DEPARTAMENTO DE ELÉCTRICA, ELECTRÓNICA Y TELECOMUNICACIONES

CARRERA DE INGENIERÍA EN ELECTRÓNICA, AUTOMATIZACIÓN Y CONTROL

TRABAJO DE TITULACIÓN, PREVIO A LA OBTENCIÓN DEL TÍTULO DE INGENIERO EN ELECTRÓNICA, AUTOMATIZACIÓN Y CONTROL

TEMA: ESTUDIO Y APLICACIÓN DE SÍNTESIS DE ALTO-NIVEL PARA DISEÑO DE SISTEMAS-ON-CHIP EMBEBIDOS DE ALTO DESEMPEÑO BASADOS EN FPGA E IP-CORES PERSONALIZADOS

AUTOR: BERRAZUETA MENA, LUIS DAVID

DIRECTOR: ING. NAVAS VIERA, BYRON ROBERTO Ph.D.

SANGOLQUÍ

2019

DEPARTAMENTO DE ELÉCTRICA, ELECTRÓNICA Y TELECOMUNICACIONES
CARRERA DE INGENIERÍA EN ELECTRÓNICA, AUTOMATIZACIÓN Y CONTROL

CERTIFICACIÓN

Certifico que el trabajo de titulación, **"ESTUDIO Y APLICACIÓN DE SÍNTESIS DE ALTO-NIVEL PARA DISEÑO DE SISTEMAS-ON-CHIP EMBEBIDOS DE ALTO DESEMPEÑO BASADOS EN FPGA E IP-CORES PERSONALIZADOS"** fue realizado por el señor **BERRAZUETA MENA, LUIS DAVID** el mismo que ha sido revisado en su totalidad, analizado por la herramienta de verificación de similitud de contenido; por lo tanto cumple con los requisitos teóricos, científicos, técnicos, metodológicos y legales establecidos por la Universidad de Fuerzas Armadas ESPE, razón por la cual me permito acreditar y autorizar para que lo sustente públicamente.

Sangolquí, enero de 2019

.....

ING. BYRON ROBERTO NAVAS VIERA Ph.D.

C. C: 1709079287

DEPARTAMENTO DE ELÉCTRICA, ELECTRÓNICA Y TELECOMUNICACIONES
CARRERA DE INGENIERÍA EN ELECTRÓNICA, AUTOMATIZACIÓN Y CONTROL

AUTORÍA DE RESPONSABILIDAD

Yo, **BERRAZUETA MENA, LUIS DAVID**, declaro que el contenido, ideas y criterios del trabajo de titulación: **"ESTUDIO Y APLICACIÓN DE SÍNTESIS DE ALTO-NIVEL PARA DISEÑO DE SISTEMAS-ON-CHIP EMBEBIDOS DE ALTO DESEMPEÑO BASADOS EN FPGA E IP-CORES PERSONALIZADOS"** es de mi autoría y responsabilidad, cumpliendo con los requisitos teóricos, científicos, técnicos, metodológicos y legales establecidos por la Universidad de Fuerzas Armadas ESPE, respetando los derechos intelectuales de terceros y referenciando las citas bibliográficas.
Consecuentemente el contenido de la investigación mencionada es veraz.

Sangolquí, enero de 2019

.....
LUIS DAVID BERRAZUETA MENA

C. C: 0502651987

DEPARTAMENTO DE ELÉCTRICA, ELECTRÓNICA Y TELECOMUNICACIONES
CARRERA DE INGENIERÍA EN ELECTRÓNICA, AUTOMATIZACIÓN Y CONTROL

AUTORIZACIÓN

Yo, **BERRAZUETA MENA, LUIS DAVID** autorizo a la Universidad de las Fuerzas Armadas ESPE publicar el trabajo de titulación: **"ESTUDIO Y APLICACIÓN DE SÍNTESIS DE ALTO-NIVEL PARA DISEÑO DE SISTEMAS-ON-CHIP EMBEBIDOS DE ALTO DESEMPEÑO BASADOS EN FPGA E IP-CORES PERSONALIZADOS"** en el Repositorio Institucional, cuyo contenido, ideas y criterios son de mi responsabilidad.

Sangolquí, enero de 2019

.....

LUIS DAVID BERRAZUETA MENA

C. C: 0502651987

Dedicado a la memoria de
Isolina, Laura y Efrén

Agradecimiento

En primer lugar, mi más sincero agradecimiento al Dr. Byron Navas, el valor de su consejo es incalculable. Su apoyo y dedicación a este trabajo ha sido realmente inspirador.

Estoy particularmente agradecido con mis padres, María del Carmen y Luis, y mi hermana, María Mercedes, por su paciencia y ayuda. En ustedes he encontrado el más profundo apoyo y consejo diario. Solo con su cariño he logrado alcanzar esta meta.

A mis abuelos, Fanny y Efrén, quienes con su sabiduría e infinito amor han hecho mis sueños posibles.

A Karolay, por compartir conmigo los mejores y más dificiles momentos a lo largo de este camino. Soy muy afortunado de tenerte a mi lado.

David Berrazueta Mena

Enero 2019

Índice general

Certificado del Director . . . . . **i**

Autoría de Responsabilidad . . . . . **ii**

Autorización . . . . . **iii**

Dedicatoria . . . . . **iv**

Agradecimiento . . . . . **v**

Índice de Contenidos . . . . . **xii**

Índice de Tablas . . . . . **xiii**

Índice de Figuras . . . . . **xv**

Índice de Códigos . . . . . **xxi**

Resumen . . . . . **xxiii**

Abstract . . . . . **xxiv**

1 Introducción **1**

1.1 Antecedentes 1

1.2 Justificación e Importancia 4

1.3 Alcance del Proyecto 6

1.4 Objetivos 8
1.4.1 Objetivo General 8
1.4.2 Objetivos Específicos 8

2 Conceptos y Estado del Arte 9
2.1 Conceptos 9
2.1.1 Field-Programmable Gate Array (FPGA) 9
2.1.2 System-on-Chip (SoC) 11
2.2 SoCs Basados en FPGA 12
2.3 Sistemas Embebidos de Alto Desempeño Basados en FPGA 16
2.4 Flujo de Diseño de SoCs 18
2.5 Complejidad en el Diseño de SoCs y Uso de Altos Niveles de Abstracción . . 21
2.6 High-Level Synthesis (HLS) 23
2.6.1 Control Data Flow Graph (CDFG) 24
2.6.1.1 CFG (Control Flow Graph) 24
2.6.1.2 DFG (Data Flow Graph) 25
2.6.1.3 Representación CDFG 26
2.6.2 Fases de HLS 26
2.6.3 Reseña Histórica de HLS 29
2.6.4 Herramientas HLS 31

3 Síntesis de Alto-Nivel para SoCs de Xilinx 36
3.1 Diseño de FPGAs Basados en Especificaciones C Usando Vivado HLS 36
3.1.1 Ventajas del Uso de Vivado HLS 36
3.1.2 Síntesis de Especificaciones C 37
3.1.2.1 Síntesis de una Especificación C en Vivado HLS 37
3.1.2.2 Métricas de Desempeño 38
3.2 Vivado High-Level Synthesis (HLS) 41

3.2.1 Metodología de Diseño de Vivado HLS 42
3.2.1.1 Flujo de Diseño 42
3.2.1.2 Entradas y Salidas de Vivado HLS 44
3.2.1.3 Interfaces de Usuario de Vivado HLS 45
3.2.2 Lenguajes, Librerías y Tipos de Datos Soportados 47
3.2.3 Tipos de Datos 48
3.2.3.1 Tipos de Datos C Estándar 48
3.2.3.2 Tipos de Datos Eficientes para Hardware 49
3.2.4 Manejo de Interfaces 51
3.3 Verificación y Optimización en Vivado HLS 55
3.3.1 Verificación Mediante Co-Simulación C/RTL 55
3.3.2 Análisis de Resultados de Síntesis en Vivado HLS 56
3.3.3 Restricciones y Directivas de Optimización 59
3.3.4 Optimizaciones de Desempeño 60
3.3.4.1 Pipelining 60
3.3.4.2 Particionamiento de Arreglos 62
3.3.4.3 Unrolling Loops 64
3.3.4.4 Dataflow 66
3.3.5 Otras Optimizaciones 68
3.3.5.1 Optimizaciones de Latencia 68
3.3.5.2 Optimizaciones de Área 69

4 Diseño de Sistemas SoC con IP-Cores Personalizados Usando Vivado . 71
4.1 Multiplicador de Matrices 73
4.1.1 Algoritmo 73
4.1.2 Implementación en Vivado HLS 75
4.1.2.1 Creación de una Especificación C 76

4.1.2.2 Simulación C 76
4.1.2.3 Especificación de Directivas de Síntesis 79
4.1.2.4 Síntesis HLS 83
4.1.2.5 Co-Simulación C/RTL 84
4.1.3 Integración del IP-core en el SoC 85
4.1.4 Software Embebido en Xilinx SDK 90
4.1.5 Optimizaciones 92
4.1.5.1 Solución 1 93
4.1.5.2 Solución 2 94
4.1.5.3 Solución 3 96
4.1.5.4 Solución 4 98
4.1.5.5 Resumen de Soluciones 99
4.2 Transformada Rápida de Fourier (FFT) 100
4.2.1 Algoritmo 101
4.2.1.1 Algoritmo Cooley – Tukey para FFT 101
4.2.2 Implementación en Vivado HLS 106
4.2.2.1 Variable Loop Bounds 106
4.2.3 Integración del IP-core en el SoC 109
4.2.4 Optimizaciones 110
4.3 AES (Advanced Encryption Standard) 111
4.3.1 Algoritmo 112
4.3.1.1 Operaciones Usadas en la Encriptación AES 112
4.3.1.2 Key Schedule 116
4.3.2 Implementación en Vivado HLS 116
4.3.3 Integración del IP-core en el SoC 119
4.3.4 Optimizaciones 119
4.4 Backpropagation Neural Network 121

4.4.1 Algoritmo 122
4.4.1.1 Redes Neuronales Artificiales (ANN) 122
4.4.1.2 Backpropagation 124
4.4.2 Implementación en Vivado HLS 127
4.4.3 Integración del IP-core en el SoC 128
4.4.4 Optimizaciones 128
4.5 Red Neuronal Artificial (ANN) para Reconocimiento de Números Manuscritos 130
4.5.1 Algoritmo 130
4.5.2 Implementación en Vivado HLS 130
4.5.3 Integración del IP-core en el SoC 132
4.5.4 Optimizaciones 132

5 Definición de un Flujo de Diseño Para SoCs e Incremento de Nivel de Automatización 134

5.1 Definición de un Flujo de Diseño Confiable para la Aceleración Mediante IP-Cores en SoCs de Xilinx 134
5.1.1 Definición de Especificación C y Test Bench 135
5.1.2 Simulación C 138
5.1.3 Síntesis C 138
5.1.4 Co-simulación C/RTL 138
5.1.5 Evaluación del Diseño Implementado 139
5.1.6 Aplicación de Directivas de Optimización 139
5.1.7 Exportación del IP 140
5.1.8 Integración del IP-core en el SoC 140
5.1.9 Creación de Software 140
5.2 Generación de Sistemas Embebidos SoCs Basados en FPGA Mediante Scripts Tcl 141

5.2.1 Lenguaje Tcl en Herramientas de Xilinx 141
5.2.2 Scripts Tcl en Vivado HLS 142
5.2.2.1 Creación Automatizada de un IP en Vivado HLS Usando Comandos Tcl 142
5.2.3 Scripts Tcl en Vivado Design Suite 146
5.2.3.1 Creación Automatizada de un SoC en Vivado Design Suite Usando Comandos Tcl 147
5.3 Demostración y Verificación de un Flujo de Diseño Automatizado para la Creación de SoCs 151
5.3.1 IP SoC Generator 152
5.3.2 Creación Automática de IP-cores de Referencia Mediante IP SoC Generator 153
5.3.3 Creación Automática de SoCs de Referencia Mediante IP SoC Generator 155
5.3.4 Creación Automática de Nuevos IP-cores Mediante IP SoC Generator 157
5.3.5 Creación Automática de Nuevos SoCs Mediante IP SoC Generator . . 162
5.3.6 Limitaciones de IP SoC Generator 162

6 Exploración de Desempeño 164
6.1 Implementación en Hard-Core Processor y Exploración de Desempeño 164
6.1.1 Implementaciones en ARM Cortex-A9 de Xilinx Zynq 165
6.1.2 Exploración de Desempeño de Implementaciones de Software en ARM Cortex-A9 166
6.2 Aceleración en IP-cores y Exploración de Desempeño 167
6.2.1 Exploración de Desempeño de Aceleradores Creados en Vivado HLS . 168
6.2.2 Aceleradores en SoCs Basados en FPGA 171
6.3 Análisis Comparativo entre Diferentes Arquitecturas 171
6.4 Discusión de Resultados 175

7 Conclusiones y Recomendaciones 178

7.1 Conclusiones 178

7.2 Recomendaciones 180

Referencias 182

Índice de tablas

Tabla 1 Sistemas computacionales embebidos basados en FPGA 15

Tabla 2 Perspectiva general de herramientas HLS relevantes 32

Tabla 3 Tipos de datos nativos de lenguaje C 49

Tabla 4 Tipos de datos enteros de precisión arbitraria en Vivado HLS 50

Tabla 5 Tipos de datos de punto fijo de precisión arbitraria en Vivado HLS . 51

Tabla 6 Protocolos soportados en interfaces de Vivado HLS 53

Tabla 7 Elementos del reporte de síntesis C en Vivado HLS 57

Tabla 8 Benchmarks seleccionados para el desarrollo de aceleradores en hardware 72

Tabla 9 Resumen de soluciones para el IP *matrixmul* 100

Tabla 10 Número de operaciones para una DFT y FFT de 1024 puntos 104

Tabla 11 Representación binaria para la selección de índices en una FFT . . . 106

Tabla 12 Resumen de soluciones para el IP AES 121

Tabla 13 Parámetros de entrenamiento del benchmark Backpropagation 127

Tabla 14 Resumen de soluciones para el IP Backprop 129

Tabla 15 Resumen de soluciones para el IP ANN 133

Tabla 16 Tarjetas de desarrollo con Xilinx Zynq más usadas 144

Tabla 17 Directorios para los diseños de referencia 157

Tabla 18 Resumen de configuración de las arquitecturas A1-ARM y A2-ARM . 166

Tabla 19 Resultados de desempeño de las arquitecturas A1-ARM y A2-ARM . 167

Tabla 20 Resultados de desempeño de aceleradores por defecto y optimizados . 169

Tabla 21 Resumen de utilización de recursos de FPGA de aceleradores en hardware 171

Tabla 22 Resumen de configuración de las arquitecturas A1-ARM+IPacc y A2-ARM+IPacc 172

Tabla 23 Resultados de desempeño de benchmarks en diferentes arquitecturas . 173

Índice de figuras

Figura 1 Arquitectura básica de un FPGA 10

Figura 2 Arquitectura básica de un SoC 12

Figura 3 Arquitecturas de sistemas computacionales embebidos basados en FPGA 15

Figura 4 SoC de arquitectura heterogénea 17

Figura 5 Flujo de diseño de SoCs 20

Figura 6 Complejidad del diseño de SoCs 22

Figura 7 CFG de la función *ejemplo* 25

Figura 8 DFG de la función *ejemplo* 25

Figura 9 CDFG de la función ejemplo 26

Figura 10 Flujo de diseño de HLS 27

Figura 11 Diferentes implementaciones de un datapath 28

Figura 12 Implementación basada en la combinación de un datapath y un controlador 29

Figura 13 Implementación mediante Vivado HLS del Código 2 40

Figura 14 Latencia e intervalos en la ejecución del ejemplo mostrado en el Código 2 41

Figura 15 Vivado HLS en el flujo de diseño de Xilinx 42

Figura 16 Flujo de diseño de Vivado HLS 43

Figura 17 Interfaz gráfica de usuario de Vivado HLS 46

Figura 18 Interfaz de línea de comandos Tcl de Vivado HLS 47

Figura 19 Interfaces generadas por defecto en Vivado HLS para la función *mac_hls* 52

Figura 20 Flujo de verificación de Vivado HLS mediante co-simulación C/RTL 56

Figura 21 Ejemplo de reporte de síntesis en Vivado HLS 57

Figura 22 Perspectiva de análisis de Vivado HLS 58

Figura 23 Periodo de reloj e incertidumbre de reloj en Vivado HLS 59

Figura 24 Ejecución de la función *example1* del Código 4 61

Figura 25 Ejecución de la función *example2* del Código 5 63

Figura 26 Particionamiento de arreglos en subarreglos o en elementos individuales 64

Figura 27 Implementación *rolled* de la función *loop* del Código 6 65

Figura 28 Implementaciones de la función *loop* del Código 6 utilizando *unrolled* 66

Figura 29 Implementación de la jerarquía de sub-funciones de la función *func_top_level* del Código 7 67

Figura 30 Loop flattening en lazos anidados 69

Figura 31 Proceso de obtención del elemento $c_{1,1}$ de la matriz para la multiplicación de las matrices A y B de orden 2×2 74

Figura 32 Importación de los archivos que forman parte de la especificación C 77

Figura 33 Comparación de resultados esperados y resultados del DUT en la simulación C de Vivado HLS 78

Figura 34 Arquitectura de un Zynq-7000 SoC 80

Figura 35 Canales de escritura y lectura de la interfaz AXI4 81

Figura 36 AXI Interconnect con múltiples maestros y esclavos 82

Figura 37 Reporte de síntesis del IP *matrixmul* para un dispositivo Zynq Z-7010 84

Figura 38 Reporte de co-simulación exitoso para el IP matrixmul 85

Figura 39 IP Matrixmul importado al catálogo de IP-cores de Vivado IP Integrator 86

Figura 40 Zynq PS y su ventana *Re-customize* IP en el diseño de bloques de Vivado IP Integrator 87

Figura 41 Zynq PS e IP matrixmul en el diseño de bloques de Vivado IP Integrator 87

Figura 42 Diagrama de bloques de la integración de IP *matrixmul* en el SoC . 88

Figura 43 Creación de un wrapper HDL de un diseño de bloques en Vivado Design Suite 89

Figura 44 Generación de bitstream en Vivado Design Suite 89

Figura 45 Modelo de capas de hardware y software para el diseño de SoC en Xilinx Zynq utilizado en la tesis 90

Figura 46 Árbol de proyecto en Xilinx SDK para un SoC basado en el modelo de capas de la Figura 45 92

Figura 47 Estimación de desempeño y recursos de la Solución 1 del IP *matrixmul*. 93

Figura 48 Perspectiva de análisis de la Solución 1 para el IP *matrixmul* 94

Figura 49 Estimación de desempeño y recursos de la Solución 2 del IP *matrixmul* 95

Figura 50 Perspectiva de análisis de la Solución 2 para el IP *matrixmul* 96

Figura 51 Estimación de desempeño y recursos de la Solución 3 del IP *matrixmul* 97

Figura 52 Perspectiva de análisis de la Solución 3 para el IP *matrixmul* 97

Figura 53 Estimación de desempeño y recursos de la Solución 4 del IP *matrixmul* 98

Figura 54 Perspectiva de análisis de la Solución 4 para el IP *matrixmul* 99

Figura 55 Diagramas de mariposa para una FFT de 4 puntos 103

Figura 56 Selección de índices de las muestras para calcular pequeñas DFTs de dos puntos 105

Figura 57 Estimación de desempeño para la función *fft* original del Código 11 108

Figura 58 Estimación de desempeño para la función *fft* modificada del Código 12109

Figura 59 Diagrama de bloques de la integración de IP *fft* en el SoC 110

Figura 60 Operación *SubByte* de AES 113

Figura 61 Operación *ShiftRows* de AES 113

Figura 62 Operación *MixColumn* de AES 114

Figura 63 Operación *AddRoundKey* de AES 115

Figura 64 Proceso de encriptación AES para una clave de 128 bits 115

Figura 65 Proceso key schedule para obtención de sub-claves AES 117

Figura 66 Reporte de co-simulación para el benchmark AES de la fuente CHStone 118

Figura 67 Modificación del archivo de cabecera AES.h. 118

Figura 68 Diagrama de bloques de la integración de IP AES en el SoC 120

Figura 69 Ejecución de encriptación AES utilizando pipelining 120

Figura 70 Modelo básico de una neurona artificial 122

Figura 71 Función sigmoide 123

Figura 72 Red neuronal *feedforward* de tres capas 124

Figura 73 Representación del método de descenso de gradiente para retropropagación 125

Figura 74 Estimación de desempeño para el IP Backprop 128

Figura 75 Diagrama de bloques de la integración de IP Backprop en el SoC . . 129

Figura 76 Mensajes de error en la síntesis con optimizaciones pipelining, unrolled para el IP Backprop 129

Figura 77 Red neuronal para el reconocimiento de dígitos manuscritos en imágenes de 20×20 pixeles 131

Figura 78 Diagrama de bloques de la integración de IP ANN en el SoC 132

Figura 79 Flujo de diseño confiable para la creación de aceleradores en hardware en Vivado HLS 135

Figura 80 Estructura básica de una especificación C en Vivado HLS 136

Figura 81 Archivos para la creación automatizada de un IP-core utilizando Vivado HLS para el ejemplo de la Sección 5.2.2.1 143

Figura 82 Carpeta con los archivos de solución creada en Vivado HLS mediante script Tcl 146

Figura 83 Reporte de síntesis RPT para el ejemplo matrixmul consultado utilizando un lector de texto plano 147

Figura 84 Archivos para la creación automatizada de un SoC integrando un IP-core generado en Vivado HLS 148

Figura 85 Arquitectura del SoC generado mediante el script Tcl del ejemplo de la Sección 5.2.3 151

Figura 86 Flujo de diseño automatizado usando scripts Tcl para la creación de SoCs con aceleradores de hardware 152

Figura 87 Archivos de IP_SoC_generator 153

Figura 88 Interfaz Vivado HLS Command Prompt 154

Figura 89 IP_SoC_generator script ejecutandose en consola 154

Figura 90 IP_SoC_generator selección de dispositivo 155

Figura 91 Archivos generados automáticamente por IP_SoC_generator durante la recreación del proyecto de referencia matrixmul 156

Figura 92 Ventana GUI de Vivado Design Suite. 156

Figura 93 Mensaje de finalización del proceso de generación del bitstream . . . 158

Figura 94 Archivos de la especificación C FIR para la creación de un nuevo IP-core mediante IP_SoC_generator 160

Figura 95 Creación de un nuevo IP a partir de la especificación C de un filtro FIR en IP_SoC_generator 161

Figura 96 Creación de scripts para la automatización del proceso de diseño de SoC en IP_SoC_generator 161

Figura 97 Archivos generados automáticamente por IP_SoC_generator durante la recreación de un nuevo IP 162

Figura 98 Arquitectura de SoCs generados mediante IP_SoC_generator 163

Figura 99 Arquitectura del processing system de un Zynq-7000 165

Figura 100 Factor de aceleración de los IP-cores optimizados en relación a los diseños generados por defecto en Vivado HLS 169

Figura 101 Representación abstracta de las arquitecturas basadas en Zynq . . . 172

Figura 102 Factor de aceleración de la arquitectura A2-ARM+IPacc en relación a A1-ARM 174

Figura 103 Reducción del tiempo de ejecución de algoritmos implementados en aceleradores en hardware 176

Índice de códigos

Código 1 Función *ejemplo* escrita en lenguaje C 24

Código 2 Función *basicf* escrita en lenguaje C 39

Código 3 Función *mac_hls* escrita en lenguaje C 52

Código 4 Función *example1* escrita en lenguaje C 61

Código 5 Función *example2* escrita en lenguaje C 62

Código 6 Función *loop* escrita en lenguaje C 64

Código 7 Función *func_top_level* escrita en lenguaje C 67

Código 8 Algoritmo para multiplicación de matrices en C++ 74

Código 9 Test Bench C para la multiplicación de matrices del Código 8 . . . 78

Código 10 Directivas para la especificación de interfaces AXI4-Lite a nivel de puertos y a nivel de bloque para la función matrixmul 83

Código 11 Algoritmo para FFT con método Cooley - Tukey en C++ 104

Código 12 Algoritmo para FFT con método Cooley - Tukey en C++ modificado para resolver problema de variable loop bounds en Vivado HLS 108

Código 13 Creación de un nuevo proyecto en Vivado HLS mediante comandos Tcl 143

Código 14 Creación de una solución en Vivado HLS mediante comandos Tcl . 144

Código 15 Aplicación de directivas de optimización en Vivado HLS mediante comandos Tcl 145

Código 16 Ejecución de simulación C síntesis C co-simulación y exportación del IP en Vivado HLS mediante comandos Tcl 145

Código 17 Creación de un nuevo proyecto en Vivado Design Suite mediante comandos Tcl 148

Código 18 Importación de IP en catálogo de IP-cores de Vivado Design Suite mediante comandos Tcl 149

Código 19 Creación de un nuevo diseño de bloques en Vivado Design Suite mediante comandos Tcl 149

Código 20 Directiva para especificar interfaz AXI a nivel de puertos 159

Código 21 Directiva para especificar interfaz AXI a nivel de bloque 159

Código 22 Función top-level fir con especificación de interfaces AXI Lite . . . 159

Resumen

La necesidad actual de SoCs heterogéneos para aplicaciones que requieren un poder computacional cada vez mayor ha incrementado la complejidad de los procesos de diseño, verificación e integración. SoCs ejecutando software en procesadores host con funciones aceleradas en IP-cores ofrecen un mayor desempeño comparado con arquitecturas tradicionales. En particular, FPGAs permiten la creación de aceleradores de funciones cuyo desempeño es limitado en procesadores de software embebido. Sin embargo, la necesidad de trabajar en bajos niveles de abstracción (e.g., HDL) puede significar una desventaja. En la actualidad, los métodos y herramientas HLS (High-Level Synthesis) ofrecen reducir la complejidad de diseño usando descripciones de alto-nivel. El propósito de esta tesis es investigar la metodología de diseño, usando Vivado HLS, para acelerar el desarrollo de SoCs heterogéneos de alto desempeño basados en FPGA. Para abordar este problema, esta tesis plantea la creación de aceleradores (IP-cores) de algoritmos usados en aplicaciones de sistemas embebidos y de control en base a los cuales se explora las optimizaciones y limitaciones de Vivado HLS. El resultado de esta tesis demuestra que el método de diseño utilizado es efectivo en la creación de IP-cores que incrementan el desempeño de la ejecución de algoritmos. Las principales contribuciones de esta tesis son: (i) documentación acerca de los métodos, teoría y herramientas de HLS, (ii) diseños y documentación de referencia de aceleradores en hardware de FPGA, (iii) flujo de diseño para la creación de aceleradores en SoCs de Xilinx, y (iv) scripts Tcl para la creación semiautomática de IP-Cores y SoCs de Xilinx.

