Proyecto Final IPD-432

Andrew Morrison Torres
201203028-5

I. Introducción y Contexto

I-A. Descripción del proyecto

Este proyecto busca obtener la radiación emitida por una llama. Tener información sobre la radiación de la llama a una velocidad comparable o más rápida que las dinámicas de esta permitiría un control rápido y una combustión más eficiente. Esto se pretende lograr aplicando una matriz de transformación a una imagen RGB, para obtener un espectro en cada pixel de la imagen. Luego este espectro se integra y se escala para obtener la radiación final.

La idea viene de "Combustion Diagnostics by Calibrated Radiation Sensing and Spectral Estimation"[1] en el cual se detalla el algoritmo. El foco del paper está en validar el resultado numérico del algoritmo. En este proyecto se buscará acelerar este algoritmo y replicar los resultados.

El procesamiento de cada pixel no depende de los píxeles a su alrededor. Esto hace que se pueda procesar en linea, por lo que parece sensato implementar el algoritmo en una FPGA. El procesamiento en la FPGA debe hacerse con números en punto fijo para aprovechar las DSPs por lo que se debe tener en cuenta una pérdida de precisión. También se debe tener en cuenta que las DSP no puede trabajar con números demasiado largos, en este caso utilizando la Nexys Video que contiene DSPs modelo DSP48E1 que permite multiplicar largos de 25 por 18. Pueden haber multiplicaciones más grandes pero utilizarían más DSPs y el timing se complicaría.

Se asumirá que las imágenes vienen pre-procesadas y no es necesario identificar que es llama y que no es. No asumir esto complicaría demasiado el proyecto y además esto se hace bastante rápido en una CPU o GPU. El cuello de botella del procesamiento se encuentra en la gran cantidad de multiplicaciones que hay que hacer para calcular el hiper-espectro por lo que este es el proceso que vale la pena acelerar.

I-B. Idea Original

En la figura 1 se puede ver un diagrama de bloques de lo propuesto a hacer en este proyecto. La idea era guardar en la RAM la imagen original y el espectro calculado, y mostrar en una pantalla la imagen original y alguna capa del hiper-espectro. Claramente esto fue muy ambicioso.

Los bloques donde una implementación en FPGA puede ser más rápida que una CPU o GPU son el multiplicador de matrices, interfaz RAM y las sumatorias que calculan la radiación puntual y global. Todavía no se tiene claro cuál es la mejor forma de detectar la llama por lo que sería poco sensato implementar en FPGA en este momento.

La primera dificultad fue utilizar la RAM. El IP que viene con Vivado no es fácil de utilizar como una block RAM, tiene muchas más opciones, parámetros y una documentación muy extensa.

Luego se encontró un diseño de ejemplo que estaba probado para la Nexys Video el cual dio muchos problemas para que funcione. En Vivado 2017 no corría pero si en 2016.4, después de mucho esfuerzo en tratar de arreglar el ejemplo para que funcione en versiones nuevas de Vivado se decidió trabajar en la versión más antigua.

Otro aspecto que no se tomó en cuenta al principio del proyecto fue el tiempo que hay que dedicar para leer y estudiar la documentación de nuevos módulos/IPs/protocolos que se van a ocupar. Ejemplo son los módulos HDMI de entrada y de salida que requieren que el reloj provenga de un PLL, esto se ve solo en la documentación del módulo. También resulto muy útil dedicar tiempo para aprender como funciona AXI ya que muchos módulos la usan, en particular AXI stream para los multiplicadores y la transición de video.

Al final no quedó tiempo para tener una pantalla conectada a la FPGA. Esto requeriría un cruce de dominio de relojes, diseñar una interfaz de usuario, y lo más importante, otra instancia de un VDMA la cual requiere ser configurada.

Básicamente, implementar un ecosistema para el IP que se pretendía diseñar fue mucho más trabajo del que se tenia planeado al principio.

Finalmente se decidió no utilizar una pantalla para mostrar resultados y no guardar el hiper-espectro en la RAM.

