que sólo necesita las licencias de Xilinx. Con respecto al bus AXI, demostró ser una alternativa versátil en la implementación, pero la sobrecarga en el uso de recursos lógicos es notoria al utilizarlo para el pipeline de procesamiento, área donde no puede superar un bus desarrollado específicamente para la aplicación.

1.3. Discusión

El procesamiento de video infrarrojo presenta varias etapas de pre-procesamiento para obtener una imagen con la que trabajar, y varias de post-procesamiento antes de obtener resultados interpretables por sus usuarios. En cualquier caso, son escasas las opciones que tienen los usuarios para personalizar el procesamiento realizado por el dispositivo. Por esta razón es necesario el desarrollo de modelos y arquitecturas que faciliten la creación de diseños personalizados.

En general, los lenguajes de programación DSL son una buena forma de abstraer un diseño de bajo nivel y acelerar el desarrollo. Sin embargo, los lenguajes analizados carecen de la capacidad de controlar aspectos de bajo nivel, como la latencia entre distintas unidades de procesamiento. Además, no cuentan con la capacidad que permita realizar co-procesamiento entre unidades de hardware dedicadas y procesadores de propósito general. Una vez agregadas estas características, estos lenguajes se transforman en poderosas herramientas adecuadas tanto para programadores que se inician en el mundo del hardware, como para programadores con experiencia y que necesitan características específicas.

Es claro que el uso de la reconfiguración parcial en FPGA lograría crear un sistema compacto que incorpore más funcionalidades que un equivalente sin reconfiguración parcial, pero aún existen algunas trabas si se requiere trabajar con más de un fabricante al mismo tiempo, pues el proceso no está actualmente estandarizado. Una vez estandarizado el proceso, estos sistemas se transformarán en comunes y la funcionalidad de los dispositivos que las usen aumentarán.

Con respecto al procesamiento de video en sistemas embebidos, los trabajos analizados son buenos en términos de rendimiento, pero no incorporan la capacidad de realizar co-procesamiento con un procesador de propósito general, lo que requeriría mayores modificaciones al sistema com-

pleto, eliminando la ventaja que tienen de usar simples interconexiones al momento de realizar el diseño.

1.4. Hipótesis

Es posible explotar las características comunes de los algoritmos de procesamiento de video infrarrojo, para diseñar un modelo de cómputo que provea los elementos fundamentales de procesamiento, creando un sistema que posea un mayor grado de abstracción y permita acelerar la creación de este tipo de aplicaciones.

1.5. Objetivos

1.5.1. Objetivos generales

El objetivo de este trabajo es diseñar un modelo de cómputo heterogéneo y una implementación de referencia, compuesto por procesadores de propósito general y aceleradores de cómputo personalizados, para aplicaciones de visión por computador en tiempo real enfocadas en el procesamiento de video infrarrojo.

1.5.2. Objetivos específicos

1. Proponer un modelo de cómputo heterogéneo compuesto por CPUs y aceleradores hardware diseñados con lógica programable.
2. Diseñar una arquitectura que soporte el modelo propuesto.
3. Validar la arquitectura propuesta con la implementación de aplicaciones de prueba sobre un FPGA Zynq de Xilinx.
4. Diseñar un método para controlar y automatizar la reconfiguración parcial dentro de un FPGA.
5. Implementar schedulers para software y hardware que permitan compartir recursos, como CPU o lógica.

1.6. 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 el modelo de cómputo propuesto. Para ello se analizan primero varios trabajos relevantes, extrayendo las similaridades y diferencias que permitan definir el modelo.
- Capítulo 3: Define formalmente un lenguaje de programación de dominio específico, basado en el modelo propuesto anteriormente.
- Capítulo 4: Entrega detalles sobre una arquitectura hardware base.
- Capítulo 5: Analiza herramientas de desarrollo computacionales y propone un flujo de diseño.
- Capítulo 6: Resume el trabajo realizado, presenta las conclusiones, y propone el trabajo a futuro.

Capítulo 2: Modelo de cómputo

El modelo de cómputo descrito en este capítulo se generó analizando varios trabajos relevantes para el procesamiento de video en sistemas embebidos. El análisis dio como resultado un conjunto de definiciones y reglas, que deben ser cumplidas por las implementaciones. Se presentan además, los razonamientos y decisiones tomadas en la generación del modelo.

2.1. Motivación

En general, cuando se diseña una arquitectura de hardware personalizada y su respectiva implementación, los objetivos son alcanzar una mayor eficiencia energética disminuyendo el consumo, y aumentar la velocidad de procesamiento, preferentemente para acelerar el cómputo realizado y procesar datos en tiempo real. Con estos objetivos, se diseña una arquitectura compuesta por elementos de cómputo personalizados, que permiten cumplir a cabalidad los objetivos.

Los elementos de cómputo personalizados hacen uso eficiente de recursos de la plataforma en que se vayan a implementar, como FPGAs de distintos fabricantes o ASICs; Otorgan una gran cantidad de grados de libertad al poder diseñar lógica de procesamiento, interfaces, y buses de comunicación ajustados a la aplicación; y logran la mayor velocidad de procesamiento al no tener que acarrear la sobrecarga de recursos que genera el uso estandarizado de componentes.

Crear un diseño de la forma recién descrita requiere que el diseñador maneje una gran cantidad de conceptos y habilidades específicas, lo que sube la barrera de entrada a diseñadores habituados a procesos de diseño de más alto nivel. Adicionalmente, los tiempos de diseño son altos, pues se debe realizar un diseño completo de la mayoría de los elementos en cada aplicación. Finalmente, una vez terminado e implementado el diseño, cuesta adaptarlo e integrarlo a otros sistemas de procesamiento, debido a que no cuenta con interfaces de comunicación estandarizadas.

2.2. Trabajos analizados

A continuación se presenta una breve descripción de algunos de los trabajos analizados que fueron considerados relevantes para la confección de los requerimientos y elementos necesarios del modelo:

- Redlich et al, [34], propone una forma de corregir la no-uniformidad en un IRFPA, utilizando un algoritmo de rangos constantes, que funciona estimando estadísticas de los pixeles en el tiempo usando un modelo de primer orden. La implementación utiliza aritmética de punto fijo y logra corregir en tiempo real 238 imágenes por segundo con una resolución de 640x480 píxeles y 14 bits por píxel.

- Wolf et al, [35], implementa un algoritmo de estabilización térmica de un IRFPA basado en microbolómetros no refrigerados, y usa un modelo autorregresivo de media móvil (del inglés *Autoregressive–Moving-Average Model*, ARMA) para modelar la dinámica presente entre los valores de temperatura del IRFPA entregados por el sensor de la cámara y el efecto que tiene en la medición.

- Vergara et al, [11], presenta un algoritmo de detección de rostros en infrarrojo. Proceso que es realizado en varias etapas: Extracción de características, clasificación, y encasillamiento. Además, se presenta una arquitectura de hardware y su implementación en un FPGA.

- Araneda et al, [36], implementa un algoritmo de estabilización digital de video, proceso que es dividido en 3 etapas secuenciales: estimación de movimiento, filtro de movimiento, y corrección de movimiento, siendo la estimación de movimiento la que requiere mayor capacidad de cómputo.

- Soto et al, [37], implementa un algoritmo de clasificación de rostros usando una cámara infrarroja. Usa patrones binarios locales (del inglés *Local Binary Patterns*, LBP) para extraer las características de los rostros. Luego la imagen es separada en regiones a las cuales se calcula el histograma, y se construye un histograma concatenado para mejorar la precisión. Finalmente se usa Análisis Discriminante Lineal (del inglés *Linear discriminant analysis*, LDA) para reducir la dimensionalidad y lograr almacenar una base de datos de 53 personas en un FPGA.

- Cárdenas et al, [38], implementa una arquitectura para la detección de manzanas golpeadas usando una selección de bandas de una cámara hiperespectral. Clasifica cada pixel en la imagen en 3 clases usando una máquina de vector soporte (del inglés *support vector*

machine, SVM): golpeada, no golpeada, y fondo. Finalmente, como resultado se calcula el porcentaje de golpes que presenta la manzana.

- Valenzuela et al, [39], implementa una compensación para el ruido de rayas (striping noise) producido en la adquisición de datos de una cámara hiperespectral. La compensación, implementada en un FPGA, hace uso de una red neuronal multi-dimensional para corregir el ruido en tiempo de adquisición.

2.3. Requerimientos mínimos de una implementación

En un sistema de procesamiento de video, la comunicación con el mundo exterior es fundamental. Se distinguen dos casos. El primero corresponde a la etapa de desarrollo, en el cual sólo son necesarias interfaces mínimas de validación, como un puerto serial. El segundo caso corresponde a un sistema totalmente funcional, destinado a ser usando en un producto, para el cual como mínimo se debe contar con una entrada y una salida de video.

Los sistemas analizados que acceden globalmente a los datos del cuadro de video requieren una forma de almacenar al menos un cuadro de video para análisis y/o posterior corrección. Los SoCs por su reducido tamaño, generalmente incluyen memorias locales que pueden servir en el mejor de los casos para almacenar un cuadro de video pequeño, por lo que un sistema que acceda globalmente a un cuadro de video debe tener como mínimo una memoria externa del tamaño suficiente para almacenar al menos un cuadro de video completo, siendo idealmente de dos o más cuadros. Este requerimiento no es necesario para sistemas simples que acceden a los datos de forma secuencial a medida que llegan al sistema.

2.4. Sistema de procesamiento de video

En todos los trabajos analizados se pudo identificar que el diseño completo estaba compuesto por varias secciones o divisiones lógicas, cada una encargada de una tarea específica. De especial interés para esté trabajo es la sección que realiza el procesamiento de video.

Al observar las entradas de datos y sus resultados, no todos los trabajos analizados utilizan una estructura similar para el procesamiento de video, teniendo algunos video como entrada y salida, [34]; video como entrada y datos como salida, [11]; o combinaciones mixtas, como video y datos como entrada, y video como salida, [35].

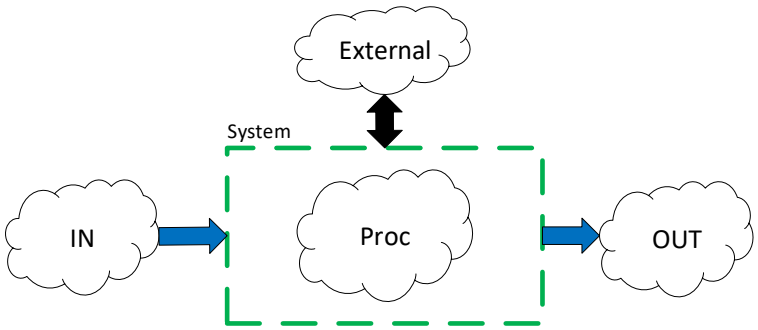


Fig. 2.1: Ejemplo de sistema de procesamiento de video

En cuanto a otros tipos de interacciones, si bien de forma externa se usaron interfaces estandarizadas para transmisión de datos, internamente se observó una gran variedad de interfaces y protocolos personalizados, diseñados específicamente para la aplicación con el objetivo de simplificar lo más posible el diseño y/o minimizar latencias.

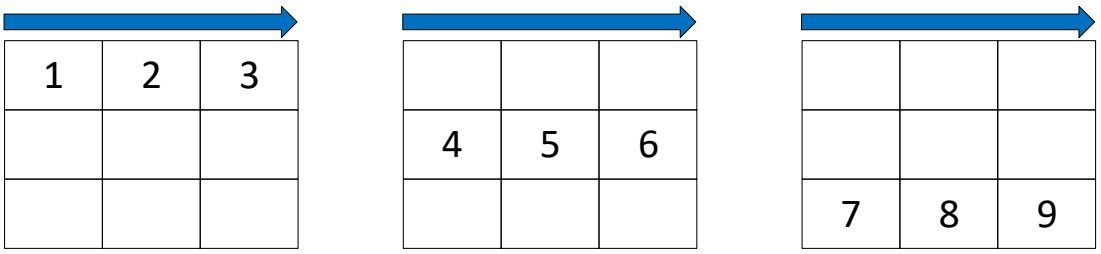
Una generalización de lo observado en los trabajos analizados permite realizar la siguiente definición para el modelo:

Definición 1. Sistema de procesamiento de video: Es un pipeline unidireccional, con la información fluyendo desde una única entrada de video, hacia una única salida de video. Opcionalmente, se permiten interacciones externas, pero deben ser totalmente estandarizadas por la implementación, como accesos a memorias alojadas fuera del sistema, o comunicación con elementos de procesamiento externo, como un procesador de propósito general.

Es importante destacar que no forman parte del sistema los elementos externos desarrollados para adaptar las entradas, salidas e interacciones externas a la forma de procesamiento del sistema.

La Figura 2.1 muestra un diseño completo, en la que el sistema de procesamiento de video está representado por líneas segmentadas verdes. La entrada y salida son obligatoriamente en forma de video, representada por flechas azules, y las interacciones externas se realizan usando un bus de comunicación estandarizado.

La estandarización de las interacciones del sistema permite acelerar el proceso de desarrollo e integración en otros diseños compatibles. Además, permite intercambiar o reemplazar funcionalidades, a la vez que facilita y fomenta la reutilización de trabajos existentes.



(a) Línea 1 (b) Línea 2 (c) Línea 3

Fig. 2.2: Ejemplos de transmisión de video

2.5. Stream de video

Un video está compuesto por una serie de cuadros (imágenes) ordenados en el tiempo. A su vez, cada uno de los cuadros está compuesto por pixeles, que contienen información asociada a su posición dentro del cuadro.

La Figura 2.2 ilustra, de forma simplificada, el proceso de transmisión de video en sistemas tradicionales (análogo, VGA, HDMI, Camera Link, etc.). La transmisión se realiza pixel a pixel, o en paquetes de información para algunos estándares, partiendo en la esquina superior izquierda, y avanzando hacia la derecha hasta completar una línea. Una vez completada la línea, generalmente hay un breve descanso (blanking horizontal), antes de comenzar a transmitir la siguiente línea. El proceso se repite hasta que el cuadro ha sido transmitido completamente, momento en el que generalmente se produce un descanso más prolongado (blanking vertical) antes de comenzar a transmitir el siguiente cuadro. La velocidad de transmisión del pixel, en conjunto con las duraciones de los descansos horizontales y verticales se conoce como timing de video, parámetros que ayudan en la caracterización de una transmisión. Finalmente, la cantidad de pixeles que componen un cuadro (tamaño) combinado con el timing de video, determinan el número de cuadros de video por segundo que se transmiten, parámetro conocido como tasa de refresco, medido en Hz.

En la mayoría de los trabajos analizados, la transferencia de video es tratada principalmente como una transferencia de líneas de video, procesando la información al completar la adquisición de una línea completa. Sin embargo, un análisis más profundo de las arquitecturas diseñadas, en particular en [36] para crear las proyecciones integrales, y en [35] para realizar correcciones de no uniformidad en línea, reveló que el procesamiento línea a línea se realiza porque las etapas anteriores producen resultados de esa forma. Es más, el acceso a la información contenida en las líneas de video se realiza de forma ordenada desde el pixel más antiguo al más nuevo (secuencial), transformando efectivamente el sistema en uno que procesa los datos pixel a pixel. Si se remueven

las primeras etapas de las arquitecturas, dejando solamente el sistema de procesamiento de video, algunos trabajos se pueden convertir fácilmente para que el procesamiento de los datos sea pixel a pixel.

Sobre el timing de video, los trabajos analizados sólo ocupan esa información para adquirir de una entrada de video fija, o para generar una salida válida que un monitor u otro periférico de despliegue pueda entender, no encontrándose una relación directa con los datos procesados por el sistema.

Como ya se mencionó, la unidad atómica en una transmisión de video es un pixel, y la información contenida por él varía mucho dependiendo de la aplicación. Entre los trabajos analizados se pueden identificar canales RGB de 8 bits cada uno y escala de grises de 8 bits, [36], cuentas digitales de una cámara infrarroja de 14 bits, [35], y bandas de una cámara hiperespectral, [38], entre otros. Debido a la gran diversidad encontrada, es deseable crear una división lógica (dimensión/canal) de la información contenida en el pixel, con el objetivo de facilitar el diseño e interpretación de los datos.

Una generalización de las observaciones anteriores permite realizar la siguiente definición para el modelo:

Definición 2. Stream de video: Es un flujo ordenado de cuadros de video. Cada cuadro se trasmite pixel a pixel, partiendo en la esquina superior izquierda, y avanzando hacia la derecha hasta completar una línea. Proceso que se repite hasta completar el cuadro completo. Opcionalmente pueden existir pausas entre pixeles, distintas líneas, o cuadros de video. Las características que definen a un stream de video son: El tamaño horizontal y vertical, en pixeles; las dimensiones o canales contenidos; y la profundidad de bits de cada dimensión.

El stream de video recién definido es lo suficientemente genérico como para implementar todos los trabajos analizados, requiriendo leves modificaciones a las arquitecturas de los trabajos analizados para adaptarlas al modelo.

Cabe destacar, que trasferir datos línea a línea es necesario en algunas aplicaciones, como en un sistema que trabaja como acelerador de cómputo para otro, y que ambos se comunican a través de un bus de propósito general. En este tipo de buses, realizar una transacción por cada pixel generalmente resulta en un rendimiento reducido debido al tiempo perdido en las señales de control para la transacción, por lo que es recomendable realizar al menos las transferencias línea a línea, para posteriormente convertir a procesamiento pixel a pixel si fuera posible. Este caso queda cubierto por los drivers de entrada y salida del modelo, definidos más adelante.

2.6. Hardware-threads

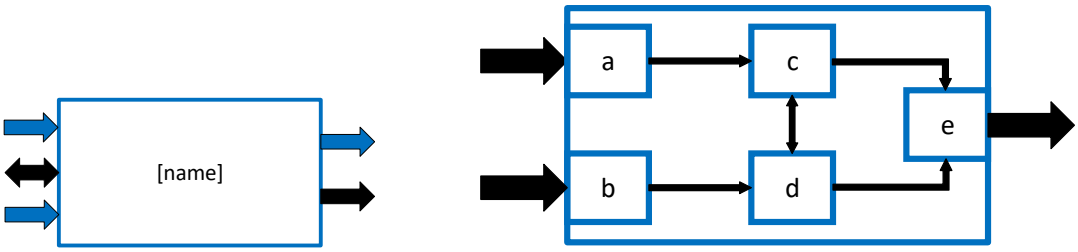
Una gran diversidad de elementos de procesamiento personalizado, específicos para cada aplicación, fueron encontrados en todos los trabajos analizados. Todos ellos, en alguna parte de la arquitectura diseñada transforman el video en la entrada a datos personalizados, con el objetivo de extraer información y obtenerla a la salida, o calcular parámetros para transformar el video y generar la salida. Si bien se reconocieron varios elementos comunes entre los trabajos, descritos en detalle más adelante, la mayoría permaneció específico a la aplicación. Estos elementos son característicos del algoritmo implementado en cada trabajo, por lo que cualquier esfuerzo por estandarizar las interfaces podría dañar la usabilidad y granularidad del modelo, al exigir al diseñador usar interfaces y elementos que tienen un alto nivel de abstracción para su aplicación. De todas formas, se considera necesario definir una mejor forma de describir un elemento de procesamiento, pues la etapa de análisis de este trabajo fue bastante ardua al no estar completamente documentadas las interfaces personalizadas usadas por los elementos de procesamiento, teniendo que consultar detalles con algunos de los autores de los trabajos. Por ello, se deben establecer criterios de diseño que apunten a la creación de elementos personalizados con descripciones claras sobre su funcionamiento, y que permita un grado de abstracción y descripción mayor al alcanzado en los lenguajes de descripción de hardware más comunes, como Verilog y VHDL.

Por las razones antes mencionadas, se realiza la siguiente definición para el modelo:

Definición 3. Hardware-thread: Es un elemento de cómputo que realiza una tarea específica. Se define a través de un protocolo o secuencia de ejecución, en donde un estímulo del medio (señal de inicio) desencadena la ejecución de una tarea, para finalmente entregar los resultados (señal de fin). No posee restricciones en las entradas y salidas, pudiendo tener varias de ellas y con protocolos de comunicación personalizados, aunque de preferencia estandarizados. La tarea realizada puede ser no determinística, y su término puede depender además de otras tareas. Internamente puede estar compuesto por otros hardware-threads, formando jerarquías, o lógica personalizada, como sumadores, multiplexores, registros, etc. Opcionalmente, posee parámetros de configuración para aumentar su versatilidad y fomentar la reutilización en otros diseños.

El hardware-thread definido permite crear elementos de cómputo universales, que procesan datos adicionales al video, y la definición de un protocolo permite una mejor caracterización al momento de tomar decisiones de diseño.

La Figura 2.3a ilustra el símbolo de un hardware-thread, compuesto por un rectángulo azul y



(a) Vista general (b) Jerarquía

Fig. 2.3: Ejemplos de un hardware-thread

un nombre descriptivo en su centro. Además, muestra un ejemplo de las diversas conexiones que se pueden realizar: La flecha azul representa un stream de video, la flecha negra unidireccional que representa una salida de datos personalizados, y la flecha negra bidireccional representa comunicación con otro hardware-thread o un acceso a memoria. Similarmente, la Figura 2.3b muestra una jerarquía de hardware-threads, con dos entradas de datos personalizados, diversas interacciones entre los hardware-threads internos, y una salida de datos personalizada.

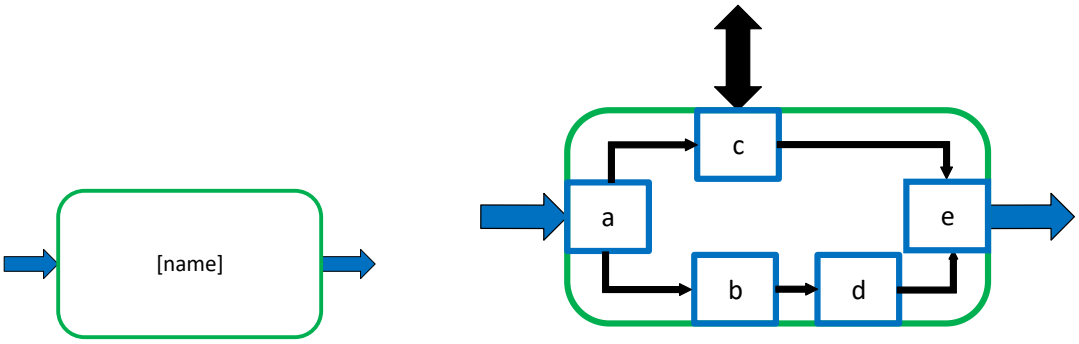
2.7. Filtros

Un hardware-thread es un elemento que permite hacer procesamiento totalmente personalizado, pero tiene tantas libertades que no ayuda mucho en la estandarización y reutilización que se quiere lograr con el modelo. Por esto, es necesario definir un elemento más restrictivo, enfocado solamente en el procesamiento de video, y que facilite la interconexión entre varios para formar un pipeline de procesamiento.