PALABRAS CLAVE:

- **SISTEMAS EMBEBIDOS**
- **SOCS BASADOS EN FPGA**
- **SÍNTESIS DE ALTO-NIVEL**
- **SISTEMAS HETEROGÉNEOS**

Abstract

The current need for heterogeneous SoCs in complex applications has increased the difficulty of the design, verification and integration processes. SoCs using host processors running embedded software with functions accelerated in IP-cores offer higher performance than traditional architectures. In particular, FPGAs allow creating hardware accelerators for algorithms whose performance is limited in embedded software processors. However, the need to work at low-levels of abstraction (e.g., HDL) is the major disadvantage of these architectures. Currently, HLS (High-Level Synthesis) methods and tools offer to reduce the design complexity using high-level descriptions. To address this problem and take advantage of HLS, this thesis investigates the design methodology to accelerate the development of IP-cores for high-performance heterogeneous SoCs based on FPGA, using Vivado HLS. Besides, this thesis creates such IP-cores to accelerate the execution of algorithms used in applications of embedded and control systems. Using these accelerators, this thesis explores the performance and limitations of the optimization features of Vivado HLS. The result of this thesis shows that the devised design methodology is adequate for the creation of IP-cores that increase the execution performance of algorithms, in several speedup factors. The main contributions of this thesis are (i) documentation about the methods, theory, and tools of HLS, (ii) reference designs and documentation of the accelerators created in FPGA hardware, (iii) design flow for the creation of accelerators in Xilinx SoCs, and (iv) Tcl scripts for the semiautomatic creation of Xilinx IP-Cores and SoCs.

KEY WORDS:

- **EMBEDDED SYSTEMS**
- **FPGA-BASED SOCS**
- **HIGH-LEVEL SYNTHESIS**
- **HETEROGENEOUS SYSTEMS**

Capítulo 1

Introducción

En este capítulo se realiza una introducción al proyecto de investigación presentando los antecedentes, la justificación e importancia, la definición del alcance y el planteamiento de objetivos.

1.1. Antecedentes

Las arquitecturas de hardware reprogramable, tales como FPGAs (*Field-Programmable Gate Arrays*) tienen el potencial para proveer aceleración en la velocidad de operación a una fracción de la energía de un procesador (Nane et al., 2016). Sin embargo, su mayor desventaja ha estado relacionada con la necesidad de diseño en un bajo nivel de abstracción, típicamente usando lenguajes de descripción de hardware o "*Hardware Description Language (HDL)*", por ejemplo, VHDL (*Very High Speed Integrated Circuit HDL*) y Verilog. Estos lenguajes requieren un alto grado de conocimiento y experiencia por parte del diseñador. Debido a esto, la programación mediante lenguajes HDL puede ser agotadora, tomando meses para desarrollar prototipos de prueba y mayor tiempo para perfeccionarlos y optimizarlos (Brodtkorb, Dyken, Hagen, Hjelmervik, & Storaasli, 2010).

El aumento de la complejidad en el diseño de SoCs en las últimas décadas ha incentivado

a la comunidad científica a crear nuevas metodologías y herramientas de diseño (Cong et al., 2011; Coussy, Gajski, Meredith, & Takach, 2009). Una de estas metodologías es la Síntesis de Alto-Nivel o HLS (*High-Level Synthesis*), la cual ha tenido un gran impacto, permitiendo modelar y diseñar sistemas de forma más eficiente a través de una descripción funcional de alto-nivel (Crockett, Elliot, Enderwitz, & Stewart, 2015). Usando HLS, el diseñador tiene la capacidad de influenciar y optimizar la síntesis mediante la aplicación de directivas y restricciones de aspectos de hardware (Ananthanarayana, Lopez, & Lukowiak, 2017). Es por esto, que HLS es cada vez más popular para el diseño de aceleradores y sistemas heterogéneos de alto rendimiento y gran eficiencia energética, acortando el tiempo de desarrollo y abordando la problemática actual (Gort & Anderson, 2015).

El uso de HLS es particularmente interesante en el contexto de arquitecturas heterogéneas basadas en FPGAs y CPUs (Unidades Centrales de Procesamiento) multi-núcleo. La lógica programable de la FPGA proporciona una plataforma perfecta para crear IP-cores (Núcleos de Propiedad Intelectual) de coprocesamiento debido a la capacidad de realizar una ejecución paralela (Ha & Teich, 2017; Park, Shires, & Henz, 2008). Entonces, tareas complejas que requieren una gran cantidad de ciclos secuenciales de reloj de CPU, se pueden ejecutar mucho más rápido en un coprocesador basado en FPGA. Adicionalmente, el uso de HLS significa que partes funcionales del sistema pueden trasladarse del software (destinado a la ejecución en el CPU) al hardware (para su implementación en FPGA), realizando modificaciones menores en el código de alto-nivel (Crockett et al., 2015). En efecto, al desarrollar sistemas complejos, el particionamiento de software/hardware se puede realizar fácilmente con el apoyo de HLS. Según sea necesario, el diseñador puede explorar diferentes particiones funcionales y evaluar alternativas antes de elegir una arquitectura final para el sistema sin requerir largos periodos de tiempo (Cong et al., 2011). Por lo tanto, HLS se ha convertido en una herramienta de diseño eficiente para el desarrollo de aceleradores en hardware.

Aunque las herramientas de HLS parecen mitigar de manera eficiente el problema de crear la descripción del hardware, su generación automática a partir del software sigue sien-

do un proceso complejo y se ha desarrollado una amplia gama de enfoques diferentes (Del Sozzo, Baghdadi, Amarasinghe, & Santambrogio, 2017; Nane et al., 2016). Comprender las investigaciones actuales acerca de HLS requiere de un análisis cuidadoso en la literatura. En el artículo titulado "*A Survey and Evaluation of FPGA High-Level Synthesis Tools*" (Nane et al., 2016), los autores presentan un análisis de las herramientas de HLS actuales, y una metodología donde se evalúa una solución comercial y tres académicas en un conjunto de *benchmarks*[1] en lenguaje C, con el objetivo de realizar un análisis a profundidad en términos de rendimiento y uso de recursos. Resultados adicionales se presentan en el artículo titulado "*High-Level Synthesis for FPGAs: From Prototyping to Deployment*" (Cong et al., 2011), se utiliza la herramienta AutoPilot HLS de AutoESL junto con las plataformas desarrolladas por Xilinx como ejemplo para demostrar la efectividad de las soluciones de síntesis C a FPGA. En el artículo "*LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems*" (Canis et al., 2011), se presenta una nueva herramienta de síntesis de alto-nivel de código abierto llamada LegUp que permite usar técnicas de software para el diseño de hardware. LegUp acepta un programa C estándar como entrada y automáticamente compila el programa en una arquitectura híbrida que contiene un *soft-processor*[2] basado en FPGA y aceleradores de hardware personalizados que se comunican a través de una interfaz de bus estándar. En "*An overview of today's high-level synthesis tools*" (Meeus, Van Beeck, Goedemé, Meel, & Stroobandt, 2012), los autores realizan la evaluación de una amplia selección de herramientas de HLS recientes en términos de capacidades operacionales, usabilidad, calidad de resultados y analizan la adopción de las mismas en flujos de diseño usados por la industria.

Como se observa en los trabajos analizados en el párrafo anterior, cada uno crea diferentes metodologías para evaluar herramientas de HLS. Entre otros, los problemas que se identifican se describen a continuación: i) en (Nane et al., 2016) y (Meeus et al., 2012) se evalúan

[1] Pruebas utilizadas para medir el desempeño de un sistema computacional.
[2] Procesadores modificables e implementables en la lógica programable de FPGA.

herramientas de HLS, su desempeño y recursos en la síntesis de alto-nivel para dos diferentes FPGAs, mas no se analiza el incremento de desempeño que se puede lograr en comparación con otras arquitecturas, ii) en (Cong et al., 2011) se contrasta HLS con métodos HDL para demostrar la reducción de recursos usados en FPGA, pero es necesario un análisis y exploración de un flujo de diseño óptimo para SoCs, iii) en (Canis et al., 2011) se analizan resultados de funciones aceleradas en IP-cores en conjunto con un *soft-processor* implementado en FPGA, sin embargo, es importante una evaluación de desempeño en una arquitectura basada en *hard-core processors*[3] y FPGA en un único SoC, usando herramientas y métodos HLS. En contraste con la mencionada literatura, el presente proyecto pretende diseñar, explorar, y comparar el desempeño de diversas arquitecturas embebidas de SoCs basados en FPGA, donde funciones serán aceleradas tanto en *hard-core processors* (i.e., ARM Cortex), como en IP-cores diseñados mediante herramientas de HLS. Además, se investigará y definirá un flujo de diseño confiable para crear SoCs basados en FPGA orientado a incrementar la automatización en el diseño de IP-cores como aceleradores usando Vivado Design Suite y scripts Tcl (*Tool Command Language*).

1.2. Justificación e Importancia

Esta propuesta de proyecto nació de la necesidad de buscar métodos de desarrollo eficientes de SoCs basados en FPGA, usando HLS para generar IP-cores de coprocesamiento. De hecho, esta es una tendencia en computación de alto desempeño, tema abordado en el proyecto "Embedded Heterogeneous Super Computing" (eHSC), al que pertenece y busca aportar la investigación y trabajo de este proyecto de titulación; cuya iniciativa e idea fue propuesta por el director de este proyecto. Además, es un tema de interés personal del estudiante responsable del mismo, quien ve la importancia de HLS para los sistemas embebidos de alto desempeño.

[3]Procesadores no modificables implementados en circuito integrado.

El uso de arquitecturas complejas de procesamiento basadas en hardware para aplicaciones de alto rendimiento no es una técnica nueva y ha sido utilizada en las últimas décadas. Sin embargo, muchos ingenieros evitan el uso de FPGAs debido a que se requiere un alto grado de conocimiento en diseño de hardware, lo cual abarca lenguajes HDL (VHDL o Verilog), arquitecturas de FPGAs, métodos de diseño y manejo de las herramientas computacionales disponibles (Windh et al., 2015). Esto hizo que fuera extremadamente difícil para estas plataformas ser ampliamente utilizadas e hizo del desarrollo en FPGAs una tecnología restringida a un pequeño grupo de programadores de hardware (Bacon, Rabbah, & Shukla, 2013; Rodríguez, A.; Valverde, J.; Portilla, J.; Otero, A.; Riesgo, 2018). El crecimiento y necesidad de sistemas cada vez más complejos basados en FPGAs requiere gran cantidad de tiempo y recursos para el desarrollo y verificación de diseños que involucran el uso de lenguajes HDL. El requerimiento de tiempo es aún mayor cuando es necesaria la creación y verificación de sistemas multiprocesador que incluyan IP-cores, donde la interconexión y comunicación entre los diferentes elementos es altamente compleja (Kazmierkowski, 2009). Precisamente esta complejidad de diseños ha hecho menos viable el continuar usando lenguajes HDL, por lo tanto, la exploración de métodos que permitan a los desarrolladores trabajar en un alto-nivel de abstracción como HLS es requerido.

Las metodologías y herramientas de HLS se encuentran todavía en desarrollo y son un tema de investigación actual. Por lo tanto, muchas soluciones son aún pruebas de concepto o herramientas de evaluación académicas como por ejemplo, LegUp, DWARV y BAMBU (Nane et al., 2016). Otras herramientas y flujos de diseño con un mayor grado de confiabilidad son soluciones propietarias y altamente costosas, por ejemplo, Stratus HLS (Qin & Berekovic, 2015). Existen también soluciones HLS en Matlab y Simulink, que proporcionan flujos automatizados para un conjunto limitado de librerías y placas FPGA de terceros (Navas, Öberg, & Sander, 2013). Por otro lado, Xilinx HLS es la solución comercial más usada, pero es muy reciente. Por ello, es necesario definir un flujo de diseño adecuado que permita acelerar y mejorar los métodos de desarrollo para SoCs basados en FPGA, más aún si es un tema no

muy ampliamente documentado, especialmente en nuestro medio.

De acuerdo a la revisión del estado del arte en los repositorios digitales de las universidades del Ecuador y el análisis previo de esta sección es necesaria una documentación sobre HLS y diseños de referencia que aporten a trabajos de investigación futuros, los cuales requieran un nivel de complejidad mayor, tomando en cuenta que HLS es un tópico de interés y enfoque de diseño actual en programas de investigación a nivel mundial conforme a los artículos científicos citados.

1.3. Alcance del Proyecto

La finalidad de este proyecto de investigación es acelerar y mejorar los métodos para el desarrollo, verificación y optimización de SoCs basados en FPGA usando HLS, en lugar de los métodos de diseño tradicionales que usan lenguajes HDL de forma exclusiva. Cabe señalar que la optimización en el diseño de SoCs embebidos de alto desempeño busca priorizar la reducción de latencias y mejorar el rendimiento con los recursos disponibles, usando técnicas como *pipelining* y *dataflow*. Entre otras, las principales contribuciones del proyecto son: i) documentación de los conceptos y métodos de HLS. ii) diseño, exploración y comparación de desempeño de diversas arquitecturas embebidas de SoCs basadas en FPGA, donde funciones serán aceleradas tanto en *hard-core processors* como en IP-cores diseñados mediante herramientas de HLS. iii) investigación y definición de un flujo de diseño confiable para crear SoCs basados en FPGA. iv) incremento del nivel de automatización en el diseño de IP-cores como aceleradores usando Vivado Design Suite y scripts Tcl.

Se realizará la exploración y comparación de desempeño de diferentes arquitecturas, tanto en *hard-core processors* (i.e., ARM Cortex), como en SoCs basados en FPGA (i.e., Xilinx Zynq) con IP-cores diseñados mediante HLS. Para la comparación se utilizará al menos cinco *benchmarks* típicos para sistemas embebidos y de control usados a nivel académico y científico. La selección de estos algoritmos se realizará tomando en cuenta que aplicaciones

del futuro requerirían de sistemas embebidos y de control que serán cada vez más complejos, demandando procesamiento de datos provenientes de múltiples sensores, análisis de grandes cantidades de datos en tiempo real, técnicas de control inteligente, manejo de comunicaciones y además un consumo energético reducido. Es por ello que en esta tesis se seleccionarán algoritmos relacionados a estas aplicaciones para explorar su desempeño en arquitecturas heterogéneas basadas en FPGA. En el diseño de SoCs se realizará un particionamiento de software/hardware, explorando la arquitectura que permita un mejor rendimiento respecto a una implementación en software en *hard-core processor*. HLS será utilizado para el desarrollo de hardware en FPGA (IP-cores) para la aceleración de pruebas de rendimiento. Se hará uso de directivas y restricciones para influenciar la síntesis de alto nivel, a fin de obtener la solución con mejor desempeño de diferentes implementaciones de la misma arquitectura.

Se efectuará la investigación y definición de un flujo de diseño confiable para crear SoCs basados en FPGA con funciones aceleradas en IP-cores, definidos en C++ usando herramientas de Xilinx. Para ello, se crearán scripts Tcl que permitan incrementar el nivel de automatización en la generación de IP-cores en Vivado Design Suite.

El alcance del proyecto es a nivel nacional y regional, sirviendo como referencia para el proyecto de investigación eHSC y futuros proyectos del DEEE relacionados al área de conocimiento y líneas de investigación (e.g., sistemas electrónicos y computacionales). El proyecto de investigación es de tipo autofinanciado, usando herramientas de acceso libre y tarjetas de desarrollo con Zynq-7000 de Xilinx, provistas por el director y el estudiante responsable de este proyecto. Las contribuciones de la investigación se concentran en la generación de documentación, métodos, scripts, software, descripción de hardware para FPGA disponibles como *soft* IP-cores[4] y diseños de referencia.

[4]IP-cores reconfigurables, generalmente descritos en HDL, e implementables en la lógica programable de una FPGA.

1.4. Objetivos

1.4.1. Objetivo General

Acelerar y mejorar los métodos de desarrollo, verificación y optimización de sistemas SoC basados en FPGA usando niveles superiores de abstracción, i.e., lenguaje C++, en lugar de los tradicionales lenguajes de descripción de hardware (HDL).

1.4.2. Objetivos Específicos

- Estudiar y documentar los métodos de HLS (*High-Level Synthesis*) actuales para SoCs, incluyendo teoría sobre HLS.
- Acelerar en IP-cores (hardware) al menos cinco *benchmarks* típicos para aplicaciones de sistemas de control y sistemas embebidos, y explorar su desempeño comparado con implementaciones de software en *hard-core processors*; los cuales servirán como diseño de referencia en el proyecto eHSC.
- Investigar y definir un flujo de diseño confiable para crear SoCs basados en FPGA con funciones aceleradas en IP-cores, definidos en C++, usando herramientas de Xilinx.
- Incrementar el nivel de automatización en la generación de sistemas embebidos SoCs basados en FPGA, usando el flujo de diseño HLS de Xilinx Vivado Design Suite y scripts Tcl.

Capítulo 2

Conceptos y Estado del Arte

En este capítulo se presentan conceptos básicos acerca de FPGAs y SoCs para facilitar la comprensión del texto. Después, se presenta el estudio de investigaciones previas acerca de sistemas embebidos de alto rendimiento basados en FPGA y el uso de altos niveles de abstracción y HLS para vencer la complejidad actual en el diseño de SoCs. Finalmente, se introducen conceptos, teoría y herramientas actuales de HLS.

2.1. Conceptos

2.1.1. Field-Programmable Gate Array (FPGA)

Los FPGAs son dispositivos semiconductores reconfigurables que permiten desarrollar hardware personalizado. El diseñador puede configurar[1] sus bloques de lógica e interconexiones para crear sistemas únicos mediante lenguajes especializados. La tecnología FPGA o lógica programable[2] es una plataforma perfecta para el desarrollo de sistemas digitales cuando se requiere un balance entre flexibilidad, aceleración en la ejecución y reducción del consumo

[1]En la literatura se usa las palabras configuración o programación para referirse a la creación de sistemas en FPGA. Sin embargo, es necesario aclarar que el término correcto es configurable, ya que en FPGA no existe un programa en memoria cuyas instrucciones se ejecuten en un procesador.

[2]Lógica programable es un término muy usado para referirse a cualquier dispositivo reconfigurable ya sea PLD, CPLD, o FPGA.

energético. Gracias a su desarrollo, el número de puertas lógicas en FPGAs se ha incrementado considerablemente permitiendo crear sistemas completos en ellos. Por consiguiente, hoy en día es posible implementar varios núcleos de procesamiento, memorias, controladores de periféricos e interconexiones.

La arquitectura de los FPGAs de Xilinx consiste en arreglos bidimensionales de bloques de lógica configurables o CLBs (*Configurable Logic Blocks*), interconexiones programables entre ellos, y bloques de entradas y salidas. Una representación simplificada de la arquitectura de un FPGA se muestra en la Figura 1. Los CLBs son los elementos básicos que lo conforman. Estos proveen lógica fundamental y recursos para almacenamiento (Farooq, Marrakchi, & Mehrez, 2012). Cada CLB está compuesto de elementos denominados *Slice*, que contienen a su vez LUTs (*Look-up Tables*) y Flip-Flops (Crockett, Elliot, Enderwitz, & Stewart, 2015). Cada LUT puede implementar pequeñas funciones lógicas, memorias ROM (*Read Only Memory*), RAM (*Random Access Memory*), o registros de desplazamiento (Crockett et al., 2015). Además, las FPGAs pueden contener recursos adicionales como DSPs (*Digital Signal Processors*) y bloques de memoria RAM.