II. Calculo Matemático

En la ecuación (1) se obtiene el espectro $\hat{E}$ a partir de la multiplicación de la matriz T y el pixel $\rho(u,k)$. En la ecuación (2) se obtiene la radiación en el pixel, sin calibrar. Finalmente en la ecuación (3) se calibra y se suman todas las radiaciones locales para obtener la radiación de toda la imagen.

$$\hat{E}(\lambda,u,k) = T\rho(u,k) \tag{1}$$

$$Rad_l(k,u) = \int_{\lambda_1}^{\lambda_{1024}} \hat{E}(\lambda,u,k), d\lambda \tag{2}$$

$$Rad_g(k) = \frac{A_f A_p}{R^2} \sum_{u=1}^{N} Rad_l(k,u) \tag{3}$$

$\rho(u,k)$ vector RGB del pixel u en el instante k.
T matriz de transformación de tamaño 1024×3.
$\hat{E}$ es el espectro estimado.
N es el numero de pixeles de la imagen.
A_f es el área de la llama en cm^2.
A_p es el área del pixel en μm^2.
R es la distancia entre la llama y la cámara, en cm.

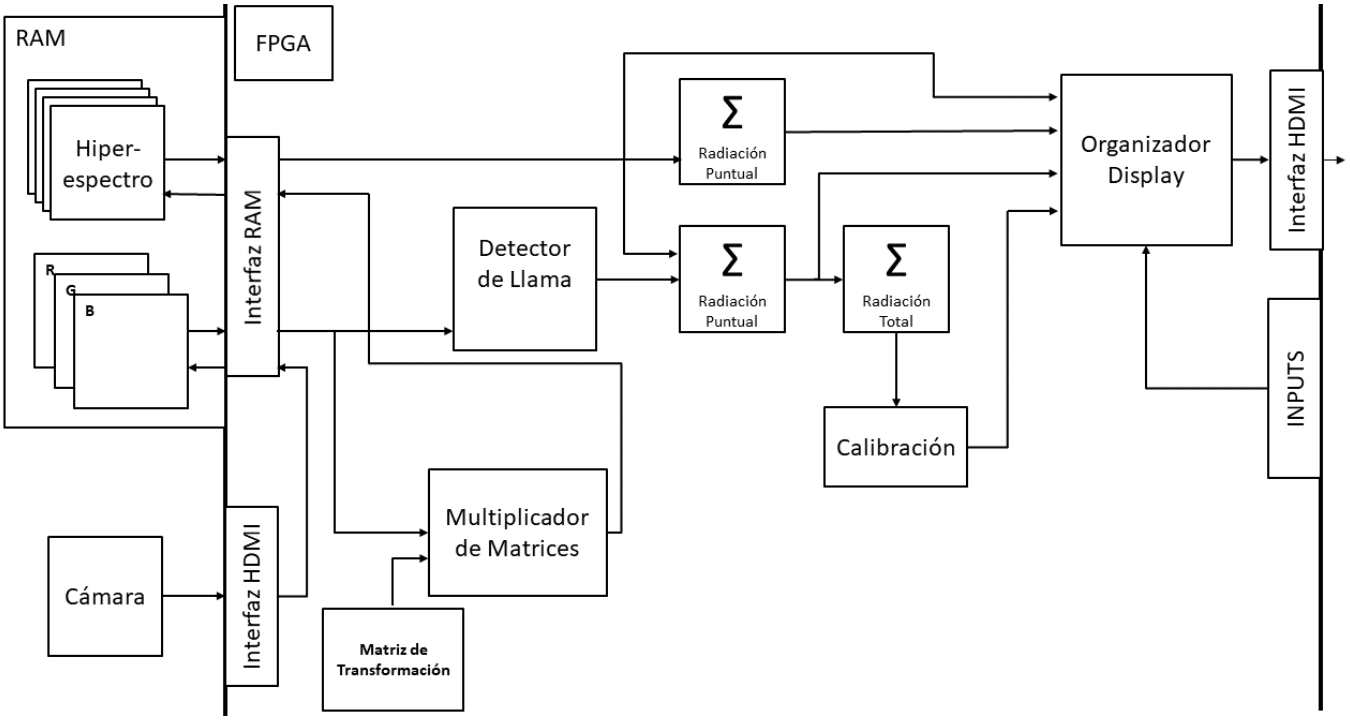

Figura 1. Diagrama de idea original al comienzo del proyecto.