Definición 4. Filtro: Es un hardware-thread con restricciones específicas para un pipeline de video estandarizado. Debe tener exactamente una entrada y una salida en formato stream de video. Opcionalmente, puede tener interacciones externas, pero con protocolos definidos de forma estándar, como de acceso a memoria. Internamente, un filtro está compuesto por uno o más hardware-threads, almacenamiento local, y otros filtros (jerarquías).

El filtro definido tiene claras restricciones sobre sus interacciones, siendo explícitamente unidireccional, lo que facilita la creación de pipelines de procesamiento al permitir evaluar y comparar el desempeño de distintos filtros que realizan tareas similares.

La Figura 2.4a ilustra el símbolo de un filtro, compuesto por un rectángulo verde con esquinas redondeadas y un nombre descriptivo en su centro. Además, muestra un ejemplo de una conexión típica, en la que las flechas azules representan streams de video de entrada y salida. Similarmente,



(a) Vista general (b) Interior de un filtro

Fig. 2.4: Ejemplos de un filtro

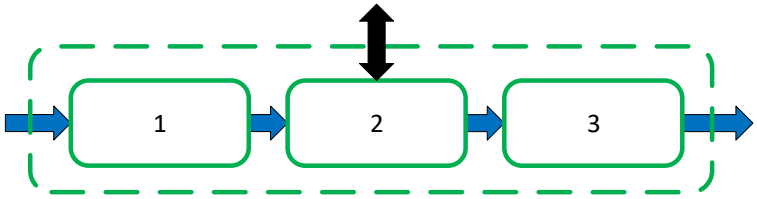


Fig. 2.5: Sistema visto como un filtro

la Figura 2.4b muestra el interior de un filtro, con la entrada y salida de video obligatoria, y una interacción externa estandarizada (acceso a memoria).

Si bien no todos los trabajos analizados tienen una salida de video, en particular, [37] y [11] tienen como salida la detección y clasificación de rostros en un formato personalizado; es necesario que todos los filtros la implementen para que puedan ser parte de un pipeline. Además, uno de los objetivos al definir un filtro como elemento de procesamiento más estricto, fue que pudiera representar completamente la parte de procesamiento de video de los trabajos, como memorias de título, o tesis de investigación, para poder comparar los resultados al reemplazar diferentes implementaciones del mismo algoritmo. Esto se puede apreciar gráficamente en la Figura 2.5, donde se podría fácilmente, por ejemplo, eliminar el filtro 1, o intercambiar el filtro 2 por el filtro 3 para probar los efectos que tienen estos cambios en la salida. También permite ilustrar que el sistema de procesamiento de video puede ser visto como un gran filtro, rodeado por una línea discontinua verde, gracias a que los filtros se pueden organizar en jerarquías.

Es importante destacar, que es necesaria una modificación a los trabajos que tienen salidas personalizadas de datos para que sean representables en el modelo. Una posible alternativa para la salida de datos personalizados, puede ser a través de la interfaz de memoria externa, definida posteriormente en este trabajo.

Finalmente, los filtros pueden ser clasificados según el acceso y el procesamiento que hagan del stream de video:

- Punto: Realizan operaciones pixel a pixel, requiriendo solamente la información del pixel actual en procesamiento para generar el pixel de salida.

- Locales: Realizan operaciones con el pixel actual y su vecindad, requiriendo acceso a una vecindad, usualmente llamada ventana, del pixel actual para generar el pixel de salida. Como este tipo de operaciones requiere más información que la contenida normalmente en un stream de video, es necesario un procesamiento previo, visto en detalle más adelante en este trabajo, para obtener la información de la vecindad.

- Globales: Realizan operaciones tomando uno o varios pixeles arbitrarios dentro del cuadro de video, pudiendo generar como pixel de salida otro pixel o pixeles arbitrarios. Los accesos requeridos por este tipo de filtro son solamente posibles internamente en un filtro usando un hardware-thread, ya que los accesos arbitrarios requieren algún tipo de almacenamiento para los cuadros de origen y destino. Además, si se realizan este tipo de operaciones, el filtro debe implementar una forma de generar o recrear un stream de video a partir de un elemento de almacenamiento. Detalles sobre este proceso son revisados con mayor profundidad más adelante en este trabajo.

2.8. Flujo de datos

Esta sección define un conjunto de reglas orientadas al flujo y manejo de datos dentro del modelo. Algunas son directamente derivadas de las definiciones anteriores, y otras de los trabajos analizados. A continuación de cada regla se presentan argumentos para su existencia e importancia en el modelo.

Regla 1: La entrada al sistema de procesamiento de video debe ser siempre en formato stream. Otros formatos deben adaptarse para ser procesados.

Como el sistema está compuesto por una jerarquía de filtros, a los cuales solamente puede entrar video en forma de stream, entonces la única posibilidad para entrar video al sistema es a través de un stream. Cabe recordar que el sistema de procesamiento de video puede ser parte de un diseño mayor, o incluso tener múltiples sistemas de procesamiento de video en un mismo diseño. Para estos casos, se deben desarrollar drivers que adapten el formato usado por el diseño, si es que no es compatible con el formato de stream de video definido por el modelo. Un ejemplo de esto sería el caso de tener un diseño que trabaje en base a líneas de video (line-buffers), o cuadros de video (frame-buffers), y se deba adaptar a un stream de video, teniendo

una implementación trivial ya que requiere solamente recorrer la línea o cuadro de video para extraer la información del pixel necesaria.

Regla 2: Durante el procesamiento interno puede no se mantenerse el timing de origen, es decir, los datos pueden salir del sistema a una frecuencia distinta a la que entran.

La definición de stream de video del modelo no incorpora la información de timing, debido a que los trabajos analizados solamente lo usan fuera del sistema de procesamiento de video. Además, debido a que en el sistema puede realizar operaciones complejas no determinísticas, no se puede garantizar que un pixel que ingresa al sistema sea procesado, y salga en un tiempo determinado del sistema. Sin embargo, el modelo permite que una implementación pueda mantener el timing si así lo desea, pero dicha implementación debe estar preparada para recibir y procesar entradas que no lo mantengan.

Regla 3: La salida del sistema siempre debe ser en formato stream. Sin timing definido.

Similarmente a lo argumentado en la regla 1, el sistema está compuesto por una jerarquía de filtros, y de ellos solamente puede salir video en forma de stream, entonces la única posibilidad para sacar video del sistema es a través de un stream. El stream de video de salida carece de información de timing, por lo que si se desea recrear, debe hacerse mediante drivers de salida específicos al formato en que se desee emitir la señal de video.

Regla 4: La llegada del último pixel de un cuadro a un filtro debe producir un vaciado del pipeline interno (flush), completando también el cuadro a su salida cuando finalice el procesamiento. Excepcionalmente, si un filtro usa varios cuadros de video de la entrada para producir una salida, debe vaciar su pipeline interno con el último pixel del cuadro de video necesario para producir una salida.

Vaciar el pipeline en el último pixel del cuadro de video crea una garantía del modelo hacia el usuario, pues asegura que si se introduce un cuadro completo al sistema, el resultado obtenido también está delimitado por un cuadro. Como característica adicional de esta delimitación, se genera un instante de tiempo entre cuadros de video en que se puede realizar procesamiento. Un ejemplo de este comportamiento se puede ver en [36], donde se realiza la estimación de movimiento cuando finaliza la recepción del cuadro y se terminan de calcular las proyecciones integrales, proceso que queda condicionado a que la entrada al sistema tenga pausas entre cuadros de video, lo que generalmente se cumple para la mayoría de los sistemas. Si no existiera

este instante de tiempo entre cuadros de video garantizado, y la pausa se produjera por ejemplo en la mitad del cuadro de video, el proceso de estimación de movimiento no tendría el tiempo suficiente para completarse antes de que inicie el próximo cuadro de video, produciéndose un error en el sistema. Este trabajo también realiza operaciones el finalizar cada línea de video, pero en una etapa de despliegue de resultados, por lo que no se consideran parte del sistema de procesamiento de video, correspondiendo a un driver de salida. Finalmente, un ejemplo de la excepción a la regla es un filtro que realiza decimación al nivel de cuadros de video, entregando un cuadro a la salida cada dos cuadros en la entrada.

Si bien la regla recién definida no entrega la mayor granularidad posible al modelo, alcanza para cubrir los trabajos analizados. Si llegase a ser necesaria una mayor granularidad, ya sea a nivel de líneas o pixeles, se podría llevar a cabo una segunda revisión del modelo, redactando nuevamente algunas reglas o agregando nuevas.

Regla 5: La transferencia de un pixel sólo se llevará a cabo si el filtro de destino está desocupado, produciendo una detención en caso de estar ocupado.

Cuando la frecuencia de los pixeles a la entrada de un filtro supera la frecuencia que el filtro puede procesar los datos, se produce un error, que dependiendo la naturaleza del sistema, como hard/soft real-time, puede provocar un fallo generalizado o simplemente un pequeño desperfecto. Para detectar estos escenarios, es necesario implementar algún tipo de control de flujo en el que el filtro de destino avise cuando su capacidad de procesamiento fue excedida, no aceptando más pixeles hasta que tenga disponibles nuevos recursos de cómputo. A esta situación se le conoce como detención, *stall* en inglés, e idealmente es no deseada, pero a veces es producida de forma natural por un diseño internamente para limitar el uso o acceso a algún recurso. Si el origen de una detención es una ráfaga, *burst* en inglés, de pixeles, y luego se vuelve a la frecuencia de operación normal, la detención puede ser absorbida colocando buffers de pixeles. Finalmente, aunque el modelo no exige a través de una regla conocer el punto exacto donde se produce la detención, es deseable que una implementación cuente con esta característica, al menos en las etapas de desarrollo, ya que podrían ayudar a encontrar los cuellos de botella del sistema y aumentar su velocidad de procesamiento o disminuir el consumo de recursos.

Regla 6: Un hardware-thread debe implementar un método de control de flujo para sus entradas y salidas. Para las entradas debe ser capaz de generar detenciones en el caso que no pueda seguir procesando datos. Para las salidas, debe ser capaz de detectar detenciones y no cambiar el estado de ninguna de sus salidas hasta que haya terminado la detención.

Esta regla es una generalización de la regla 5, pero para un hardware-thread, extendiendo el control de flujo y el concepto de detenciones a todos los protocolos de comunicación que se utilicen entre distintos hardware-threads. Esta regla representa un requerimiento importante del modelo, pues la mayoría de los trabajos analizados utilizaban un método de control de flujo solamente en operaciones consideradas no determinísticas. Sin embargo, para el modelo de cómputo la operación de un hardware-thread es considerada no determinística.

Se reconoce que el forzar un método de control de flujo en toda la jerarquía genera un aumento en el uso de recursos, y posiblemente una disminución en la capacidad de cómputo máxima del sistema, pero se consideró necesario para fomentar un diseño que sea claro, reutilizable, y menos propenso a errores.

2.9. Almacenamiento

Esta sección define un conjunto de reglas orientadas al almacenamiento dentro del modelo. Algunas son directamente derivadas de las definiciones anteriores, y otras de los trabajos analizados. A continuación de cada regla se presentan argumentos para su existencia e importancia en el modelo.

En todos los trabajos analizados que requieren realizar un procesamiento más allá de del pixel actual, se encontraron elementos de almacenamiento o memorias. Los tipos de almacenamiento encontrado fueron variados, desde simples registros de desplazamiento hasta complejas interfaces con memorias externas al sistema de procesamiento y específicas a la aplicación desarrollada. El uso dado al almacenamiento también es variado, pudiéndose clasificar entre uso para manipular video, y uso para almacenar datos totalmente dependientes de los algoritmos implementados en el sistema. El uso en video es principalmente para retrasar un stream de video durante unos pixeles o líneas, accediendo a los datos de forma secuencial, y para el almacenamiento de cuadros de video completos, de acceso aleatorio. El uso para datos personalizados es dominado principalmente por memorias de una dimensión y de acceso aleatorio, usadas en el almacenamiento de parámetros y resultados temporales de los algoritmos.

Físicamente, se encontraron dos tipos de memorias en los trabajos. Las primeras, son memorias internas al chip, capaces de almacenar sólo una pequeña cantidad de líneas de video, pero con la ventaja de operar en bajas latencias, usualmente en un ciclo de reloj, y de forma determinística. Las segundas, son memorias externas al chip, usualmente una memoria de acceso aleatorio (del inglés *random-access memory*, RAM), capaces de almacenar una gran cantidad

de cuadros de video, pero requiriendo una gran latencia para sus operaciones, por lo que su acceso suele realizarse en transacciones compuestas ráfagas de datos y varias señales de control. Además, debido a su tamaño generalmente son usadas por varios sistemas para intercambiar información dentro del diseño completo, por lo que su funcionamiento no es determinístico.

Regla 7: El almacenamiento orientado a video debe soportar las distintas características de un stream de video, siendo de vital importancia las dimensiones y el número de bits por cada una. En el caso del almacenamiento de un cuadro de video completo, también deben soportarse distintas resoluciones.

Como fue mencionado anteriormente, el uso del almacenamiento para video es común entre los trabajos analizados, ya sea en forma de líneas completas o varios cuadros de video. Debido a esto, es necesario que los elementos de almacenamiento soporten las características de un cuadro de video, y que puedan ser alojados dentro o fuera del chip a conveniencia del diseñador, como se verá más adelante.

Regla 8: Las memorias, internas o externas, deben contar con una interfaz de acceso unificada, que debe ser elegida cuidadosamente para minimizar la latencia de las memorias locales, y para soportar las operaciones complejas en memorias externas.

Si bien la mayoría de los elementos de almacenamiento encontrados en los trabajos analizados tienen interfaces de acceso diferentes, siendo las más simples las memorias internas, deben compartir la misma interfaz de acceso, con el objetivo de facilitar una fácil migración entre memorias internas y externas. Esto provoca una unificación de las interfaces de memorias externas, que en la mayoría de los casos simplifica su uso; y la adición de señales de control para las memorias internas, que incrementa levemente el uso de recursos, pero brinda la unificación deseada en el modelo. La interfaz de acceso debe ser cuidadosamente seleccionada, para que las memorias internas puedan mantener la latencia reducida que las caracteriza y no ralentizar la implementación de los algoritmos.

2.10. Lógica y aritmética

Esta sección define un conjunto de reglas orientadas a recursos lógicos y operaciones aritméticas del modelo. Algunas son directamente derivadas de las definiciones anteriores, y otras de los trabajos analizados. A continuación de cada regla se presentan argumentos para su existencia e importancia en el modelo.

Regla 9: Una implementación debe tener elementos básicos de lógica combinacional y sincrónica disponibles para el usuario, como multiplexores, registros, etc. Estos elementos permiten que el diseño pueda alcanzar una gran granularidad de ser necesario.

En todos los trabajos analizados, los elementos de lógica, tanto combinacional como sincrónica, forman parte de los bloques fundamentales con los que se construye el diseño. Es más, incluso si se creara un modelo de cómputo con un alto nivel de abstracción deberían existir al menos algunos elementos de lógica, como un multiplexor para analizar condiciones y elegir resultados.

Regla 10: Aritmética entera: Números enteros tienen un número de bits arbitrario, con o sin signo. Se deben implementar como mínimo operaciones de suma resta, multiplicación, división y comparación.

Todos los trabajos analizados realizan operaciones con aritmética entera en la implementación de los algoritmos, utilizando sólo la cantidad de bits necesarios para aumentar el rendimiento y ahorrar recursos. Para la representación de enteros con signo, la mayoría de las implementaciones usan complemento a dos, encontrando sólo un trabajo que realiza una implementación distinta, magnitud-signo. Las operaciones de suma, resta multiplicación y división son comunes en la transformación de los datos, mientras que las comparaciones son usadas para tomar decisiones sobre las operaciones siguientes a realizar.

Regla 11: Aritmética decimal: Números de punto fijo tienen un número de bits arbitrario, tanto para la parte decimal como para la fraccionaria, con o sin signo. Se deben implementar como mínimo operaciones de suma resta, multiplicación, división y comparación.

En los trabajos analizados, la aritmética decimal es dominada por implementaciones de punto fijo. Sin embargo, se encontraron varias implementaciones distintas, personalizadas ligeramente a la aplicación. Por ejemplo, [36] implementa aritmética de punto fijo, con el bit de signo asumiendo un valor de 0 para números negativos y 1 para positivos, en cambio, [35] implementa aritmética de punto fijo usando complemento a dos para la convención de signo. Este tipo de convenciones y decisiones de diseño pueden ser fácilmente unificadas al definirlas de forma estricta en la implementación del modelo.

Dentro de la literatura analizada también encontraron algunos trabajos que implementan aritmética en punto flotante, pero de una forma totalmente personalizada y sólo para las operaciones necesarias. Debido a que existen varios estándares de como operar en punto flotante;

son raramente usados en el diseño de sistemas de procesamiento de video; y sus implementaciones utilizan muchos recursos lógicos, se decidió no incluirlos en el modelo. Sin embargo, una implementación es libre de agregarlos como elementos personalizados de cómputo, en el caso específico de que se necesitara realizar operaciones con un gran rango dinámico.

2.11. Uso compartido de recursos

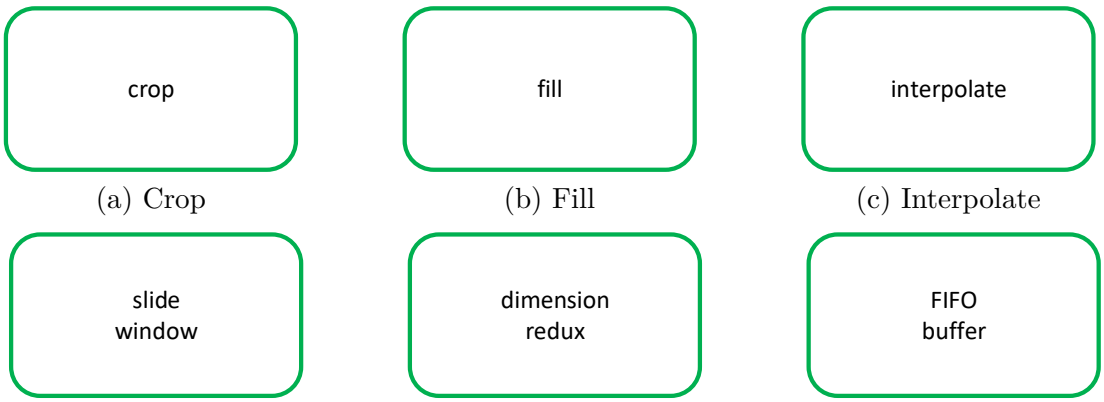
Esta sección define un conjunto de reglas orientadas al uso compartido de recursos del modelo. Algunas son directamente derivadas de las definiciones anteriores, y otras de los trabajos analizados. A continuación de cada regla se presentan argumentos para su existencia e importancia en el modelo.

Regla 12: Opcionalmente, se puede implementar el uso compartido de recursos, como acceso múltiple por división de tiempo (del inglés *Time Division Multiple Access*, TDMA), cuando sea necesario.

Algunos trabajos, en la presencia de un recurso considerado valioso por su alto costo de implementación o escasez, utilizan alguna forma de acceso compartido, especialmente cuando los accesos al recurso no son simultáneos. En un caso extremo, esta característica permite compartir el acceso a un único recurso disponible en todo el sistema. Debido a que sólo algunos trabajos implementaban esta forma de acceso, dependerá de la implementación entregar esta característica, quedando como opcional en el modelo. A modo de ejemplo, en [36] se utiliza TDMA para compartir el acceso a un puerto de memoria, y también para compartir unidades que realizan aritmética de punto fijo, como sumadores y multiplicadores.

Regla 13: Si se implementa el acceso compartido de recursos, los buses y señales de la implementación deben ser multiplexables, es decir, deben soportar su conexión a arbitradores de uso de recursos.

El acceso a un recurso compartido se realiza generalmente con un arbitrador, que conoce el estado actual del recurso y otorga un acceso temporal a la parte interesada cuando estima conveniente, pudiendo denegar el acceso si el recurso se encuentra actualmente ocupado. Debido a esto, los buses usados para acceder al recurso compartido deben ser multiplexables, y soportar esperas hasta que el acceso al recurso sea otorgado.



(a) Crop (b) Fill (c) Interpolate

(d) Slide window (e) Dimension redux (f) FIFO buffer

Fig. 2.6: Primitivas: Filtros

2.12. Primitivas

Esta sección define un conjunto elementos del modelo que las implementaciones deben tener disponibles para el usuario. Algunos de los elementos descritos a continuación se derivan de las definiciones del modelo, y el resto representan elementos comunes encontrados en los trabajos analizados, aplicando algunas modificaciones para hacerlos más genéricos y reutilizables.

Es importante destacar que las primitivas descritas son obligatorias para todas las implementaciones. Además, con el objetivo de fomentar la reutilización, el modelo define una cantidad mínima pero necesaria de parámetros para las primitivas. Idealmente, las implementaciones añadirán sus propios parámetros que sirvan para incrementar la reutilización.

2.12.1. Filtros

Los filtros realizan operaciones directo sobre un stream de video, transformándolo, y obteniendo el stream procesado a su salida. El modelo define sólo filtros básicos, siendo la implementación libre de definir los que sean necesarios para el tipo de aplicación deseada.

A continuación se listan y detallan los filtros comunes encontrados en los trabajos y definidos en el modelo:

1. **crop**: Símbolo de referencia en la Figura 2.6a. Recorta un área del stream de entrada, y la entrega a su salida como otro stream de tamaño más pequeño. Dentro de sus parámetros deben estar el tamaño del área a recortar y su ubicación dentro del cuadro de video. Importante: Este filtro no debe modificar el valor que tienen los pixeles a conservar.

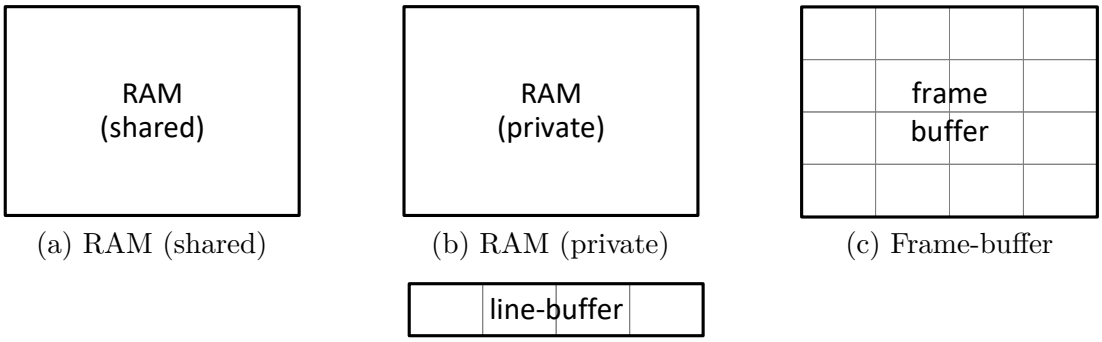
2. **fill**: Símbolo de referencia en la Figura 2.6b. Rellena el stream de entrada, aumentando su tamaño y produciendo un nuevo stream a la salida, donde se copiarán los datos de la entrada en una ubicación indicada por el usuario. Dentro de sus parámetros deben estar el tamaño del stream de salida, la ubicación del stream de entrada dentro del de salida, y el valor con el que rellenar cada nuevo pixel. Importante: Este filtro no debe modificar el valor que tienen los pixeles del stream de entrada.