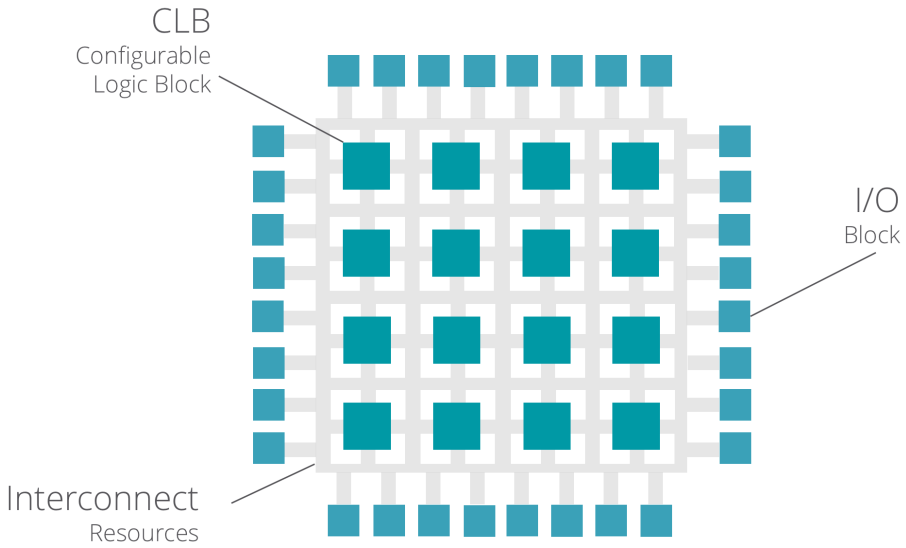


Figura 1. Arquitectura básica de un FPGA. Los elementos típicos que lo conforman son CLBs, interconexiones programables, y bloques de entradas y salidas.

Sin llegar a alcanzar el desempeño de un ASIC (*Application Specific Integrated Circuit*), los FPGAs se han popularizado por otras ventajas. El diseñar en FPGA implica una alta flexibilidad, siendo esta la principal motivación para su uso. Otras ventajas son el menor costo de diseño para volúmenes de producción menores y menor *time-to-market* en comparación con ASICs. Además, su inherente paralelismo y posibilidad de reconfiguración parcial mientras se encuentra en funcionamiento (*partial and run-time reconfiguration*) lo hacen atractivo para varias aplicaciones (Farooq et al., 2012; Navas et al., 2013). Debido a su importancia, los FPGAs son actualmente considerados partes fundamentales para el desarrollo de muchos sistemas de alto desempeño, y han sido incorporados en SoCs programables y arquitecturas heterogéneas.

2.1.2. System-on-Chip (SoC)

Un SoC puede ser definido como un IC (Circuito Integrado) complejo que abarca todos los componentes funcionales de un sistema computacional embebido. Por lo tanto, remplaza la combinación de diferentes ICs físicamente separados. Un SoC está formado por componentes que se denominan IP-cores, elegidos según la aplicación, y que pueden ser implementados en hardware y software (Sinha, Roop, Salcic, & Basu, 2014). Además, necesariamente debe contar con elementos que permitan la comunicación entre todos sus IP-cores.

Las arquitecturas de SoCs pueden ser tan variadas como sus aplicaciones, pero todo SoC está constituido tanto de software como de hardware. El software comprende aplicaciones sobre algún tipo de OS (Sistema Operativo) o aplicaciones *bare-metal*[3] ejecutadas por un procesador. Por otro lado, el hardware está compuesto del propio procesador, memorias, y periféricos. La comunicación entre software y hardware es llevada a cabo mediante un sistema de interconexión haciendo uso de un determinado protocolo. El sistema de interconexión puede estar constituido por un conjunto de canales directos, o un bus usado por múltiples componentes (Crockett et al., 2015). Una representación de la arquitectura básica de un SoC

[3]Firmware que interactúa directamente con el hardware sin niveles de abstracción superiores.

se muestra en la Figura 2.

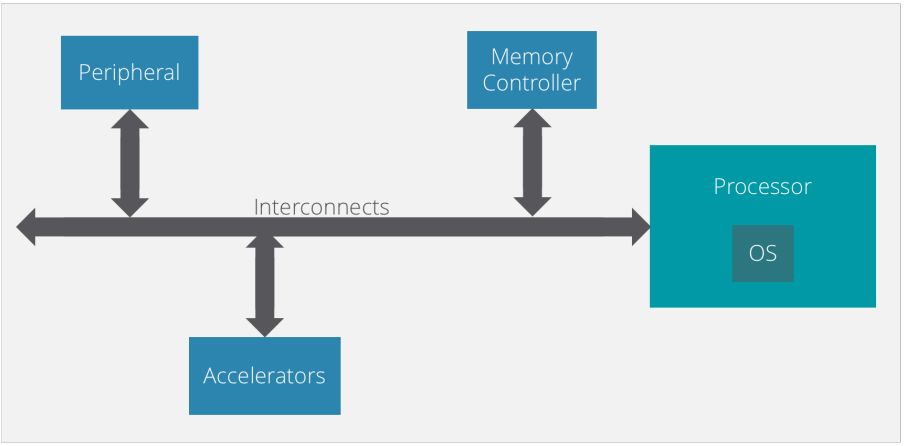


Figura 2. Arquitectura básica de un SoC. Los SoCs estan generalmente compuestos por un procesador, memoria, periféricos e interconexiones.

La acogida de SoCs en las últimas décadas ha conllevado su rápida evolución, potenciada por el uso cada vez mayor de dispositivos móviles. Ya desde hace un tiempo es común que muchos SoCs incluyan procesadores programables RISC (*Reduced Instruction Set Computer*), DSPs, canales de comunicación de alta velocidad, memorias y periféricos (Zurawski, 2004). Hoy en día, la evolución de los mismos ha permitido crear SoCs multi-procesador (MPSoCs), los cuales incluyen muchos o todos los elementos requeridos en una aplicación que hace uso de varios tipos de procesadores (Wolf, Jerraya, & Martin, 2008).

2.2. SoCs Basados en FPGA

El desarrollo de FPGAs en las últimas décadas se ha visto marcado por la creciente tendencia de incluir bloques de hardware dedicado, tales como memorias, unidades DSP, y más recientemente, procesadores embebidos en un único SoC (Fort et al., 2014). FPGA-SoCs o SoCs basados en FPGA son dispositivos que contienen un *hard-core processor* y lógica programable en un mismo chip, lo que les permite estar estrechamente acoplados (Göbel, Elhossini, Chi, Alvarez-Mesa, & Juurlink, 2017). Esto posibilita crear sistemas híbridos de

hardware y software con aceleradores que mejoren el desempeño y la eficiencia energética en aplicaciones de gran costo computacional.

El uso de procesadores en conjunto con FPGAs para sistemas computacionales no es reciente. Estos sistemas híbridos nacieron en los años 2000 ante la necesidad de procesamiento tanto de algoritmos o aplicaciones de software (ideales para un procesador dedicado), como de cálculos de intensos flujos de datos (adecuado para FPGA), donde los procesadores no son eficientes. Por lo tanto, se requería de una solución con un mayor desempeño comparado con el ofrecido por implementaciones puramente de software (Crockett et al., 2015). Además, es bien conocido que implementaciones de ciertos algoritmos en hardware pueden ofrecer mejores resultados comparados a su ejecución en software, sobre todo si se aprovecha niveles de paralelismo.

FPGAs y procesadores han sido usados durante algún tiempo como elementos físicamente separados. Uno de estos primeros sistemas fue el Atmel AT94K, el cual era un *System-on-Board*[4] (SoB) que incluía un procesador de 8-bits RISC. Sin embargo, la desventaja de estas arquitecturas es el tiempo necesario para transferir datos e instrucciones entre software y hardware, constituyendo una latencia adicional que se compensa con la aceleración del procesamiento (Crockett et al., 2015). Además, estas configuraciones pueden estar limitadas por el uso de un bus estándar, lo cual constituye una gran desventaja.

Otro enfoque usado por mucho tiempo ha sido la integración de *soft-core processors* en la lógica programable de FPGA. Estos procesadores son provistos por los desarrolladores, Xilinx y Altera, tales como MicroBlaze (2002) y Nios II (2004) respectivamente, o desarrollados por terceros (por ejemplo, OpenSPARC de Sun Microsystems, y LatticeMico32 de Lattice Semiconductor). Sin embargo, el desempeño ofrecido por estos es limitado y depende de la familia de FPGA utilizada (Fort et al., 2014; Weber & Chin, 2006). Los *hard-core processors* ofrecen un mejor desempeño en comparación con *soft-processors*, pudiendo ser tres o cuatro veces más rápidos (Fletcher, 2005). Por ejemplo, un SoC típico con MicroBlaze puede alcanzar

[4]Sistema computacional que abarca todos sus componentes en una placa de circuito impreso.

200 o 300 MHz dependiendo del FPGA escogido, velocidad muy inferior en comparación a la de un SoC con un ARM Cortex-A9 que alcanza entre 600 MHz a 1 GHz (Crockett et al., 2015).

El desarrollo de FPGA-SoCs ha sido clave para superar las limitaciones descritas en las anteriores configuraciones. Los dos grandes desarrolladores de FPGAs, Xilinx y Altera[5], han dedicado esfuerzos al desarrollo de SoCs basados en FPGA debido a la importancia de los mismos para cumplir las exigencias de aplicaciones de alto desempeño (Fletcher, 2005; Fort et al., 2014). De hecho, ya en el año 2002 Xilinx lanzó al mercado uno de los primeros FPGA-SoCs, el Virtex-II Pro, el cual incluía un *hard-core processor* IBM PowerPC, que después también estaría presente en Virtex-4 y Virtex-5, siendo estos los predecesores de tecnologías actuales. Hoy en día, ambos desarrolladores producen familias de FPGAs que incorporan un procesador físico en una sección dedicada del chip. Por ejemplo, procesadores de 32-bits *dual-core* ARM Cortex-A9 incluidos en la familia Zynq-7000 de Xilinx y Cyclone V de Altera; y más recientemente procesadores de 64-bits *quad-core* ARM Cortex-A53 en los Xilinx UltraScale+ MPSoCs e Intel Stratix 10 SoCs (Göbel et al., 2017). Para ilustrar las arquitecturas discutidas en esta sección se presenta la Figura 3, mientras que en la Tabla 1 se resumen ejemplos significativos, de tipo comercial, en el desarrollo de sistemas computacionales basados en FPGA.

Las ventajas de embeber un procesador dentro de un FPGA son varias. Principalmente se obtiene gran flexibilidad de diseño combinada con la posibilidad de realizar transferencias de datos entre hardware y software a fin de maximizar la eficiencia y el desempeño de SoCs. Un algoritmo de software que requiera intensos cálculos computacionales resultará en un cuello de botella. Este tipo de algoritmos pueden ser trasladados a hardware para su ejecución en un acelerador de coprocesamiento. Los FPGA-SoCs permiten conectar estos aceleradores al *hard-core processor* a través de canales de baja latencia. Con las herramientas de diseño actuales para FPGAs el proceso para trasladar secciones de software a hardware es mucho

[5]Intel adquirió Altera en 2015 debido a la importancia de FPGAs en sistemas computacionales actuales.

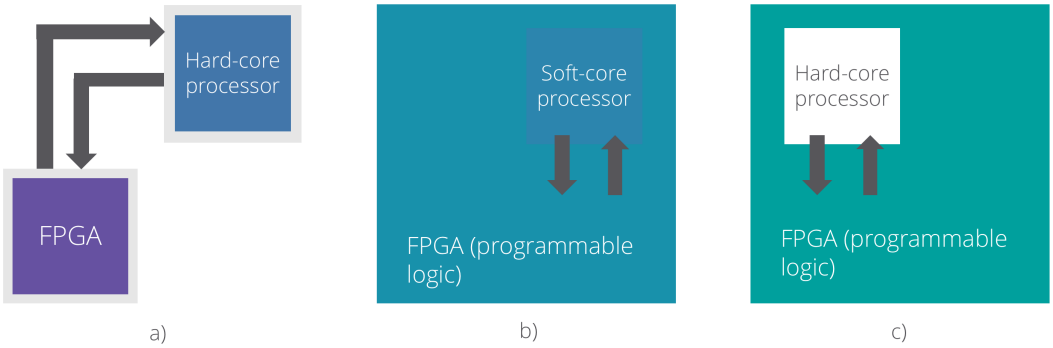


Figura 3. Arquitecturas de sistemas computacionales embebidos basados en FPGA. a) FPGA con hard-core processor como elementos discretos, b) FPGA con soft-core processor implementado en lógica programable, c) FPGA con hard-core processor en un único chip.

Tabla 1

Ejemplos significativos en el desarrollo de sistemas computacionales embebidos basados en FPGA

Dispositivo	Procesador	Tipo	Arquitectura	Velocidad	Año
Atmel AT94K	8-bit RISC AVR	Hard-core	SoB	25 MHz	2001
Xilinx Virtex-II Pro	MicroBlaze	Soft-core	FPGA	150 MHz	2002
Altera Excalibur	ARM922T	Hard-core	FPGA-SoC	200 MHz	2002
Xilinx Virtex-II Pro	IBM PowerPC	Hard-core	FPGA-SoC	450 MHz	2002
Altera Stratix II	NIOS	Soft-core	FPGA	180 MHz	2004
Microsemi SmartFusion	ARM Cortex-M3	Hard-core	FPGA-SoC	100 MHz	2010
Altera Arria 10	ARM Cortex-A9	Hard-core	FPGA-SoC	800 MHz	2013
Xilinx Zynq-7100	ARM Cortex-A9	Hard-core	FPGA-SoC	667 MHz	2013
Altera Stratix 10	ARM Cortex-A53	Hard-core	FPGA-SoC	1.5 GHz	2015
Xilinx Zynq UltraScale	ARM Cortex-A53	Hard-core	FPGA-SoC	1.5 GHz	2016

más simple gracias a la posibilidad de adaptar código C/C++ para su síntesis a hardware usando HLS (Fletcher, 2005). Otra ventaja significativa es la simplificación del sistema a nivel de placa, reduciendo el número de componentes y aumentando la confiabilidad del sistema. Esto contribuye directamente a una significativa reducción de consumo energético debido a que las conexiones internas entre la lógica programable y el procesador consumen menos energía comparado con los enlaces externos en un sistema de componentes físicamente separados (Avelino et al., 2017; Crockett et al., 2015).

2.3. Sistemas Embebidos de Alto Desempeño Basados en FPGA

Sistemas basados en FPGA se usan cada vez más para la implementación de aplicaciones críticas, donde es necesario lograr altos niveles de aceleración tanto en hardware como en software. Son varios los campos de aplicación que requieren sistemas embebidos de alto desempeño, entre ellos se encuentran el procesamiento de señales e imágenes, vehículos no tripulados, sistemas de aviónica, IoT (*Internet of Things*), entre otras (de Oliveira, Tambara, & Kastensmidt, 2017). Por ejemplo, la necesidad de procesamiento en tiempo real en dispositivos de imágenes médicas ha conducido a evaluar la posibilidad de utilizar la aceleración de FPGA para proporcionar los recursos computacionales necesarios (Hoozemans, Heij, Straten, & Al-Ars, 2017). De hecho, el uso de coprocesadores en FPGA es particularmente efectivo como solución a estos nuevos retos, siendo una plataforma ideal para la implementación de paralelismo masivo y procesadores en tiempo real (Hoozemans et al., 2017; S. Zhou, Jiang, & Prasanna, 2014). Por ejemplo, en procesamiento de imágenes, donde muchas operaciones matemáticas son llevadas a cabo en un gran número de muestras o pixeles de forma simultánea. A pesar de que existen recursos específicos disponibles en algunos procesadores para llevar a cabo este tipo de aplicaciones (tales como NEON en los procesadores ARM Cortex-A9), su desempeño no necesariamente alcanza el nivel de optimización de hardware dedicado para dichas tareas (Crockett et al., 2015).

Los sistemas embebidos pueden aprovechar las ventajas de FPGAs para lograr un mejor desempeño. Al analizar el rendimiento general de un procesador en una determinada aplicación, se debe tomar en cuenta el desempeño específico obtenido en cada tarea para poder identificar posibles cuellos de botella. Por ejemplo, si un procesador ocupa una gran cantidad de ciclos de ejecución de CPU para realizar una tarea específica, se debe investigar si dicha tarea es susceptible a acelerarse usando algún grado de paralelismo (Crockett et al., 2015). Si esto es posible, entonces esta tarea es candidata a ser implementada en hardware, lo

cual produciría una mejora significativa en el desempeño general del sistema. Debido a esto, la creciente tendencia en la industria de sistemas embebidos es usar los FPGAs para crear aceleradores de hardware o coprocesadores en arquitecturas heterogéneas como complemento de CPUs y GPUs (Choi et al., 2016; Podobas, Zohouri, Maruyama, & Matsuoka, 2017). Un ejemplo son los Zynq UltraScale MPSoCs los cuales son SoCs multiprocesador heterogéneos, que combinan procesadores ARM Cortex-A53, GPUs ARM Mali-400 MP2 y lógica programable de FPGA en un único chip. Una representación de un SoC de arquitectura heterogénea se muestra en la Figura 4.

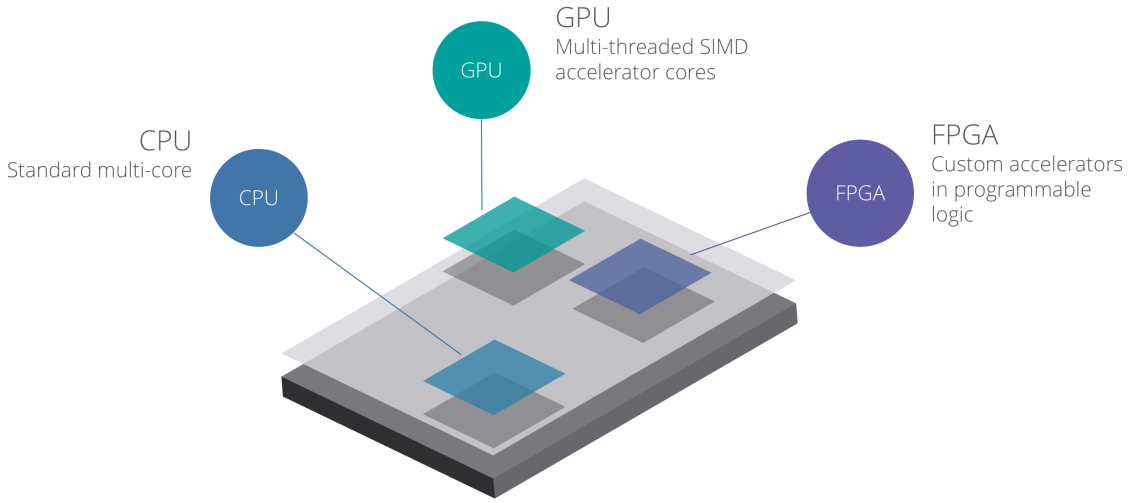


Figura 4. SoC de arquitectura heterogénea. Este SoC consta de: CPU multi-núcleo, GPU y FPGA con aceleradores implementados en hardware en un único chip.

El uso de FPGA-SoCs para aplicaciones de alto desempeño se ha incrementado a lo largo de la última década. Una de las principales motivaciones para diseñadores ha sido la aparición de herramientas de diseño que hacen uso de descripciones de alto-nivel. Se puede encontrar numerosos ejemplos en la literatura, tal como el artículo "*FPGA-Based High-Performance Embedded Systems for Adaptive Edge Computing in Cyber-Physical Systems: The ARTICo Framework*" (Rodríguez, A.; Valverde, J.; Portilla, J.; Otero, A.; Riesgo, 2018), donde se presenta una referencia de diseño para el desarrollo de sistemas embebidos de alto rendimiento basados en FPGA para *Edge Computing* en sistemas Ciber-Físicos, donde se hace uso de

varias plataformas FPGA tanto con *soft-processors* y *hard-processors* (específicamente ARM Cortex-A9 en Zynq). En el artículo "*High-Level Design using Intel FPGA OpenCL: a Hyperspectral Imaging Spatial-Spectral Classifier*" (Domingo et al., 2017), se revisan estrategias de optimización y evaluación de aceleradores clasificadores de imágenes hiperespectrales usando un Cyclone V SoC de Intel basado en FPGA y *hard-processors* ARM. En "*Zedwulf: Power-Performance Tradeoffs of a 32-node Zynq SoC cluster*" (Moorthy & Kapre, 2015), se presenta un prototipo de clúster de 32 nodos compuesto a partir de SoCs Zynq para acelerar las aplicaciones orientadas a gráficos dispersos ligados a la comunicación, como las simulaciones de redes neuronales. Mientras que en el artículo "*High-Performance Low-Energy Implementation of Cryptographic Algorithms on a Programmable SoC for IoT Devices*" (B. Zhou, Egele, & Joshi, 2017), se usa un FPGA reconfigurable para operaciones criptográficas usando una tarjeta de desarrollo Zedboard, la cual posee dos núcleos ARM y un FPGA Zynq. Como se puede observar en la literatura, existe una tendencia clara y prometedora en el uso FPGAs en sistemas embebidos de alto rendimiento. Sin embargo, a pesar de que actualmente existen herramientas HLS, la complejidad es aún elevada y abarca retos significativos. El desarrollo de herramientas de descripción de alto-nivel que simplifiquen el diseño, integración y verificación de sistemas SoCs heterogéneos es un tema todavía en investigación.

2.4. Flujo de Diseño de SoCs

Varias metodologías se han planteado como referencia para el diseño de SoCs desde su creación, sin embargo, hasta la actualidad no existe una única metodología. En esta sección se revisa de forma breve un flujo de diseño simple con seis etapas basado en el análisis de metodologías usadas a nivel académico y de industria. También se hace hincapié en la importancia de la reutilización de IP-cores como estrategia en la industria de los sistemas embebidos para lograr una mayor productividad.

Considerando las etapas fundamentales durante el proceso de desarrollo de SoCs, se pre-

senta un flujo de diseño en la Figura 5, el mismo que se describe a continuación mediante etapas: i) El primer paso en el diseño de SoCs, representado en la parte superior, es el análisis de requerimientos. Este análisis parte de la definición tanto de la funcionalidad deseada como de requerimientos específicos, en la mayoría de los casos: el costo, consumo energético, desempeño y latencia máxima (Zurawski, 2004). ii) En base a esta descripción, el diseñador define una especificación abstracta del sistema que determina su comportamiento, funcionalidad y capacidades. Siendo esta la base durante el proceso de diseño del SoC. iii) En la tercera etapa, se explora la arquitectura adecuada para el SoC requerido. Esto incluye la selección de procesadores, dispositivos de entrada y salida, elementos de memoria e interconexión entre los diferentes componentes. Dentro de los procesadores escogidos se pueden encontrar procesadores que ejecuten software embebido o aceleradores para funciones específicas creados en hardware programable (Lin, 2006). iv) Se realiza la partición entre hardware y software para la arquitectura escogida. Esta etapa puede ser entendida como un ciclo realimentado en el cual se analiza el desempeño del sistema, permitiendo a los diseñadores realizar mejoras durante el proceso hasta encontrar el particionamiento con el que se obtenga un resultado óptimo. Esta metodología es conocida como co-diseño y es ampliamente usada en el desarrollo de sistemas embebidos. v) En esta etapa se diseña el hardware y software de forma concurrente. Para que el equipo de desarrollo de software pueda trabajar de forma simultanea se usan capas de abstracción de hardware (Zurawski, 2004). vi) La integración del hardware y software es evaluada en base a pruebas de desempeño, verificando que se cumpla con las métricas requeridas para asegurar la funcionalidad deseada.