Para implementar la ecuación (2) se aplica una integración trapezoidal:

$$\int_{\lambda_1}^{\lambda_{1024}} \hat{E}(\lambda,u,k)\,d\lambda \approx \sum_{\lambda=1}^{\lambda=1023} \frac{\hat{E}(\lambda_{k+1},u,k)+\hat{E}(\lambda_k,u,k)}{2}(\lambda_{k+1}-\lambda_k) \quad (4)$$

En este caso, las variaciones $\Delta\lambda_k$ no son constantes por lo tanto no se pueden factorizar. Para hacer un uso más eficiente de los recursos en una FPGA se requiere simplificar las ecuaciones para bajar el número de multiplicaciones que se tienen que hacer y además reducir latencia. En la ecuación (5) se omiten los parámetros u y k en $\hat{E}$ que se entienda mejor.

$$Rad_l(k,u) \approx \frac{1}{2}\sum_{\lambda=1}^{\lambda=1023}(\hat{E}(\lambda_{k+1},u,k)+\hat{E}(\lambda_k,u,k))(\lambda_{k+1}-\lambda_k) = \frac{1}{2}\sum_{\lambda=1}^{\lambda=1023}\hat{E}(\lambda_{k+1})\lambda_{k+1}-\hat{E}(\lambda_k)\lambda_k+\hat{E}(\lambda_k)\lambda_{k+1}-\hat{E}(\lambda_{k+1})\lambda_k = \frac{1}{2}\Bigg(\hat{E}(\lambda_{1024})\lambda_{1024}-\hat{E}(\lambda_1)\lambda_1+\sum_{\lambda=1}^{\lambda=1023}\hat{E}(\lambda_k)\lambda_{k+1}-\hat{E}(\lambda_{k+1})\lambda_k\Bigg) \quad (5)$$

Teniendo en cuenta que la radiación proviene de (1) se puede ver que en (5) hay multiplicaciones entre la fila k de T, el vector ρ y una constante λ. A continuación se definen dos matrices que simplifican la ecuación, donde la notación para indexar las matrices es la misma que en Matlab.

$$Q1 = [T(1:1023,:).*\lambda(2:1024); T(1024,:).*\lambda(1024)] \quad (6)$$

$$Q2 = [T(2:1024,:).*\lambda(1:1023); T(1,:).*\lambda(1)] \quad (7)$$

Así se puede reescribir la ecuación (5):

$$\frac{1}{2}\sum_{\lambda=1}^{\lambda=1024} Q1\rho(u,k) - Q2\rho(u,k) \quad (8)$$

Lamentablemente, la ecuación (8) fue implementada en hardware antes de que fuera analizada a profundidad. El resultado es correcto pero las operaciones se pueden simplificar aun más. $Q1$ y $Q2$ se pueden juntar en otra matriz y así ahorrar la mitad de las multiplicaciones. No solo eso, sino que después de analizar nuevamente las ecuaciones resulta evidente que en la ecuación (2) el pixel no depende de la variable de integración por lo tanto se puede sacar como constante. La integral de las columnas de T se puede hacer offline.

$$Rad_l(k,u) = \int_{\lambda_1}^{\lambda_{1024}} T(\lambda)d\lambda\ \rho(u,k) \quad (9)$$

Con B un vector de dimensiones 1×3 es definido como la integral de las columnas de T.

$$B = \int_{\lambda_1}^{\lambda_2} T d\lambda \quad (10)$$

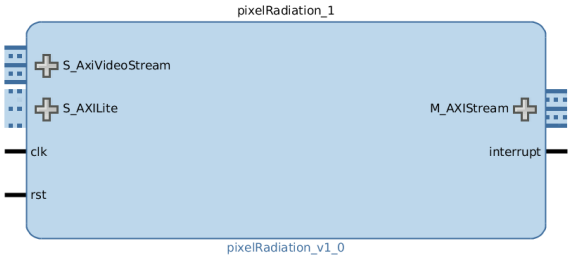

Figura 2. IP diseñada en este proyecto.