3. **interpolate**: Símbolo de referencia en la Figura 2.6c. Realiza una interpolación al stream de entrada, pudiendo achicar o agrandar el stream de salida. Para ello, puede modificar el valor de los pixeles del stream de entrada mediante un algoritmo de interpolación. Dentro de sus parámetros deben estar el tamaño del stream de salida, y varias elecciones de algoritmos para realizar la interpolación.

4. **slide window**: Símbolo de referencia en la Figura 2.6d. Crea una ventana deslizante del stream de entrada, obteniendo un stream de salida con una dimensiones adicional por cada pixel de la vecindad del pixel actual. Dentro de sus parámetros deben estar el tamaño de la ventana, y varias elecciones para definir el comportamiento de la ventana en el borde del cuadro de video (rellenar con un ceros, duplicar último valor, reflejar cuadro, etc.). Importante: Como este filtro crea dimensiones adicionales en el stream de salida, el de entrada debe contener exactamente una dimensión para evitar confusiones a los usuarios.

5. **dimension redux**: Símbolo de referencia en la Figura 2.6e. Concatena todas las dimensiones del stream de entrada, entregando un stream de salida con sólo una dimensión. No posee parámetros requeridos por el modelo. Importante: Este filtro no debe modificar el valor que tienen los pixeles del stream de entrada, sólo concatena sus bits.

6. **FIFO buffer**: Símbolo de referencia en la Figura 2.6f. Permite retrasar y absorber ráfagas de pixeles en su entrada que no pueden ser procesadas inmediatamente por el filtro conectado a su salida. Internamente incorpora alguna forma de almacenamiento para guardar los pixeles que vienen llegando, sacando de forma ordenada los pixeles solamente si han superado el tamaño mínimo del buffer y el stream de salida los puede procesar. Dentro de sus parámetros deben estar el tamaño total del buffer, y el tamaño mínimo. Cuando el tamaño mínimo es 0, el buffer es ideal para absorber ráfagas de pixeles a la entrada. En cambio, con un tamaño mínimo mayor a 0, se pueden implementar, por ejemplo, retardos completos de una línea para crear la vecindad de un pixel en una ventana deslizante. Importante: Este filtro sólo puede retrasar los pixeles del stream de entrada, teniendo prohibido alterar su valor o reordenarlos.



(a) RAM (shared) (b) RAM (private) (c) Frame-buffer

(d) Line-buffer

Fig. 2.7: Primitivas: Almacenamiento

Durante el análisis de los trabajos se encontraron varios filtros adicionales que no fueron definidos en esta sección. La decisión de no incluirlos se debió a que se diseñó el modelo para ser minimalista y que permitiera construir implementaciones en poco tiempo que cumplen totalmente con el modelo. Dependiendo de su orientación, las implementaciones proveerán filtros más avanzados, como corrección de gama, operadores de sobel, umbralización, limitadores, etc.

2.12.2. Almacenamiento

A continuación se listan y detallan los elementos de almacenamiento comunes encontrados en los trabajos y definidos en el modelo:

1. **RAM (shared)**: Símbolo de referencia en la Figura 2.7a. Almacenamiento genérico de acceso aleatorio compartido (global). Siempre reside fuera del chip, con un espacio de direcciones compartido por el diseño completo. Debe tener al menos una interfaz de acceso aleatorio a memoria para operaciones de lectura y escritura. Dentro de sus parámetros deben estar el número de bits de la dirección, y de las palabras.

2. **RAM (private) on-chip**: Símbolo de referencia en la Figura 2.7b. Almacenamiento genérico de acceso aleatorio privado. Siempre reside dentro del chip, con un espacio de direcciones privado. Debe tener al menos una interfaz de acceso aleatorio a memoria para operaciones de lectura y escritura. Dentro de sus parámetros deben estar el número de bits de la dirección, y de las palabras. Importante: Sus interfaces de acceso deben ser completamente idénticas a **RAM (private) off-chip**, con el objetivo de poder intercambiarlas fácilmente.

3. **RAM (private) off-chip**: Símbolo de referencia en la Figura 2.7b. Almacenamiento genérico de acceso aleatorio privado. Siempre reside fuera del chip, con un espacio de di-

recciones privado. Debe tener al menos una interfaz de acceso aleatorio a memoria para operaciones de lectura y escritura. Dentro de sus parámetros deben estar el número de bits de la dirección, y de las palabras. Importante: Sus interfaces de acceso deben ser completamente idénticas a **RAM (private) on-chip**, con el objetivo de poder intercambiarlas fácilmente.

4. **frame-buffer on-chip**: Símbolo de referencia en la Figura 2.7c. Almacena un cuadro de video, con todas sus dimensiones. Reside dentro del chip. Debe tener al menos una interfaz de acceso aleatorio para operaciones de lectura y escritura de un pixel, utilizando sus coordenadas como direccionamiento. Dentro de sus parámetros deben estar el tamaño, número de dimensiones, y bits de cada dimensión del cuadro de video. Importante: Sus interfaces de acceso deben ser completamente idénticas a **frame-buffer off-chip**, con el objetivo de poder intercambiarlas fácilmente.

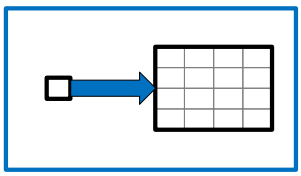
5. **frame-buffer off-chip**: Símbolo de referencia en la Figura 2.7c. Almacena un cuadro de video, con todas sus dimensiones. Reside fuera del chip. Debe tener al menos una interfaz de acceso aleatorio para operaciones de lectura y escritura de un pixel, utilizando sus coordenadas como direccionamiento. Dentro de sus parámetros deben estar el tamaño, número de dimensiones, y bits de cada dimensión del cuadro de video. Importante: Sus interfaces de acceso deben ser completamente idénticas a **frame-buffer on-chip**, con el objetivo de poder intercambiarlas fácilmente.

6. **line-buffer**: Símbolo de referencia en la Figura 2.7d. Almacena una línea de video, con todas sus dimensiones. Reside dentro del chip. Debe tener al menos una interfaz de acceso aleatorio para operaciones de lectura y escritura de un pixel, utilizando su coordenada como direccionamiento. Dentro de sus parámetros deben estar el tamaño, número de dimensiones, y bits de cada dimensión de la línea de video.

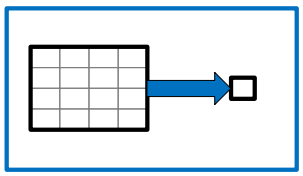
Los elementos de almacenamiento recién definidos permiten tener una gran versatilidad, tanto en características como en ubicación física. Debido a esto, se pueden utilizar para realizar cosas con que no fueron objetivos del modelo en el momento de su diseño, como por ejemplo almacenar varios cuadros de video en el tiempo, cada uno en una dimensión de un frame-buffer.

2.12.3. Hardware-threads

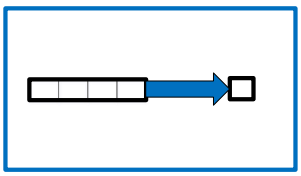
A continuación se listan y detallan los hardware-threads comunes encontrados en los trabajos y definidos en el modelo:



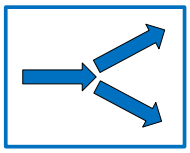
(a) Stream to frame-buffer



(b) Frame-buffer to stream



(c) Line-buffer to stream



(d) Stream splitter

Fig. 2.8: Primitivas: Hardware-threads

1. **stream to frame-buffer**: Símbolo de referencia en la Figura 2.8a. Almacena un cuadro de video proveniente de un stream, en un **frame-buffer on-chip/off-chip** (no incorporado). Como entrada, tiene un stream de video; y como salidas, una interfaz de escritura al **frame-buffer** de destino, y una señal para la notificación de que se ha escrito un cuadro de video completo en el **frame-buffer**. Dentro de sus parámetros deben estar el tamaño, número de dimensiones, y bits de cada dimensión del cuadro de video.

2. **frame-buffer to stream**: Símbolo de referencia en la Figura 2.8b. Crea un stream de video a partir de un **frame-buffer on-chip/off-chip** (no incorporado). Como entradas, tiene una interfaz de lectura al **frame-buffer** de origen, y una señal para controlar el inicio de la lectura de un cuadro de video completo desde el **frame-buffer**; y como salida un stream de video. Dentro de sus parámetros deben estar el tamaño, número de dimensiones, y bits de cada dimensión del stream de video.

3. **line-buffer to stream**: Símbolo de referencia en la Figura 2.8c. Crea un stream de video a partir de un **line-buffer** (no incorporado). Como entradas, tiene una interfaz de lectura al **line-buffer** de origen, y una señal para controlar el inicio de la lectura de una línea de video completa desde el **line-buffer**; y como salida un stream de video. Dentro de sus parámetros deben estar el tamaño, número de dimensiones, y bits de cada dimensión del stream de video.

4. **stream splitter**: Símbolo de referencia en la Figura 2.8d. Crea múltiples copias de un stream de video. Produce una detención en la entrada si alguna de las salidas está ocupada y no puede recibir el pixel actual, forzando una sincronización de todos los streams de salida.

Durante el análisis de los trabajos se encontraron varias otras estructuras comunes, pero su implementación era tan específica que no se encontró una forma estandarizada para ellas. Sin embargo, tales estructuras se pueden implementar con las primitivas descritas anteriormente. Una de las estructuras encontradas fue una pirámide Gaussiana, que consiste en tomar una imagen y crear varios niveles, cada uno conteniendo versiones más pequeñas de la imagen, lo que permite procesar la imagen más chica e ir refinando el resultado a medida que se cambia a los niveles más grandes (detallados) de la pirámide. Esta estructura tiene gran variabilidad entre implementaciones, pues algunas reducen la imagen a la mitad en cada nivel, y otras en un 20 % para más exactitud. Además, los algoritmos usados para interpolar los niveles varían mucho entre las implementaciones. Otra estructura encontrada fue un histograma. Si bien el cálculo tenía pocas variaciones en su implementación, llegando generalmente los datos en forma de stream, el almacenamiento dependía mucho de las necesidades de la aplicación, y también el acceso a sus datos, pues en algunas el acceso era en forma aleatoria, y en otras en forma de stream.

2.12.4. Drivers I/O

Los drivers de entrada y salida permiten adaptar el sistema de procesamiento de video definido en el modelo, para que pueda interactuar con otros sistemas inicialmente no compatibles, e integrarse como parte de un diseño más grande. Además, algunos pueden conectarse directamente con equipos, como computadores personales, que permiten acelerar el proceso de diseño y validación.

Una implementación debe tener por lo menos un driver de entrada y uno de salida para interactuar con el medio. La siguiente lista menciona algunos métodos de entrada/salida utilizados por los trabajos analizados:

- **serial port**: Puerto serie, con trasferencias pixel a pixel.
- **Ethernet**: Frame Ethernet, con transferencias que pueden alcanzar hasta una línea de video por frame.
- **HDMI**: En formato de stream de video, usualmente para visualización y aplicaciones finales.
- **memory map**: Transferencias directas de memoria global para que el sistema de procesamiento pueda ser usado como un acelerador de cómputo.

- **general purpose bus**: Transferencias directas de un bus de propósito general para que el sistema de procesamiento pueda ser usado como un acelerador de cómputo.

2.13. Limitaciones del modelo

A continuación se presenta una lista con las limitaciones, identificadas hasta el momento, del modelo diseñado:

- Resolución definida al momento de compilación/síntesis: Desde un principio se caracterizó un stream de video a través de características invariantes en el tiempo, lo que permitió realizar simplificaciones en el modelo. Sin embargo, una consecuencia de esto es que no se pueden realizar cambios en tiempo de funcionamiento a la resolución de entrada. Esta puede ser una limitante para algunos sistemas de procesamiento, pero como todos los trabajos analizados estaban orientados a aplicaciones específicas, no implementaban el soporte de múltiples resoluciones, o el cambio dinámico en tiempo de funcionamiento.
- No se pueden compartir elementos (como almacenamiento) entre los filtros: El modelo impone restricciones sobre un filtro para que sea auto-contenido y que pueda ser intercambiado fácilmente por otro de simular funcionalidad. Sin embargo, esto provoca que no se puedan compartir elementos de procesamiento entre los filtros, obligando a realizar una duplicación de recursos si se necesita la misma funcionalidad en dos filtros distinto. Un caso extremo que el modelo no puede representar, es un sistema en el que hay un recurso escaso, único, que es necesario por dos o más filtros.
- Acceso a sólo un espacio de direcciones para memoria compartida global: El tipo de almacenamiento definido en el modelo no incorpora múltiples espacios de direcciones, pues ningún trabajo analizado lo necesitaba. Sin embargo, se han encontrado dispositivos, como FPGAs, que incorporan varios controladores de memoria internos, y por consecuencia, tienen varios espacios de direcciones asociados, por lo que se cree que sería una buena adición en una futura revisión del modelo.

Es importante destacar que algunas de las limitaciones presentadas tienen directa relación con las reglas estrictas impuestas por el modelo, por ser la primera versión estable. Sin embargo, en una futura revisión, cuando el modelo haya sido validado con la implementación de varios trabajos, algunas reglas podrían relajarse y dar paso a un modelo con menos limitaciones y más características dinámicas.

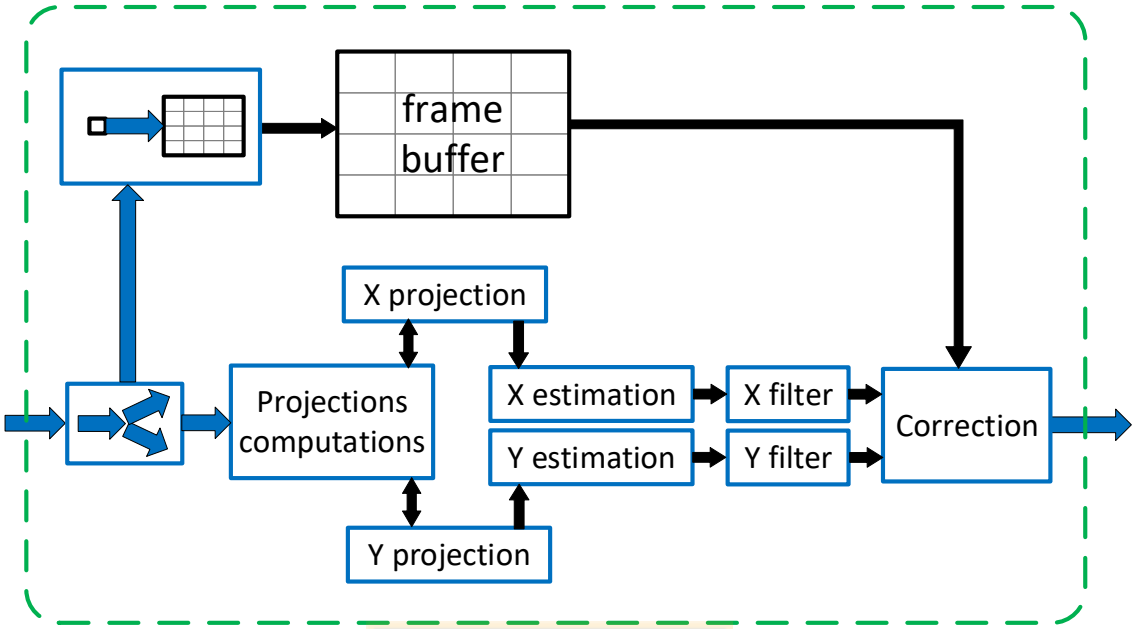


Fig. 2.9: Estabilización digital de video

2.14. Ejemplos

A continuación se describen ejemplos de uso del modelo para describir algunas aplicaciones.

2.14.1. Estabilización digital de video

La Figura 2.9 representa el esquemático de la aplicación descrita con el modelo de cómputo. La aplicación completa es un filtro, compuesto por una jerarquía de hardware-threads. Un pixel es duplicado cuando llega el sistema. La primera copia se almacena en un frame-buffer para su posterior corrección. La segunda copia es utilizada en el cálculo de las proyecciones integrales, que son almacenadas de forma separada para cada eje, X e Y. Como la estimación de movimiento necesita las versiones anterior y actual de las proyecciones, son almacenadas en un doble buffer construido con dos line-buffers. El doble buffer cuenta con dos direcciones de acceso, una para el buffer anterior y otra para el actual. Debido a que se requieren tres accesos, uno para cálculo de proyecciones y dos para estimación de movimiento, y que los accesos no se realizan de forma simultanea (calculo de proyecciones y luego estimación), se puede colocar un arbitrador (TDMA) para controlar el acceso al doble buffer, ya que sin el debería recurrirse a otras estrategias para obtener múltiples accesos. Cuando se finaliza el cálculo de las proyecciones, se notifica a los hardware-threads de estimación de movimiento. Estos realizan una búsqueda exhaustiva

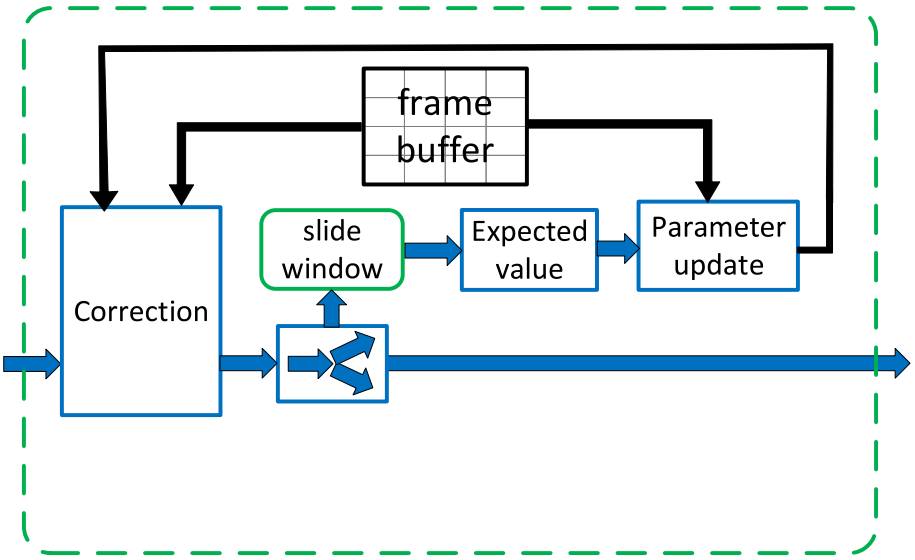


Fig. 2.10: Estabilización térmica de video infrarrojo

probando todos los desplazamientos posibles, y eligiendo el que mejor se ajuste a la proyección anterior. Al terminar la estimación de movimiento, los desplazamientos encontrados son filtrados, usando aritmética de punto fijo, para eliminar componentes voluntarios. Finalmente, el cuadro de video guardado en el frame-buffer es corregido, eliminando los desplazamientos involuntarios y generando un nuevo stream de video.

2.14.2. Estabilización térmica de video infrarrojo

La Figura 2.10 representa el esquemático de la aplicación descrita con el modelo de cómputo. La aplicación completa es un filtro, compuesto por una jerarquía de hardware-threads. Un pixel es corregido inmediatamente cuando entra al sistema usando la información de calibración por pixel contenida en un frame buffer de sólo lectura, y parámetros de corrección. El pixel ya corregido es duplicado, pasando el primero directamente a la salida. Con el segundo pixel se crea una ventana deslizante para obtener su vecindad, con la cual se calcula un valor esperado. Este valor es usado en conjunto con la información de calibración para calcular nuevos parámetros de corrección, que son pasados nuevamente al hardware-thread de corrección para ser utilizados en la siguiente corrección.

Capítulo 3: Lenguaje de programación

Este capítulo presenta un lenguaje de programación derivado del modelo de cómputo definido en el capítulo anterior. Es un DSL que combina características de C y Verilog, resultando en buenas abstracciones y una granularidad a nivel de bits. El flujo de ejecución es basado en eventos, lo que permite fácilmente crear u obtener paralelismo a partir del código.

3.1. Generalidades

La sintaxis formal del lenguaje es descrita usando la notación Backus–Naur Form (BNF), que es también usada en la definición de Verilog. Las siguientes convenciones serán usadas:

- Palabras en minúsculas, que pueden contener guiones bajos, son usadas para denotar categorías sintácticas. Por ejemplo:
  filter_declaration.
- Palabras en negrita denotan palabras clave reservadas, operadores, y signos de puntuación como partes necesarias de la sintaxis. Por ejemplo:
  filter => ;
- Una barra vertical separa elementos alternativos a no ser que aparezca en negrita, en cuyo caso se representa a si misma. Por ejemplo:
  test_operators ::= + | - | & | | | *
- Paréntesis cuadrados encierran elementos opcionales. Por ejemplo:
  test_declaration ::= [test_attr] test_var;
- Paréntesis de llaves encierran elementos repetidos a no ser que aparezcan en negrita, en cuyo caso se representa a si mismos. El elemento puede aparecer cero o más veces, y las repeticiones pueden ocurrir de izquierda a derecha como una regla recursiva por la izquierda. Por ejemplo: Las dos siguientes reglas son equivalentes:
  list_of_test_objets ::= test_object { , test_object }
  list_of_test_objets ::=
  test_object
  | list_of_test_objets , test_object

Durante la descripción del lenguaje, se mostrarán varios ejemplos cortos con el propósito de aclarar su uso.

3.2. Convenciones léxicas

Los archivos de texto de código fuente serán un stream de componentes léxicos, donde cada uno de ellos estará constituido por uno o más caracteres. La disposición de los componentes en un archivo de código fuente será en formato libre; es decir, espacios y saltos de línea no tendrán significado sintáctico, aparte de ser separadores de componentes, excepto en identificadores escapados.

Los tipos de componentes léxicos en el lenguaje son: Espacio en blanco, comentario, identificador, palabra clave, número, y operador.

3.2.1. Espacio en blanco

El espacio en blanco contendrá caracteres de espacios, tabulaciones, saltos de línea, y saltos de página. Estos caracteres serán ignorados, excepto cuando sirven para separar otros componentes léxicos.