El diseño de SoCs involucra cumplir con demandantes exigencias de la industria. Existe una constante presión debido a aspectos de reducción de costos y tiempos de lanzamiento (*time-to-market*). Estas exigencias sumadas a la complejidad de sistemas actuales, ha forzado a desarrolladores a buscar formas de incrementar la productividad del flujo de diseño discutido en el párrafo anterior (Abdurohman, Kuspriyanto, Sutikno, & Sasongko, 2010). Existen básicamente dos enfoques para mejorar la productividad: i) reusar partes tanto de software

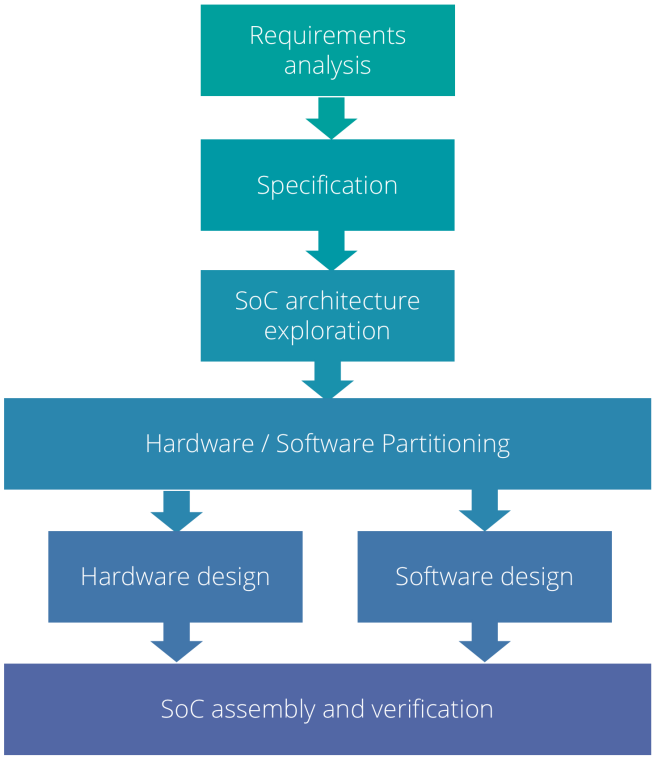


Figura 5. Flujo de diseño de SoCs.

como de hardware, y ii) elevar el nivel de abstracción (Bobda, Mead, Whitaker, Kamhoua, & Kwiat, 2017; Crockett et al., 2015). El primero es discutido a continuación, mientras que el segundo enfoque se trata en secciones subsecuentes.

La reutilización de IP-cores disponibles tanto como *soft-cores* y *hard-cores* es una práctica común y una filosofía dentro del flujo de diseño de sistemas embebidos modernos. Esta práctica tiene varias ventajas, por un lado, una significativa reducción del tiempo necesario para el diseño, y por otro lado, el uso de IP-cores diseñados por terceros garantiza su funcionalidad ahorrando tiempo destinado a su verificación (Crockett et al., 2015). Sin embargo, al momento de diseñar el SoC se debe analizar que IP-cores pueden ser adquiridos a terceros, lo cual depende de varios factores: performance, consumo energético, área de utilización, y flexibilidad de configuración de los mismos. En caso de que estos no cumplan con los requerimientos de la aplicación se tendrá que diseñar un nuevo IP, lo que conlleva un inevitable incremento de recursos y tiempo de diseño (Lin, 2006).

2.5. Complejidad en el Diseño de SoCs y Uso de Altos Niveles de Abstracción

La complejidad en el diseño de SoCs radica en varios aspectos. Los procesos de diseño, verificación e integración incrementan la complejidad de su desarrollo como se ilustra en la Figura 6. Entre los principales problemas que enfrenta un diseñador se encuentran: i) La implementación física implica colocar billones de transistores dentro del chip, venciendo restricciones físicas del silicio. ii) La falta de componentes (IP-cores) reusables exige diseñarlos desde cero a partir de descripciones de bajo nivel, lo cual requiere de diseñadores expertos en hardware. iii) El manejo de comunicaciones a nivel de chip es extremadamente complicado por el gran tráfico de señales. Es necesario usar subsistemas *Network-on-Chip*[6] (NoC) dentro de SoCs. iv) La integración y verificación de SoCs implican simulaciones conjuntas de software/hardware complejas y demandantes de tiempo. v) Es necesario cumplir con un *time-to-market* exigente en la industria.

El diseño de SoCs ha sido siempre un proceso complejo y multidisciplinario (Zurawski, 2004). Esta complejidad se ha visto incrementada junto con el deseo de aumentar el poder computacional de estos dispositivos, de los que además se espera que sean cada vez más pequeños, menos costosos, más eficientes energéticamente, más confiables, etc (Lin, 2006). Como se revisó en secciones anteriores, muchos SoCs actuales son sistemas multi-núcleo o MPSoCs (*Multi-processor System-on-Chip*) que además brindan una gran cantidad de servicios. Es por ello que hoy en día el diseño de SoCs requiere de desarrolladores de hardware y software trabajando de forma concurrente dentro de una metodología altamente productiva.

La necesidad del diseñador de especificar la funcionalidad y comportamiento del sistema en un bajo nivel de abstracción implica varios problemas. Entre estos problemas se tiene la demanda de tiempo y recursos, además, si el SoC a ser diseñado es complejo, la productividad se verá inevitablemente afectada siguiendo esta metodología. Por ejemplo, el uso de FPGA

[6]Subsistema basado en red para manejar la comunicación intra-chip de un SoC.

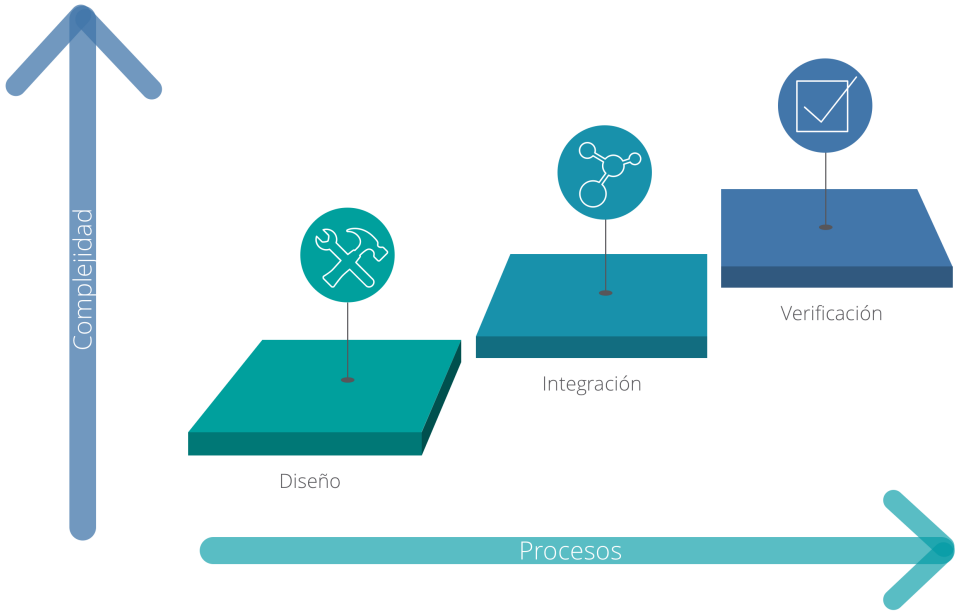


Figura 6. Complejidad del diseño de SoCs.

para crear aceleradores ofrece muchas ventajas ya descritas, pero su adopción ha sido limitada debido al tiempo que consume un diseño usando lenguajes HDL como VHDL, Verilog, SystemC y SystemVerilog, en donde se realiza descripciones de muy bajo nivel (Windh et al., 2015). Otro aspecto complejo en diseño de SoCs es la comunicación interna entre los diferentes componentes, involucrando el uso de protocolos y topologías en donde es necesaria una sofisticada sincronización, sobre todo cuando se usan arquitecturas heterogéneas (Zurawski, 2004). Además, una vez que se tienen todos los componentes integrados formando un sistema, es necesaria una verificación de su funcionalidad, lo cual se realiza mediante una simulación conjunta de software y hardware, sin embargo, si el SoC es complejo la simulación puede no proporcionar un nivel de confianza suficiente (Lin, 2006). Estos retos, han obligado a la comunidad científica a buscar una abstracción de diseño que ofrezca una mejor productividad tanto en el diseño de software y hardware (Cong et al., 2011).

El reto actual es acelerar y automatizar la creación de SoCs. Precisamente la elevación del nivel de abstracción incluyendo a HLS ofrece soluciones para el diseño, integración y verificación de sistemas altamente complejos. La finalidad del uso de altos niveles de abstracción

es lograr un proceso de diseño mucho más productivo. Es por ello que muchas investigaciones actuales, tanto a nivel académico como comercial, centran su atención en herramientas de alto-nivel. Es bien conocido que usando lenguajes HLL (*High-Level Languages*) se puede lograr mejores resultados tanto en el desarrollo de hardware y software comparados con los alcanzados usando lenguajes que involucren el conocimiento de detalles complejos de diseño. Además, en las últimas dos décadas se ha generado mucho interés por el estudio de metodologías *top-down* de automatización de diseño de SoCs. Un ejemplo de ello es el interés actual en el diseño electrónico a nivel de sistema o ESL *design* (*Electronic system-level design*) para SoCs complejos y MPSoCs (Coussy et al., 2009). ESL define una metodología de automatización de la síntesis a nivel de sistema, incrementando el nivel de abstracción para su descripción y verificación mediante lenguajes como C, C++ y SystemC (Nikolov et al., 2008). De hecho, HLS tiene un rol central en el diseño de SoCs actuales, reduciendo no solo el tiempo en la creación de hardware sino también el de su verificación (Coussy et al., 2009). Además, la automatización del proceso ayuda a eliminar la inserción de errores que se generan en un proceso manual (Fingeroff, 2010). En resumen, las herramientas de diseño de alto-nivel ofrecen grandes oportunidades, convirtiéndose en partes vitales del desarrollo de sistemas complejos.

2.6. High-Level Synthesis (HLS)

HLS es una de las principales apuestas para el desarrollo de FPGAs y ASICs, y una de las grandes promesas para automatizar el desarrollo de futuros sistemas computacionales. La capacidad de generar implementaciones RTL (*Register-Transfer Level*) a partir de descripciones de alto-nivel ha cautivado la atención de la comunidad científica particularmente en las últimas décadas. Hoy en día se continua con la investigación y desarrollo de herramientas y metodologías que exploten sus potencialidades. En esta sección se revisan conceptos fundamentales sobre HLS como las representaciones gráficas intermedias, la compilación, *allocation*,

scheduling, *binding* y generación de RTL. Estos conceptos permiten tener una visión global del proceso. Además, se revisa de forma breve herramientas comerciales relevantes, así como importantes investigaciones académicas en desarrollo.

2.6.1. Control Data Flow Graph (CDFG)

El proceso de HLS inicia con una representación intermedia de alto-nivel denominada Gráfico de Flujo de Control y Datos (CDFG). El CDFG captura la descripción de comportamiento de alto-nivel en dependencias de control y datos, las cuales son abstraídas como un controlador y un datapath. Un CDFG se crea a partir de la combinación de dos gráficos, CFG (*Control Flow Graph*) y DFG (*Data Flow Graph*). La construcción de cada uno y su combinación se explican a continuación.

2.6.1.1. CFG (Control Flow Graph)

El gráfico de flujo de control está compuesto por nodos y flechas que representan las funciones básicas y el flujo de control entre nodos (Koch, Hannig, & Ziener, 2016). Cada nodo representa una instrucción o conjunto de instrucciones en las cuales no existe un punto de divergencia, mientras que cada flecha simboliza dependencias en el orden de ejecuciones. A continuación, se presenta la función *ejemplo* en el Código 1, la cual será usada para ilustrar la construcción del CDFG. En base a esta función se ha realizado el CFG mostrado en la Figura 7, en el cual se puede observar los dos nodos presentes en la estructura `while` de la función *ejemplo*.

Código 1
Función ejemplo escrita en lenguaje C

```
1  int ejemplo (int a,int b,int c,int d){
2          int a_aux,b_aux,c_aux;
3          while (a>b) {                    //Nodo1
4                  a_aux=a-b;               //Nodo2 inicio
5                  b_aux=c+d;
6                  c_aux=a_aux+b_aux;
7                  a=a_aux;b=b_aux;c=c_aux;//Nodo2 final
```

```
8           }
9           return a;
10  }
```

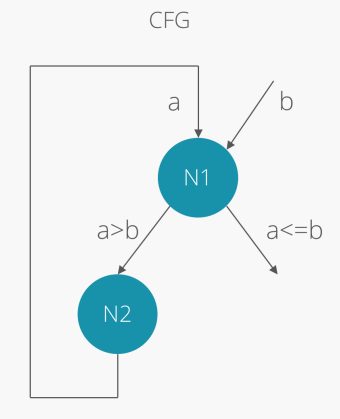


Figura 7. CFG de la función *ejemplo*. El primer nodo representa la comparación entre a y b; el segundo nodo representa el conjunto de instrucciones dentro del ciclo while.

2.6.1.2. DFG (Data Flow Graph)

El gráfico de flujo de datos representa a detalle cada nodo separándolo a su vez en nodos más pequeños. Esta representación es conocida como árbol de operaciones. Las flechas representan las dependencias existentes entre operaciones para determinar si deben realizarse de forma secuencial o pueden ser paralelizadas. En la Figura 8 se representa el DFG para cada nodo del CFG de la Figura 7.

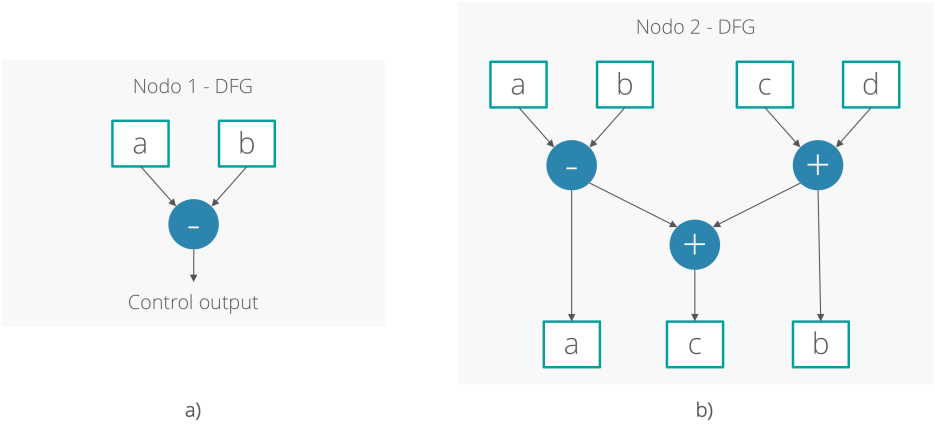


Figura 8. DFG de la función *ejemplo*. En la figura se representan los DFGs con las dependencias de datos para: a) primer nodo, b) segundo nodo.

2.6.1.3. Representación CDFG

Finalmente se combina el CFG y DFG. Los nodos del CFG en los cuales se toman decisiones de divergencia son representados con triángulos, y los nodos donde se ejecutan operaciones son representados por rectángulos (Koch et al., 2016). En la Figura 9 se muestra el CDFG para la función ejemplo.

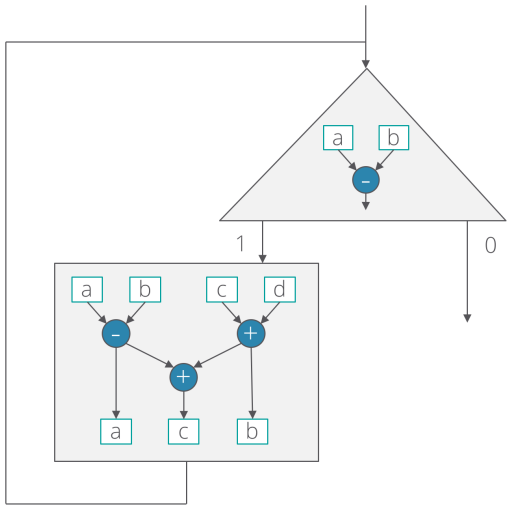


Figura 9. CDFG de la función ejemplo.

2.6.2. Fases de HLS

El proceso de diseño mediante HLS empieza con una especificación de alto-nivel de una aplicación, y las restricciones (*constraints*) provistas por el diseñador. En base a ellas, las herramientas HLS de forma automática o semiautomática generan una arquitectura RTL que implementa esta especificación. La arquitectura RTL es generada como un sistema digital que consiste en i) un *datapath*, y ii) una máquina de estados finita (FSM) (Koch, Hannig, & Ziener, 2016). Para lograr esto, las herramientas requieren ejecutar una serie de tareas, las cuales se muestran en la Figura 10 y son detalladas a continuación.

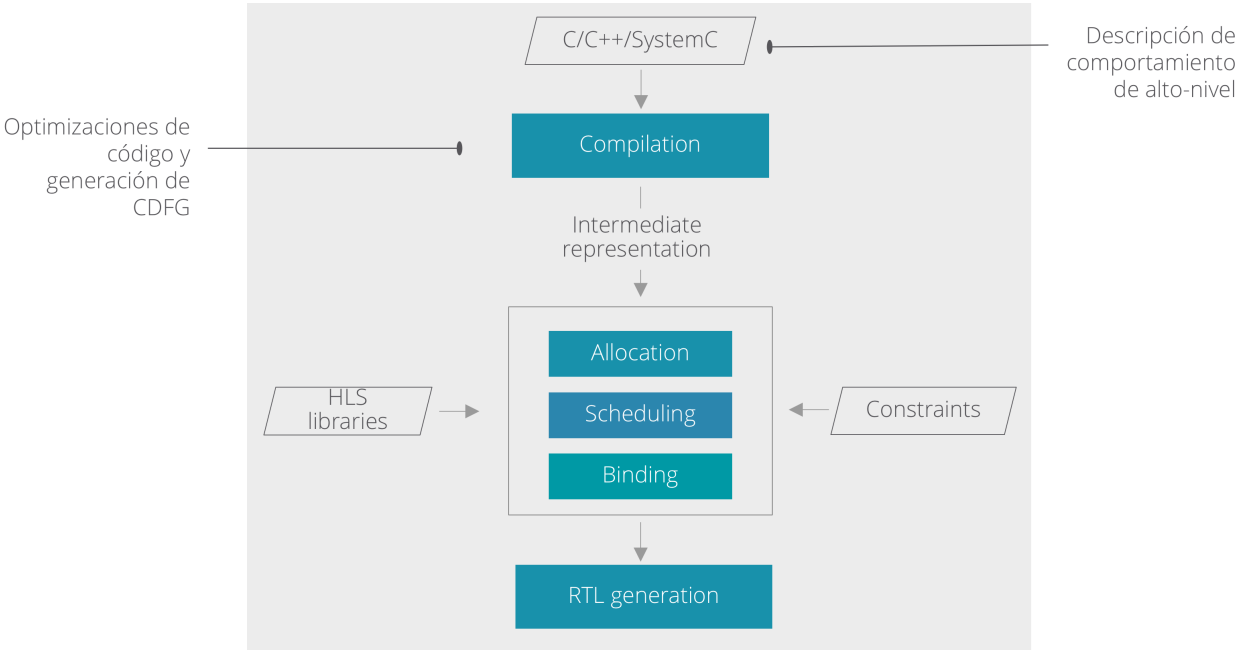


Figura 10. Flujo de diseño de HLS.

1. **Compilación**. HLS inicia con un proceso de compilación de la descripción de comportamiento del sistema digital especificada en lenguaje C/C++/SystemC. Esto incluye optimizaciones como la eliminación de código infructuoso o redundante y de falsas dependencias de datos (Coussy et al., 2009; Koch et al., 2016). Este código de alto-nivel optimizado es transformado a la representación intermedia denominada CDFG (Sección 2.6.1).

2. ***Allocation***. Determina el número y tipo de recursos de hardware necesarios para la implementación de la aplicación. Estos recursos incluyen unidades de hardware como: elementos de almacenamiento (i.e., registros y memorias), unidades funcionales (e.g., sumadores, multiplicadores, etc.) y recursos de interconexión (e.g., buses, multiplexores, etc.) (Koch et al., 2016; Vinet & Zhedanov, 2009). El tipo de recurso específico es seleccionado de una librería de componentes RTL, la cual contiene información acerca del área, latencia y consumo energético de cada componente (Coussy et al., 2009). El proceso de *allocation* realiza la selección de elementos de tal forma que aseguren el

cumplimiento de los requisitos del diseñador.

3. ***Scheduling***. Asigna cada operación a un ciclo de control (*control steps*) (Vinet & Zhedanov, 2009). Cada ciclo de control corresponde a un ciclo de reloj del sistema (Koch et al., 2016). El objetivo de *scheduling* es encontrar y explotar las opciones de paralelismo en la representación intermedia (CDFG). Para ello, se analizan las relaciones de dependencia, es decir, si una operación depende de la salida de otra. Así, operaciones no dependientes pueden ser ejecutadas en paralelo, durante un mismo ciclo de control y en diferentes elementos de hardware. Una de las restricciones para alcanzar un máximo nivel de paralelismo es la disponibilidad de recursos, la cual es determinada durante la etapa de *allocation* (Vinet & Zhedanov, 2009). Regresando al ejemplo de la Figura 8, la implementación del Nodo 2 puede realizarse de dos formas, ejecutando secuencialmente todas las operaciones, o paralelizando las operaciones posibles. Estas dos alternativas se muestran en la Figura 11.

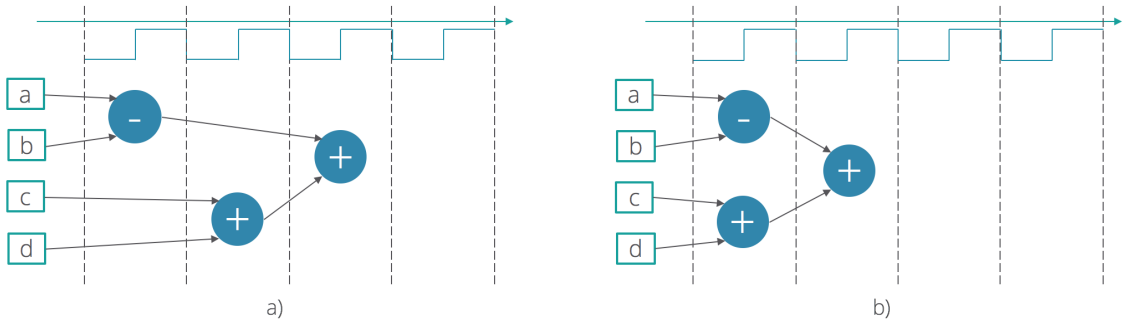


Figura 11. Diferentes implementaciones de un datapath: a) ejecución secuencial, b) ejecución paralela de operaciones.

4. ***Binding***. Se asigna cada operación a una unidad funcional, y el valor del resultado de la misma a registros. Las transferencias de datos son a su vez asignadas a recursos de interconexión (Vinet & Zhedanov, 2009). Uno de los objetivos de la etapa de *binding* es determinar que recursos de hardware pueden ser compartidos por operaciones que no requieren una ejecución simultánea. Si existen recursos compartidos, entonces los

valores de entrada correctos en cada ciclo de control deben ser seleccionados mediante técnicas de multiplexación (Koch et al., 2016).

5. ***Generación de RTL***. A partir de las tareas de síntesis antes descritas, se genera una arquitectura RTL la cual consiste en la combinación de un controlador y un datapath, como se muestra en la Figura 12. La descripción generada en RTL (ej., VHDL/Verilog/SystemC), está lista para su síntesis a hardware (Vinet & Zhedanov, 2009).