$$Rad_l(k,u) = B \cdot \rho(u,k) \tag{11}$$

La ecuación (3) puede ser reescrita como:

$$Rad_g(k,u) = \frac{A_f \cdot A_p}{R^2} \cdot B \cdot \sum_{u=1}^{N} \rho(k,u) \tag{12}$$

Así el problema se reduce a un acumulador de pixeles, mucho más fácil de implementar en FPGA y probablemente haga que no sea necesario utilizar una FPGA para este problema.

Lo que se presenta en este proyecto es lo obtenido con la ecuación (8).

III. Descripción de Módulos

Se creó un ip llamado pixelRadiation (figura 2). Se utilizaron interfaces AXI para que sea compatible con otros IPs y no reinventar la rueda.

A continuación se describen las entradas y salidas del IP:

- $S\_AXIVideoStream$ es la interfaz AXI video stream de 24 bits por donde entran las imágenes a procesar en formato RGB (8 bits por color).
- $S\_AXILite$ es la interfaz AXI Lite de 32 bits por donde se puede acceder a los registros del IP.
- $M\_AXIStream$ es una interfaz AXI Stream de 64 bits por la cual sale el resultado del cálculo. Fue implementada para tener otra opción de leer los resultados.
- $interrupt$ es una interrupción que se levanta por un ciclo indicando que hay un resultado listo, en los registros, para leer.
- clk y rst son las señales de reloj y reset que son compartidas por todas las interfaces AXI del IP.

Los parámetros son los siguientes:

- $BASEADDR$: Dirección de la interfaz AXI Lite.
- $NPIXELS$: Número de píxeles en la imagen a procesar.
- $BITS$: Número de bits por canal de color.
- $STEPS$: Número de pasos en los que se va a multiplicar.
- $LVECTOR$: Largo del vector que se multiplica en cada paso.
- $QWIDTH$: Ancho en bits de cada elemento de las matrices Q1 y Q2.

En la figura 3 se puede ver un bosquejo general de cómo están relacionados los bloques dentro del IP.

A continuación se describe el funcionamiento de los módulos (el primer nombre corresponde a la instancia del módulo en este proyecto, el entre paréntesis corresponde al nombre del módulo):

- $pixelRadiationBrain$ ($pixelRadiationStateMachine$): máquina de estado que controla la carga de datos, indica cuándo puede recibir otro pixel, guarda en un registro el pixel con el que se va a trabajar, controla los buffer circulares y los shift registers.
- $registers$ ($pixelRadiation\_regs$): unidad de registros que se pueden leer y escribir por interfaz AXI Lite. Este módulo fue generado en $airhdl.com$. La documentacion de estos registros se encuentra en el archivo $pixelRadiationRegistersQ\_regs.html$.
- $multiplicator1$ y $multiplicator2$ ($addmultiply$): este módulo multiplica se encarga de multiplicar una matriz por el vector RGB. La matriz viene del buffer circular y el vector viene de la máquina de estados. El resultado es un vector de largo $LVECTOR$. Utiliza 3 DPS en cascada para cada elemento del vector de salida.
- $adderTree1$ y $adderTree2$ ($addertree\_sec$): suman todos los elementos de los vectores de salida de los multiplicadores.
- $acumulator$ ($acumulator$): va sumando la resta de las salidas de los $adderTree$. Cuando junta $STEPS \cdot NPIXELS$ resultados, la suma corresponde a la radiación sin calibrar.
- $fix2FloatAccumulator$ ($fix2float$): pasa el resultado de representación en punto fijo a punto flotante. Sin embargo, lo trata como un número entero, falta correr la coma. El corrimiento de la coma se hace con una multiplicación por un factor 2^{-n} en $precalibrator$.
- $nonZeroPixelCounter$ ($nonZeroCounter$): cuenta los píxeles que son distintos de 0,0,0 ya que estos corresponden a píxeles que forman parte de la llama. El resultado está en punto flotante.
- $preCalibrator$ ($floatMultiplicator$): multiplica el número de píxeles distintos de cero con la constante que fue cargada en los registros. La constante engloba distintos parámetros de la cámara, su distancia a la llama y el factor para convertir un número punto fijo en flotante.
- $calibrator$ ($floatMultiplicator$): multiplica la salida de $preCalibrator$ con la salida de $fix2FloatAccumulator$.
- $calibratedInterrupt$ ($interruptSource$): una vez que recibe un resultado válido genera una interrupción, guarda en los registros $upperresult$ y $lowerresult$ de $registers$ y no los vuelve a actualizar hasta que sean leídos. Así se garantiza que los datos leídos correspondan a los que estaban en el momento que se hizo la interrupción. Los resultados que se obtengan y no se puedan guardar son descartados.