3.2.2. Comentario

```
comment ::=
          one_line_comment
        | block_comment
one_line_comment ::= // comment_text \n
block_comment ::= /* comment_text */
comment_text ::= { Any_ASCII_character }
```

El lenguaje tendrá dos formas de introducir comentarios. Comentarios de una línea empezarán con los dos caracteres “//” y finalizar con un salto de línea. Un comentario de bloque deberá comenzar con “/*” y terminar con “*/”, y no podrán ser anidados.

3.2.3. Identificadores y palabras claves

single_identifier ::= **a-zA-Z0-9_$** { [**a-zA-Z0-9_$**] }

Un identificador será usado para dar a un objeto un nombre único por el cual puede ser referenciado. Estará compuesto por una secuencia de letras, dígitos, y guiones bajos. El primer carácter no puede ser un dígito. Ejemplos de identificadores bien definidos se muestra a continuación:

```
my_stream
index_num
cascade
_mosfet
```

La longitud máxima de un identificador puede estar definida por la implementación, pero debe ser como mínimo 1024 caracteres.

Las palabras claves son identificadores predefinidos que son usados para definir la estructura del lenguaje. Todas las palabras claves son definidas en minúsculas. Una lista con ellas se presenta a continuación:

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

3.2.4. Definiciones previas

A continuación se presentan varias definiciones que serán utilizadas en diversas partes del lenguaje.

Las agrupaciones de dígitos que formarán las tres bases soportadas por el lenguaje, binario, decimal, y hexadecimal:

unsigned_number ::= decimal_digit { _ | decimal_digit }
binary_value ::= binary_digit { _ | binary_digit }
hex_value ::= hex_digit { _ | hex_digit }
decimal_digit ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
binary_digit ::= **0** | **1**
hex_digit ::=
0 | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
| **a** | **b** | **c** | **d** | **e** | **f** | **A** | **B** | **C** | **D** | **E** | **F**

Independiente de la base numérica de la agrupación de dígitos, estos podrán estar separados por un guion bajo para mejorar su legibilidad.

Agrupaciones de dígitos decimales utilizadas en un número que debe ser mayor a 0:

non_zero_unsigned_number ::= non_zero_decimal_digit { _ | decimal_digit }
non_zero_decimal_digit ::= **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

3.2.5. Constantes numéricas

Una constante entera puede ser especificada en base decimal, binaria, y hexadecimal. En cambio, una constante fraccionaria sólo podrá ser especificada en base decimal. Todas las constantes que tengan signo serán representadas en complemento a dos.

Una constante, entera o fraccionaria, se especificará inicialmente con el operador "+" para números positivos, o con "**-**" para números negativos. Si ningún operador estuviera presente, la constante será sin signo (positiva). Luego se indicará la base, siendo "**d**" para decimal, "**b**" para binario, o "**h**" para hexadecimal. Si la base no estuviera presente, la constante tendrá base decimal. A continuación sigue una secuencia de dígitos, dependientes de la base elegida, que representarán la parte entera de la constante. Opcionalmente, y sólo para base decimal, se definirá la parte fraccionaría de la constante. Para ello, primero se antepondrá un "**.**" seguido de una secuencia de dígitos del 0 al 9.

El número de bits necesario para una constante se calculará automáticamente para las constantes de tipo entero. Para las constantes fraccionarias, se deberá realizar un *cast* explícitamente antes de asignarlas. Cuando se asigne una constante a un tipo de dato que contiene más bits de los que posee la constante, esta será extendida con el signo para la constantes con signo, y con 0 para las constantes sin signo. En ningún caso se podrá asignar una constante a un tipo de dato que contenga menos bits de los que posee la constante.

Formalmente, la sintaxis de una constante es:

```
number_constant ::=
        decimal_number
      | binary_number
      | hex_number
      | frac_number
frac_number ::=
        [ sign ] decimal_base unsigned_number . unsigned_number
decimal_number ::=
        [ sign ] unsigned_number
      | [ sign ] decimal_base unsigned_number
sign ::= + | -
binary_number ::=
        binary_base binary_value
hex_number ::=
        hex_base hex_value
decimal_base ::= d | D
binary_base ::= b | B
hex_base ::= h | H
```

Algunos ejemplos de constantes:

```
b101      // Binary: Positive (no sign)
d123      // Decimal: Positive (no sign)
123       // Decimal: Positive (no sign, no base)
+d123     // Decimal: Positive
-d123     // Decimal: Negative
+d123.0   // Fractional: Positive
-d45.99   // Fractional: Negative
```

3.3. Tipos de datos

A continuación se definirán los distintos tipos de datos del lenguaje. Es importante mencionar que lo habrá conversión implícita de tipos, teniendo que aplicar un *cast* para realizar operaciones entre tipos de datos distintos.

3.3.1. Básicos

event

event_declaration ::= event_type list_of_events_identifiers **;**
event_type ::= event_keyword
list_of_events_identifiers ::= event_identifier {, event_identifier}
event_identifier ::= single_identifier
event_keyword ::= **event**

Un **event** permite controlar el flujo de ejecución. Se propaga de forma *multicast*, es decir, tiene un emisor y múltiples receptores. El emisor inicia la propagación al llamar a la función **set_event()**, definida como:

set_event_call ::= set_event_keyword **(** event_identifier **) ;**
set_event_keyword ::= **set_event**

Esta función recibe un argumento de tipo **event**, y su llamado bloqueará la ejecución hasta que todos los receptores hayan recibido satisfactoriamente el evento.

Los receptores recibirán el evento en una sentencia **on_event()**, definida más adelante. Esta se ejecutará cada vez que se reciba el evento, y sólo podrá aceptar un nuevo evento si es que ya ha finalizado su ejecución.

raw_bits

```
raw_bits_declaration ::= raw_bits_type raw_bits_ident_init_list ;
raw_bits_type ::= raw_bits_keyword ( total_bits )
raw_bits_ident_init_list ::= raw_bits_ident_init { , raw_bits_ident_init }
raw_bits_ident_init ::= raw_bits_identifier [ = raw_bits_initializer ]
total_bits ::= non_zero_unsigned_number
raw_bits_initializer ::=
          raw_bits_identifier
        | raw_bits_slice
        | number_constant
raw_bits_identifier ::= single_identifier
raw_bits_keyword ::= raw_bits
raw_bits_slice ::= raw_bits_identifier [ constant_range_expression ]
constant_range_expression ::=
          unsigned_number
        | msb_number : lsb_number
msb_number ::= unsigned_number
msb_number ::= unsigned_number
```

Es un conjunto de bits de `total_bits` elementos numerados desde 0 a (`total_bits`-1). Es usado en la representación de datos que carecen de una interpretación numérica inicial, como un stream de pixeles, o los datos de una memoria. Solamente será posible usar operadores de bits, y las operaciones aritméticas requerirán un *cast*. Se puede acceder a un bit o subconjunto de bits usando paréntesis cuadrados, el que seguirá siendo de tipo **raw_bits**. Para acceder a exactamente 1 bit, se hará directamente con su número. En cambio, para seleccionar varios bits, se escribirá primero el número del bit más significativo, el carácter "**:**" como separador, y el número del menos significativo.

```
raw_bits(10) my_var;
my_var[0];    // First bit
my_var[4:0]; // bits 4 through 0 (5 bits total)
```

number

number_declaration ::= number_type number_ident_init_list **;**
number_type ::= number_keyword **(** int_bits **,** frac_bits **)**
number_ident_init_list ::= number_ident_init { **,** number_ident_init }
number_ident_init ::= number_identifier [**=** number_initializer]
int_bits ::= non_zero_unsigned_number
frac_bits ::= unsigned_number
number_initializer ::= number_identifier | number_constant
number_identifier ::= single_identifier
number_keyword ::= **number**

Es la representación generalizada de un número con signo, ya sea entero o fraccional (punto fijo). En su declaración se deberán especificar el número de bits de la parte entera, `int_bits`, que incluirá el signo, por lo que siempre será mayor o igual a 1; y el número de bits de la parte fraccionaria, `frac_bits`, que podrá ser 0 para números puramente enteros. El rango de números que se podrán representar en la parte entera es de $[-2^{\texttt{int\_bits}-1}, +2^{\texttt{int\_bits}-1} - 1]$. En cambio, la parte fraccionaria siempre tendrá un rango de $[0, 1[$ con una precisión de $1/2^{\texttt{frac\_bits}}$, con lo que se obtiene un rango total de $]-(2^{\texttt{int\_bits}-1}+1), +2^{\texttt{int\_bits}-1}[$. Finalmente, se podrán realizar operaciones aritméticas de forma directa entre distintas variables de tipo **number**, calculándose automáticamente la cantidad de bits del resultado, incluso si las operaciones se realizan entre variables con distinta cantidad de bits cada una. A continuación se muestran algunos ejemplos de declaración e inicialización:

```
number(3,0) var1;          // 3-bits integer
number(3,4) var2;          // 3-bits integer, 4-bits fractional
number(8,0) var3 = +45;    // 8-bits integer with initializer (pos)
number(8,0) var4 = -45;    // 8-bits integer with initializer (neg)
number(8,0) var5 = var3;   // 8-bits integer with initializer (var)
```

unumber

```
unumber_declaration ::= unumber_type unumber_ident_init_list ;
unumber_type ::= unumber_keyword ( int_bits , frac_bits )
unumber_ident_init_list ::= unumber_ident_init { , unumber_ident_init }
unumber_ident_init ::= unumber_identifier [ = unumber_initializer ]
int_bits ::= non_zero_unsigned_number
frac_bits ::= unsigned_number
unumber_initializer ::= unumber_identifier | number_constant
unumber_identifier ::= single_identifier
unumber_keyword ::= unumber
```

Es la representación generalizada de un número sin signo, ya sea entero o fraccional (punto fijo). Se declarará y usará de la misma forma que **number**, pero explícitamente para números sin signo. Como no se ocupará un bit para el signo, el rango de la parte entera es $[0, +2^{\texttt{int\_bits}} - 1]$, con lo que se obtendrá un rango total de $[0, 2^{\texttt{int\_bits}} + 1[$. A continuación se muestran algunos ejemplos de declaración e inicialización:

```
unumber(5,0) var1;          // 5-bits integer
unumber(5,4) var2;          // 5-bits integer, 4-bits fractional
unumber(8,0) var3 = 90;     // 8-bits integer with initializer
unumber(8,0) var4 = var3;   // 8-bits integer with initializer
```

parameter

```
parameter_declaration ::= parameter_type parameter_ident_init_list ;
parameter_type ::= parameter_keyword
parameter_ident_init_list ::= parameter_ident_init { , prameter_ident_init }
parameter_ident_init ::= parameter_identifier = parameter_initializer
parameter_initializer ::= parameter_identifier | number_constant
parameter_identifier ::= single_identifier
parameter_keyword ::= parameter
```

Permitirá la creación de elementos paramétricos en el lenguaje, ya que para efectos prácticos, serán una constante con nombre. Debido a esto, podrá albergar cualquier expresión que sea constante en tiempo de compilación, y el número de bits necesarios se calculará automáticamente. La inicialización será siempre requerida, y cuando se haga en una lista de argumentos, servirá

para dar un valor por defecto al **parameter**. A continuación se muestran algunos ejemplos:

```
parameter par1 = b11011; // 5-bits integer
parameter par2 = 31;     // 5-bits integer
parameter par3 = -90;    // 8-bits integer
parameter par4 = par3;   // 8-bits integer
```

Arreglos (vectores)

```
array_declaration ::= array_type array_identifier [ array_size ] ;
array_type ::=
        raw_bits_type
      | number_type
      | unumber_type
      | memory_port_type
array_size ::= non_zero_unsigned_number
array_identifier ::= single_identifier
```

Un arreglo agrupará uno o más elementos de un mismo tipo bajo un único identificador. El arreglo tendrá `array_size` elementos, numerados desde 0 a (`array_size`-1). La versión actual del lenguaje no incorpora una forma de inicializar a un valor los elementos del arreglo, por lo que deberán ser inicializados manualmente, uno a uno. Cada elemento se accederá con un índice, `array_index`, usando paréntesis cuadrados, de la forma:

```
array_element ::= array_identifier [ array_index ] ;
array_type ::=
        raw_bits_type
      | number_type
      | unumber_type
array_index ::=
        unsigned_number
      | unumber_identifier
```

A continuación se muestran algunos ejemplos:

```
raw_bits(6) my_array[5]; // Array of 5 elements of type raw_bits
my_array[0] = b101010;   // Assignation of the first element
my_array[1];             // Second element
```

```
my_array[2][4:0];          // bits 4 through 0 of the third element
```

Es importante destacar que si bien se pueden declarar memorias usando arreglos, es mejor usar tipos especializados de almacenamiento, por las ventajas que tienen al ser intercambiables y estar mejor optimizados para una mayor cantidad de elementos.

3.3.2. Video

pix_stream

```
pix_stream_declaration ::= pix_stream_type pix_stream_ident_init_list ;
pix_stream_type ::= pix_stream_keyword
pix_stream_ident_init_list ::= pix_stream_ident_init { , pix_stream_ident_init }
pix_stream_ident_init ::= pix_stream_identifier [= pix_stream_initializer]
pix_stream_initializer ::= pix_stream_identifier
pix_stream_identifier ::= single_identifier
pix_stream_keyword ::= pix_stream
```

Tipo de dato que se usará para representar un stream de video. Es una estructura con miembros de varios tipos, que se accederán usando el operador "**.**". Los miembros de la estructura se listan a continuación:

- `res_h`: Resolución horizontal, en pixeles. De tipo **parameter**, modificable una vez.
- `res_v`: Resolución vertical, en pixeles. De tipo **parameter**, modificable una vez.
- `depth`: Arreglo que contiene el tamaño en bits de cada una de las dimensiones del stream. De tipo **parameter**, modificable una vez.
- `data`: Arreglo de **raw_bits** que contiene los datos del pixel actual. La cantidad de elementos del arreglo depende del número de dimensiones que tenga el stream, `dim`.
- `dim`: Número de dimensiones del stream. Será calculado automáticamente a partir de la definición de `depth`. De tipo **parameter**, sólo lectura.
- `new_data`: Evento que se generará automáticamente para un stream de entrada cuando llegue un pixel, por lo que puede ser capturado con la función **on_event()**. Para un stream de salida, es decir, que sea generado por un filtro, deberá ser generado manualmente con una llamada a la función **set_event()** cada vez que cambie el valor de un pixel.

- `set_format()`: Función que permitirá establecer valores para los miembros que son modificables una vez.
- `copy_format()`: Función que permitirá copiar el formato de otro **pix_stream**.

Inmediatamente después de su declaración, y si no se usa un inicializador que haga una copia del stream, todos los miembros de **pix_stream** deberán tener un valor inválido (como 0 para las resoluciones), lo que permitirá asegurar sean definidos antes de usarlos. Los miembros se inicializan llamando la función `set_format()`, definida como:

```
set_format_call ::= pix_stream_identifier . set_format ( property_value_list ) ;
property_value_list ::= property_value { , property_value }
property_value ::=
        pix_stream_prop = unsigned_number
      | depth = { unsigned_number { , unsigned_number } }
pix_stream_prop ::=
        res_h
      | res_v
```

La función deberá ejecutarse en tiempo de compilación, y producirá un error si alguno de los miembros ya ha sido inicializado anteriormente con un valor diferente. Llamadas posteriores a la función con el mismo valor para los miembros del **pix_stream** no deberán producir errores. Este comportamiento permite que la función se pueda utilizar para requerir un formato específico de un stream de video, pues si la entrada tiene un valor distinto al requerido se producirá un error al tratar de modificar por segunda vez el formato.

Adicionalmente, se define una función para copiar el formato de otro **pix_stream**. Es importante destacar que esta función no deberá crear una copia del **pix_stream**, solamente copiar el valor de los miembros que definen el formato. La función es definida como:

```
copy_format_call ::= pix_stream_identifier . copy_format ( pix_stream_identifier ) ;
```

A continuación se presenta un ejemplo de declaración y uso:

```
pix_stream in_pix, out_pix;                                    // Declaration
in_pix.set_format( res_h=640, res_v=480, depth={8} );          // Format (1)
out_pix.copy_format( in_pix );                                 // Format (2)
on_event( in_pix.new_data ) {                                  // Event
        out_pix.data[0] = in_pix.data[0];
        set_event( out_pix.new_data );
```

```
}
```

3.3.3. Almacenamiento

memory

```
memory_declaration ::= memory_type memory_ident_list ;
memory_type ::= memory_keyword ( mem_width , mem_len1 , mem_len2 )
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_keyword ::= memory
```

Tipo de dato que se usará para representar una memoria, interna o externa, que pueda almacenar como mínimo un cuadro de video. En su declaración se deberán especificar el ancho de palabra en bits, `mem_width`, el largo de la primera dimensión, `mem_len1`, lo que permitirá crear un *line-buffer*; y el largo de la segunda dimensión, `mem_len2`, si es que se desea crear un *frame-buffer*, o un valor 0 en caso contrario. Una **memory** será una estructura con miembros de varios tipos, que se accederán usando el operador ".". Los miembros de la estructura se listan a continuación:

- `width`: Ancho en bits de los elementos contenidos. Siempre será mayor a 0. De tipo **parameter**, sólo lectura.
- `len1`: Longitud de la primera dimensión. Siempre será mayor a 0. De tipo **parameter**, sólo lectura.
- `len2`: Longitud de la segunda dimensión. Si es 0, corresponde a una memoria unidimensional, como un *line-buffer*, y si es mayor a 0, corresponde a una memoria bidimensional, como un *frame-buffer*. De tipo **parameter**, sólo lectura.
- `port`: Arreglo que contiene los puertos de comunicación. No tiene un tamaño definido, pero se recomienda acceder a los elementos en orden. Para que el puerto sea detectado como "conectado" se deberá realizar por lo menos una asignación, no siendo posible usarlo

directamente para acceder al contenido de la memoria. De tipo **memory_port**, sólo lectura.

El acceso a los datos no se realizará directamente desde una **memory**, si no que se utilizará un puerto de comunicación, **memory_port**, conectado a ella. El número de puertos usado y su acceso (r, w, r/w) es inferido automáticamente por el compilador al analizar el código fuente.

memory_port

```
memory_port_declaration ::= memory_port_type memory_port_ident_init_list ;
memory_port_type ::= memory_port_keyword
memory_port_ident_init_list ::= memory_port_ident_init { , memory_port_ident_init }
memory_port_ident_init ::= memory_port_identifier [ = memory_port_initializer ]
memory_port_initializer ::= memory_port_identifier
memory_port_identifier ::= single_identifier
memory_port_keyword ::= memory_port
```

Un **memory_port** se conectará a una **memory** al asignarle un elemento del vector `port`. Las asignaciones podrán ocurrir en el momento de la declaración, en asignaciones siguientes, e incluso en tiempo de funcionamiento, lo que permitirá intercambiar y/o redirigir los puertos. El acceso al contenido de la **memory** se realizará como:

```
memory_access ::= memory_port_identifier ( mem_idx1 , mem_idx2 ) ;
mem_idx1 ::=
          unsigned_number
        | unumber_identifier
mem_idx2 ::=
          unsigned_number
        | unumber_identifier
```

En donde `mem_idx1` representa el índice de la primera dimensión, pudiendo tomar valores entre 0 y (`mem_idx1`-1), y `mem_idx2` es el índice de la segunda dimensión, con valores entre 0 y (`mem_idx2`-1). Esta forma de acceder al contenido de la **memory** permite utilizar directamente las coordenadas del pixel en un cuadro de video, eliminando la necesidad de tener que calcular la dirección equivalente si fuera una memoria de una dimensión.

Adicionalmente, se definirá una función de sistema que permita compartir el acceso de múltiples puertos (virtuales) a un puerto (físico) de memoria, útil cuando los accesos al contenido no

sean concurrentes, o la implementación física de una mayor cantidad de puertos no sea factible.

```
memory_port_mux_call ::=
        memory_port_mux ( port_ident_dst, port_ident_src1 , port_ident_src2 ) ;
port_ident_dst ::= memory_port_identifier
port_ident_src1 ::= memory_port_identifier
port_ident_src2 ::= memory_port_identifier
```

A continuación se presenta un ejemplo de declaración y uso de **memory** en conjunto con un **memory_port**:

```
memory(24,640,480) my_memory;
memory_port my_port = my_memory.port[0];
raw_bits(24) my_var;
my_var = my_port(0,0); // Read element (0,0) (first)
my_port(9,9) = my_var; // Write element (9,9)
// port_mux usage
// Access my_port2 and my_port3 using only my_port1
memory_port my_port1, my_port2, my_port3;
my_port1 = my_memory.port[1];
memory_port_mux( my_port1, my_port2, my_port3 );
```

3.3.4. Conversión

cast

```
cast_call ::= cast ( cast_dst_type , cast_src ) ;
cast_dst_type ::=
        raw_bits_type
      | number_type
      | unumber_type
cast_src ::=
        number_constant
      | raw_bits_identifier
      | number_identifier
      | unumber_identifier
```

Realizará la conversión entre tipos de datos, ya que el lenguaje tiene con un tipado fuerte, y como consecuencia los operadores aritméticos no podrán realizar operaciones en bits sin formato. Para ello, deberá tomar los bits de `cast_src` y les cambiará el tipo al de destino, `cast_dst_type`. Es importante destacar que no se debe modificar el contenido de los bits, solamente su interpretación. A continuación se presenta un ejemplo de uso:

```
pix_stream a;
unumber(8,0) b;
b = cast( unumber(8,0), a.data[0][7:0] );
```

num_approx

num_approx_call ::= **num_approx** (number_constant , int_bits , frac_bits) ;

Realizará la aproximación de una constante fraccionaria a un número específico de bits. Específicamente, la parte entera de la constante deberá caber en `int_bits` bits (con el signo incluido), produciendo un error si el los bits son insuficientes. La parte fraccionaria se aproximará buscando el valor más cercano que sea representable en `frac_bits` bits. Esta operación será requerida en todos los lugares donde se use una constante fraccionaria, pues podría perderse precisión al pasar de su representación en dígitos decimales a bits. A continuación se presenta un ejemplo de uso:

```
number(2,3) my_num;
my_num = num_approx( +d0.3333333, 2, 3 ); // 2 bits int, 3 bits frac
```

unum_approx

unum_approx_call ::= **unum_approx** (number_constant , int_bits , frac_bits) ;

Realizará la aproximación de una constante fraccionaria a un número específico de bits. Específicamente, la parte entera de la constante deberá caber en `int_bits` bits, produciendo un error si el los bits son insuficientes. La parte fraccionaria se aproximará buscando el valor más cercano que sea representable en `frac_bits` bits. A continuación se presenta un ejemplo de uso:

```
unumber(1,3) my_num;
my_num = unum_approx( d0.3333333, 1, 3 ); // 1 bit int, 3 bits frac
```

round

```
round_call ::= round ( src_identifier ) ;
src_identifier ::=
        number_identifier
      | unumber_identifier
```

Realizará el redondeo al entero más cercano de un número, con o sin signo. El redondeo se hará alejándose del cero. Por ejemplo, -2.5 se redondea a -3 y 2.5 se redondea a 3. A continuación se presenta un ejemplo de uso:

```
number(2,5) a, b;
a = num_approx( +1.25, 2, 5 );
b = round( a ); // Round to 1.0
```

3.4. Sentencias y expresiones

```
statement ::=
        compound_statement
      | declaration_statement
      | assignment_statement
      | funcion_call_statement
      | conditional_statement
      | loop_statement
      | on_event_statement
      | unary_expression ;
compound_statement ::= { statement { statement } }
```

Una sentencia será cualquier expresión que ejecute una acción y esté terminada en "**;**". Una sentencia puede estar compuesta por otras sentencias, formando una sentencia compuesta, que deberá ser encerrada entre paréntesis de llave "**{}**". Se crearon varias categorías de sentencias para facilitar su clasificación.