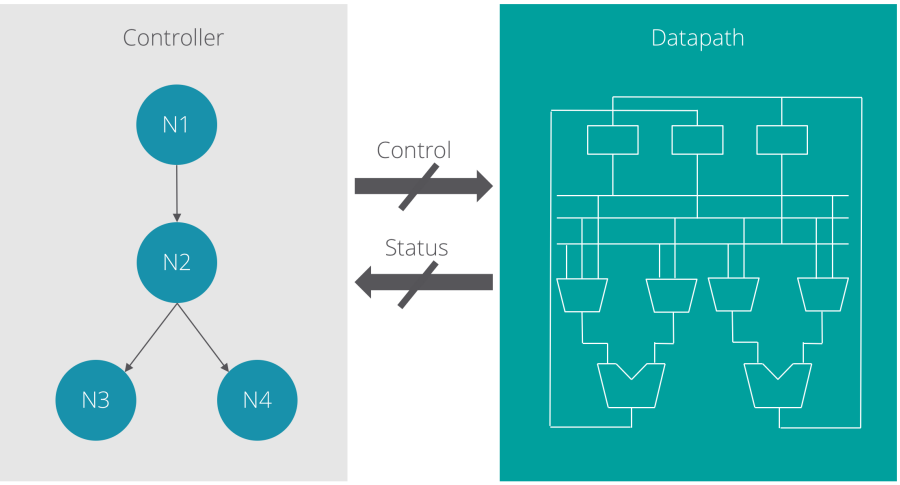


Figura 12. Implementación basada en la combinación de un datapath y un controlador.

Tradicionalmente HLS involucra los tres mayores procesos: *allocation*, *scheduling* y *binding*. Sin embargo, dependiendo de los objetivos de diseño, HLS puede incluir también otras técnicas de análisis, paralelización y optimización. Dichas técnicas dependerán de la herramienta utilizada (Koch et al., 2016). En la Sección 2.6.4 se revisan algunas herramientas y técnicas usadas por las mismas.

2.6.3. Reseña Histórica de HLS

El desarrollo de herramientas HLS abarca un largo camino evolutivo. Desde que surgieron las primeras ideas en la década de 1970, los esfuerzos en su investigación se han incrementado hasta nuestros días (Coussy et al., 2009). Varios estudios dedicados a su evolución se pueden

encontrar en la literatura, donde los autores dividen su desarrollo por periodos o generaciones con una extensa cobertura. Esta sección no pretende estudiar de forma detallada la evolución de HLS. Por lo contrario, se presenta una breve reseña histórica acerca de HLS en la que se resaltan principales puntos en su desarrollo hasta la actualidad.

Las primeras investigaciones surgieron en una época muy temprana, abordando síntesis RTL y síntesis de alto-nivel. Sin embargo, no sería hasta los años 80 donde surgiría un mayor interés, y se empezarían a investigar los conceptos de HLS (Martin & Smith, 2009). Como resultado, durante esta década se presentan importantes artículos que tendrían una fuerte repercusión a futuro. Principalmente estudios acerca de *scheduling* a partir de descripciones de comportamiento. Adicionalmente, se investiga cómo desarrollar técnicas efectivas de *allocation* y *binding* (Coussy & Morawiec, 2016). Ya en los años 90, las primeras herramientas comerciales se lanzan al mercado, entre ellas: Synopsys Behavioral Compiler y Cadence (Koch et al., 2016). A pesar de las grandes promesas de HLS, estas herramientas no tuvieron la acogida esperada, siendo usadas solamente por un grupo selecto en la industria y universidades. Entre sus problemas estaban la baja calidad de sus resultados, diseños difíciles de validar, y uso parcial de lenguajes HDL como entrada a las herramientas HLS (Martin & Smith, 2009). Finalmente, en la primera década de los años 2000 el mercado de herramientas comerciales enfocadas a ASICs y FPGAs creció significativamente. Entre estas herramientas se encuentran Mentor Catapult C Synthesis, Celoxica Agility, Bluespec, AutoESL AutoPilot, Xilinx AccelDSP, Synopsys Synplicity Synplify DSP, y Cadence C-to-Silicon (Martin & Smith, 2009). La significativa adopción de estas herramientas se debe a diferentes razones: i) la mayoría usa lenguajes C, C++, SystemC o extensiones de los mismos como entrada, lo que facilita a ingenieros de hardware y software poder diseñar directamente en FPGAs para crear aceleradores sin tener que usar necesariamente HDLs, ii) la calidad de resultados mejora permitiendo optimizaciones durante el proceso de síntesis, iii) rápida creación de implementaciones de hardware en FPGAs y ASICs, siendo clave para alcanzar restricciones como *time-to-market* (Martin & Smith, 2009).

2.6.4. Herramientas HLS

Las herramientas actuales son categorizadas de acuerdo a parámetros que las definen. Desde esta perspectiva, los principales parámetros son: A) Tipo de licencia: determina si la herramienta es comercial, de licencia accesible bajo solicitud, o de licencia libre (*open source*). B) Lenguaje de entrada: pueden ser lenguajes de programación de propósito-general (GPLs) o lenguajes conocidos como de dominio específico (DSLs). Lenguajes GPL típicos como C, C++, y SystemC pueden ser una versión limitada del estándar o extensiones que soporten tipos de datos específicos para diseño de hardware (Koch et al., 2016). Lenguajes DSLs por el contrario son lenguajes nuevos específicamente diseñados con un enfoque particular para cada herramienta (Nane et al., 2016). C) Nivel de optimización y paralelización: el nivel de optimización determina la eficiencia en la síntesis a RTL en base a restricciones como, limitación de recursos, o priorización de rendimiento. Igualmente, es importante el nivel de paralelización, el cual puede ser realizado a nivel de instrucción, nivel de lazo, o múltiples iteraciones de lazos simultáneos (software *pipelining*), etc. Ambos parámetros determinarán la calidad de los resultados alcanzados.

En base a los parámetros descritos en el párrafo anterior, se resume a continuación algunas de las herramientas HLS usadas actualmente. Se han escogido siete importantes herramientas comerciales: Altera OpenCL, Bluespec, Catapult HLS, HDL Coder, Stratus HLS, Synphony HLS, Vivado HLS y dos herramientas fruto de investigaciones académicas: LegUp y ROCCC, cuyo análisis se resume en la Tabla 2.

1. **Altera SDK/OpenCL:** es una herramienta comercial desarrollada por Altera, la cual está orientada a crear funciones de kernel aceleradas en FPGA a partir OpenCL (*Open Computing Language*) (Koch et al., 2016). Este lenguaje es una variante de C, y estándar de programación de arquitecturas heterogéneas. La herramienta se encarga de sintetizar OpenCL de forma que se obtenga un circuito en FPGA con un profundo pipeline (Nane et al., 2016), el cual es generado como Verilog RTL. Esta herramienta usa

Tabla 2
Perspectiva general de herramientas HLS relevantes

Herramienta	Desarrollador	Licencia	Arquitecturas	Lenguaje	Salida
SDK / OpenCL	Altera	Comercial	FPGA de Altera	OpenCL, C	Verilog
Bluespec	BlueSpec Inc.	Comercial	ASIC, FPGA	BSV	Verilog, SystemC
Catapult HLS	Mentor Graphics	Comercial	ASIC, FPGA	C, SystemC	VHDL, Verilog, SystemC
LegUp	U. Toronto	Open-source	FPGA	C	Verilog
HDL Coder	MathWorks Inc.	Comercial	FPGAs de Xilinx y Altera	Matlab, Simulink	VHDL, Verilog
ROCCC	U. California	Open-source	FPGA	C	VHDL
Stratus HLS	Cadence	Comercial	ASIC, FPGA	C, C++, SystemC	Verilog, SystemC
Synphony HLS	Synopsys	Comercial	ASIC, FPGA	C, C++, Matlab	VHDL, Verilog, SystemC
Vivado HLS	Xilinx	Comercial	FPGA de Xilinx	C, C++, SystemC	VHDL, Verilog, SystemC

Fuente: (Koch et al., 2016; Nane et al., 2016)

dos tipos de paralelismo: explicit multi-threading e implicit loop pipelining (Zohouri, Maruyamay, Smith, Matsuda, & Matsuoka, 2017).

2. **Bluespec:** es una herramienta comercial HLS creada por Bluespec Inc. El lenguaje de entrada es BSV (*Bluespec System Verilog*), un tipo de HDL de alto-nivel basado en Verilog (Nane et al., 2016). En esta herrramienta se realizan descripciones de comportamiento en base a una serie de reglas denominadas *Guarded Atomic Actions*. Estas descripciones se consideran a medio camino entre el diseño *high-level* y el diseño RTL, automatizando ciertos aspectos de la creación de hardware pero no en su totalidad (Meeus et al., 2012). Debido a esta limitante, se requiere de una mayor experiencia por parte del diseñador usando descripciones RTL, pero como ventaja permite tener un preciso control sobre los recursos y el desempeño. Bluespec puede generar como salida

tanto Verilog RTL como SystemC. El compilador es capaz de optimizar los diseños usando *dynamic scheduler*, lo cual le permite ejecutar múltiples reglas en un mismo ciclo de reloj.

3. **Catapult HLS:** es una herramienta comercial desarrollada por Mentor Graphics (adquirida a Calypto Design Systems en 2015). Acepta descripciones funcionales en ANSI C++ y SystemC, entregando un modelo optimizado RTL VHDL, Verilog o SystemC (Meeus et al., 2012). Catapult está enfocado a generar diseños con un reducido consumo energético, permitiendo también optimizar el área y desempeño.

4. **LegUp:** es una herramienta HLS *open-source* desarrollada por investigadores de la Universidad de Toronto, la cual continua en permanente desarrollo. Acepta como lenguaje de entrada una descripción en ANSI C para sintetizarla a hardware de FPGA (Verilog RTL) o a un sistema híbrido. Este sistema híbrido consiste en un procesador embebido (MIPS I o ARM Cortex-A9) y aceleradores personalizados por el diseñador para realizar cómputos intensivos (Fort et al., 2014; Koch et al., 2016). LegUp provee soporte para paralelismo a nivel de hilos y de lazos para realizar *pipeline* en hardware (Fort et al., 2014).

5. **HDL Coder:** es una herramienta comercial disponible como toolbox de Matlab ofrecida por The MathWorks Inc. HDL Coder es capaz de generar código VHDL y Verilog a partir de funciones de MATLAB y modelos de Simulink (Mathworks, 2018). El código HDL puede ser independiente del FPGA u optimizado para ciertas FPGAs de Xilinx y Altera (Koch et al., 2016).

6. **ROCCC:** es una herramienta HLS *open-source* desarrollada por la Universidad de California Riverside. Está enfocada al diseño de aceleradores en FPGA a partir de una versión limitada de lenguaje C. ROCCC se centra principalmente en la paralelización de aplicaciones de alta densidad informática (Nane et al., 2016), y se enfoca en mejorar

el manejo de memoria usando *smart buffers* (Meeus et al., 2012). Esta herramienta hace uso de optimizaciones como: *loop unrolling*, *data reuse* y *clock cycle time* (Koch et al., 2016). ROCCC genera como salida una descripción VHDL.

7. **Stratus HLS:** es una herramienta comercial ofrecida por Cadence enfocada al diseño de ASICs y FPGAs. Junto con el IDE de Cadence se puede crear modelos con IP-cores optimizados mediante HLS. Stratus acepta como entrada descripciones de comportamiento en lenguaje C, C++ y SystemC, y entrega RTL Verilog y SystemC. Esta herramienta permite optimizar la arquitectura RTL para cumplir con métricas de consumo energético, desempeño y área.

8. **Synphony HLS:** es una herramienta comercial HLS desarrollada por Synopsys Inc. Synphony está orientado a la creación de RTL optimizado para FPGA y ASICs. Además, Synopsys ofrece un conjunto de herramientas que complementan y facilitan la implementación y verificación de diseños. Esta herramienta acepta como entrada modelos C, C++ e incluso archivos de Matlab, entregando RTL en VHDL, Verilog o SystemC. Synphony optimiza la arquitectura RTL según los objetivos de diseño, como requerimientos de área, velocidad o consumo energético. Esta herramienta usa técnicas como *loop unrolling* y *loop pipelining* (Nane et al., 2016).

9. **Vivado HLS:** es una herramienta comercial creada por Xilinx, la cual forma parte del entorno Vivado Design Suite. Vivado HLS está orientada a la generación de núcleos de hardware y se complementa con otras herramientas de Xilinx para la creación de sistemas híbridos (*hard-core* o *soft-core* processors comunicados con aceleradores) (Windh et al., 2015). La herramienta acepta descripciones en lenguajes C, C++, SystemC o incluso *kernels* usando OpenCL (Xilinx, 2016). Tipos de datos de precisión arbitraria son soportados en C++ y SystemC. Las implementaciones RTL son obtenidas como módulos VHDL, Verilog, o SystemC, las cuales pueden ser sintetizadas en familias soportadas de FPGAs Xilinx. Esta herramienta cuenta con varias posibilidades de optimización,

entre ellas: optimización de latencia, área, y paralelización de lazos y funciones, etc. Una descripción más extensa acerca de Vivado HLS se presenta en el Capítulo 3.

Capítulo 3

Síntesis de Alto-Nivel para SoCs de Xilinx

En este capítulo se presentan conceptos fundamentales de la herramienta para síntesis de alto-nivel de Xilinx, Vivado HLS. Primero, se introduce a la síntesis de especificaciones C de Vivado. Segundo, se aborda la integración de esta herramienta al flujo de diseño de SoCs de Xilinx. Tercero, se detallan las funcionalidades de Vivado HLS y las optimizaciones de diseño disponibles.

3.1. Diseño de FPGAs Basados en Especificaciones C Usando Vivado HLS

3.1.1. Ventajas del Uso de Vivado HLS

Los beneficios de usar HLS para el diseño de sistemas digitales son varios. Los conceptos que se presentan a continuación se centran en la herramienta Vivado HLS de Xilinx, sin embargo, muchos pueden aplicarse a HLS en general. Entre otras, sus ventajas son: i) Mejorar la productividad para los diseñadores de hardware, permitiendo trabajar en un nivel de

abstracción superior sin perder calidad en los resultados. ii) Acercar el diseño de hardware a programadores de software, permitiéndoles trasladar algoritmos de alto costo computacional a FPGA. iii) Acelerar el diseño de sistemas digitales al evitar detalles complejos acerca de implementaciones de hardware que son demandantes de tiempo. iv) Facilitar la verificación de la funcionalidad de diseños, realizándola en alto-nivel. v) Aceptar la influencia de la síntesis mediante la aplicación de directivas de optimización y uso de restricciones. vi) Permitir la creación de múltiples implementaciones de una misma especificación C, explorando varios diseños hasta encontrar la solución óptima. vii) Facilitar la reutilización de algoritmos de software (Xilinx - UG902, 2017).

3.1.2. Síntesis de Especificaciones C

La síntesis HLS involucra la realización de las fases *Scheduling*, *Allocation* y *Binding*, las cuales se explican en la Sección 2.6.2. La presente sección busca aclarar conceptos adicionales acerca de la síntesis de una especificación C mediante Vivado HLS. Una especificación C es definida como una descripción de comportamiento en C, C++, o SystemC, la cual puede incluir una función *top-level* y una jerarquía de sub-funciones (Xilinx - UG902, 2017). Vivado HLS realiza la síntesis de una especificación C tomando en cuenta las siguientes directrices.

3.1.2.1. Síntesis de una Especificación C en Vivado HLS

1. Se sintetiza únicamente como puertos I/O (entrada/salida) los argumentos de la función *top-level* y sus valores de retorno.

2. Las funciones C son sintetizadas como bloques en una jerarquía RTL. Es decir, la jerarquía de sub-funciones C se sintetizan como módulos o entidades RTL.

3. Los lazos dentro de sub-funciones son mantenidos en la forma llamada *"rolled"*. Es decir, HLS crea el hardware necesario para realizar una iteración, y ocupará los mismos

recursos secuencialmente para las demás iteraciones hasta completar las operaciones del lazo.

4. Los arreglos de datos son sintetizados como block RAM. Si estos arreglos son argumentos de la función *top-level*, entonces HLS implementa puertos para acceder a un block RAM fuera del diseño.

3.1.2.2. Métricas de Desempeño

HLS permite utilizar directivas de optimización para influenciar la síntesis. Estas directivas permiten modificar y controlar el comportamiento del sistema y la creación de puertos I/O (Xilinx - UG902, 2017). Para determinar las optimizaciones requeridas se puede realizar un análisis en base a los reportes de síntesis. Estos reportes contienen información acerca de las métricas de desempeño detalladas a continuación.

- **Área:** cantidad de recursos de hardware utilizados para la implementación del diseño.
- **Latencia:** número de ciclos de reloj requeridos para calcular los valores de salida.
- **Intervalo de iniciación:** número de ciclos de reloj que transcurren antes de que la función acepte nuevos valores de entrada.
- **Latencia de iteración de lazo:** número de ciclos de reloj requeridos para completar una iteración del lazo.
- **Intervalo de iniciación de lazo:** número de ciclos de reloj previos antes de que la siguiente iteración inicie.
- **Latencia de lazo:** número de ciclos de reloj necesarios para completar todas las iteraciones del lazo.

Ejemplo 3.1

Para ilustrar los conceptos básicos revisados en esta sección, se presenta la función *basicf* descrita en el Código 2. La extracción de la lógica de control a una FSM y creación de puertos I/O a partir de argumentos de esta función se muestran en la Figura 13. La función *top-level* tiene tres argumentos de tipo `char` y dos arreglos. HLS sintetiza los argumentos `a`, `b` y `c` como puertos de datos de 8 bits, mientras que `in` y `out` son sintetizados como puertos de lectura y escritura de bloques RAM respectivamente. HLS de forma automática extrae la lógica de control de la especificación C y crea la FSM para el control de las operaciones. En el ejemplo son creados cuatros estados C0, C1, C2, C3. Cada uno de los estados es descrito a continuación.

Código 2
Función basicf escrita en lenguaje C

```
1  void basicf (int in[3], char a, char b, char c, int out[3]) {
2       int x,y;
3       for(int i = 0; i < 3; i++) {
4         x = in[i];
5         y = a + b +c*x;
6         out[i] = y;
7           }
8  }
```

1. **C0:** Se realiza la suma entre `a` y `b`. Esta suma se debe realizar una sola vez en todo el lazo, por lo cual, HLS mueve esta operación fuera del mismo y la asigna al estado C0. El resultado se almacena en un registro cuyo valor es leído dentro del lazo.

2. **C1:** Se direcciona los elementos `i` del arreglo `in` según corresponda a la iteración.

3. **C2:** Se lee `in[i]` y se guarda su valor en un registro `x`.

4. **C3:** Se realiza el producto de `c` y `x`, el cual se suma al resultado obtenido en C0.

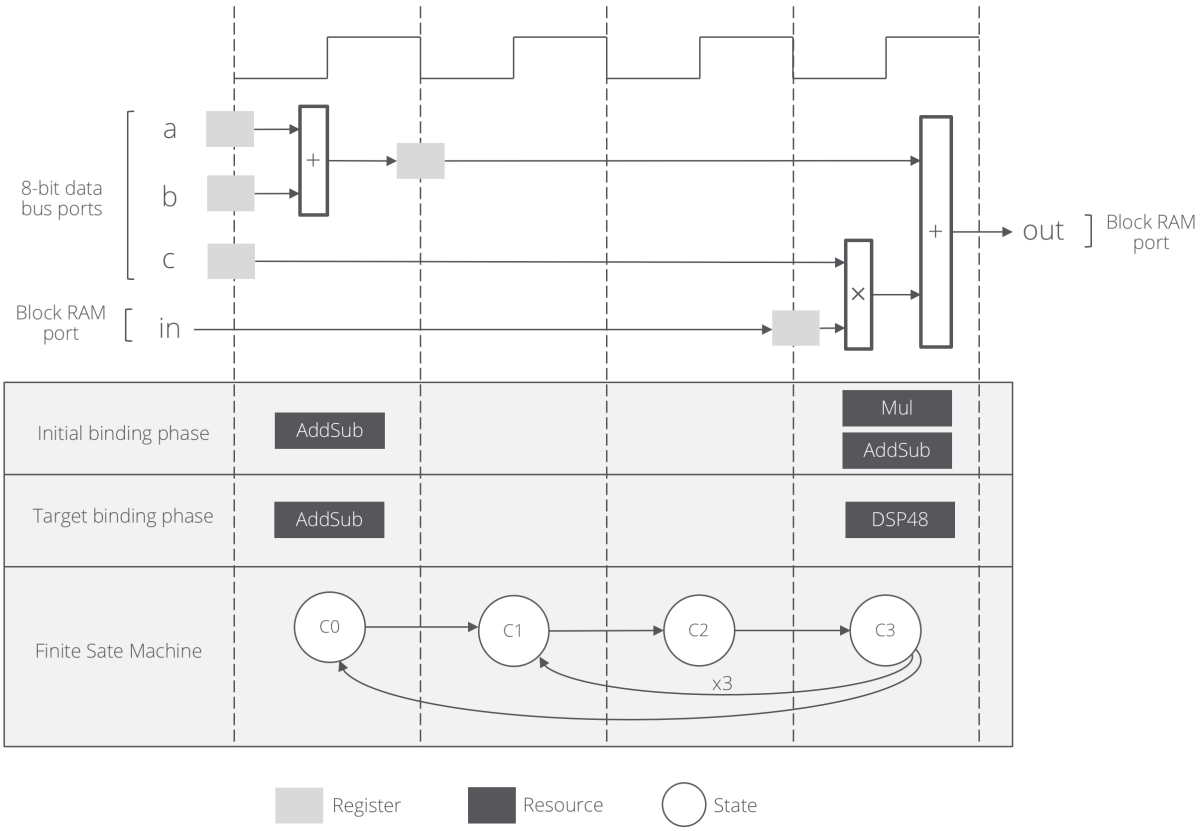


Figura 13. Ejemplo de implementación HLS del Código 2. Extracción de la lógica de control a una FSM y creación de puertos I/O a partir de argumentos de la función top-level. La etapa de binding inicialmente implementa un multiplicador (Mul) y dos sumadores/restadores (AddSub). La implementación final utiliza un DSP48 para las operaciones de suma y multiplicación.
Fuente: (Xilinx - UG902, 2017)

En la Figura 14 se muestra la secuencia de operaciones realizada durante una ejecución de la función *basicf*. Así también, se presentan las métricas de desempeño como latencias e intervalos para el ejemplo de la Figura 13. La latencia de la función para realizar todos sus cálculos es de nueve ciclos, mientras que el intervalo de iniciación para aceptar nuevos valores de entrada es de diez ciclos. La latencia de lazo y el intervalo de iteración coinciden en tres ciclos, y finalmente, la latencia de lazo es de nueve ciclos.