El módulo $ShiftRegistersandCircularBuffers$ engloba a varias instancias de buffers y shift registers. Básicamente por cada elemento de color de Q1 y Q2 hay un buffer y un shift register.

- $shiftRegister$: un shift register para cada ir juntando

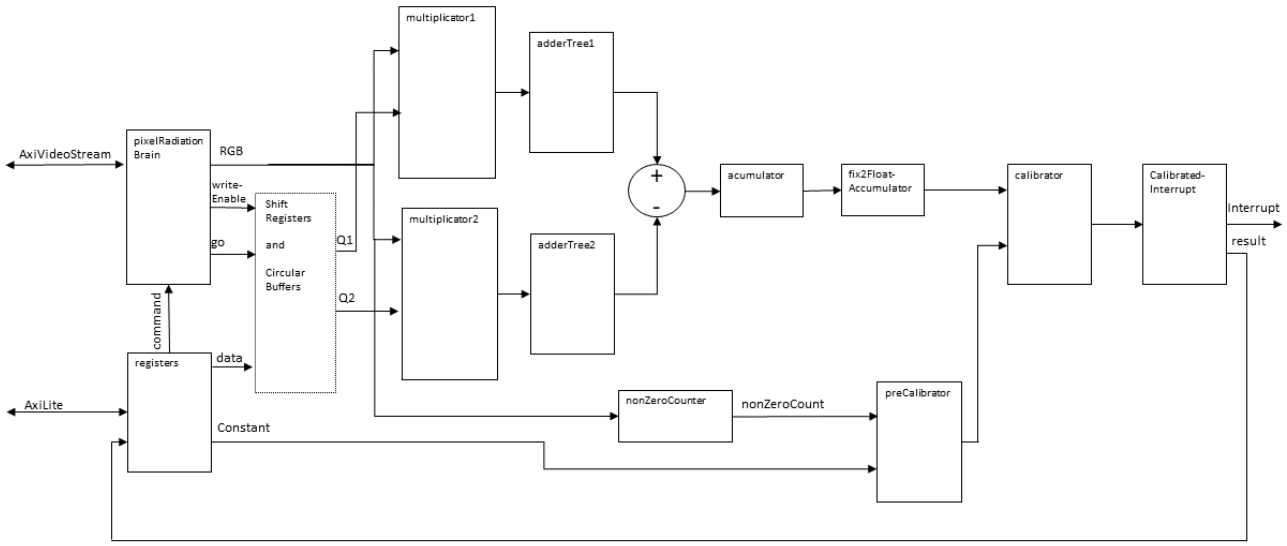

Figura 3. Diagrama de la relación entre los bloques implementados en pixelRadiation.

los elementos que llegan de a uno por AXI Lite. Guarda $LVECTOR$ elementos.

- $circularBuffer$: los elementos de las matrices Q se guardan en buffers separados por color. Tiene una profundidad de $STEPS$. Este módulo es una FIFO first word fall though que vuelve a insertar en la FIFO cada elemento que fue leído. Se utilizó una XPM_FIFO ya que es parametrizable y no requiere utilizar un wizard para hacer cambios en los largos o profundidad.