Una sentencia de declaración permitirá declarar una variable (identificador). Definida formalmente como:

```
declaration_statement ::=
          event_declaration
        | raw_bits_declaration
        | number_declaration
        | unumber_declaration
        | parameter_declaration
        | array_declaration
        | pix_stream_declaration
        | memory_declaration
        | memory_port_declaration
```

Una sentencia de asignación permitirá asignar el valor resultante de una expresión a una variable (identificador). Definida formalmente como:

```
assignment_statement ::= assignment_lvalue = expression
assignment_lvalue ::=
          raw_bits_identifier
        | number_identifier
        | unumber_identifier
        | array_element
        | pix_stream_identifier
        | memory_port_identifier
        | memory_access
```

Una sentencia de llamada a función realizará el llamado a una función, que no tiene ningún tipo de retorno, como *void* en C. Definida formalmente como:

```
funcion_call_statement ::=
          set_event_call
        | set_format_call
        | copy_format_call
        | memory_port_mux_call
```

3.4.1. Sentencia if

```
conditional_statement ::= if ( expression ) statement [ else statement ]
```

Se usará para tomar una decisión sobre si una sentencia es ejecutada. Si la expresión se

evalúa verdadera, es decir, es distinta de 0, se ejecutará la primera sentencia. En cambio, si se evalúa falsa, no se ejecutará la primera sentencia, y opcionalmente se ejecutará la segunda sentencia. A continuación se presenta un ejemplo de uso:

```
number(8,0) a, b, c = 10;
// Only if
if( c < 5 ) {
        a = 4;
}
// if-else
if( c > 15 ) {
        b = 10;
}
else {
        b = 45;
}
```

3.4.2. Sentencia for

```
loop_statement ::= for ( variable_assignment ; expression ; variable_assignment ) statement
variable_assignment ::=
        assignment_statement
      | unary_expression
```

Se usará para ejecutar una sentencia de forma repetitiva mientras la expresión sea evaluada verdadera (valor distinto de 0). La primera asignación de variable se ejecutará siempre. Luego, mientras la expresión se evalúe verdadera, se ejecutará la sentencia y la segunda asignación de variable. Si la expresión se evalúa falsa, se ejecutará la siguiente sentencia. A continuación se presenta un ejemplo de uso:

```
unumber(16,0) x_out = 100;
unumber(16,0) idx;
for( idx = 0; idx < 10; idx = idx + 1 ) {
        x_out = x_out + idx;
}
```

3.4.3. Sentencia on_event

on_event_statement ::= **on_event** (event_expression) compound_statement
event_expression ::=
event_identifier { **or** event_identifier }
| event_identifier { **and** event_identifier }

Se usará para ejecutar una sentencia cuando se produzca un evento. El evento permanecerá bloqueado hasta que la sentencia se haya ejecutado. El flujo de ejecución de esta sentencia es distinto al resto, pues se puede disparar su ejecución en cualquier momento, y no de forma secuencial como el resto de las sentencias. Ejemplos de uso se verán en la definición de **hardware_thread**.

3.4.4. Expresiones

```
expression ::=
          number_constant
        | rvalue_expresion
        | unary_expression
        | expression binary_operator expression
        | conversion_call_expression
rvalue_expresion ::=
          raw_bits_identifier
        | raw_bits_slice
        | number_identifier
        | unumber_identifier
        | parameter_identifier
        | array_element
        | pix_stream_identifier
        | memory_port_identifier
        | memory_access
unary_expression ::= unary_operator single_identifier
conversion_call_expression ::=
          cast_call
        | num_approx_call
        | unum_approx_call
        | round_call
```

Una expresión es una construcción del lenguaje que combina operandos con operadores para producir un resultado que depende del valor de los operandos y el significado semántico del operador. Cualquier operando, incluso sin un operador, es considerado una expresión. Cada vez que se necesite un valor en una sentencia se puede usar una expresión.

Una expresión puede ser una constante numérica, un identificador, una expresión unaria (operador unario + operando), el resultado de aplicar un operador binario a dos expresiones, o el resultado de llamadas a funciones de sistema para conversión de tipos. Una expresión compuesta, que contiene varias expresiones con operadores, se evalúa de izquierda a derecha, respetando la precedencia de los operadores. Sin embargo, se pueden agrupar las expresiones entre paréntesis, "()", para cambiar la precedencia de los operadores.

3.4.5. Operadores

Un operador operará sobre uno o más operandos, dependiendo de su tipo. Formalmente, se definen como:

```
unary_operators ::= ~ | ! | ++ | -- | + | -
binary_operators ::=
        & | |
      | * | / | + | -
      | && | || | == | ! = | < | <= | > | >=
```

La precedencia de los operadores se puede examinar en su definición, teniendo más precedencia los operadores unarios que los binarios, y en cada tipo de operador, el que tiene mayor precedencia es el de la izquierda.

Los operadores unarios operarán sobre un operando, obteniendo una expresión, que será del mismo tipo del operando, como resultado. A continuación se realiza una descripción de cada uno:

Operador	**nombre**	**Descripción**
~	negación de bits	Invierte todos los bits del operado. Sólo es aplicable a operandos de tipo **raw_bits**
!	negación lógica	Invierte el valor lógico de una expresión, es decir, el nuevo valor será 0 si es que el valor anterior era distinto de 0. Similarmente, el nuevo valor será distinto de 0 si el valor anterior era 0. Es aplicable a operandos de tipo **raw_bits**, **number** y **unumber**
++	incrementar	Incrementa en 1 el valor de la expresión. Sólo es aplicable a operandos de tipo **number** y **unumber**
--	decrementar	Decrementa en 1 el valor de la expresión. Sólo es aplicable a operandos de tipo **number** y **unumber**
+	positivo	Signo positivo
-	negativo	Invierte el signo de una expresión

Los operadores binarios operarán sobre dos operandos del mismo tipo, obteniendo una expresión, que será del mismo tipo de los operandos, como resultado. A continuación se realiza una descripción de cada uno:

Operador	nombre	Descripción
&	*and* de bits	Realiza un *and* de bits entre sus dos operandos. Ambos deben tener la misma cantidad de bits, y ser de tipo **raw_bits**
|	*or* de bits	Realiza un *or* de bits entre sus dos operandos. Ambos deben tener la misma cantidad de bits, y ser de tipo **raw_bits**
*	multiplicación	Multiplica dos números. Sólo en operandos de tipo **number** y **unumber**
/	división	Divide dos números. Sólo en operandos de tipo **number** y **unumber**
+	suma	Suma dos números. Sólo en operandos de tipo **number** y **unumber**
−	resta	Resta dos números. Sólo en operandos de tipo **number** y **unumber**
&&	*and* lógico	Realiza un *and* lógico entre sus dos operandos. Es aplicable a **raw_bits**, **number** y **unumber**
||	*or* lógico	Realiza un *or* lógico entre sus dos operandos. Es aplicable a **raw_bits**, **number** y **unumber**
==	igual a	Es verdadero (distinto de 0) si sus dos operandos son iguales, falso en otro caso. Es aplicable a **raw_bits**, **number** y **unumber**
!=	distinto a	Es verdadero si sus dos operandos son distintos, falso en otro caso. Es aplicable a **raw_bits**, **number** y **unumber**
<	menor que	Es verdadero si el operando de la izquierda tiene un valor menor que el de la derecha. Es aplicable a **number** y **unumber**
<=	menor o igual que	Es verdadero si el operando de la izquierda tiene un valor menor o igual que el de la derecha. Es aplicable a **number** y **unumber**
>	mayor que	Es verdadero si el operando de la izquierda tiene un valor mayor que el de la derecha. Es aplicable a **number** y **unumber**
>=	mayor o igual que	Es verdadero si el operando de la izquierda tiene un valor mayor o igual que el de la derecha. Es aplicable a **number** y **unumber**

3.5. Hardware-threads

```
hw_thread_def ::= hw_thread_identifier ( hw_thread_def_arg_list ) compound_statement
hw_thread_call ::= hw_thread_identifier ( hw_thread_call_arg_list ) ;
hw_thread_def_arg_list ::= hw_thread_def_arg { , hw_thread_def_arg }
hw_thread_call_arg_list ::= hw_thread_call_arg { , hw_thread_call_arg }
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_call_arg ::=
          event_identifier
        | raw_bits_identifier
        | number_identifier
        | unumber_identifier
        | parameter_identifier
        | pix_stream_identifier
        | memory_port_identifier
hw_thread_identifier ::= single_identifier
```

Es la unidad de ejecución más genérica del lenguaje. Su definición se hará con una lista de argumentos separados por "**,**". El rol de cada uno de los argumentos, entrada, salida, o ambos, será detectado automáticamente por el compilador. La definición también deberá incorporar el cuerpo, delimitado por paréntesis de llaves, "**{}**". El cuerpo estará compuesto por sentencias, llamados a otros **hardware_thread**s, y llamados a **filter**s. El llamado (instanciación) se realizará con el identificador, más los argumentos entre paréntesis, finalizando con el carácter "**;**". El cuerpo estará dividido lógicamente en dos partes, la primera tendrá sentencias generales, y la segunda la sentencia **on_event**. Como analogía se puede pensar a la primera parte como un bloque *initial* de Verilog. Las sentencias generales estarán limitadas a declaraciones y asignaciones estáticas, pues son resultas en tiempo de compilación. Este requerimiento impide que se realicen accesos a memoria, sentencias condicionales, o ciclos iterativos. En cambio, en la sentencia **on_event** no existe ninguna restricción, pues será la encargada de definir la parte

dinámica o algorítmica del **hardware_thread**. A continuación se presentan algunos ejemplos de definición y uso:

```
// Empty declaration
dummy_job( pix_stream a, pix_stream b ) {
    // ...
}
```

```
// Declaration and a simple statement
dummy_copy( number(5,0) in_a, number(5,0) out_b ) {
    out_b = in_a;
}
```

```
// Simple pipeline
dummy_copy1( number(5,0) in_a, number(5,0) out_b ) {
    out_b = in_a;
}

dummy_copy2( number(5,0) in_a, number(5,0) out_b ) {
    out_b = in_a;
}

dummy_copy_pipeline( number(5,0) in_a, number(5,0) out_b ) {
    number(5,0) tmp1;
    dummy_copy1( in_a, tmp1 );
    dummy_copy2( tmp1, out_b );
}
```

```
// Copy with an on_event statement
dummy_copy( number(5,0) in_a, number(6,0) out_b, event operate ) {
    on_event( operate ) {
        out_b = in_a;
    }
}
```

3.6. Filtros

```
filter_def ::= filter filter_identifier ( filter_def_arg_list ) compound_statement
filter_call ::= filter_identifier ( filter_call_arg_list ) ;
filter_def_arg_list ::= filter_def_arg { , filter_def_arg }
filter_call_arg_list ::= filter_call_arg { , filter_call_arg }
filter_def_arg ::=
        pix_stream_type pix_stream_identifier
      | memory_port_type memory_port_identifier
      | parameter_type parameter_identifier
filter_call_arg ::=
        pix_stream_identifier
      | memory_port_identifier
      | parameter_identifier
filter_identifier ::= single_identifier
```

Es una unidad de ejecución más restrictiva que un **hardware_thread**, con el objetivo de fomentar la reutilización a través de la estandarización. Como entrada, obligatoriamente debe tener un **pix_stream**, lo que también aplica para la salida. Opcionalmente, puede tener uno o varios **parameter** y **memory_port**, estando prohibido incluir cualquier variable con otro tipo de dato. Su definición es igual a la de un **hardware_thread**, pero se debe anteponer la palabra clave **filter** para diferenciarlos. El cuerpo estará compuesto y deberá seguir las mismas reglas. Así como también el llamado. A continuación se presenta un ejemplo de definición y uso:

```
filter dummy_flt( pix_stream a, pix_stream b ) {
    on_event( a.new_data ) {
        b.data[0] = a.data[0];
        set_event( b.new_data );
    }
}
```

Finalmente, se pueden utilizar las definiciones anteriores para definir formalmente un archivo de código fuente como:

```
source_text ::=
        { hw_thread_def } { filter_def }
        filter top ( filter_def_arg_list ) compound_statement
```

El filtro que se compilará (main/top) debe tener como nombre **top** para que el compilador lo reconozca. La versión más simple de un archivo de código fuente tendrá sólo el filtro **top**, y versiones más completas tendrán varios filtros formando un pipeline. A continuación se presenta un ejemplo de archivo de código fuente:

```
// Some definitions were omitted to improve legibility
filter flt1( pix_stream in_p, pix_stream out_p ) { /* ... */ }

filter flt2( pix_stream in_p, pix_stream out_p ) { /* ... */ }

filter top( pix_stream in_pix, pix_stream out_pix ) {
    pix_stream tmp;
    flt1( in_pix, tmp );
    flt2( tmp, out_pix );
}
```

3.7. Elementos incorporados

A continuación se definirán los elementos que incorpora el modelo. Haciendo una analogía con C, serían parte de la biblioteca estándar.

3.7.1. Hardware-threads

```
hwt_pix_to_memory( pix_stream src, memory_port dst_port,
                   event save_done );
```

Almacena un cuadro de video proveniente de un stream, en una memoria. Genera un evento para notificar que se ha escrito un cuadro de video completo en la memoria. Donde:

- `src`: Stream de video de origen.
- `dst_port`: Puerto de escritura a la memoria de destino. Las dimensiones deben coincidir.
- `save_done`: Evento que se dispara al terminar de copiar un cuadro de video completo.

```
hwt_memory_to_pix( memory_port src_port, pix_stream dst,
                   event start );
```

Crea un stream de video a partir de una memoria. El evento de entrada controla el inicio de la conversión de un cuadro de video, teniendo que dispararse periódicamente por cada cuadro. Donde:

- `src_port`: Puerto de lectura a la memoria de origen. Las dimensiones deben coincidir.
- `dst`: Stream de video de destino.
- `start`: Evento que controla el inicio de la conversión.

3.7.2. Filtros

```
flt_crop( pix_stream src, pix_stream dst,
          parameter size_h, parameter size_v,
          parameter offet_h, parameter offset_v);
```

Recorta un área del stream de entrada, y la entrega a su salida como otro stream de tamaño más pequeño, sin modificar el valor de los pixeles conservados. Donde:

- `src`: Stream de entrada.
- `dst`: Stream de salida.
- `size_h`: Tamaño horizontal a conservar.
- `size_v`: Tamaño vertical a conservar.
- `offset_h`: Offset horizontal de la zona conservada.
- `offset_v`: Offset vertical de la zona conservada.

```
flt_fill( pix_stream src, pix_stream dst,
          parameter new_size_h, parameter new_size_v,
          parameter location_h, parameter location_v,
          parameter fill_value );
```

Rellena el stream de entrada, aumentando su tamaño y produciendo un nuevo stream a la salida, donde se copiarán los datos de la entrada en una ubicación indicada por el usuario. Todo esto sin modificar el valor de los pixeles de entrada. Donde:

- `src`: Stream de entrada.
- `dst`: Stream de salida.
- `new_size_h`: Tamaño horizontal a conservar.
- `new_size_v`: Tamaño vertical a conservar.
- `location_h`: Ubicación horizontal de la zona. En pixeles, entre 0 y `new_size_h`-1.
- `location_v`: Ubicación vertical de la zona. En pixeles, entre 0 y `new_size_v`-1.
- `fill_value`: Valor con el que rellenar el espacio recortado.

```
flt_interpolate( pix_stream src, pix_stream dst,
                 parameter new_size_h,
                 parameter new_size_v,
                 parameter method );
```

Realiza una interpolación al stream de entrada, pudiendo achicar o agrandar el stream de salida. Para ello, puede modificar el valor de los pixeles del stream de entrada usando un algoritmo de interpolación, que es elegido mediante un parámetro. Donde:

- `src`: Stream de entrada.
- `dst`: Stream de salida.
- `new_size_h`: Nuevo tamaño horizontal.
- `new_size_v`: Nuevo tamaño vertical.
- `method`: Método de interpolación:
  - 0 = Vecino más cercano
  - 1 = Bilineal
  - 2 = Bicúbico

- 3 = Gauss

```
flt_dim_redux( pix_stream src, pix_stream dst );
```

Concatena todas las dimensiones del stream de entrada, entregando un stream de salida con sólo una dimensión. Esta operación es realizada sin modificar el valor de los pixeles de entrada. Donde:

- `src`: Stream de entrada.
- `dst`: Stream de salida.

```
flt_slide_window( pix_stream src, pix_stream dst,
                  parameter win_size,
                  parameter border_method );
```

Crea una ventana deslizante del stream de entrada, obteniendo un stream de salida con una dimensione adicional por cada pixel de la vecindad del pixel actual. Por ejemplo, una ventana de 3×3 produce un stream de salida de 9 dimensiones. Como requerimiento, el stream de entrada debe contener exactamente una dimensión. Donde:

- `src`: Stream de entrada.
- `dst`: Stream de salida.
- `win_size`: Tamaño de la ventana. Ej: Un valor de 3 produce una ventana de 3×3.
- `border_method`: Comportamiento de la ventana al borde de un cuadro de video:
  - 0 = Rellenar con ceros
  - 1 = Duplicar último pixel
  - 2 = Reflejar frame

3.8. Ejemplos

A continuación se mostrarán varios ejemplos sobre le uso de las distintas características en conjunto del lenguaje de programación.

3.8.1. Simples

• Ejemplo 1: Un simple de procesamiento pixel a pixel en el que solamente se invierte el valor de todos los bits del pixel, creando una salida en *negativo*.

```
filter top( pix_stream in_pix, pix_stream out_pix ) {
    // Set (require) input format
    in_pix.set_format( res_h=640, res_v=480, depth={24} );
    // Set output format (same as input)
    out_pix.copy_format( in_pix );
    // Color invertion
    on_event( in_pix.new_data ) {
        out_pix.data[0] = ~in_pix.data[0];
        set_event( out_pix.new_data );
    }
}
```

• Ejemplo 2: Copia el stream de video de entrada a una memoria y luego genera un stream de video a las salida usando la memoria como origen.

```
filter top( pix_stream in_pix, pix_stream out_pix ) {
    // Set (require) input format
    in_pix.set_format( res_h=640, res_v=480, depth={24} );
    // Set output format
    out_pix.copy_format( in_pix );
    // Create memory
    memory( 24, in_pix.res_h, in_pix.res_v ) frame_buffer1;
    memory_port fb1_port1 = frame_buffer1.port[0];
    memory_port fb1_port2 = frame_buffer1.port[1];
    // To frame buffer
    event to_memory_ready;
```

```
    hwt_pix_to_memory( in_pix, fb1_port1, to_memory_ready );
    // To pix_sstream
    hwt_memory_to_pix( fb1_port2, out_pix, to_memory_ready );
}
```

• Ejemplo 3: Convierte una imagen a escala de grises, y luego la binariza (blanco y negro) usando un umbral.

```
filter rgb24_to_grayscale( pix_stream in_pix, pix_stream out_pix ) {
    // Set (require) input format
    in_pix.set_format( depth={24} );
    // Set output format
    out_pix.set_format( depth={8} );
    // Declare temporal variables
    unumber(8,0) rgb_1, rgb_2, rgb_3, gray_conv;
    on_event( in_pix.new_data ) {
        rgb_1 = cast( unumber(8,0), in_pix.data[0][7:0] );
        rgb_2 = cast( unumber(8,0), in_pix.data[0][15:8] );
        rgb_3 = cast( unumber(8,0), in_pix.data[0][23:16] );
        gray_conv = ( rgb_1 + rgb_2 + rgb_3 ) / 3;
        out_pix.data[0] = cast( raw_bits(8), gray_conv );
        set_event( out_pix.new_data );
    }
}

filter umbralize( pix_stream in_pix, 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( pix_stream in_pix, pix_stream out_pix ) {
    // Set (require) input format
    in_pix.set_format( res_h=640, res_v=480, depth={24} );
    // Set output format
    out_pix.set_format( res_h=640, res_v=480, depth={1} );
    // Pipeline
    pix_stream tmp;
    rgb24_to_grayscale( in_pix, tmp );
    umbralize( tmp, out_pix );
}
```

3.8.2. Estabilización de video

A continuación se muestran algunas partes relevantes del código fuente, pudiendo encontrar la versión completa en el anexo A.