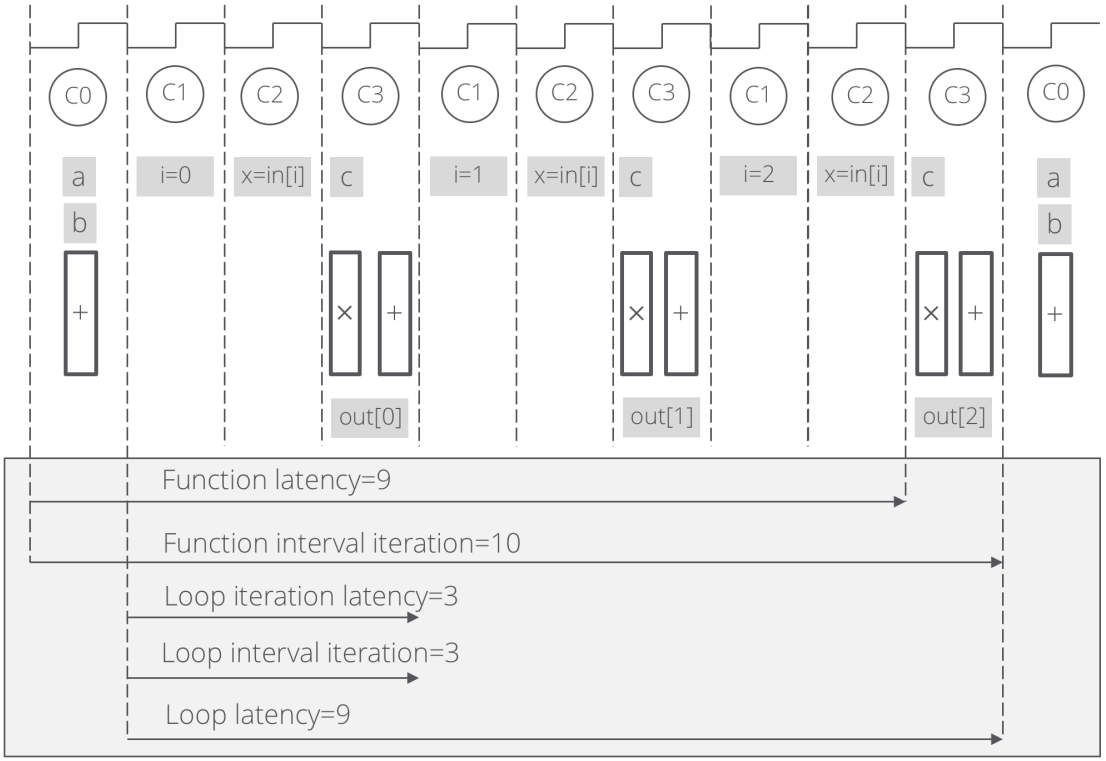


Figura 14. Latencia e intervalos en la ejecución del ejemplo mostrado en el Código 2. Aquí se observa como el estado C0 es ejecutado una sola vez fuera del loop for, mientras que C1, C2 y C3 son ejecutados en cada una de las tres iteraciones.
Fuente: (Xilinx - UG902, 2017)

3.2. Vivado High-Level Synthesis (HLS)

Vivado HLS es la herramienta de síntesis de alto-nivel de Xilinx. Esta herramienta está enfocada a la creación de IP-cores a partir de descripciones de comportamiento de alto-nivel, ofreciendo facilidades para obtener una alta productividad de diseño. Además, Vivado HLS está estrechamente ligada al conjunto de herramientas de Xilinx, lo que permite que los IP-cores creados puedan ser integrados a sistemas más grandes de hardware en FPGAs y SoCs[1]. En la Figura 15 se representa la integración de Vivado HLS al flujo de diseño de Xilinx.

[1]En el desarrollo de la tesis se usa Vivado IP Integrator para la generación de SoCs. Otras herramientas de Xilinx para la integración de IP-cores son Vivado RTL Integration, y System Generator for DSP, las cuales están fuera del alcance de esta tesis.

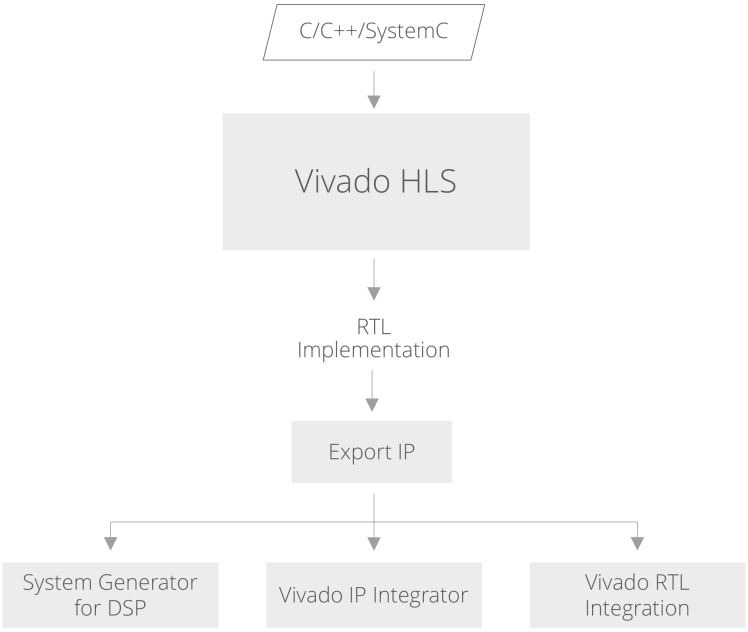


Figura 15. Vivado HLS en el flujo de diseño de Xilinx. Los IP-cores generados en Vivado HLS pueden ser importados en otras herramientas de Xilinx, las cuales permiten la integración de los mismos a sistemas de hardware más grandes.

3.2.1. Metodología de Diseño de Vivado HLS

3.2.1.1. Flujo de Diseño

El flujo de diseño de Vivado HLS involucra varias etapas. El proceso empieza a partir de la creación de una especificación C y termina con la obtención de una implementación RTL final junto a reportes de síntesis. De forma opcional, se puede exportar esta implementación como un IP que puede ser utilizado en la creación de un SoC, el cuál es el método de diseño utilizado en esta tesis. El flujo de diseño se representa en la Figura 16, donde cada etapa del proceso es detallada a continuación.

Etapas del flujo de diseño de Vivado HLS

1. **Creación de una especificación C a ser sintetizada.** La descripción puede constar de una única función, o de una función *top-level* y una jerarquía de sub-funciones. Esta especificación es denominada por Xilinx como *Design Under Test* (DUT) o diseño bajo

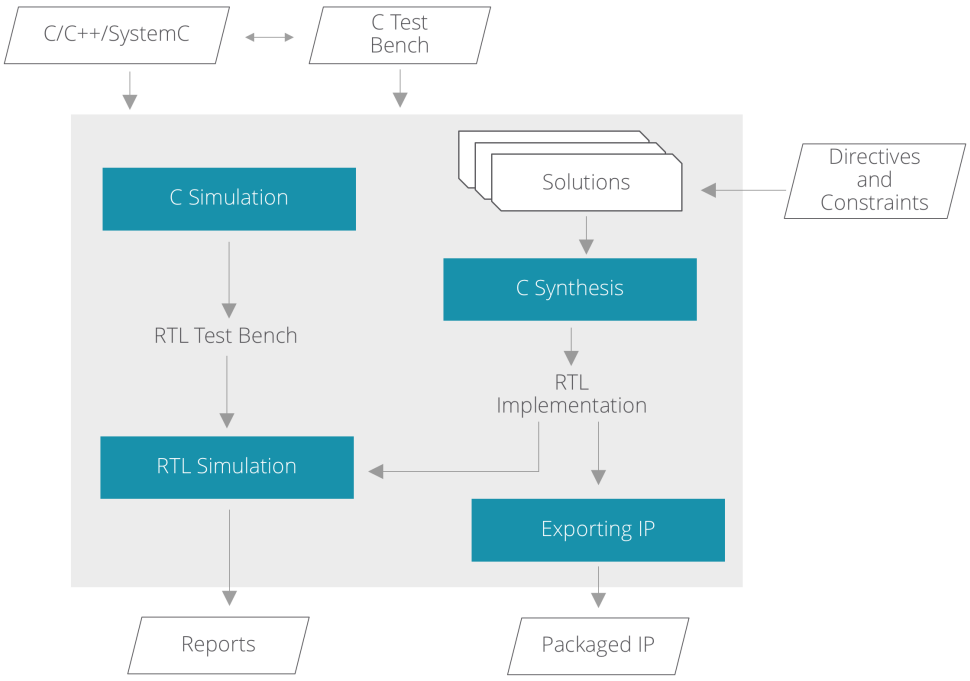


Figura 16. Flujo de diseño de Vivado HLS.
Fuente: (Xilinx - UG902, 2017)

prueba.

2. **Creación de un test bench[2] en C.** El *test bench* llama a la función C a ser sintetizada y compara sus resultados con un conjunto de valores esperados o *golden results*.

3. **Simulación C.** Vivado HLS compila la descripción de alto-nivel, y ejecuta el *test bench*. La especificación C es validada en base a los resultados obtenidos.

4. **Creación de una solución.** Se define directivas y restricciones de síntesis para obtener una implementación RTL personalizada de la especificación C. Vivado HLS permite crear varias soluciones, donde cada una será una implementación RTL de la misma especificación C, pero con diferentes directivas de optimización y restricciones. Esto permite explorar fácilmente diferentes implementaciones de hardware seleccionando la solución que mejor se adapte a los requerimientos específicos de cada aplicación.

[2]En Vivado HLS un test bench se entiende como un algoritmo destinado a la verificación de la funcionalidad del DUT.

5. **Síntesis, optimización y análisis.** Se sintetiza la solución mediante las fases de *scheduling*, *binding* y *allocation*. Los resultados pueden ser analizados y comparados para cada solución. Para facilitar el análisis de resultados Vivado HLS facilita una perspectiva de análisis, en la cual se muestra un detalle de las operaciones realizadas por la función (revisar Sección 3.3.2). En base a esto se pueden aplicar directivas de optimización para alcanzar las métricas de desempeño, área o latencia requeridas (revisar Sección 3.3.3).

6. **Verificación RTL.** Se verifica el resultado con una simulación a partir de un *test bench* RTL, el cual es generado automáticamente[3] por Vivado HLS a partir del *test bench* C.

7. **Generación de reportes.** Vivado entrega reportes detallados de síntesis, los cuales pueden ser analizados y servir de referencia para mejorar los resultados a través de la aplicación de directivas de optimización.

8. **Creación de IP-cores a partir de implementaciones RTL.** Es un proceso opcional que permite exportar el diseño para integrarlo como parte de un sistema de hardware mediante otras herramientas de Xilinx.

3.2.1.2. Entradas y Salidas de Vivado HLS

En esta sección se describen las entradas y salidas en el flujo de diseño de Vivado HLS de acuerdo a lo mostrado en la Figura 16.

Entradas

- **Funciones C escritas en C, C++, SystemC.** Estos archivos contienen las funciones a sintetizarse y son la principal entrada de Vivado HLS. Se puede tratar de una jerarquía de sub-funciones y múltiples archivos (Crockett et al., 2015).

[3]Para esta generación automática se debe cumplir con ciertas directrices, las cuales son discutidas en la Sección 4.1.2.2.

- **Test bench C.** Archivo que simula la función C a ser sintetizada para comprobar su funcionalidad. A partir del *test bench* C, Vivado HLS genera de forma automática una co-simulación C/RTL en donde verifica la salida del modelo RTL final.

- **Restricciones.** El diseñador debe proveer ciertas restricciones para cada una de las soluciones. Las restricciones definidas en Vivado HLS son el periodo de reloj, incertidumbre de reloj, y el dispositivo FPGA de destino.

- **Directivas de optimización.** La aplicación de directivas es un proceso opcional y su objetivo es influenciar la síntesis. De esta forma se busca implementar la arquitectura que cumpla las métricas requeridas. Mediante las directivas se puede alcanzar optimizaciones para cumplir con un determinado número de recursos, protocolos I/O para la comunicación del bloque final, latencia, intervalo de iniciación, latencia de iteración de lazos, intervalo de iniciación de lazos, y latencia de lazos (Xilinx - UG902, 2017).

Salidas

- **Implementaciones RTL.** Es la principal salida de Vivado HLS. Las implementaciones son generadas como archivos en formato HDL en los estándares VHDL (IEEE 1076-2000) y Verilog (IEEE 1364-2001) (Xilinx - UG902, 2017).

- **Archivos de reporte.** Informes de resultados de síntesis, co-simulación C/RTL, y del empaquetado de IP-cores (Xilinx - UG902, 2017).

3.2.1.3. Interfaces de Usuario de Vivado HLS

El proceso de diseño antes descrito se puede llevar a cabo tanto en una Interfaz Gráfica de Usuario (GUI), semejante a entornos de desarrollo de software, como en una Interfaz de Línea de Comandos (CLI) usando comandos Tcl[4]. A continuación, se realiza una breve descripción

[4]Tcl es un lenguaje de código abierto de sintaxis sencilla usando para la creación de scripts.

de estas interfaces sin ahondar en mayores detalles.

Interfaz Gráfica de Usuario (GUI)

La interfaz GUI es un entorno de desarrollo completo para la creación y manejo de proyectos de Vivado HLS. Ofrece todas las facilidades para editar, depurar, especificar directivas y analizar resultados durante el proceso. La interfaz está compuesta por los cuatro paneles mostrados en la Figura 17, los cuales son: **(1) Explorador**: ubicado a la izquierda, muestra la jerarquía del proyecto. **(2) Panel de información**: muestra el contenido de archivos para su visualización o edición de código. **(3) Panel auxiliar**: muestra detalles de los archivos abiertos en el panel de información. **(4) Panel de consola**: muestra la salida de los procesos ejecutados por Vivado HLS.

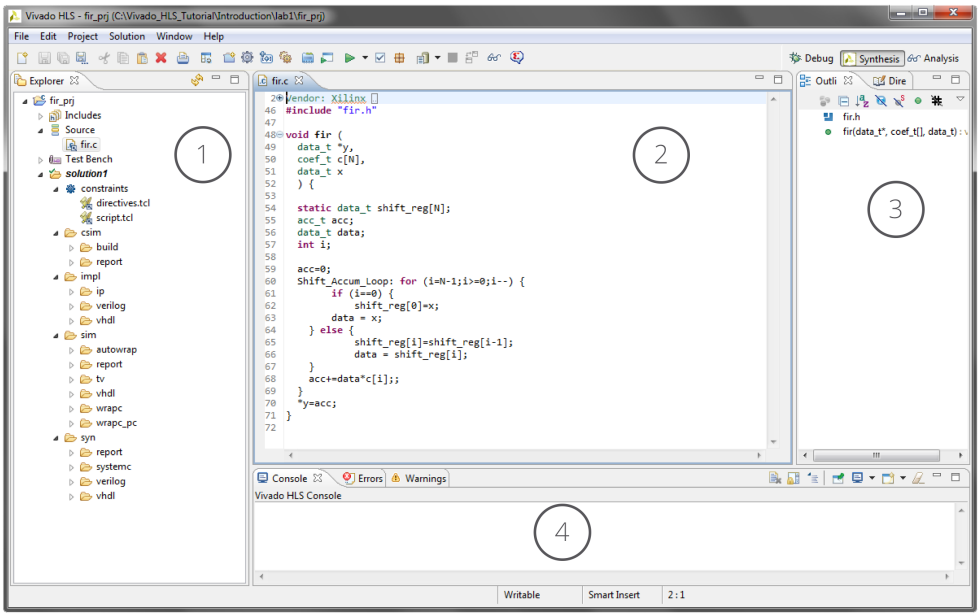


Figura 17. Interfaz gráfica de usuario de Vivado HLS. La ventana muestra la vista de *Synthesis*, donde el Panel de Información presenta una especificación C de un filtro FIR.

Interfaz de Línea de Comandos (CLI) Tcl

La interfaz CLI está diseñada para ejecutar tareas repetitivas que de otra forma serían

llevadas a cabo manualmente en la interfaz GUI. Usando esta interfaz, ciertas tareas pueden ser realizadas de forma semi-automática a fin de reducir el tiempo y asegurar reproducibilidad de resultados. De hecho, se pueden crear archivos Tcl para ejecutar los comandos en conjunto. Vivado HLS CLI usa un entorno GNU minGW en Windows, lo que permite usar comandos DOS y ciertos comandos de Linux (Xilinx - UG902, 2017). Una ventana *command prompt* de Vivado HLS se muestra en la Figura 18.

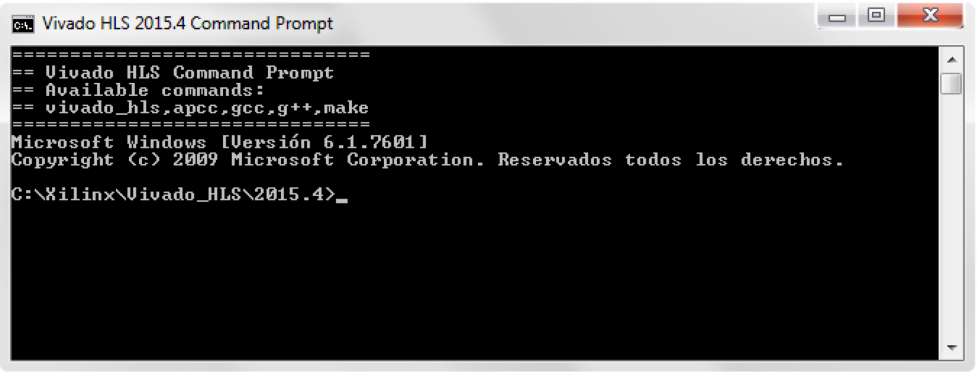


Figura 18. Interfaz de línea de comandos Tcl de Vivado HLS.

3.2.2. Lenguajes, Librerías y Tipos de Datos Soportados

Vivado HLS soporta la compilación y simulación de ciertos estándares C. Los lenguajes soportados son ANSI-C (GCC 4.6), C++ (G++ 4.6), OpenCL API (1.0) y SystemC (IEEE 1666-2006, version 2.2) (Xilinx - UG902, 2017). La mayoría de los constructores C, C++ y SystemC son soportados, con la excepción de aquellos que hagan uso de operaciones típicas de OS. Por ejemplo, constructores para asignación dinámica de memoria, escritura y lectura de archivos[5] no pueden ser usados. Por otro lado, los tipos de datos para estos lenguajes son soportados en su totalidad.

Vivado HLS provee extensiones para los estándares C. Estas extensiones corresponden a librerías, las cuales contienen funciones y constructores optimizados para la implementación

[5]La escritura y lectura de archivos no está soportada en las funciones a ser sintetizadas. Sin embargo, pueden ser usadas en *test benchs*.

en FPGA (Xilinx - UG902, 2017). El uso de estas librerías está enfocado a alcanzar resultados de alta calidad que aprovechen de forma eficiente los recursos. Sin embargo, la cantidad de librerías es limitada, por lo cual, aunque puede facilitar el diseño, no es un método general para sintetizar especificaciones C. Las librerías C disponibles en Vivado HLS son: i) tipos de datos de precisión arbitraria, ii) tipos de datos de punto flotante con media precisión (16 bits), iii) operaciones matemáticas, iv) funciones para video, v) FFT (Transformada Rápida de Fourier) y filtros FIR (Respuesta Finita al Impulso), y vi) funciones para maximizar el uso registros de desplazamiento en LUTs (Xilinx - UG902, 2017). Si se requiere información detallada de cada una de las librerías mencionadas, se puede consultar el documento "*Vivado Design Suite User Guide – HLS UG902*" (Xilinx - UG902, 2017).

3.2.3. Tipos de Datos

La selección de los tipos de datos para Vivado HLS es fundamental debido a que incide directamente sobre los resultados, recursos y consumo energético de la implementación final. Para comprender esto, es necesario conocer las diferencias entre los tipos de datos nativos de estándares C y tipos de datos de precisión arbitraria de Vivado HLS. Estos últimos son también llamados tipos de datos para hardware eficiente y es precisamente el objetivo que persiguen, reducir el número de recursos innecesarios que serían ocupados al usar tipos de datos nativos, para así obtener un diseño de hardware eficiente.

3.2.3.1. Tipos de Datos C Estándar

Los lenguajes C y C++ tienen un conjunto de datos básicos. Estos son: carácter (`char`), entero (`int`), punto flotante de precisión simple (`float`), y punto flotante de precisión doble (`double`). De este conjunto existen otros tipos derivados mediante los llamados especificadores de tipo, los cuales son: `long`, `long long`, `short`, `unsigned`, y `signed` (Kochan, 2014). Todos ellos pueden ser usados en Vivado HLS. En la Tabla 3 se resumen los tipos de datos nativos

de estándares C con su descripción, el número de bits y rango de cada uno.

Tabla 3
Tipos de datos nativos de lenguaje C

Tipo de dato	Descripción	Bits	Rango
`char`	Carácter simple	8	-128 a 127
`short int`	entero de precisión reducida	16	-32768 a 32767
`unsigned short int`	entero de precisión reducida	16	0 a 65535
`int`	entero básico	32	-2,147,483,648 a 2,147,483,647
`unsigned int`	entero de precisión extendida	32	0 a 4,294,967,295
`long long int`	entero de precisión extendida	64	-2,147,483,648 a 2,147,483,647
`unsigned long long int`	entero de precisión extendida	64	0 a 4,294,967,295
`float`	punto flotante de precisión simple	32	-3.403e+38 a 3.403e+38
`double`	punto flotante de precisión doble	64	-1.798e+308 a 1.798e+308
`long double`	punto flotante de precisión doble extendida	80	-3.4e-4932 a 1.1e+4932

Fuente: (Kochan, 2014)

Como se observa, estos tipos de datos son múltiplos de 8 bits. Esto se debe a que están dirigidos a software en procesadores. Sin embargo, el uso de este tipo de datos no es ideal para la creación de hardware y resultaría en un diseño ineficiente. Por ejemplo, una multiplicación de 18×18 bits puede llevarse a cado en un DSP48, pero si el tipo de dato de cada factor es definido como `int` (32 bits), esto requeriría de tres DSP48 (Xilinx - UG902, 2017). El uso de este tipo de datos representaría un uso ineficiente de recursos, ya que se usaría dos DSP48 más de lo requerido para el objetivo deseado. Esto evidencia la necesidad de utilizar tipos de datos que se adecuen al diseño de hardware.

3.2.3.2. Tipos de Datos Eficientes para Hardware

Los tipos de datos de precisión arbitraria permiten crear variables de longitudes personalizadas. El uso de este tipo de datos significa una reducción de los recursos de lógica programable utilizados y del consumo energético, ademas de un posible incremento de las

frecuencias de reloj máximas alcanzables (Xilinx - UG902, 2017). Vivado HLS ofrece soporte para precisión arbitraria de tipos enteros y punto fijo para C y C++, y, además tipos de datos de precisión arbitraria propios de SystemC.

Tipos enteros de precisión arbitraria

Los tipos de datos enteros de precisión arbitraria pueden ser usados mediante la inclusión de cabeceras de referencia a librerías incluidas en Vivado HLS. En la Tabla 4 se resumen estos tipos de datos para los lenguajes C, C++ y SystemC.

Tabla 4
Tipos de datos enteros de precisión arbitraria en Vivado HLS

Lenguaje	Tipo de dato	Descripción	Header requerido
C	`[u]int<W>`	Entero de W bits de precisión hasta 1024 bits	#include “ap_cint.h”
C++	`ap_[u]int<W>`	Entero de W bits de precisión hasta 1024 bits	#include “ap_int.h”
SystemC	`sc_[u]int<W>`	Entero de W bits de precisión hasta 64 bits	#include “systemc.h”
	`sc_[u]bigint<W>`	Entero de W bits de precisión hasta 512 bits	

Fuente: (Xilinx - UG902, 2017)

Tipos de punto fijo de precisión arbitraria

Los tipos de datos de punto fijo de precisión arbitraria tienen un formato específico. Estos tipos constan de un cierto número de bits que representa la parte entera y otro número de bits que representa la parte fraccionaria. Para su uso deben ser añadidas las cabeceras de las librerías mostradas en la Tabla 5. Cabe resaltar que este tipo de datos está disponible para C++ y SystemC, mas no para lenguaje C.