Todos los módulos menos $pixelRadiationStateMachine$ pueden ser utilizados sin problema en otro proyecto. El módulo $addmultiply$ está hecho para multiplicar un vector de $3 \times LVECTOR$ por otro de 3×1 lo cual resulta óptimo para procesamiento de imágenes. Y el IP se puede ocupar para cualquier cálculo que tenga la estructura de la ecuación (8) y si los tamaños de las variables son los mismos no es necesario implementar de nuevo el hardware ya que las variables son programables a través de la interfaz AXILite. Todos los módulos implementados fueron diseñados pensando en que se podrían ocupar en otra aplicación. Esto se ve en que las entradas y salidas tienen nombres genéricos y no específicos a esta aplicación.

La latencia es de $(33 + 17 * NPIXEL) * p$ y el throughput es de $1/(17 * NPIXEL * p)$ donde p es el periodo del reloj utilizado. Esto se puede verificar en el testbench. De esos 33 ciclos constantes 9 son para pasar de punto fijo a flotante, otros 9 para la multiplicación en $calibrator$ y el resto es entre los adder tree y el acumulador.

En la tabla I se tiene el reporte de utilización de recursos y slack del diseño con reloj de 160MHz con varias estrategias de implementación. Siempre hay un slack en el modulo de HDMI y como no es parte del IP diseñado se va a ignorar. Se puede observar que a esta frecuencia solo la implementación 9 no tiene slack (salvo -0.89 en HDMI).

El critical path se encuentra en el módulo $circularBuffer$ específicamente en el read enable de la FIFO implementada. Esto se puede deber a que son varias FIFO con entradas y salidas bastante grandes. Otro factor que influye es que es first word fall though. Puede ser que la implementación por XPM_FIFO no sea muy óptima ya que la señal en cuestión pasa por 3 LUT antes de entrar a la RAM. No se esperaba un error así en este modulo porque no se veía tan complicado.

Se puede solucionar utilizando una RAM, pero se perdería flexibilidad en el diseño ya que habría que utilizar un wizard cada vez que se hace un cambio. Además habría que advertir al usuario del ip que cuando cambia el parámetro $STEPS$ también tiene que cambiar la RAM, claramente esto podría llevar a errores si es que se le olvida cambiar la RAM. También se espera que cambiando $STEP$ y $LVECTOR$ se obtengan diferentes resultados.

IV. Demo

El demo se basa en un ejemplo de Digilent "Nexys Video HDMI Demo"[2]. Los principales componentes en el demo son un procesador MicroBlaze, una entrada HDMI, un módulo Video Direct Memory Address, un controlador para la RAM y un módulo $pixelRadiation$.

Para crear el proyecto se deben seguir las instrucciones de "Using Digilent Github Demo Projects"[3]. Si es que se modifica algún parametro de $pixelRadiation$ se debe activar

TABLA I
UTILIZACIÓN DE RECURSOS Y SLACK PARA DISTINTAS ESTRATEGIAS DE IMPLEMENTACIÓN, CON UN RELOJ DE 160MHZ

Name	Strategy	WNS	TNS	LUT	FF	BRAM	DSP	LUTRAM	Elapsed
synth_1	Flow_AreaMultThresholdDSP			0	0	0	0	0	00:00:42
impl_1	Congestion_SpreadLogic_high	-0,77	-0,81	35164	52406	131	411	2251	00:59:42
impl_2	Congestion_SpreadLogic_medium	-0,82	-2,08	35153	52406	131	411	2249	00:58:49
impl_3	Congestion_SpreadLogic_low	-0,77	-2,06	35158	52406	131	411	2252	00:52:23
impl_4	Congestion_SpreadLogic_Explore	-0,77	-0,81	35164	52406	131	411	2251	00:57:38
impl_5	Congestion_SSI_SpreadLogic_high	-0,89	-1,04	35150	52406	131	411	2252	00:57:00
impl_6	Congestion_SSI_SpreadLogic_low	-0,77	-3,08	35134	52406	131	411	2249	00:50:44
impl_7	Congestion_SSI_SpreadLogic_Explore	-0,89	-1,04	35150	52406	131	411	2252	00:56:44
impl_8	Performance_ExtraTimingOpt	-0,77	-0,97	35160	52406	131	411	2251	00:55:15
impl_9	Performance_Retiming	-0,89	-0,89	35150	52406	131	411	2252	01:00:32
Out-of-Context pixelRadiation	Vivado Synthesis Defaults			12778	24487	87	402	380	00:02:17