• Un **hardware_thread** que crea un doble buffer a partir de una memoria unidimensional. Demuestra la capacidad de asignación dinámica de los puertos de memoria.

```
double_buffer( memory_port front, memory_port back, event swap,
               parameter buff_width, parameter buff_len ) {

    memory(buff_width,buff_len,0) buff1;
    memory(buff_width,buff_len,0) buff2;
    unumber(1,0) swap_state = 0;

    on_event( swap ) {
        swap_state = ~swap_state;

        if( swap_state == 0 ) {
            front = buff1.port[0];
            back  = buff2.port[0];
```

```
        }
        else {
            front = buff2.port[0];
            back  = buff1.port[0];
        }
    }
}
```

• Un **hardware_thread** que calcula las proyecciones integrales en línea, es decir, a medida que van llegando desde el stream de video, y las almacena en memorias. Realiza varias asignaciones condicionales dependiendo de la línea y pixel actual.

```
calc_proj( pix_stream in_pix, memory_port x_proj_mem, memory_port y_proj_mem,
           event proj_ready ) {

    unumber(16,0) x_count = 0;
    unumber(16,0) y_count = 0;
    unumber(16,0) proj_x_val;
    unumber(18,0) proj_y_accum;
    //
    on_event( in_pix.new_data ) {
        // New frame
        if( x_count == 0 && y_count == 0 ) {
            proj_y_accum = 0;
        }
        // New line
        if( x_count == 0 ) {
            proj_y_accum = proj_y_accum +
                           cast( unumber(18,0), y_proj_mem(y_count,0) );
            y_proj_mem(y_count,0) = cast( raw_bits(18), proj_y_accum );
            proj_y_accum = 0;
            ++y_count;
        }
        else {
            proj_x_val = cast( unumber(16,0), x_proj_mem(x_count,0) );
            proj_x_val = proj_x_val
                         cast( unumber(8,0), in_pix.data[0] );
            x_proj_mem(x_count,0) = cast( raw_bits(18), proj_x_val );
            proj_y_accum = proj_y_accum +
                           cast( unumber(8,0), in_pix.data[0] );
            ++x_count;
        }
        // End frame
```

```
        if( x_count == in_pix.res_h && y_count == in_pix.res_v ) {
            set_event( proj_ready );
        }
    }
}
```

• Un **hardware_thread** que realiza la estimación de movimiento entre dos proyecciones. Implementa un algoritmo iterativo dentro de dos ciclos **for** anidados para calcular el mínimo valor de la suma de diferencias absoluta entre todos los desplazamientos posibles.

```
motion_estimation( memory_port proj_new, memory_port proj_old,
                   number(6,0) desp_found,
                   event start, event ready,
                   parameter MAX_D, parameter search_len ) {

    unumber(27,0) min_SAD;
    //
    on_event( start ) {
        min_SAD = MAX( unumber(27,0) );
        unumber(8,0) val_new;
        unumber(8,0) val_old;
        unumber(8,0) val_abs;
        desp_found  = 0;
        unumber(16,0) p_idx;
        unumber(6,0) test_desp;
        for( test_desp = -MAX_D; test_desp < MAX_D; ++test_desp ) {
            curr_SAD = 0;
            for( p_idx = MAX_D; p_idx < search_len-MAX_D; ++p_idx ) {
                val_new = cast( unumber(8,0), proj_new(p_idx+test_desp,0) );
                val_old = cast( unumber(8,0), proj_old(p_idx,0) );
                //
                if( val_new > val_old )
                    val_abs = val_new - val_old;
                else
                    val_abs = val_old - val_new;
                //
                curr_SAD = curr_SAD + val_abs;
            }
            //
            if( curr_SAD < min_SAD ) {
                min_SAD    = curr_SAD;
                desp_found = test_desp;
            }
```

```
        }
        //
        set_event( ready )
    }
}
```

• Un **hardware_thread** que realiza el filtrado de movimientos no deseados. Usa aritmética de punto fijo para filtrar, limitar, y redondear la entrada.

```
// Simple MVI implementation
motion_filter( number(6,0) desp_in, number(6,0) desp_filt,
               event start, event ready, parameter MAX_D ) {

    unumber(1,2) delta = unum_approx( 0.5, 1, 2 );
    number(8,5) fmv, fmv_prev, fmv_round;

    on_event( start ) {
        // -- Filter
        fmv = delta*fmv_prev + desp_in;
        fmv_prev = fmv;
        // -- Limit
        if( fmv > MAX_D )
            fmv = MAX_D;
        else if( fmv < -MAX_D )
            fmv = -MAX_D;
        // -- Round
        fmv_round = unum_approx( fmv, 6, 0 );
        //
        desp_filt = cast( number(6,0), fmv_round );
        set_event( ready );
    }
}
```

3.8.3. Estabilización térmica de video infrarrojo

A continuación se muestran algunas partes relevantes del código fuente, pudiendo encontrar la versión completa en el anexo B.

• Un **hardware_thread** que realiza la corrección del video entrante usando datos de calibración almacenados en un frame buffer y parámetros de corrección. Además, usa contadores para obtener la ubicación del pixel actual.

```
correction( pix_stream in_pix, pix_stream out_pix, memory_port cal_data,
            number(14,0) f0_par, number(14,0) u_par ) {
    //
    unumber(10,0)  h_count = 0;
    unumber(10,0)  v_count = 0;
    raw_bits(32,0) s_data;
    number(16,0)   Sij_1, Sij_2;
    unumber(14,0)  Yij_div_Sij_0;
    unumber(14,0)  Xij;
    //
    on_event( in_pix.new_data ) {
        // Get parameters from memory
        s_data = cal_data( h_count, v_count );
        Sij_1  = cast( number(16,0), s_data[15:0] );
        Sij_2  = cast( number(16,0), s_data[31:16] );
        // Get pixel value
        Yij_div_Sij_0 = cast( unumber(14,0), in_pix.data[0] );
        // Perform correction
        Xij = Yij_div_Sij_0 + Sij_1*f0_par + Sij_1*u_par;
        // Increment or reset counters
        if( (h_count == in_pix.res_h-1) && (v_count == in_pix.res_v-1) ) {
            h_count = 0;
            v_count = 0;
        }
        else {
            h_count = h_count + 1;
            v_count = v_count + 1;
        }
        // Output
        out_pix.data[0] = cast( raw_bits(14,0), Xij );
        set_event( out_pix.new_data );
    }
}
```

• Un **filter** que realiza el promedio de los elementos de una ventana deslizante y luego calcula la diferencia entre el promedio y el pixel central.

```
filter exp_value( pix_stream in_pix, pix_stream out_pix ) {
    // Set output format
    in_pix.set_format( res_h=in_pix.res_h, res_v=in_pix.res_v, depth={14} );
    //
    number(24,0) accum;
    on_event( in_pix.new_data ) {
```

```
        accum = 0;
        accum = accum + cast( unumber(14,0), in_pix.data[0] );
        accum = accum + cast( unumber(14,0), in_pix.data[1] );
        accum = accum + cast( unumber(14,0), in_pix.data[2] );
        accum = accum + cast( unumber(14,0), in_pix.data[3] );
        accum = accum + cast( unumber(14,0), in_pix.data[5] );
        accum = accum + cast( unumber(14,0), in_pix.data[6] );
        accum = accum + cast( unumber(14,0), in_pix.data[7] );
        accum = accum + cast( unumber(14,0), in_pix.data[8] );
        accum = 256*cast( unumber(14,0), in_pix.data[4] ) - accum;
        accum = accum / 9;
        out_pix.data[0] = cast( raw_bits(14), accum );
        set_event( out_pix.new_data );
    }
}
```

Capítulo 4: Arquitectura

Este capítulo presenta una arquitectura base para las posibles implementaciones del modelo propuesto. Específicamente, se presentan detalles sobre eventos, el bus de video, el bus de almacenamiento, el bus de propósito general, y sobre algunos hardware-threads.

4.1. Eventos y control de flujo

Un evento tiene como propósito coordinar la operación y transferencias de información entre varios elementos de procesamiento. Es producido por un emisor, y consumido por un receptor. La situación descrita a continuación es la más completa requerida por el modelo, pues el evento se utiliza para una transferencia de datos con soporte de detenciones.

Se distinguen varios casos que un evento debe ser capaz de manejar:

- Emisor produce resultados a la misma velocidad que el receptor los puede procesar, estando perfectamente coordinados y sin ningún problema de comunicación.
- Emisor produce resultados a una velocidad menor a la que el receptor puede procesar, teniendo este último que esperar por resultados válidos del emisor.
- Emisor produce resultados a una velocidad mayor a la que el receptor puede procesar, provocando detenciones en el emisor.

Dentro de las posibilidades para realizar control de flujo de forma sincrónica y con un mismo reloj, la más simple que satisface los casos necesarios para el modelo y que otorga una menor latencia es conocida como “interfaz FIFO”. Requiere agregar dos señales de control adicionales al sistema, ambas binarias (valores 0 o 1). La señal **valid** permite notificar al receptor que hay datos validos en su entrada, y la señal **ready** permite notificar al emisor que el receptor se encuentra disponible para recibir datos. Dependiendo del valor de las señales de control, el bus puede estar en 3 estados:

valid	**ready**	Estado	Descripción
0	1	**idle**	En reposo. El receptor está listo para recibir datos.
1	1	**tx**	Transmisión. Emisor se encuentra transmitiendo datos.
x	0	**busy**	Ocupado. El receptor se encuentra procesando datos, por lo que el emisor no puede transmitir datos.

Una transferencia como mínimo debe pasar por la secuencia de estados **idle**, **tx**, y volver a **idle** nuevamente. Pudiendo existir transferencias que no pasen por el estado **busy**, debido a que el receptor es capaz de recibir el siguiente dato inmediatamente (en el siguiente canto de reloj).

A continuación se describen los pasos que deben seguir el emisor y receptor para comunicarse utilizando el protocolo. El emisor debe seguir los siguientes pasos para realizar una trasferencia, usualmente en una máquina de estado:

(reset) La señal **valid** debe tener un valor igual a 0.

1. Muestrear la señal **ready**. Si el valor es 0 (receptor desocupado), avanzar al siguiente paso. Si el valor es 1, permanecer acá (detención detectada), esperar un ciclo de reloj y repetir el muestreo.

2. Poner el primer dato en la salida, y un valor 1 a la señal **valid**. Esperar el siguiente ciclo de reloj y avanzar al paso siguiente.

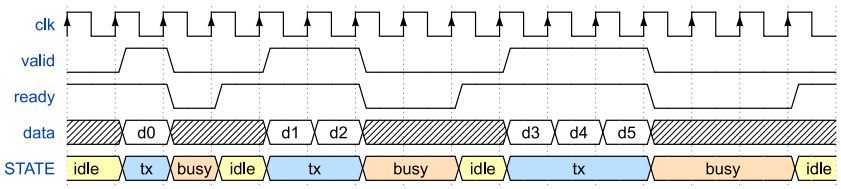
3. Si no hay más datos que transferir, finalizar la trasferencia al poner un valor 0 en la señal **valid**. Pasar el siguiente paso si todavía faltan datos.

4. Muestrear la señal **ready**. Si el valor es 1 (detención detectada), permanecer acá; esperar un ciclo de reloj y repetir el muestreo. Si el valor es 0 (receptor desocupado), poner el siguiente dato en la salida, manteniendo la señal **valid** en 1; esperar un ciclo de reloj, e ir al paso 3. Importante: No se debe cambiar el contenido de las salidas si el receptor está produciendo una detención, porque pueden producirse pérdidas de datos.

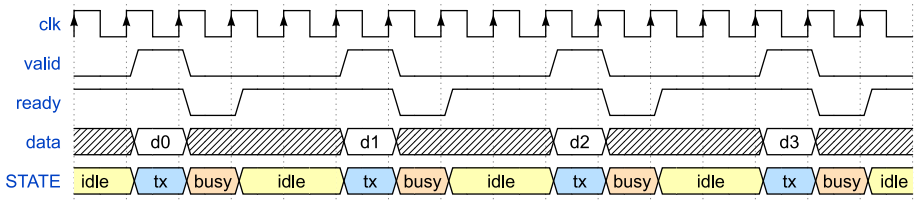
Similarmente, el receptor debe realizar los siguientes pasos:

(reset) La señal **ready** debe tener un valor igual a 1.

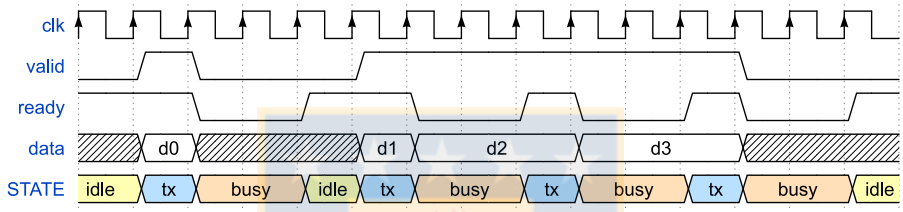
1. Muestrear la señal **valid**. Si el valor es 1 (emisor enviando), avanzar al siguiente paso. Si el valor es 0, permanecer acá; esperar un ciclo de reloj y repetir el muestreo.



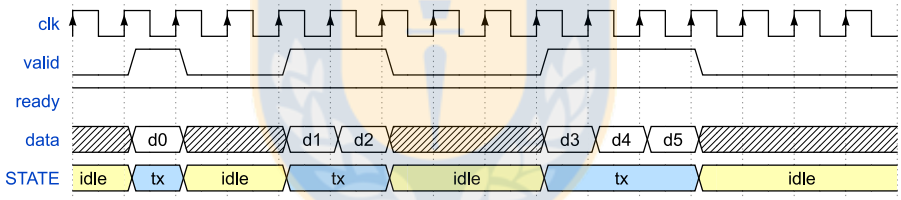
(a) Emisor y receptor a la misma velocidad



(b) Emisor más lento que receptor



(c) Emisor más rápido que receptor (detenciones)



(d) Receptor con procesamiento inmediato (pipeline)

Fig. 4.1: Ejemplos de transmisión en una interfaz FIFO

2. Guardar o procesar los datos en la entrada. Avanzar al siguiente paso.

3. Muestrear variables internas para determinar si hay espacio para procesar el dato que llegará en el siguiente ciclo de reloj. Si no hay espacio, poner un valor 0 en la señal **ready** (generará una detención), esperar un ciclo de reloj y repetir muestreo. Si hay espacio, poner un valor 1 en la señal **ready**, esperar un ciclo de reloj e ir al paso 1.

Es importante destacar que varios de los pasos recién mencionados pueden ser ejecutados paralelamente en una implementación, pero fueron separados para una lectura más conveniente.

La Figura 4.1 muestra los tipos de interacciones que se pueden dar entre el emisor y receptor. En particular, la Figura 4.1a muestra tres transferencias, de uno, dos, y tres datos respectivamente, y representa una situación en que el emisor envía datos a la misma velocidad que el

receptor los recibe y procesa. Si bien existen retardos de procesamiento de parte del receptor, en que el bus está en el estado **busy**, están perfectamente sincronizados con los del emisor. La Figura 4.1b presenta cuatro transferencias, con un dato cada una. En este caso, el emisor envía datos a una velocidad menor a la que puede recibir y procesar el receptor, realizando un envío cada cuatro ciclos de reloj. En cambio, el receptor es capaz de procesar los datos en un ciclo de reloj, lo que produce dos ciclos en que el bus está en estado **idle** por cada transferencia. La Figura 4.1c muestra dos transferencias, de uno y tres datos respectivamente, y representa una situación en que el emisor es capaz de enviar datos a una velocidad mayor a la que el receptor puede recibir y procesar, que es un dato por cada tres ciclos de reloj. En este caso, el receptor produce una detención por cada dato transferido, lo que se traduce en el bus está un gran cantidad de tiempo en el estado **busy**. La Figura 4.1d representa una situación ideal desde el punto de vista de la transmisión de datos, ya que el receptor puede recibir y procesar un dato en cada ciclo de reloj. Desde la perspectiva del bus, las transferencias sólo está ocupan dos estados, **idle** y **tx**, por lo que se obtiene una latencia mínima y una máxima cantidad de datos transferidos.

Del análisis anterior, en especial la Figura 4.1d se puede notar que la interfaz FIFO tiene, en el mejor caso, una latencia de un ciclo de reloj al inicio de cada envío, transfiriendo además un dato por cada ciclo de reloj, lo que ratifica la elección realizada.

4.2. Bus de video

Por el bus de video debe pasar el stream de video definido por el modelo, y además realizar el control de flujo entre los filtros. A continuación se describen las distintas señales que lo componen.

En los buses de transmisión de video comunes los datos del pixel se trasmiten en sincronía con el reloj de pixel. La frecuencia de funcionamiento de este reloj depende de características como la resolución, profundidad de bits, y tasa de refresco, entre otros. Debido a que los filtros pueden modificar estas características, y que la frecuencia final depende de ellas, el reloj debe ser transferido en conjunto con el stream de video, sufriendo las modificaciones adecuadas para mantener el tiempo de procesamiento del filtro y minimizar las detenciones. Además, se deben sincronizar todas las señales del bus al nuevo domino de reloj. Por ejemplo, si un filtro cambia el tamaño al doble, el reloj deberá subir también al doble para poder mantener la misma velocidad de procesamiento y no provocar detenciones en la entrada.

El bus debe contar también con alguna forma de realizar inicialización, que es usualmente

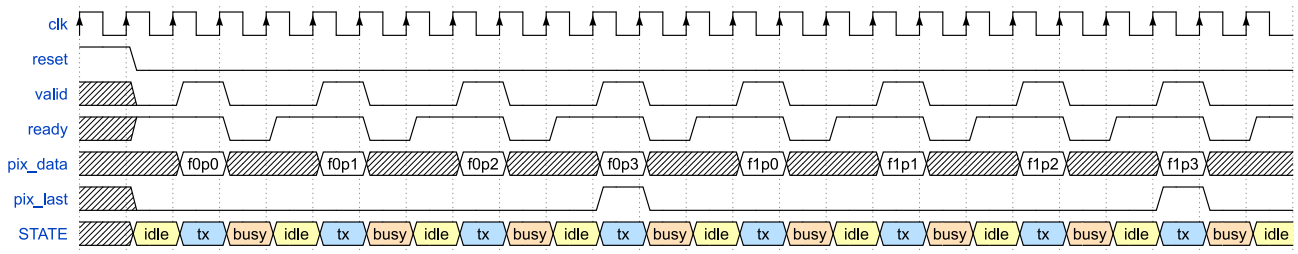


Fig. 4.2: Transmisiones en el bus de video

implementada en diseños de hardware con una señal de reset. Esta será una señal de reset global, sincronizada con el reloj del bus, por las mismas razones mencionadas para el reloj.

En un stream de video, los datos de video pueden ser representados de distintas formas o modelos de color. Este divide una imagen en distintos canales o componentes (dimensiones en el modelo), que a su vez pueden tener una profundidad de bits específica para la aplicación. A modo de ejemplo, en [35] se obtiene una imagen con 14 bits por píxel desde la cámara infrarroja. La imagen es procesada por distintos filtros, para finalmente ser transformada mediante un mapa de colores al modelo de color rojo, verde y azul (del inglés *red, green, blue*, RGB) con un total de 24 bits por píxel. Debido a la las distintas posibilidades de representación que se deben soportar, y que la implementación debe optimizar el uso de recursos lo mejor posible, es que el los datos de video se transportaran por señales de ancho variable, configurable al momento de diseño, dependiendo de las características del stream de video. Además, al momento de transmitir, todas las dimensiones deben ser físicamente concatenadas para mejorar el orden dentro del bus. Una vez recibidas, se puede aplicar nuevamente la separación lógica en dimensiones. Por ejemplo, para transmitir una imagen binarizada (blanco y negro) se usa sólo 1-bit, y para trasmitir una imagen RGB de ocho bits por canal, se utilizan 24-bits.

Como se mencionó en los capítulos anteriores, debe haber una forma de vaciar el pipeline (flush) de un filtro y sincronizarlo cuando haya terminado de llegar un cuadro de video completo. Esto se implementará mediante una señal que estará activa en el último pixel del cuadro de video. Además, esta señal permitirá la sincronización de los contadores en los filtros que necesiten la posición del pixel para trabajar, evitando así pasar dos señales adicionales por el bus, y ahorrando recursos lógicos en lo posible.

Para implementar el control de flujo se pude reutilizar la misma interfaz FIFO descrita previamente en el capítulo, ya que también se adapta muy bien a las necesidades de transporte unidireccionales del bus.

La Figura 4.2 muestra una transmisión de dos cuadros pertenecientes a un stream de video con un tamaño de 2×2 pixeles. La activación de la señal reset al principio inicializa todas las

señales a sus valores por defecto. Un pixel es transmitido por el emisor cada tres ciclos de reloj, y procesado por el receptor en un ciclo. En conjunto con la transferencia del cuarto pixel del primer cuadro de video, “f0p3”, se activa la señal de sincronización que indica que corresponde al último pixel del cuadro. La transferencia del segundo cuadro de video ocurre de la misma forma que el primero, finalizando con la transferencia del último pixel, “f1p3”, y la señal que indica que el cuadro de video está completo.

Es importante destacar que el ejemplo mostrado es solamente para fines ilustrativos, y que la transmisión puede funcionar tan rápido como la mostrada en la Figura 4.1d.

4.3. Bus de almacenamiento

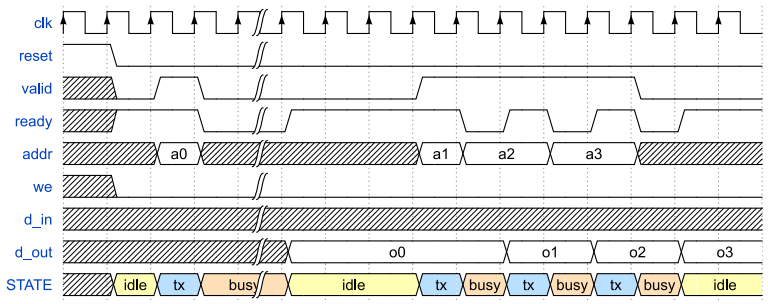
El bus de almacenamiento debe soportar transacciones (accesos) de lectura y escritura a memorias internas y externas que pueden ser no determinísticas. Adicionalmente, varias instancias del bus, llamadas canales, se pueden conectar a un mismo elemento de almacenamiento, el que internamente contendrá un arbitrador para completar las transacciones.

Como estructura básica para el bus se utilizará la interfaz FIFO vista previamente, con varias señales adicionales, pues tiene una baja latencia para memorias internas al chip, y soporta detenciones para accesos a memorias externas no determinísticas. El elemento de procesamiento que desea realizar una transacción cumple el rol del emisor, y el elemento de almacenamiento el rol del receptor. Debido a que una transacción de memoria tiene datos que viajan del receptor a emisor, se añadieron señales adicionales al bus que permitan obtener estos datos.

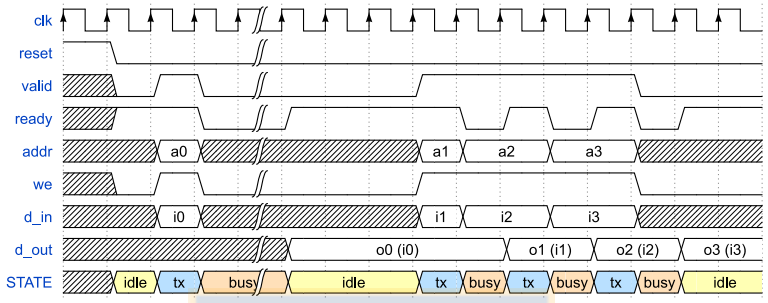
Se definen las siguientes señales adicionales que permiten controlar el tipo de transacción y la entrada/salida de datos:

- **we** (hacia almacenamiento): Señal binaria que controla el tipo de acceso, siendo 0 para lectura, y 1 para escritura.
- **addr** (hacia almacenamiento): Dirección de lectura/escritura.
- **d_in** (hacia almacenamiento): Datos de que se escribirán.
- **d_out** (desde almacenamiento): Datos leídos.

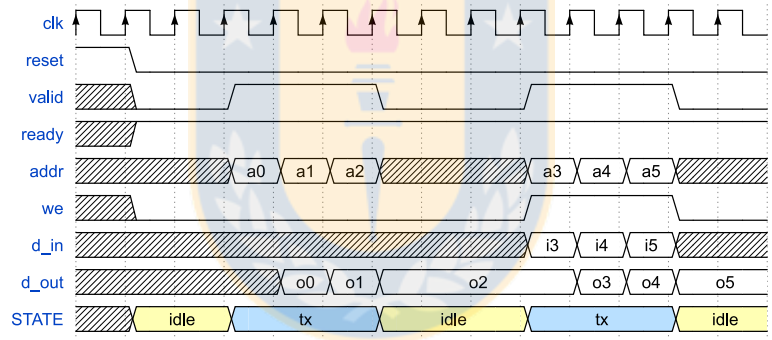
Las señales **d_in**, **d_out**, y **addr** deben tener un número de bits apropiado al tamaño de los datos y el espacio de direcciones asociado.