Tabla 5
Tipos de datos de punto fijo de precisión arbitraria en Vivado HLS

Lenguaje	Tipo de dato	Descripción	Header requerido
C++	`ap_[u]fixed<W,I,Q,O,N>`	Punto fijo de W bits enteros y W-I bits fraccionarios	#include "ap_fixed.h"
SystemC	`sc_[u]fixed<W,I,Q,O,N>`	Punto fijo de W bits enteros y W-I bits fraccionarios	#include "systemc.h"

Fuente: (Xilinx - UG902, 2017)

3.2.4. Manejo de Interfaces

El proceso de síntesis de una especificación C a un módulo de hardware involucra la creación de interfaces que permitan su comunicación. Este proceso es conocido como síntesis de interfaces. Vivado HLS sintetiza los argumentos de una función *top-level* y sus valores de retorno como puertos. Cada uno de estos puertos debe tener asociado un protocolo de comunicación que posibilite su intercambio de datos. El conjunto puerto y protocolo es definido como interfaz. Vivado HLS crea interfaces a nivel de puerto (argumentos de la función *top-level*) y, además, interfaces a nivel de bloque que coordinan la comunicación con otros subsistemas. Este concepto se muestra en la Figura 19 y será ampliado durante esta sección.

Ejemplo 3.2

El proceso de síntesis de interfaces puede llevarse a cabo por defecto en Vivado HLS o influenciarse mediante el uso de directivas. En el Código 3 se muestra la función *mac_hls* la cual corresponde a un MAC (*Multiplier Accumulator*). En la Figura 19 se muestra la representación simplificada de sus interfaces después de la síntesis por defecto del Código 3. La función *mac_hls* tiene tres argumentos definidos como tipo `int`, por lo que `a`, `b` y `accum` (*accum_i*) son sintetizados como puertos de datos de 32 bits. Como no existe un valor especificado de retorno, Vivado HLS asume que `accum` es el valor de salida del bloque y lo implementa como un puerto de datos de 32 bits (*accum_o*). Además, son añadidas por defecto las señales como

clock y *reset* a todos los diseños HLS en Vivado que requieran más de un ciclo de reloj para completar sus operaciones. Si se requiere, una señal de *enable* también puede ser añadida en la configuración de interfaces. En la Figura 19, también se aprecian otras señales a nivel de bloque, las mismas que serán discutidas posteriormente.

Código 3
Función mac_hls escrita en lenguaje C

```
1  void mac_hls(int a, int b, int *accum)
2  {
3      int accum_aux= 0;
4      accum_aux += a * b;
5      *accum = accum_aux;
6  }
```

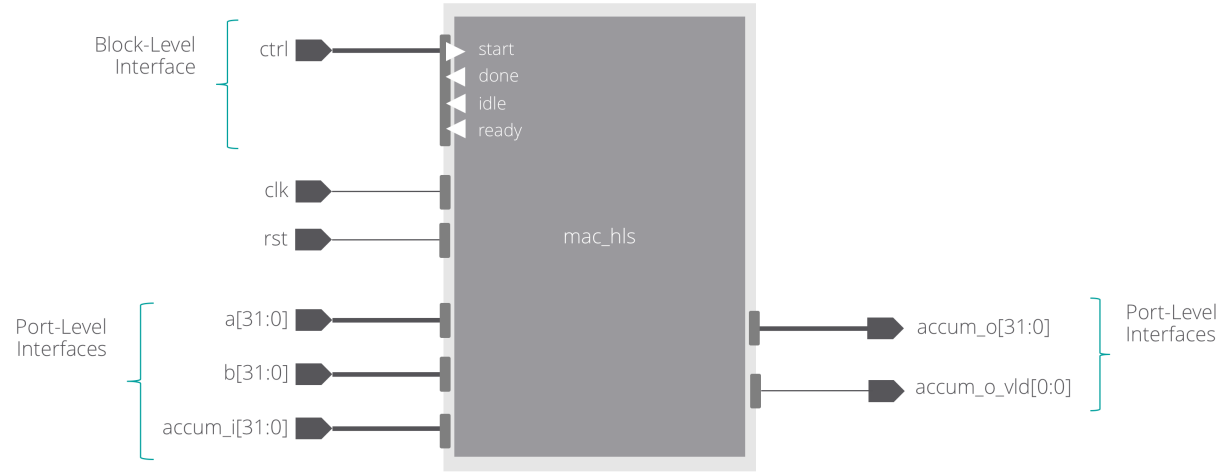


Figura 19. Interfaces generadas por defecto en Vivado HLS para la función *mac_hls*. La figura muestra una representación simplificada de las interfaces generadas por defecto en Vivado HLS. La dirección de los puertos es inferida a partir de las operaciones de lectura y escritura. Señales de *clock* (Clk) y *reset* (rst) son añadidos por defecto a todos los diseños.

Interfaces a nivel de puerto

Las interfaces a nivel de puerto permiten enviar datos hacia el bloque y hacia fuera del bloque (Xilinx - UG902, 2017). Vivado HLS genera por defecto puertos como cables simples sin ningún protocolo definido, por ejemplo, los mostrados en la Figura 19. Esto puede complicar la lectura y escritura de datos, por lo cual el uso de protocolos debe ser considerado.

Vivado HLS ofrece varios protocolos disponibles para los puertos, donde el diseñador tiene la opción de seleccionar el protocolo para cada uno. Sin embargo, existen restricciones según la dirección del puerto y el tipo de argumento (Crockett et al., 2015). Un resumen completo de los protocolos soportados y por defecto por tipo de argumento se muestra en la Tabla 6.

Tabla 6
Protocolos soportados en interfaces de Vivado HLS

Tipo de Argumento		Escalar		Matriz			Puntero		
Tipo		I	O	I	I/O	O	I	I/O	O
Interfaces AXI4	axix	S		S		S	S		S
	s_axilite	S	S	S	S	S	S	S	S
	m_axi			S	S	S	S	S	S
Sin protocolo I/O	ap_none	D					D	S	S
	ap_stable	S					S		
Wire Handshakes	ap_ack	S					S	S	S
	ap_vld	S					S	S	D
	ap_ovld							D	S
	ap_hs	S		S		S	S	S	S
Interfaces de memoria	ap_memory			D	D	D			
	Bram			S	S	S			
	ap_fifo			S		S			
Interfaz de bus	ap_bus			S	S	S	S	S	S

Nota: I=entradas; O=salidas; D=por defecto; S= soportado.

En la Tabla 6 se pueden observar que los protocolos de interfaces a nivel de puerto están agrupados en cuatro principales categorías: 1) interfaces AXI4, 2) sin protocolo I/O, 3) *Wire Handshakes*, y 4) interfaces de memoria. Cada uno de estos protocolos se describe a continuación.

1. **Interfaces AXI4.** AXI (*Advanced eXtensible Interface*) es un estándar abierto para comunicaciones on-chip que es parte de ARM AMBA y ampliamente usado en SoCs basados en FPGA (Xilinx - UG761, 2011). El estándar define tres tipos de interfaces: (1) AXI4-Stream, para transmisión de datos *streaming* a alta velocidad, (2) AXI4-Lite, para mapeo de memoria sin transmisiones *burst* de datos, y (3) AXI4 master, igual que AXI4-Lite pero con soporte para transmisiones *burst* de datos.

2. **Sin protocolo I/O.** Ningún protocolo es añadido al puerto de datos. Este tipo es usado únicamente para entradas de configuración del bloque.

3. **Wire handshakes.** Son interfaces con el estándar industrial *handshake* bidireccional.

4. **Interfaces de memoria.** Interfaces para puertos que acceden a block RAMs con puertos de datos, dirección y *enable* de escritura.

5. **Interfaz de bus.** Interfaz para comunicarse con buses.

Los protocolos 2, 3, 4 y 5 descritos arriba, son utilizados para comunicarse a nivel de IP-cores (hardware). Por el contrario, el uso de interfaces AXI4 está orientado a comunicar el módulo con un CPU o un microcontrolador ejecutando software. Para ello, Vivado HLS crea de forma automática Wrappers, lógica de las interfaces AXI4 y drives, lo cual representa una ventaja significativa debido a que los módulos pueden ser complejos, y anteriormente se requería de codificación en HDL. Vivado HLS genera también los drivers (software) para controlar el módulo de hardware. De esta forma un CPU puede iniciar o detener la ejecución de las operaciones del IP, así como también enviar datos y obtener sus resultados. En el desarrollo de la tesis se usa las interfaces AXI4-Lite para comunicar la lógica programable de FPGA con un procesador embebido.

Interfaces a nivel de bloque

Las interfaces a nivel de bloque permiten controlar el funcionamiento y ejecución de un IP. Vivado HLS ofrece soporte para tres tipos de protocolos: 1) ap_ctrl_none, 2) ap_ctrl_hs y 3) ap_ctrl_chain. Vivado selecciona por defecto el protocolo ap_ctrl_hs, como se muestra en el ejemplo de la Figura 19. Cada uno de estos tres protocolos se describe a continuación.

1. **ap_ctrl_none.** No se añade ningún protocolo específico. El control del funcionamiento se deja por completo a cargo de los protocolos asignados a los puertos.

2. **ap_ctrl_hs.** Hace uso del denominado *handshaking*, el cual ocupa cuatro señales: a) *ap_start*: señal de entrada para la activación de funcionamiento del bloque, b) *ap_ready*: señal de salida que indica la disponibilidad del bloque para recibir nuevas entradas, c) *ap_idle*: señal de salida que indica que el IP se encuentra procesando datos en ese momento, d) *ap_done*: señal de salida que indica que los resultados se encuentran disponibles (Crockett et al., 2015).

3. **ap_ctrl_chain.** Funciona de igual forma que *ap_ctrl_hs* con la diferencia de una señal extra. Esta señal es denominada *ap_continue* y es útil cuando varios bloques son conectados en cadena. *ap_continue* indica que el siguiente bloque en la cadena se encuentra listo para recibir nuevos datos.

3.3. Verificación y Optimización en Vivado HLS

3.3.1. Verificación Mediante Co-Simulación C/RTL

Una de las ventajas de Vivado HLS es la automatización del proceso de verificación post-síntesis. Esta automatización se logra gracias a una co-simulación C/RTL, la cual reúsa el *test bench* C para verificar la salida RTL (Xilinx - UG902, 2017). Mediante este proceso se logra eliminar la necesidad de crear manualmente un *test bench* RTL, ahorrando tiempo al diseñador. Vivado HLS realiza el proceso de verificación en tres fases: i) Simulación C, ii) Simulación RTL, y iii) Verificación Post-Simulación. Estas fases se representan en la Figura 20 y son descritas a continuación.

- **Simulación C.** Se ejecuta la simulación mediante el *test bench* C. Se llama a la función *top-level* del DUT y se aplican vectores de entrada para obtener sus resultados y compararlos con los resultados esperados o *golden results*.

- **Simulación RTL.** Los vectores de entrada son aplicados al modelo RTL en una simu-

lación generada automáticamente por Vivado HLS. Los resultados de este modelo son denominados como vectores de salida RTL.

- **Verificación post-simulación.** Los vectores de salida de la simulación RTL son comparados con los resultados esperados para encontrar posibles diferencias.

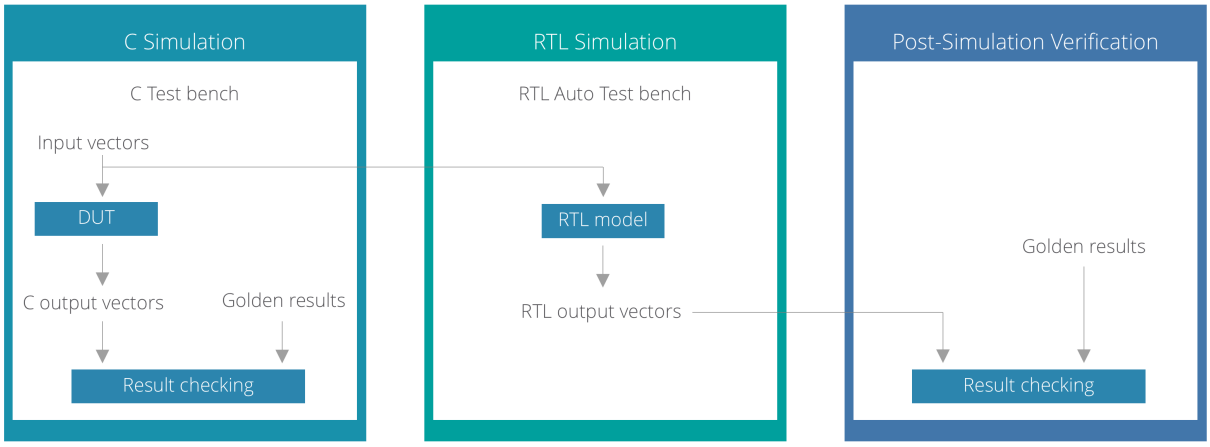


Figura 20. Flujo de verificación de Vivado HLS mediante co-simulación C/RTL.

3.3.2. Análisis de Resultados de Síntesis en Vivado HLS

Reporte de síntesis

Vivado HLS genera reportes de síntesis como recursos para el análisis de cada solución RTL. El reporte provee detalles acerca del desempeño y el área utilizada de cada implementación. Estos reportes están divididos por categorías, entre las cuales se cuenta con información sobre: i) desempeño estimado, ii) estimación de uso de recursos, y iii) reporte de interfaces. En la Tabla 7 se amplía información sobre los elementos que conforman cada una de las categorías, mientras que en la Figura 21 se muestra un ejemplo de un reporte de síntesis.

Tabla 7
Elementos del reporte de síntesis C en Vivado HLS

Categoría	Elemento	Descripción
Desempeño estimado	Timing	Frecuencia de reloj objetivo, incertidumbre de reloj, y estimación de la frecuencia de reloj.
	Latency	Latencia e intervalo de ejecución. Estas métricas son especificadas en ciclos de reloj. Así también, se muestra el detalle de latencia e intervalo para todas las sub-funciones y lazos.
Uso de recursos	Summary	Muestra la cantidad de LUTs, Flip-Flops, y DSP48s usados en la implementación del diseño.
	Details	Muestra el detalle de los recursos para implementar: memorias, FIFOs, registros de desplazamiento, multiplexores, etc.
Reporte de interfaces	Interfaz	Muestra el detalle de síntesis de interfaces para cada argumento de la función top-level

Fuente: (Xilinx - UG902, 2017)

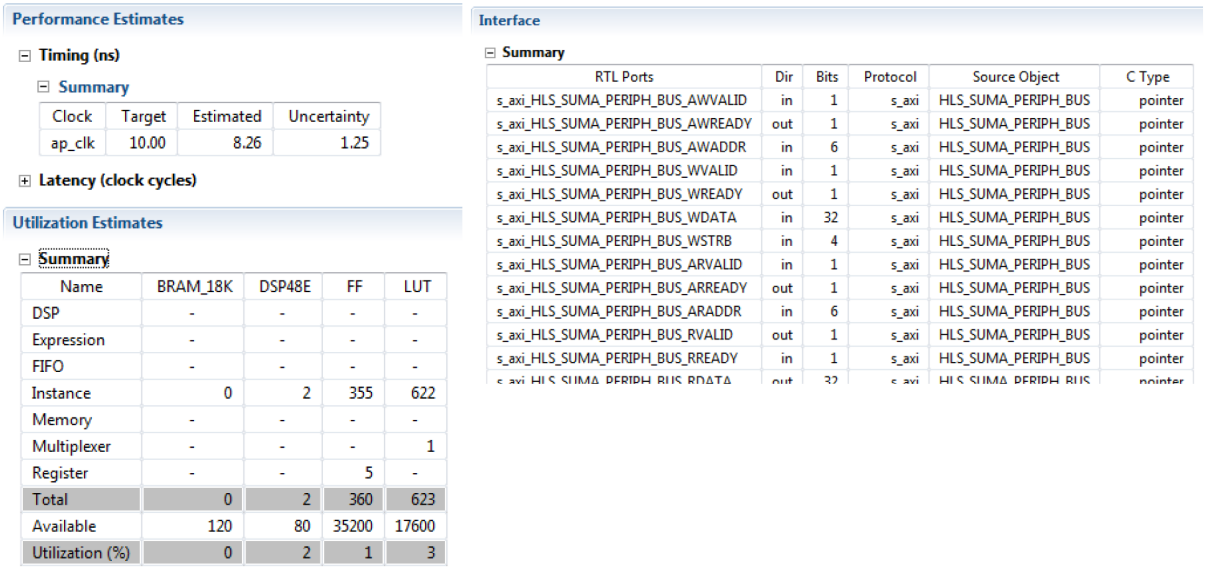


Figura 21. Ejemplo de reporte de síntesis en Vivado HLS. La ventana muestra las tres categorías principales: Desempeño Estimado, Estimación del Uso de Recursos y Reporte de Interfaces.

Perspectiva de análisis

Vivado HLS ofrece una perspectiva de análisis en su interfaz GUI. Esta perspectiva está orientada a facilitar el análisis de desempeño de los bloques sintetizados. Su función es facilitar

el análisis en base al cual se puede optimizar el diseño. Un ejemplo de esta ventana puede observarse en la Figura 22. En ella se muestran tres secciones: (1) jerarquía del módulo en la parte superior izquierda donde se resumen los recursos utilizados por bloques y sub-bloques (block RAM, *pipelined multipliers*, etc), (2) resumen de performance en la parte inferior izquierda en donde se resume el performance de cada componente del bloque y sub-bloques (lazos) en base a la latencia, (3) vista gráfica donde se representa la programación o *scheduling* de las operaciones en cada ciclo de reloj.

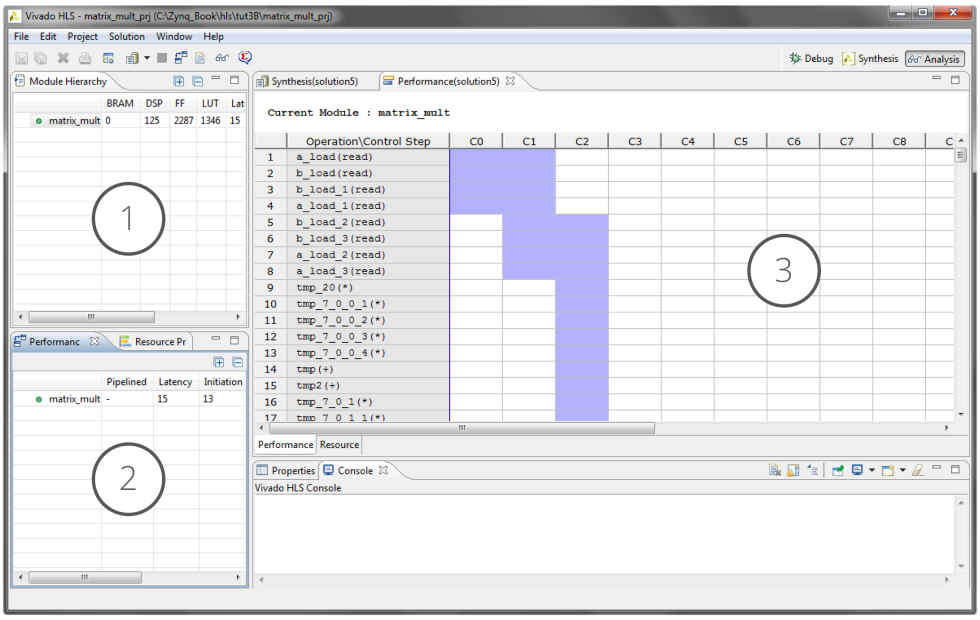


Figura 22. Perspectiva de análisis de Vivado HLS.

La vista gráfica de la perspectiva de análisis se denomina *Schedule View*, la cual consta de: a) lista de recursos en la columna de la izquierda, de forma que se presenta en color verde los sub-bloques, en amarillo las operaciones resultantes de lazos, y en morado operaciones estándar, b) la fila superior lista los estados de control de la FSM usada en Vivado HLS para realizar la etapa de *scheduling* (Xilinx - UG902, 2017).

3.3.3. Restricciones y Directivas de Optimización

En esta sección se revisa conceptos acerca de restricciones (denominadas generalmente como *constraints*) y directivas de optimización, las cuales son entradas a Vivado HLS usadas por el diseñador para ejercer cierto control sobre el proceso de síntesis (ver Figura 16). El diseñador debe obligatoriamente definir restricciones para realizar la síntesis, mientras que las directivas de optimización son opcionales para lograr una implementación de hardware que satisfaga las métricas de diseño.

Restricciones (constraints)

La síntesis de Vivado HLS es dirigida en base a restricciones de diseño, las cuales se describen a continuación.

1. **El periodo de reloj.** Vivado requiere la definición del periodo de reloj en nanosegundos, con ello puede estimar el tiempo de cada operación y realizar el proceso de *scheduling*.

2. **La incertidumbre de reloj.** Debido a que Vivado no puedo calcular en primera instancia el tiempo necesario para realizar los procesos de *placement* y *routing*, es necesario definir una incertidumbre de reloj, la misma que es restada del periodo de reloj y provee un margen para la realización de estos procesos (Xilinx - UG902, 2017), como se muestra en la Figura 23.

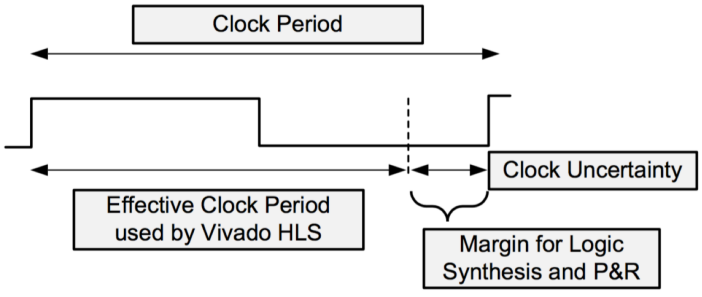


Figura 23. Periodo de reloj e incertidumbre de reloj en Vivado HLS.
Fuente: (Xilinx - UG902, 2017)

3. **El dispositivo FPGA de destino.** El identificador del dispositivo FPGA proporciona a Vivado HLS información acerca del número de recursos de hardware disponibles (LUTs, registros, RAM bloques y DSP48s) necesario para el proceso de *allocation*. En base a esta información Vivado calcula el área máxima utilizable para la cual optimizará la implementación del diseño.

Directivas de optimización

Vivado HLS permite usar varias directivas de optimización para satisfacer los objetivos de diseño. Estas directivas pueden ser agrupadas de acuerdo al tipo de optimización que persiguen como: **i) optimizaciones de desempeño:** permiten generar implementaciones con técnicas de paralelización como pipelinig y dataflow, **ii) optimizaciones de latencia:** pretenden reducir el número de ciclos de reloj requeridos para completar sus operaciones, o **iii) optimizaciones de área:** buscan reducir el número de recursos de hardware utilizados para la implementación.

3.3.4. Optimizaciones de Desempeño

3.3.4.1. Pipelining

Pipelining es una técnica diseñada para maximizar el desempeño en el diseño de hardware. Normalmente una tarea que está compuesta por un conjunto de operaciones debe ser ejecutada secuencialmente para obtener un resultado final. Esto se debe a que en la mayoría de casos las operaciones son dependientes de resultados previos. En el Código 4 se muestra la función *example1*, cuya ejecución secuencial se presenta en la Figura 24 a), donde cada operación es ejecutada únicamente cuando la anterior se ha completado. Cuando se habla de desempeño es importante analizar dos conceptos, la latencia y el intervalo de iniciación. La latencia será el tiempo en ciclos de reloj desde que una nueva entrada es leída hasta que se entrega su respectiva salida. El intervalo de iniciación es el tiempo entre lecturas de nuevas entrada. En el ejemplo de la Figura 24 a), la latencia es igual a dos ciclos de reloj y el intervalo

de iniciación es de tres ciclos.