la opción "Keep_equivalent_registers.[en] la implementación de ese modulo fuera de contexto.

Una vez cargado el bitstream en la FPGA y el código en C cargado, se debe utilizar algún programa que permita comunicación serial con la FPGA. La conexión a un baud rate de 115200 y se debe hacer por el puerto dedicado a UART en la FPGA.

El menú que se despliega en en terminal indica si le llega o no algo por HDMI. También se puede cambiar el buffer al que llega el video o el buffer al cual se se calcula la radiación. También se pueden imprimir patrones de color directamente en la RAM si es que no hay input por HDMI.

Para obtener la radiación actual se debe enviar un '7' y el menú se va a actualizar con el nuevo valor. Cada vez que el menú se refresca se va a actualizar el valor de la radiación. Display Frame Index corresponde al frame que se está leyendo y Video Frame Index el frame al cual se le esta grabando la imagen que llega por HDMI.

No se encontró una forma de garantizar que el IP esta recibiendo la imagen exactamente como se está mandando. La imagen se está guardando en un sector de la RAM pero aparentemente la imagen que se lee no está alineada correctamente. Por lo que la radiación que se indica en el menú no coincide con la calculada en Matlab. Pero el resultado que se obtiene es constante y va a ser la misma para esa imagen. También está la posibilidad que se sea un error del IP que no se vio en el testbench. Una forma de asegurar el correcto funcionamiento seria hacer otro proyecto donde el video pase directamente al IP para un mejor control de la imagen.

Además, se incluye un testbench para el IP $pixelRadiation$. Para usarlo se debe correr alguno de los tests en el script Matlab $testbench.m$ para crear un vector de prueba y correr el script $writeQandConst.m$. Luego en el proyecto se debe modificar la variable $NPIXELS$ del testbench y la variable $path$ para que apunte a la dirección donde se crearon los archivos anteriores. Se recomienda utilizar imágenes pequeñas, menores a 50x50 para que no se demore tanto. Al final del test se comparan los resultados obtenidos con los esperados, estos casi siempre difieren en los últimos bit. Pero como es un numero flotante, la diferencia es prácticamente nula.

V. CONCLUSIONES

El proyecto está funcionando y hace lo esencial. Se le puede entregar imágenes y les calcula sus radiaciones. Sin embargo, todavía queda trabajo. Lo más necesario sería una forma de hacer pruebas automáticas, ya que no se alcanzó codificar un driver en C para hacerse cargo de la interrupción generada por la IP. Tampoco se encontró una forma de mandar imágenes por HDMI y estar 100 % seguro de la resolución y de la radiación. También faltó implementar la ecuación (12).

Un error que no se logró resolver plenamente fue que cuando se implementa este IP, Vivado se queda pegado optimizando registros que se repiten entre los dos multiplicadores. El $work\ arround$ fue activar la directiva de síntesis "Keep_equivalent_registers".

Una mejora que puede ayudar con descongestionar el diseño es utilizar menos shift registers. En este diseño hay uno por cada columna de las matrices Q, sin embargo, se podría utilizar solo uno y que $pixelRadiationStateMachine$ controle a cual buffer se cargan los datos. Esto no se hizo en su momento debido a que trae consigo una lógica a la máquina de estados y se prefirió dar más importancia a otros problemas del diseño. Esto podría relajar un poco la implementación y crear menos errores de timing.