(a) Lectura no determinística y secuencial



(b) Escritura no determinística y secuencial



(c) Lectura y escritura con latencia mínima

Fig. 4.3: Ejemplos de transacciones de almacenamiento

Una transacción de memoria comienza sólo si señal **ready** es 1, pues es usada por el elemento de almacenamiento para indicar que hay una transacción en curso. Luego, se deben establecer todas las señales necesarias para la transacción: **we**, **addr**, **d_in** (de ser necesaria), y poner la señal **valid** en 1 para notificar al elemento de almacenamiento el comienzo de la transacción. La lectura o escritura estará completada cuando la señal **ready** sea 1, teniendo al menos un ciclo de latencia. Mientras la señal **ready** permanezca en el valor 1, se pueden seguir cambiando las señales **we**, **addr**, y **d_in** para operar en otras direcciones. Al finalizar una escritura, la salida, **d_out**, tendrá el valor escrito en esa dirección. El emisor termina la transacción al volver la señal **valid** al valor 0, pudiendo haber varios ciclos de reloj hasta que el receptor la finalice por su lado y entregue resultados.

La Figura 4.3 muestra varios ejemplos de transacciones en un bus de almacenamiento. En particular, la Figura 4.3a muestra dos transacciones de lectura, de uno y tres datos respectivamente. La primera transacción toma una cantidad no definida de ciclos, provocando una espera no determinística. Mientras que la segunda transacción toma 2 ciclos de reloj por dato leído. La Figura 4.3b muestra dos transacciones de escritura, de uno y tres datos respectivamente, con las mismas características de las transacciones de lectura anteriores. Es importante destacar que los datos escritos en las transacciones son también obtenidos en la salida cuando finaliza la escritura. La Figura 4.3c muestra dos transacciones de tres datos cada una, de lectura y escritura respectivamente. Esta situación representa transacciones con la mínima latencia posible, realizando los accesos en sólo un ciclo de reloj, lo que coincide con el comportamiento observado en las memorias alojadas al interior del chip. Esto ratifica que la elección de la interfaz FIFO, pues permite latencias bajas, manteniendo las ventajas de las memorias internas.

4.4. Bus de propósito general

Los buses de comunicación son cada vez más necesarios a medida que crece el tamaño y complejidad de los SoCs, pues permiten ordenar el sistema al momento del diseño y reducir el uso de recursos en el chip al usar buses compartidos. Entre los buses de propósito general comúnmente usados en el desarrollo de SoCs están Wishbone, [40], un bus de código abierto desarrollado por OpenCores y usado en la mayoría de los proyectos disponibles en su sitio web; y AXI, [41], un bus desarrollado por ARM para ser usado en sus productos y facilitar la integración con componentes desarrollados por sus clientes. Este tipo de buses trabaja realizando una interconexión de varias unidades funcionales de forma directa o a través de conmutadores, con cada elemento realizando transacciones para comunicarse con otro. Esto se traduce en buses no determinísticos, donde el número de ciclos de reloj ocupado en el intercambio de información no está asegurado.

Es de especial interés para este trabajo el bus AXI y sus variaciones, pues actualmente el mercado de SoCs que incluyen FPGAs y procesadores de propósito general está dominado por tecnologías de ARM, como la serie *Cyclone V* de Altera, [1], y *Zynq* de Xilinx, [2].

Las dos primeras variantes, AXI4 y AXI4-Lite, son buses mapeados a memoria y cuentan con elementos de infraestructura para la interconexión, permitiendo tener una jerarquía de elementos conectados a un conmutador en común. A diferencia de AXI4, AXI4-Lite no soporta transferencias de datos en ráfagas y está pensado para intercambiar información de control y estado. Es importante destacar que los conmutadores pueden tener internamente buffers para

las transacciones, dificultando la tarea de diseñar algunas aplicaciones sensibles a este tipo de parámetros. Otra variante, AXI4-Stream, [42], está pensado para aplicaciones que no necesitan de un mapeo a memoria y que pueden ocupar un gran ancho de banda. Entre sus características está el soporte para la multiplexación de varios canales de datos en uno físico, ahorrando recursos al interior del SoC. Está variante puede llevar también información adicional, como el tipo de datos que transmite e información de ruta, lo que baja la cantidad de información útil que lleva.

Debido a las características recién mencionadas, se utilizará un bus AXI mapeado a memoria para la comunicación en situaciones que requieran realizar co-procesamiento de datos entre un filtro y un procesador de propósito general. Además, los dispositivos que cuentan con soporte para este bus tienen transceptores incorporados, lo que permite alcanzar una mayor velocidad de comunicación entre ambos.

4.5. Elementos del modelo

La arquitectura específica de los elementos dependerá de la implementación a realizar. Razón por la cual no se pueden fijar muchos detalles desde el punto de vista de la arquitectura, pues las reglas para la implementación fueron definidas por el modelo, y las interfaces de comunicación fueron definidas en las secciones anteriores. Sin embargo, se pueden plantear algunos detalles útiles para las implementaciones:

- Un frame-buffer fuera del chip debe tener internamente una memoria caché compuesta por line-buffers para acelerar el acceso de los datos, pues usualmente los pixeles son accedidos de forma secuencial.
- Un elemento que modifique el reloj en el bus de video debe hacer uso de las primitivas disponibles en cada plataforma, como PLLs, pues no existe una forma universal de modificar el reloj en ningún lenguaje de programación de descripción de hardware.
- Algunos de los trabajos analizados, como [35], incorporan la capacidad de realizar cambios en tiempo de funcionamiento como activar o desactivar el comportamiento de un filtro o el paso de parámetros. Este tipo de información se puede pasar en forma de un canal de almacenamiento externo, en donde el emisor se encuentra fuera del sistema de procesamiento de video, y el receptor (filtro) define cuales son direcciones de memoria válidas y su significado.
- Para los drivers de entrada y salida se distinguen dos casos. El primero son los drivers

orientados a diseño y validación, como un puerto serie o Ethernet, en el que los datos deben pasar al sistema pixel a pixel, otorgando el mayor grado de control sobre el valor de la entrada y salida, y teniendo la posibilidad de añadir señales de depuración. El segundo caso lo conforman los drivers orientados a sistemas de producción, en el que los datos deben pasar en la forma que se maximice la utilización del canal con datos de video. Sin embargo, y teniendo en cuenta ambos casos, pueden existir varios drivers que ocupen el mismo canal con propósitos diferentes. Por ejemplo, puede existir un driver Ethernet para el diseño, que en cada frame agregue señales de depuración, y otro driver Ethernet orientado a sistemas de producción, en el que sólo se pasen datos de video.

Capítulo 5: Flujo de trabajo

Este capítulo presenta un análisis de las herramientas comúnmente usadas en la síntesis e integración para lenguajes de descripción de hardware. Finalmente, se detalla el flujo de diseño recomendado para la reconfiguración parcial de FPGAs Xilinx, que puede ser usado para reemplazar funcionalidades (filtros) en tiempo de funcionamiento.

5.1. Herramientas de desarrollo

Las nuevas tecnologías que se han aplicado a la industria de los semiconductores han provocado un incremento sustancial en la densidad de los circuitos integrados, causando que ahora se puedan integrar casi todas las funciones necesarias en un solo chip. El aumento ha traído algunos problemas en el diseño de los SoCs, pues ahora hay que manejar una gran cantidad de archivos de código fuente. Este problema aumenta en los diseños que usan FPGAs, pues cada fabricante ofrecía un entorno de programación con componentes propietarios que resaltaban las características de sus productos, y no permitían reutilizar los componentes en productos de otros fabricantes.

Debido a la situación anterior, hace algunos años se formó el consorcio SPIRIT (actualmente Accellera Systems Initiative), con diversos miembros como fabricantes de SoCs y creadores de herramientas de diseño. Su objetivo era definir normas para el intercambio de información en el diseño de SoCs, permitiendo interoperabilidad entre herramientas de diseño de distintos fabricantes, acelerando el tiempo de desarrollo y la disponibilidad de los productos en el mercado.

En el año 2009 fue estandarizada la primera versión de IP-XACT (IEEE 1685-2009, [43]), la que define un esquema en lenguaje de marcas extensible (del inglés *Extensible Markup Language*, XML) para describir el componente, y establece un API para ayudar en la integración con herramientas de diseño. En esta primera versión se introducen formas de definir componentes con sus respectivos puertos, registros, y parámetros entre otros, permitiendo además la descripción de buses de comunicación, interfaces, y dispositivos mapeados a memorias con espacios de direcciones. Cada componente puede albergar distintas vistas, así, se puede tener una vista de simulación, de síntesis para FPGAs, y de síntesis para ASICs. Adicionalmente, el API de IP-XACT tiene soporte para generadores de código, permitiendo alcanzar un mayor grado de

reutilización de los componentes. Estas capacidades logran la integración de casi cualquier tipo de componente en el diseño, como procesadores de propósito general mezclados con lógica.

En la actualidad ya son varias herramientas que soportan IP-XACT, entre ellas están Synplify y DesignWare de Synopsys, Vivado de Xilinx, y Platform Express de Mentor Graphics. Sin embargo, IP-XACT parece no cubrir todas las necesidades de la industria y posee ciertas limitaciones, algunas de ellas son descritas por ingenieros de ARM en [44]. En el documento se mencionan problemas a la hora de configurar los parámetros de un componente dentro de otro, interdependencia de parámetros, y existencia condicional de puertos; y se plantean algunas soluciones a ellos. Aún con los problemas recién mencionados, IP-XACT es un paso en la dirección correcta para solucionar la fragmentación de componentes e interoperabilidad entre las herramientas de diseño. Además, las limitaciones pueden ser disminuidas o eliminadas en las futuras versiones del estándar, como sucedió con la revisión más reciente en 2014 (IEEE 1685-2014, [45]).

Entre los entornos de diseño disponibles actualmente para IP-XACT, resalta Kactus2, [46], por ser una herramienta gratuita, de código abierto y altamente configurable. Kactus2 permite manejar una biblioteca de componentes descritos en IP-XACT, y además agrega extensiones que ayudan con algunas de las limitaciones anteriormente mencionadas, todo esto sin romper el estándar ya que está dentro de las extensiones posibles de IP-XACT. Además, se pueden desarrollar plugins, abriendo la posibilidad de integración con herramientas de los mayores fabricantes de la industria. Sin embargo, su principal debilidad es la carencia de un análisis profundo de archivos de código fuente lenguaje de descripción de hardware (del inglés *Hardware Description Language*, HDL), lo que reduce la velocidad del flujo de diseño.

Debido a que los fabricantes no entregan todas las especificaciones necesarias sobre sus FPGAs, el flujo de trabajo debe involucrar al menos una herramienta del fabricante. Este problema se ve agravado con los SoCs que incorporan un procesador, pues los entornos de diseño creados por los fabricantes a veces incluyen librerías propietarias y cuya redistribución está prohibida, ralentizando el tiempo que toma el diseño.

Por todo lo anterior, se considera que IP-XACT es una muy buena alternativa para resolver muchos problemas presentes en las herramientas existentes y el flujo de diseño actual. Sin embargo, al momento de la redacción de este informe, su uso no está lo suficientemente esparcido ni estandarizado, por ejemplo, no es soportado ni lo será en FPGAs Xilinx de la serie 6, chips en los cuales es posible implementar aplicaciones de procesamiento de video sin ningún problema. Debido a esto, las herramientas de integración recomendadas serán las del fabricante del dispositivo en que se vaya a implementar la aplicación, usando una biblioteca de elementos

altamente estandarizada y reutilizable, que implemente el modelo de cómputo descrito en los capítulos anteriores. Cada elemento de la biblioteca debe estar escrito en Verilog o VHDL, pues son los dos grandes lenguajes que tienen la mayor cobertura en las herramientas de desarrollo ofrecidas por los fabricantes.

5.1.1. Flujo de trabajo con el modelo

La siguiente secuencia de pasos representa el flujo de trabajo recomendado cuando se trabaja directamente con el modelo de cómputo y una implementación en un FPGA.

1. Diseño en software, a nivel de sistema. Uso de lenguajes de programación de propósito general para realizar simulaciones del algoritmo a implementar.
2. Diseño en software, con menos abstracciones. Utilizar lenguajes de propósito general, pero limitado a un subconjunto que pueda ser mapeado a una arquitectura hardware compatible con el modelo.
3. Diseño de la arquitectura de hardware. Realizar diagramas de bloque, especificando la función de cada uno, y cuidando que cumplan con las reglas del modelo.
4. Implementación de la arquitectura. Usar algún lenguaje de descripción de hardware para implementar los elementos de la arquitectura ya diseñada, usualmente Verilog o VHDL. Realizar además simulaciones para validar resultados.
5. Usar herramientas de síntesis de los fabricantes para completar la implementación.
6. Realizar la validación funcional. Comprobar resultados obtenidos entre las implementaciones de software y hardware del algoritmo. Volver a alguno de los pasos anteriores si las comparaciones no son satisfactorias.

5.1.2. Flujo de trabajo con el lenguaje de programación

La siguiente secuencia de pasos representa el flujo de trabajo recomendado cuando se usa el lenguaje de programación y una implementación en un FPGA.

1. Diseño en software, a nivel de sistema. Uso de lenguajes de programación de propósito general para realizar simulaciones del algoritmo a implementar.

2. Diseño en software, con lenguaje de programación DSL. Esto permite definir directamente una arquitectura de hardware.
3. Compilar el código fuente. Usando un compilador para el lenguaje de programación que genere código en Verilog o VHDL.
4. Usar herramientas de síntesis de los fabricantes para completar la implementación.
5. Realizar la validación funcional. Comprobar resultados obtenidos entre las implementaciones de software y hardware del algoritmo. Volver a alguno de los pasos anteriores si las comparaciones no son satisfactorias.

Se observa que el flujo de trabajo con el lenguaje de programación propuesto elimina dos pasos manuales y agrega uno automático, comparado con el flujo de trabajo directo con el modelo de cómputo.

5.2. Especificación del flujo de trabajo para reconfiguración parcial en FPGAs Xilinx

Una de las características únicas que poseen los FPGAs es la reconfiguración parcial, que a groso modo consiste en reconfigurar sólo una pequeña parte del FPGA. Esta característica permite, entre otras cosas, la reducción en los recursos totales usados debido a que se comparten entre las unidades reconfigurables. El proceso consiste en separar la lógica reconfigurable en dos conjuntos de áreas dentro del dispositivo. El primer conjunto es estático, y se configura sólo una vez al inicio del funcionamiento, ya que su configuración borra cualquier estado en el que se encuentre toda la lógica del FPGA. El segundo conjunto es reconfigurable, de forma individual para cada una de las áreas que lo componen, lo que le brinda una gran flexibilidad al poder reemplazar partes específicas del diseño sin afectar el funcionamiento del resto.

El proceso puede ser divido en dos grandes tareas, síntesis de archivos y configuración de la lógica, descritas a continuación.

5.2.1. Síntesis de los archivos de configuración

El procedimiento expuesto fue basado en su mayoría por [23], donde aparece en extenso la metodología.

En el diseño, cada una de las áreas reconfigurables se asocia con una instancia reconfigurable (instancia de un módulo), lo que permite identificarla inequívocamente. Además, cada instancia reconfigurable tiene dos o más variaciones (implementaciones) distintas, siendo sólo una de ellas la instancia reconfigurable por defecto.

La síntesis se realiza en varias etapas con un orden estricto, de la siguiente forma:

1. Definir las instancias reconfigurables y sus variaciones asociadas.
2. Realizar la síntesis lógica del diseño, usando cajas negras para marcar las instancias reconfigurables.
3. Por cada una de las variaciones, realizar la síntesis lógica.
4. Realizar el Place and Route del diseño, agregando las variaciones por defecto de las instancias reconfigurables.
5. Por cada una de las variaciones, cargar el resultado de 2 y 3, realizar la síntesis lógica, y finalmente el Place and Route.
6. Cargar el resultado de 4 y generar el archivo que contiene la configuración estática y las variaciones por defecto.
7. Por cada uno de los resultados de 5, generar el archivo de reconfiguración parcial.

A modo de resumen, en la entrada de esta tarea están todos los archivos de código fuente para sintetizar el proyecto, y a la salida se obtiene un archivo que incluye la configuración estática junto con las variaciones por defecto de todas las instancias reconfigurables. Además, por cada instancia reconfigurable se obtienen una cantidad de archivos equivalentes a sus variaciones, que corresponden a sus configuraciones parciales.

5.2.2. Configuración y posterior reconfiguración de la lógica

A la tarea normal de configuración del FPGA se le agrega una adicional, que es la reconfiguración dinámica del conjunto de áreas reconfigurables. Esta nueva tarea puede realizarse de diversas formas, pero en este trabajo se implementará para un FPGA de la serie Zynq de Xilinx. Como se vio anteriormente en [28], las formas que más acomodan a este tipo de dispositivo es la reconfiguración desde el procesador, y desde la lógica misma. En ambos casos, la tarea consiste

en leer desde algún tipo de almacenamiento los archivos de reconfiguración parciales y aplicarlos cuando corresponda.

Capítulo 6: Conclusiones

En este trabajo se presentó el diseño de un modelo de cómputo que facilita la descripción de sistemas de procesamiento para video infrarrojo. Para la creación del modelo, se definió un conjunto de características y requerimientos mínimos de la plataforma objetivo. Luego se analizó una multitud de trabajos, buscando y extrayendo características comunes entre ellos. Con ellas, se definió el modelo a través de la definición de elementos de cómputo altamente estandarizados y reglas estrictas sobre la interacción entre ellos, que deben cumplir las implementaciones que sean descritas usando el modelo. El modelo finaliza con la definición de un conjunto de elementos de procesamiento primitivos, derivados de los trabajos analizados, que pueden ser reutilizados fácilmente por cualquier implementación. Con el modelo ya definido, se presentaron descripciones que ejemplifican el uso del modelo en sistemas de procesamiento de video.

En el trabajo presentó además un lenguaje de programación de dominio específico para la descripción de sistemas de procesamiento de video, descrito formalmente usando Backus–Naur Form. Su sintaxis es una combinación de C y Verilog, rescatando las abstracciones de C y la granularidad de Verilog, y agregando también elementos propios para crear abstracciones de mayor nivel enfocadas en el procesamiento de video. El lenguaje tiene un modelo de ejecución basado en eventos, producidos en la entrada del sistema cuando llega un pixel del stream de video. También permite la creación y uso de eventos personalizados, que son los encargados de orquestar todas las acciones en el sistema. Con el lenguaje ya definido, se presentaron ejemplos simples y un ejemplo completo de sistema de procesamiento de video. Finalmente, la única característica del modelo que no fue implementada en el lenguaje, y que se puede agregar en una futura versión, es el trabajo en conjunto con un procesador de propósito general.

Se propuso además, una arquitectura base de hardware para el modelo de cómputo, definiendo detalles sobre eventos y control de flujo, permitiendo la detección de errores en sistemas que lo requieran. También se presentaron detalles sobre el bus usado para la transmisión del stream de video; un bus para realizar transacciones de acceso a memoria interna y externa; y la elección de un bus de propósito general para la comunicación con un elemento de procesamiento externo. Finalmente, se analizaron algunas herramientas de diseño usadas en la creación de hardware, y se propuso un flujo de trabajo para el diseño de sistemas de procesamiento de video que permite, en el caso de un FPGA, realizar reconfiguración parcial en tiempo de funcionamiento, intercambiando funcionalidades.

Se pudo comprobar que efectivamente hay características comunes entre los sistemas de procesamiento de video infrarrojo analizados, lo que permitió la definición del modelo de cómputo y la creación del lenguaje de programación, alcanzando un grado de abstracción satisfactorio. Sin embargo, se dejaron fuera del modelo varios elementos con funcionalidad similar, pero con implementaciones totalmente personalizadas, pues realizar una abstracción de ellas dañaría la efectividad de la implementación.

El modelo de cómputo propuesto resultó efectivo en describir directamente algunas aplicaciones de procesamiento de video. Sin embargo, otras deberán ser rediseñadas y adaptadas a la forma de procesamiento definida en el modelo, usualmente realizado por drivers de entrada y salida.

Definir un modelo de cómputo restrictivo fomenta la estandarización y reutilización de diseños, lo que resulta en implementaciones auto contenidas que pueden ser fácilmente intercambiables por otras que compartan las mismas interfaces. Esto permite acelerar los procesos de desarrollo e integración.

El lenguaje de programación propuesto permite describir sistemas de procesamiento de video de una forma fácil, sin necesidad de conocimientos avanzados sobre hardware, bajando la barrera de entrada para programadores interesados en este tipo de sistemas. La fácil descripción de los sistemas también contribuye a reducir los tiempos de desarrollo.

Para finalizar, se debe destacar que no se cumplieron todos los objetivos propuestos. Específicamente, no se realizó la implementación en hardware y comparación de las aplicaciones de prueba con distintos lenguajes de programación similares. Sin embargo, se presentó la definición de un lenguaje de programación que no estaba dentro de los objetivos del trabajo original. Por lo tanto, el primero de los trabajos futuros puede ser la realización de una validación extensa del modelo de cómputo y el lenguaje propuesto, comparando con distintos criterios como el uso de recursos lógicos y velocidad de procesamiento. Luego, se podrían desarrollar un conjunto de herramientas, partiendo por un generador de código que traduzca el código escrito en el lenguaje, a Verilog o VHDL para que posteriormente sea procesado por herramientas de síntesis de hardware. Una posibilidad más avanzada sería el desarrollo de un compilador, que entendiera y optimizara el código escrito por el usuario. Finalmente, se podría extender el lenguaje de programación para que soporte la inclusión de bloques que se ejecuten en un procesador de propósito general, generando un sistema de procesamiento heterogéneo.

Bibliografía

[1] Altera, "Cyclone V SoCs overview." [Online]. Available: https://www.altera.com/products/soc/portfolio/cyclone-v-soc/overview.tablet.html

[2] Xilinx, "Zynq-7000 All Programmable SoC." [Online]. Available: http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[3] B. Gao and D. Rees, "Asap: an asynchronous array processor for hardware-software co-processing and codesign," in *ASIC, 1996., 2nd International Conference on*, Oct 1996, pp. 151–154.

[4] O. Cheng, W. Abdulla, and Z. Salcic, "Hardware-software codesign of automatic speech recognition system for embedded real-time applications," *Industrial Electronics, IEEE Transactions on*, vol. 58, no. 3, pp. 850–859, March 2011.