Lo más subestimado al comienzo de este proyecto fue la cantidad de tiempo que se debe dedicar para investigar sobre otras IP, evaluar cual utilizar y luego debuggear problemas en ellas. El 75 % del tiempo dedicado a este proyecto fue dedicado a ese tipo de cosas. Lo más rescatable de lo aprendido de otras IPs es la interfaz AXI que simplifica y estandariza la comunicación entre módulos. En particular, AXILite y AXIStream son fáciles de implementar y hacer hardware compatible. Lo segundo más rescatable es el módulo generado por $airhdl.com$, funciona muy bien para lo que se necesitaba en este proyecto. Además, permite manejar bits individuales o grupales dentro de un registro tal como se utiliza en un microprocesador. No solamente genera código en VHDL y System Verilog, también genera un header en C, documentación en formato html, testbenches en VHDL y System Verilog. Pero los testbenches son pagados.

El hacer uso de un testbench durante el desarrollo del IP fue muy útil. En este proyecto se vieron tres grandes ventajas al utilizar un testbench.

La primera ventaja es que permite un diseño incremental mucho más fácil y rápido. No es necesario hacer síntesis e implementación después de cada cambio lo que facilita hacer pruebas por cada cambio que se hace y no juntar varios para luego probarlos físicamente el la FPGA y de alguna forma

averiguar que cambio fue el que trajo problemas.

La segunda ventaja es que no hay que hacer cambios en el diseño para ver señales internas. En las tareas se ocupó un $Integrated\ Logic\ Analizer\ (ILA)$ que requería ser implementado con las señales especificas que se quieren ver, si luego se requieren más señales se debe re-implementar. Pero con testbench solo se debe correr la simulación de nuevo. Además, luego de tener un diseño verificado por testbench es mucho más fácil encontrar errores en la implementación en FPGA. Se puede utilizar un ILA y comparar directamente los resultados con los obtenidos en el testbench y así no tener que calcular a mano, o con un programa en que estado debería estar ese pin.

La última ventaja que se vio en este proyecto es que asila el ip del resto del hardware implementado. Al hacer testbench se tiene un control sobre las entradas que permite hacer pruebas especificas y estar seguro de lo que esta entrando al IP. Con el diseño implementado en la FPGA es más difícil saber exactamente que píxeles están entrando al IP.

La única desventaja es que hay que dedicar tiempo a hacer, debuggear y mantener un testbench. Sin embargo, todo esto valió la pena en el proyecto.

El proyecto de ejemplo utiliza diseño de bloques. Al utilizar diseño de bloques es mucho más fácil entender como están relacionados los módulos, hubiese sido mucho más difícil de haber estado en código. Pero también tiene una desventaja, que es que no se pueden instanciar códigos propios que incluyan otras IP. La solución es hacer un IP que incluya a otras. El diseño del IP mismo pudo haber sido en bloques pero habría que haber hecho más IPs propios (para $circularBuffer$ y $nonZeroPixelCounter$) y rápidamente podría ponerse engorroso el desarrollo.

Por ultimo, la lección más valiosa que se obtuvo en este proyecto fue en la parte del análisis matemático. A penas se llegó a un resultado implementable (ecuación (8)) se implementó, lo cual es muy mala idea. Muchas horas de trabajo se hubiesen ahorrado de haber dedicado un poco más de tiempo a analizar las ecuaciones y buscar otra forma de implementarlas (ecuación (12)). El problema quedó mucho más sencillo, fácil de implementar y tiene mucho potencial de ser más rápido.

REFERENCIAS

[1] H. O. Garces, L. E. Arias, A. J. Rojas, J. Cuevas, and A. Fuentes. Combustion diagnostics by calibrated radiation sensing and spectral estimation. *IEEE Sensors Journal*, 17(18):5871–5879, Sept 2017.

[2] *Nexys Video HDMI Demo*. https://reference.digilentinc.com/learn/programmable-logic/tutorials/nexys-video-hdmi-demo/start.

[3] *Using Digilent Github Demo Projects*. https://reference.digilentinc.com/learn/programmable-logic/tutorials/github-demos/start.