[5] Xilinx, "Vivado High-Level Synthesis." [Online]. Available: http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[6] Altera, "Altera SDK for OpenCL - Overview." [Online]. Available: https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.tablet.html

[7] D. Scribner, K. Sarkady, M. Kruer, J. Caulfield, J. Hunt, M. Colbert, and M. Descour, "Adaptive retina-like preprocessing for imaging detector arrays," in *Neural Networks, 1993., IEEE International Conference on*, 1993, pp. 1955–1960 vol.3.

[8] J. Han, E. J. Pauwels, and P. De Zeeuw, "Visible and infrared image registration in man-made environments employing hybrid visual features," *Pattern Recogn. Lett.*, vol. 34, no. 1, pp. 42–51, jan 2013.

[9] A. Jammoussi, S. Ghribi, and D. Masmoudi, "Implementation of face recognition system in virtex ii pro platform," in *Signals, Circuits and Systems (SCS), 2009 3rd International Conference on*, Nov 2009, pp. 1–6.

[10] H. Ngo, R. Tompkins, J. Foytik, and V. Asari, "An area efficient modular architecture for real- time detection of multiple faces in video stream," in *Information, Communications Signal Processing, 2007 6th International Conference on*, Dec 2007, pp. 1–5.

[11] M. Vergara, A. Wolf, and M. Figueroa, "A texture-based architecture for face detection in IR images on an FPGA," *Proc. SPIE*, vol. 9249, pp. 92 490L–92 490L–12, 2014.

[12] M. Vollmer and K. Möllmann, *Infrared Thermal Imaging: Fundamentals, Research and Applications*. Wiley, 2010.

[13] S. N. Torres and M. M. Hayat, "Kalman filtering for adaptive nonuniformity correction in infrared focal-plane arrays," *J. Opt. Soc. Am. A*, vol. 20, no. 3, pp. 470–480, Mar 2003.

[14] W. Qian, Q. Chen, and G. Gu, "Space low-pass and temporal high-pass nonuniformity correction algorithm," *Optical Review*, vol. 17, no. 1, pp. 24–29, 2010.

[15] C. Zuo, Q. Chen, G. Gu, and X. Sui, "Scene-based nonuniformity correction algorithm based on interframe registration," *J. Opt. Soc. Am. A*, vol. 28, no. 6, pp. 1164–1176, Jun 2011.

[16] W. Isoz, T. Svensson, and I. Renhorn, "Nonuniformity correction of infrared focal plane arrays," *Proc. SPIE*, vol. 5783, pp. 949–960, 2005.

[17] B. M. Ratliff, J. S. Tyo, J. K. Boger, W. T. Black, D. L. Bowers, and M. P. Fetrow, "Dead pixel replacement in lwir microgrid polarimeters," *Opt. Express*, vol. 15, no. 12, pp. 7596–7609, Jun 2007.

[18] R. C. Gonzalez and R. E. Woods., *Digital Image Processing*, 3rd ed. Prentice-Hall, 2007.

[19] Z. Hocenski, I. Aleksi, and R. Mijakovic, "Ceramic tiles failure detection based on fpga image processing," in *Industrial Electronics, 2009. ISIE 2009. IEEE International Symposium on*, July 2009, pp. 2169–2174.

[20] 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

[21] 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.

[22] Altera, *Quartus II Handbook*, 13th ed. Altera, 2013, vol. 1: Design and Synthesis.

[23] Xilinx, *Partial Reconfiguration (UG947)*, Xilinx, April 2015, v2015.1.

[24] Altera, *Partial Reconfiguration IP Core (UG-PARTRECON)*, Altera, May 2015, 2015.05.04.

[25] A. Jacoby, D. Llamocca, R. Jordan, and G. Vera, "Proteus: An open source dynamically reconfigurable system-on-chip with applications to digital signal processing," in *Devices, Circuits and Systems (ICCDCS), 2014 International Caribbean Conference on*, April 2014, pp. 1–6.

[26] L. Maggiani, C. Salvadori, M. Petracca, P. Pagano, and R. Saletti, "Reconfigurable FPGA architecture for computer vision applications in smart camera networks," in *Distributed Smart Cameras (ICDSC), 2013 Seventh International Conference on*, Oct 2013, pp. 1–6.

[27] C. Kohn, *Partial Reconfiguration of a Hardware Accelerator on Zynq-7000 All Programmable SoC Devices (XAPP1159)*, Xilinx, January 2013, v1.0.

[28] K. Vipin and S. Fahmy, "ZyCAP: Efficient partial reconfiguration management on the Xilinx Zynq," *Embedded Systems Letters, IEEE*, vol. 6, no. 3, pp. 41–44, Sept 2014.

[29] C. Bourrasset, L. Maggiani, J. Serot, F. Berry, and P. Pagano, "DreamCAM: A FPGA-based platform for smart camera networks," in *Distributed Smart Cameras (ICDSC), 2013 Seventh International Conference on*, Oct 2013, pp. 1–2.

[30] C. Bourrasset, L. Maggianiy, J. Serot, F. Berry, and P. Pagano, "Distributed FPGA-based smart camera architecture for computer vision applications," in *Distributed Smart Cameras (ICDSC), 2013 Seventh International Conference on*, Oct 2013, pp. 1–2.

[31] C. Bourrasset, J. Serot, and F. Berry, "FPGA-based smart camera mote for pervasive wireless network," in *Distributed Smart Cameras (ICDSC), 2013 Seventh International Conference on*, Oct 2013, pp. 1–6.

[32] B. Feng, *Real Time Video Engine 2.1 Implementation in Xilinx Zynq-7000 All Programmable SoCs (XAPP1095)*, Xilinx, January 2014, v1.0.

[33] M. Bergeron, S. Elzinga, G. Szedo, G. Jewett, and T. Hill, *1080p60 Camera Image Processing Reference Design (XAPP794)*, Xilinx, December 2013, v1.3.

[34] R. Redlich, M. Figueroa, S. N. Torres, and J. E. Pezoa, "Embedded nonuniformity correction in infrared focal plane arrays using the Constant Range algorithm," *Infrared Physics & Technology*, vol. 69, no. 0, pp. 164 – 173, 2015.

[35] A. Wolf, R. Redlich, M. Figueroa, and J. E. Pezoa, "On-line nonuniformity and temperature compensation of uncooled IRFPAs using embedded digital hardware," *Proc. SPIE*, vol. 8868, pp. 88 680H–88 680H–12, 2013.

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

[37] J. E. Soto and M. Figueroa, "An embedded face-classification system for infrared images on an FPGA," *Proc. SPIE*, vol. 9249, pp. 92 490K–92 490K–12, 2014.

[38] J. Cárdenas, M. Figueroa, and J. E. Pezoa, "A custom hardware classifier for bruised apple detection in hyperspectral images," pp. 95 992K–95 992K–11, 2015.

[39] W. E. Valenzuela, M. Figueroa, J. E. Pezoa, and P. Meza, "A digital architecture for striping noise compensation in push-broom hyperspectral cameras," pp. 95 992H–95 992H–10, 2015.

[40] *WISHBONE System-on-Chip (SoC) Interconnection. Architecturefor Portable IP Cores*, OpenCores, 2010, revision B.4. [Online]. Available: http://cdn.opencores.org/downloads/wbspec_b4.pdf

[41] *AMBA AXI and ACE Protocol Specification. AXI3, AXI4, and AXI4-Lite. ACE and ACE-Lite*, ARM, Oct 2011, issue D. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ihi0022d/index.html

[42] *AMBA 4 AXI4-Stream Protocol Specification*, ARM, Mar 2010, issue A. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ihi0051a/index.html

[43] "IEEE Standard for IP-XACT, standard structure for packaging, integrating, and reusing IP within tool flows," *IEEE Std 1685-2009*, pp. 1–374, Feb 2010.

[44] D. Murray and S. Rance, "Solving next generation IP configurability," ARM, White Paper, 2014. [Online]. Available: http://www.arm.com/files/pdf/New_Whitepaper_Layout_Solving_Next_Generation_IP_Configurability.pdf

[45] "IEEE Standard for IP-XACT, standard structure for packaging, integrating, and reusing IP within tool flows," *IEEE Std 1685-2014 (Revision of IEEE Std 1685-2009)*, pp. 1–510, Sept 2014.

[46] A. Kamppi, L. Matilainen, J. Maatta, E. Salminen, T. Hamalainen, and M. Hannikainen, "Kactus2: Environment for embedded product development using IP-XACT and MCAPI," in *Digital System Design (DSD), 2011 14th Euromicro Conference on*, Aug 2011, pp. 262–265.

Anexo A: Código aplicación de estabilización de video

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

```
1  // Version: 2016-10-20
2  // Version: year-month-day
3  double_buffer( memory_port front, memory_port back, event swap,
4                 parameter buff_width, parameter buff_len ) {
5
6      memory(buff_width,buff_len,0) buff1;
7      memory(buff_width,buff_len,0) buff2;
8      unumber(1,0) swap_state = 0;
9
10     on_event( swap ) {
11         swap_state = ~swap_state;
12
13         if( swap_state == 0 ) {
14             front = buff1.port[0];
15             back  = buff2.port[0];
16         }
17         else {
18             front = buff2.port[0];
19             back  = buff1.port[0];
20         }
21     }
22 }
23
24 x_proj( memory_port ch1_new, memory_port ch2_new, memory_port ch2_old,
25         event swap ) {
26
27     memory_port new_muxed;
28     memory_port_mux( new_muxed, ch1_new, ch2_new );
29
30     double_buffer( new_muxed, ch2_old, swap, 16, 640 );
31 }
32
33 y_proj( memory_port ch1_new, memory_port ch2_new, memory_port ch2_old,
34         event swap ) {
```

```
35
36      memory_port new_muxed = port_mux( ch1_new, ch2_new );
37
38      double_buffer( new_muxed, ch2_old, swap, 18, 480 );
39  }
40
41
42  calc_proj( pix_stream in_pix, memory_port x_proj_mem, memory_port y_proj_mem,
43              event proj_ready ) {
44
45      unumber(16,0) x_count = 0;
46      unumber(16,0) y_count = 0;
47      unumber(16,0) proj_x_val;
48      unumber(18,0) proj_y_accum;
49      //
50      on_event( in_pix.new_data ) {
51          // New frame
52          if( x_count == 0 && y_count == 0 ) {
53              proj_y_accum = 0;
54          }
55          // New line
56          if( x_count == 0 ) {
57              proj_y_accum = proj_y_accum +
58                             cast( unumber(18,0), y_proj_mem(y_count,0) );
59              y_proj_mem(y_count,0) = cast( raw_bits(18), proj_y_accum );
60              proj_y_accum = 0;
61              ++y_count;
62          }
63          else {
64              proj_x_val = cast( unumber(16,0), x_proj_mem(x_count,0) );
65              proj_x_val = proj_x_val
66                           cast( unumber(8,0), in_pix.data[0] );
67              x_proj_mem(x_count,0) = cast( raw_bits(18), proj_x_val );
68              proj_y_accum = proj_y_accum +
69                             cast( unumber(8,0), in_pix.data[0] );
70              ++x_count;
71          }
72          // End frame
73          if( x_count == in_pix.res_h && y_count == in_pix.res_v ) {
74              set_event( proj_ready );
75          }
76      }
77  }
```

```
78
79  motion_estimation( memory_port proj_new, memory_port proj_old,
80                     number(6,0) desp_found,
81                     event start, event ready,
82                     parameter MAX_D, parameter search_len ) {
83
84      unumber(27,0) min_SAD;
85      //
86      on_event( start ) {
87          min_SAD = MAX( unumber(27,0) );
88          unumber(8,0) val_new;
89          unumber(8,0) val_old;
90          unumber(8,0) val_abs;
91          desp_found  = 0;
92          unumber(16,0) p_idx;
93          unumber(6,0) test_desp;
94          for( test_desp = -MAX_D; test_desp < MAX_D; ++test_desp ) {
95              curr_SAD = 0;
96              for( p_idx = MAX_D; p_idx < search_len-MAX_D; ++p_idx ) {
97                  val_new = cast( unumber(8,0), proj_new(p_idx+test_desp,0) );
98                  val_old = cast( unumber(8,0), proj_old(p_idx,0) );
99                  //
100                 if( val_new > val_old )
101                     val_abs = val_new - val_old;
102                 else
103                     val_abs = val_old - val_new;
104                 //
105                 curr_SAD = curr_SAD + val_abs;
106             }
107             //
108             if( curr_SAD < min_SAD ) {
109                 min_SAD     = curr_SAD;
110                 desp_found = test_desp;
111             }
112         }
113         //
114         set_event( ready )
115     }
116 }
117
118 // Simple MVI implementation
119 motion_filter( number(6,0) desp_in, number(6,0) desp_filt,
120                event start, event ready, parameter MAX_D ) {
```

```
121
122     unumber(1,2) delta = unum_approx( 0.5, 1, 2 );
123     number(8,5) fmv, fmv_prev, fmv_round;
124
125     on_event( start ) {
126         // -- Filter
127         fmv = delta*fmv_prev + desp_in;
128         fmv_prev = fmv;
129         // -- Limit
130         if( fmv > MAX_D )
131             fmv = MAX_D;
132         else if( fmv < -MAX_D )
133             fmv = -MAX_D;
134         // -- Round
135         fmv_round = unum_approx( fmv, 6, 0 );
136         //
137         desp_filt = cast( number(6,0), fmv_round );
138         set_event( ready );
139     }
140 }
141
142 motion_correction( number(6,0) corr_x, event corr_x_ready,
143                    number(6,0) corr_y, event corr_y_ready,
144                    memory_port fr_buff_mem,
145                    pix_stream out_pix, parameter MAX_D ) {
146
147     unumber(16,0) x_out;
148     unumber(16,0) y_out;
149     unumber(16,0) x_corr;
150     unumber(16,0) y_corr;
151
152     on_event( corr_x_ready and corr_y_ready ) {
153         for( y_out = 0; y_out < 480; ++y_out ) {
154             for( x_out = 0; x_out < 640; ++x_out ) {
155                 if( (x_out < MAX_D) || (x_out > 640-MAX_D) )
156                     out_pix.data[0] = cast( raw_bits(8), 0 );
157                 else if( (y_out < MAX_D) || (y_out > 480-MAX_D) )
158                     out_pix.data[0] = cast( raw_bits(8), 0 );
159                 else {
160                     // Calculate the displaced coordinates
161                     x_corr = x_out - corr_x;
162                     y_corr = y_out - corr_y;
163                     out_pix.data[0] =
```

```
164                         cast( raw_bits(8), fr_buff_mem(x_corr,y_corr) );
165                 }
166                 set_event( out_pix.new_data );
167             }
168         }
169     }
170 }
171
172 filter top( pix_stream in_pix, pix_stream out_pix ) {
173     // Set (require) input format
174     in_pix.set_format( res_h=640, res_v=480, depth={8} );
175
176     // Set output format
177     out_pix.copy_format( in_pix );
178
179     // Storage
180     memory( 8, 640, 480 ) frame_buffer1;
181     memory_port fb1_port1 = frame_buffer1.port[0];
182     memory_port fb1_port2 = frame_buffer1.port[1];
183     //
184     memory_port x_proj_ch1_new, x_proj_ch2_new, x_proj_ch2_old;
185     memory_port y_proj_ch1_new, y_proj_ch2_new, y_proj_ch2_old;
186     event swap_proj;
187     x_proj( x_proj_ch1_new, x_proj_ch2_new, x_proj_ch2_old, swap_proj );
188     y_proj( y_proj_ch1_new, y_proj_ch2_new, y_proj_ch2_old, swap_proj );
189
190     // --- Save video to frame-buffer ---
191     event save_done;
192     hwt_pix_to_memory( in_pix, fb1_port1, save_done );
193
194
195     // --- Pipeline ---
196     // 1: Calc proj
197     event proj_ready;
198     calc_proj( in_pix, x_proj_ch1_new, y_proj_ch1_new, proj_ready );
199     // 2: Motion estimation
200     event x_estim_ready, y_estim_ready;
201     // - Connect swap projections event -
202     swap_proj = x_estim_ready;
203     number(6,0) desp_x, desp_y;
204     motion_estimation( x_proj_ch2_new, x_proj_ch2_old, desp_x,
205                        proj_ready, x_estim_ready, 31, 640 );
206     motion_estimation( y_proj_ch2_new, y_proj_ch2_old, desp_y,
```

```
207                         proj_ready, y_estim_ready, 31, 480 );
208     // 3: Motion filtering
209     event x_filter_ready, y_filter_ready;
210     number(6,0) desp_x_filt, desp_y_filt;
211     motion_filter( desp_x, desp_x_filt, x_estim_ready, x_filter_ready, 31 );
212     motion_filter( desp_y, desp_y_filt, y_estim_ready, y_filter_ready, 31 );
213     // 4: Motion correction
214     motion_correction( desp_x_filt, x_filter_ready, desp_y_filt, y_filter_ready,
215                         fb1_port2, out_pix, 31 );
216 }
```

Anexo B: Código aplicación de estabilización térmica de video infrarrojo

El código fuente completo de la aplicación se presenta a continuación. Es importante destacar que si bien el código representa la arquitectura del trabajo, el tamaño en bits de las variables se fijó de forma aproximada, pues no se contaba con información detallada en la arquitectura del trabajo original. Teniendo esto en cuenta, sólo bastaría con ajustar los tamaños de las variables para que el código represente la arquitectura definida en [35].

```
1  // Version: 2016-10-20
2  correction( pix_stream in_pix, pix_stream out_pix, memory_port cal_data,
3              number(14,0) f0_par, number(14,0) u_par ) {
4      //
5      unumber(10,0)  h_count = 0;
6      unumber(10,0)  v_count = 0;
7      raw_bits(32,0) s_data;
8      number(16,0)   Sij_1, Sij_2;
9      unumber(14,0)  Yij_div_Sij_0;
10     unumber(14,0)  Xij;
11     //
12     on_event( in_pix.new_data ) {
13         // Get parameters from memory
14         s_data = cal_data( h_count, v_count );
15         Sij_1  = cast( number(16,0), s_data[15:0] );
16         Sij_2  = cast( number(16,0), s_data[31:16] );
17         // Get pixel value
18         Yij_div_Sij_0 = cast( unumber(14,0), in_pix.data[0] );
19         // Perform correction
20         Xij = Yij_div_Sij_0 + Sij_1*f0_par + Sij_1*u_par;
21         // Increment or reset counters
22         if( (h_count == in_pix.res_h-1) && (v_count == in_pix.res_v-1) ) {
23             h_count = 0;
24             v_count = 0;
25         }
26         else {
27             h_count = h_count + 1;
28             v_count = v_count + 1;
29         }
```

```
30          // Output
31          out_pix.data[0] = cast( raw_bits(14,0), Xij );
32          set_event( out_pix.new_data );
33      }
34  }
35
36  // (pixel value) - (expected value)
37  filter exp_value( pix_stream in_pix, pix_stream out_pix ) {
38      // Set output format
39      in_pix.set_format( res_h=in_pix.res_h, res_v=in_pix.res_v, depth={14} );
40      //
41      number(24,0) accum;
42      on_event( in_pix.new_data ) {
43          accum = 0;
44          accum = accum + cast( unumber(14,0), in_pix.data[0] );
45          accum = accum + cast( unumber(14,0), in_pix.data[1] );
46          accum = accum + cast( unumber(14,0), in_pix.data[2] );
47          accum = accum + cast( unumber(14,0), in_pix.data[3] );
48          accum = accum + cast( unumber(14,0), in_pix.data[5] );
49          accum = accum + cast( unumber(14,0), in_pix.data[6] );
50          accum = accum + cast( unumber(14,0), in_pix.data[7] );
51          accum = accum + cast( unumber(14,0), in_pix.data[8] );
52          accum = 256*cast( unumber(14,0), in_pix.data[4] ) - accum;
53          accum = accum / 9;
54          out_pix.data[0] = cast( raw_bits(14), accum );
55          set_event( out_pix.new_data );
56      }
57  }
58
59  par_update( pix_stream in_pix, memory_port cal_data, pix_stream out_pix,
60              number(14,0) f0_par, number(14,0) u_par
61              parameter learn_par1, parameter learn_par2 ) {
62      //
63      unumber(10,0)  h_count = 0;
64      unumber(10,0)  v_count = 0;
65      raw_bits(32,0) s_data;
66      number(16,0)   Sij_1, Sij_2;
67      unumber(14,0)  exp_val_err;
68      number(14,0)   dCF_df0 = 0;
69      number(14,0)   dCF_du  = 0;
70      number(34,0)   dCF_df0_accum = 0;
71      number(34,0)   dCF_du_accum  = 0;
72      //
```

```
73      on_event( in_pix.new_data ) {
74          // Get parameters from memory
75          s_data = cal_data( h_count, v_count );
76          Sij_1  = cast( number(16,0), s_data[15:0] );
77          Sij_2  = cast( number(16,0), s_data[31:16] );
78          //
79          exp_val_err = cast( unumber(14,0), in_pix.data[0] );
80          // First step
81          dCF_df0_accum = dCF_df0_accum + Sij_1*exp_val_err;
82          dCF_du_accum  = dCF_du_accum + Sij_2*exp_val_err;
83          // Update parameters
84          f0_par = f0_par - learn_par2*dCF_df0;
85          u_par  = u_par - learn_par1*dCF_du;
86          // End of frame
87          if( (h_count == in_pix.res_h-1) && (v_count == in_pix.res_v-1) ) {
88              h_count = 0;
89              v_count = 0;
90              //
91              dCF_df0 = dCF_df0_accum/1024; // Right bit-shift
92              dCF_du  = dCF_du_accum/1024; // Right bit-shift
93              dCF_df0_accum = 0;
94              dCF_du_accum  = 0;
95          }
96          else {
97              h_count = h_count + 1;
98              v_count = v_count + 1;
99          }
100     }
101 }
102
103 filter top( pix_stream in_pix, pix_stream out_pix ) {
104     // Set (require) input format
105     in_pix.set_format( res_h=640, res_v=512, depth={14} );
106
107     // Set output format
108     out_pix.copy_format( in_pix );
109
110     // Storage
111     memory( 32, 640, 512 ) frame_buffer1;
112     memory_port fb1_port1 = frame_buffer1.port[0];
113     memory_port fb1_port2 = frame_buffer1.port[1];
114
115     // --- Processing ---
```

```
116     number(14,0) f0_par = 0, number(14,0) u_par = 0;
117     correction( in_pix, out_pix, fb1_port1, f0_par, u_par );
118     //
119     // Slide window: 3x3; Fill border with last pixel
120     pix_stream pix_win1;
121     flt_slide_window( out_pix, pix_win1, 3, 1 );
122     //
123     pix_stream pix_exp_val;
124     exp_value( pix_win1, pix_exp_val );
125     //
126     par_update( pix_exp_val, fb1_port2, f0_par, u_par,
127                 num_approx(0.01,2,14), num_approx(0.001,2,14) );
128 }
```