Código 4
Función example1 escrita en lenguaje C

```
1  void example1 (...){
2       read;
3       compute;
4       write;
5  }
```

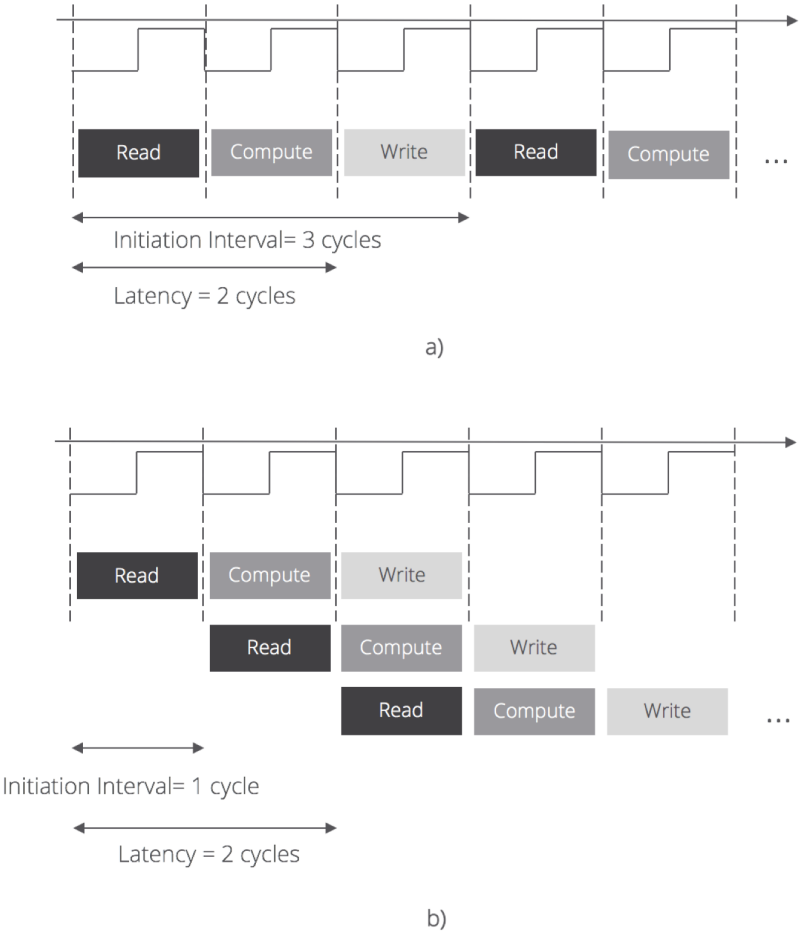


Figura 24. Ejecución de la función *example1* del Código 4. Implementación: a) sin *pipelining*, b) con *pipelining*

Utilizando *pipelining*, las operaciones previas no tienen que haber finalizado para que una nueva operación empiece a ejecutarse, es decir, varias operaciones pueden ejecutarse de forma concurrente. Vivado HLS permite utilizar *pipelining* tanto para funciones como para

lazos. La misma función *example1* implementada usando *pipeling* se muestra en la Figura 24 b). Esta implementación obtiene un intervalo de iniciación de un ciclo de reloj, y sigue manteniendo la latencia de dos ciclos.

Una vez explicado el concepto de *pipelinig*, se presenta un ejemplo donde se evidencia mayormente las ventajas de su uso. En el Código 5 se muestra la función *example2*, la cual incluye un lazo que es iterado tres veces. Su implementación sin *pipelining* se muestra en la Figura 25 a), donde se puede observar que son necesarios ocho ciclos de reloj para finalizar sus operaciones. La misma función *example2* implementada con pipelining se muestra en la Figura 25 b). La latencia de esta implementación se reduce a cuatro ciclos de reloj, obteniendo una mejora de desempeño significativa. Vivado permite realizar *pilining* a lazos y funciones a través de la directiva denominada `PIPELINING`.

Código 5
Función example2 escrita en lenguaje C

```
1  void example2(...){
2    for(i=0;i<3;i++){
3       read;
4       compute;
5       write;
6    }
7  }
```

3.3.4.2. Particionamiento de Arreglos

Al usar *pipelining* en Vivado HLS pueden existir dificultades en alcanzar un intervalo de iniciación especificado. Esto sucede cuando el intervalo es menor que el tiempo que toma realizar las operaciones de lectura de datos. Esto es común cuando se tiene arreglos, los cuales son implementados como bloques RAM con dos puertos de datos, lo que limita la eficiencia en operaciones de lectura y escritura (Xilinx - UG902, 2017). Este cuello de botella puede solucionarse mediante la división de arreglos en partes más pequeñas o incluso en elementos individuales consiguiendo más puertos de datos y mejorando la eficiencia de lectura y escritura, lo cual permite mejorar el *pipelining*.

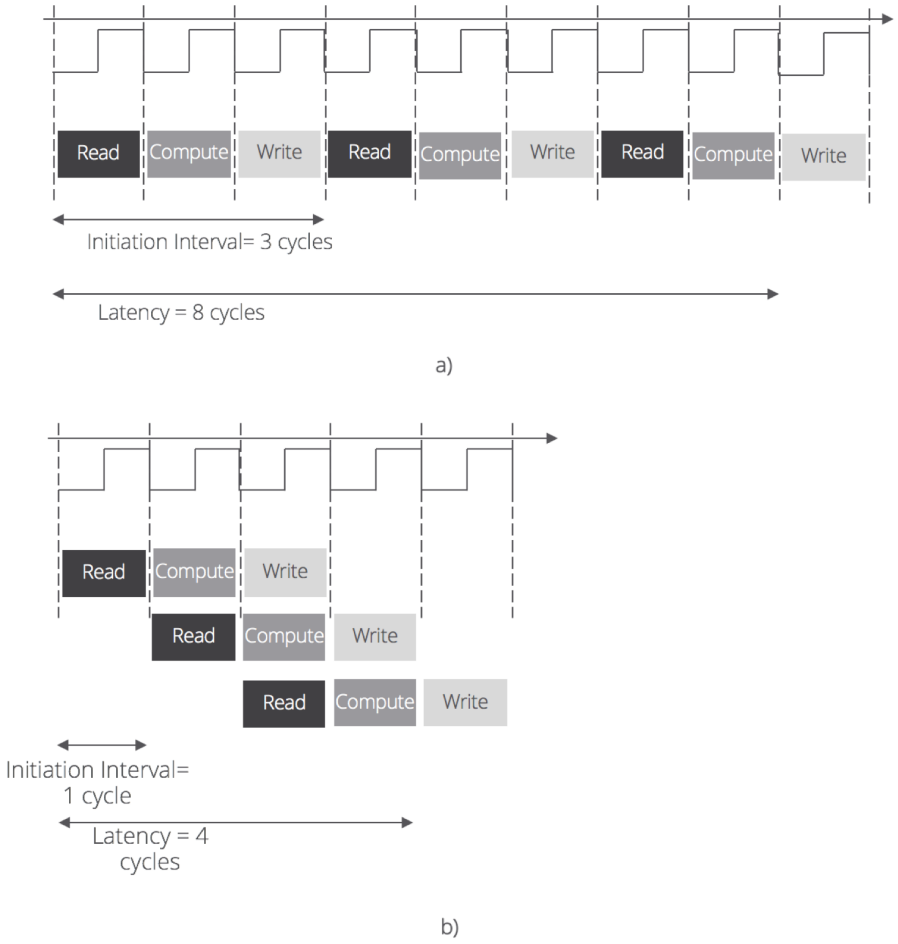


Figura 25. Ejecución de la función *example2* del Código 5. Implementación: a) sin pipelining, b) con pipelining
Fuente: (Xilinx - UG902, 2017)

Vivado permite usar la directiva `ARRAY_PARTITION` para conseguir uno de los tres tipos de particionamiento disponibles, los cuales se ilustran en la Figura 26. Se definen como: **i) block**: divide el arreglo en varios subarreglos más pequeños de una misma dimensión y organiza los elementos consecutivamente, **ii) cyclic**: funciona igual que block, pero los elementos son organizados de forma intercalada, **iii) complete**: divide el arreglo en elementos individuales. Para i) y ii) el factor de particionamiento puede ser seleccionado, y representa el número de arreglos a ser creados a partir del original. Si la división del número de elementos para el factor de particionamiento no es entero, entonces el último arreglo tendrá menos elementos (Xilinx - UG902, 2017). En caso de usar arreglos multidimensionales se puede especificar la

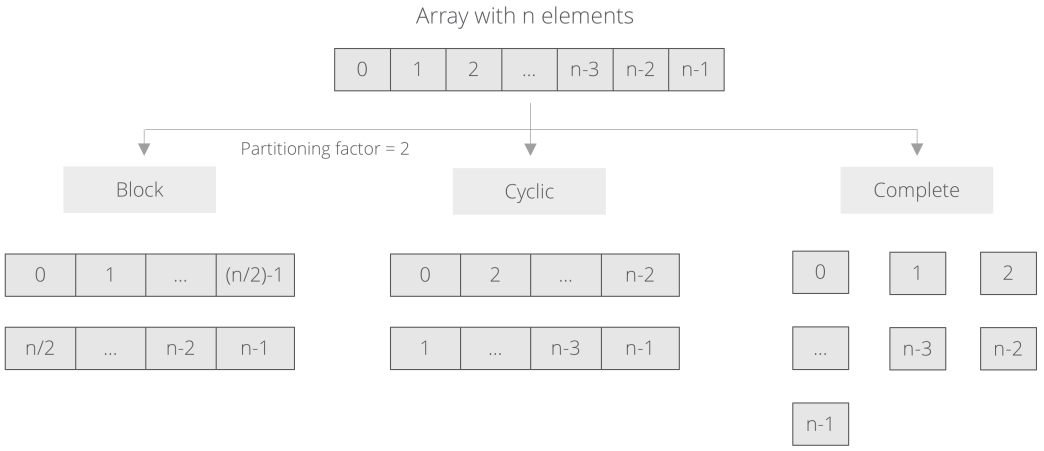


Figura 26. Particionamiento de arreglos en subarreglos o en elementos individuales. Fuente: (Xilinx - UG902, 2017)

dimensión a particionarse.

3.3.4.3. Unrolling Loops

Los lazos dentro de sub-funciones son mantenidos por defecto en la forma denominada *rolled*, como se mencionó en la Sección 3.1.2.1. La forma *rolled* de un lazo quiere decir que HLS crea el hardware necesario para realizar una iteración, y ocupará los mismos recursos para las demás iteraciones. Sin embargo, el diseñador puede variar la forma en la que se implementan los lazos mediante la directiva `UNROLL`. HLS permite realizar un *unroll* parcial o total, con lo cual se creará hardware para ejecutar varias o todas las iteraciones de un lazo de forma paralela siempre que las dependencias de datos lo permitan (Xilinx - UG902, 2017). Para ilustrar estos conceptos se presenta la función *loop* en el Código 6, el cual realiza la suma de dos vectores, elemento a elemento, en un ciclo `for`. La implementación de este lazo con las tres opciones posibles: i) *Rolled*, ii) *Unrolled* parcial y iii) *Unrolled* se describe a continuación.

Código 6
Función loop escrita en lenguaje C

```
1  void loop (...){
2       for(i=0;i<4;i++){
```

```
3               sum[i]=a[i]+b[i];
4           }
5   }
```

- **Rolled.** La implementación realiza una iteración en cada ciclo de reloj usando los mismos recursos de hardware. En la Figura 27 se muestra la implementación de la función *loop* de la forma *rolled*. En esta figura se observa que para completar las operaciones del lazo se requiere cuatro ciclos de reloj usando el recurso *AddSub_0* en cada uno de los ciclos.

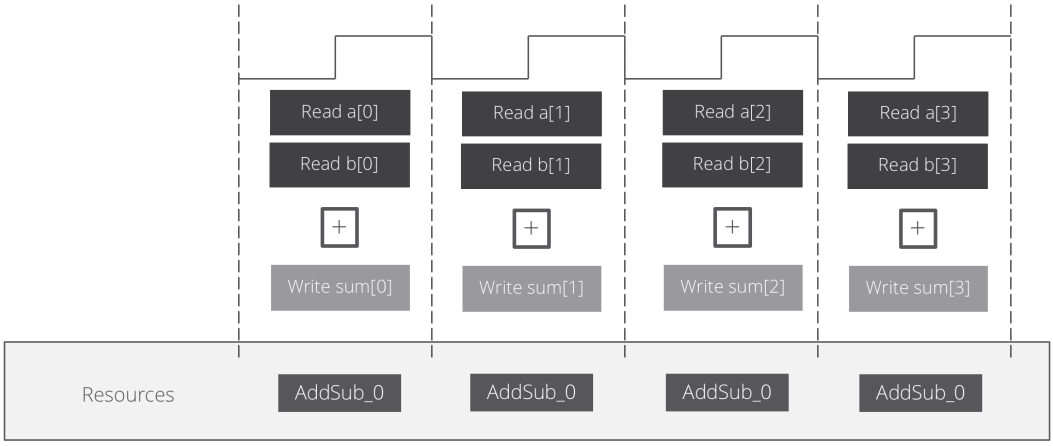


Figura 27. Implementación *rolled* de la función *loop* del Código 6.
Fuente: (Xilinx - UG902, 2017)

- **Unrolled parcial.** Esta implementación permite realizar *unroll* de un lazo en un factor determinado. Por ejemplo, en la Figura 28 a) se muestra la implementación de la función *loop* en un factor de 2, la cual usa dos AddSub y toma dos ciclos de reloj para completar sus operaciones.

- **Unrolled.** Una implementación totalmente *unrolled* crea todos los recursos de hardware necesarios para realizar todas las iteraciones de un lazo de forma paralela. Esta implementación de la función loop se muestra en la Figura 28 b), en donde se usan cuatro AddSubs. En este caso cabe resaltar que se realizan cuatro lecturas simultaneas de los arreglos `a` y `b`, y se escribe cuatro valores de forma simultánea en el vector `sum`.

Si estos argumentos están implementados como puertos RAM, entonces deben ser particionados para poder conseguir la cantidad suficiente de puertos de escritura/lectura.

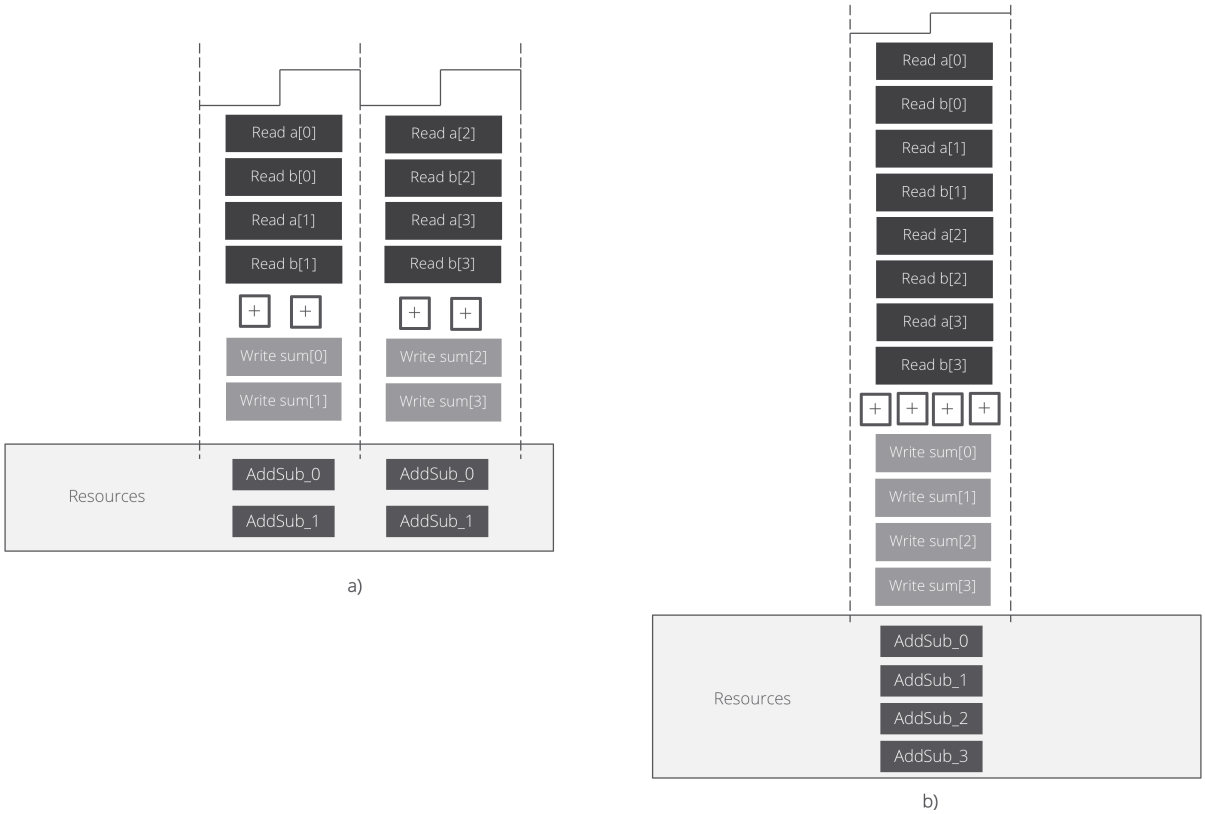


Figura 28. Implementaciones de la función *loop* del Código 6 utilizando unrolled. Implementación con: (a) *unrolled* total, (b) *unrolled* parcial
Fuente: (Xilinx - UG902, 2017)

3.3.4.4. Dataflow

Dataflow es un tipo de optimización orientada a mejorar el desempeño de una implementación de forma similar a *pipelining*. La diferencia radica en que *pipelining* se aplica a las operaciones dentro de funciones y lazos, mientras que *dataflow* se aplica a un nivel superior, paralelizando la ejecución de varias funciones (Crockett et al., 2015). La directiva de Vivado HLS es denominada `DATAFLOW`. A continuación, se presenta la función *func_top_level* en el Código 7, la cual llama a una jerarquía de sub-funciones, las mismas que pueden ser implementadas con o sin *dataflow* como se muestra en la Figura 29.

Código 7
Función func_top_level escrita en lenguaje C

```
1  void func_top_level (in_0, in_1, out_1, out_2) {
2      func_1(in1, in2, aux_out1);
3      func_2(aux_out1, aux_out2);
4      func_3(aux_out3, out_1, out_2)
5  }
```

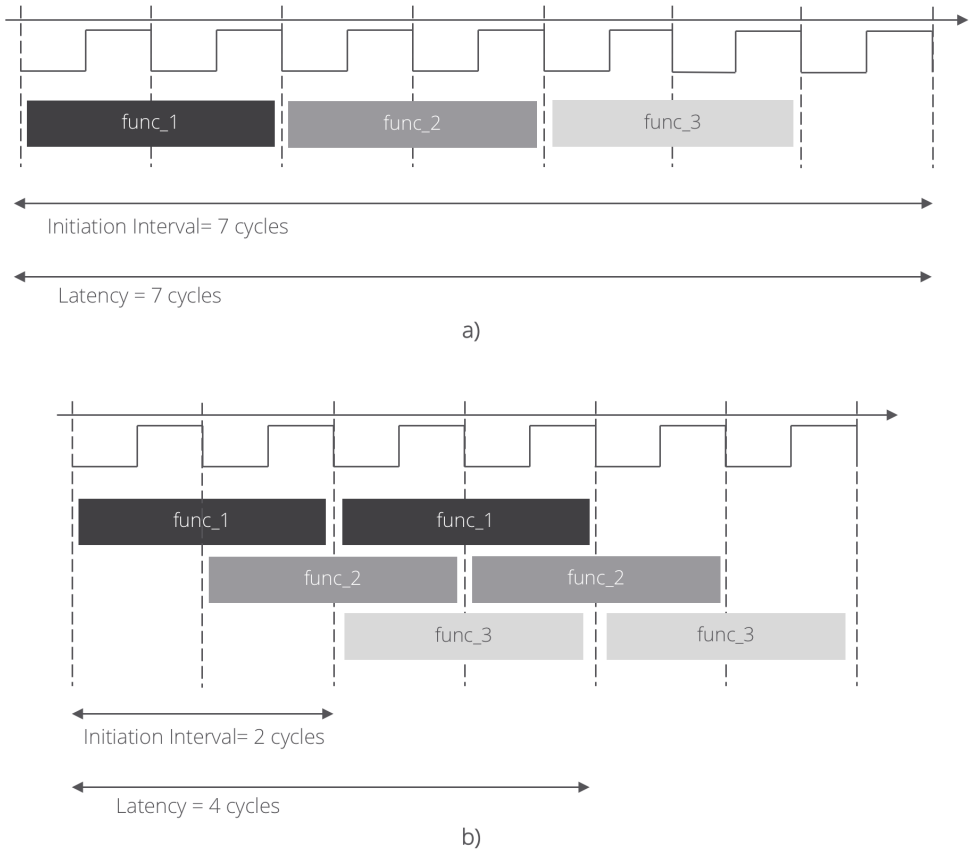


Figura 29. Implementación de la jerarquía de sub-funciones de la función *func_top_level* del Código 7: (a) sin dataflow, (b) con dataflow

En la Figura 29 a) se muestra una implementación secuencial de las sub-funciones, donde el intervalo de iniciación de la función *top-level* es de siete ciclos, y tiene una latencia también de siete ciclos. Por otro lado, en la Figura 29 b) se presenta una implementación usando *dataflow*, donde el intervalo de iniciación de la función se reduce a dos ciclos y la latencia a cuatro ciclos, siendo evidente la mejora en desempeño ofrecida por esta optimización.

Es evidente que la implementación de *dataflow* para maximizar el desempeño es mucho más compleja que la implementación de *pipelinig*. HLS requiere que las funciones se encuentren conectadas de una forma específica para que en su análisis pueda crear el *data-path* necesario para su implementación. Para ello, el diseñador tiene que definir la función *top-level* y lazos en su interior en la forma estándar de Vivado HLS. Esta forma es la utilizada en la función *func_top_level* mostrada en el Código 7. La primera función lee las entradas y genera valores internos que son utilizados por la siguiente función, y así sucesivamente hasta que la última función escriba las salidas.

3.3.5. Otras Optimizaciones

3.3.5.1. Optimizaciones de Latencia

En muchas aplicaciones puede ser necesario que la implementación cumpla con una latencia específica. Es decir, que todas las operaciones de una función o un lazo sean ejecutadas en un número de ciclos de reloj específico. Para ello, Vivado HLS permite utilizar restricciones de latencia máxima y mínima mediante la directiva `LATENCY`. Por ejemplo, Vivado permite especificar la latencia máxima de una iteración dentro de un lazo, o la latencia total de un lazo para completar sus operaciones. Cuando esto es llevado acabo, puede ser posible que Vivado HLS no pueda crear una implementación que cumpla con una cierta latencia. De ser el caso, se generará la solución con la menor latencia posible. Por el contrario, y en casos menos frecuentes, puede especificarse una latencia mínima. En tal caso Vivado utiliza tiempos muertos en la implementación para cumplir esta restricción. Vivado es capaz de usar dos técnicas orientadas a optimizar la latencia, "*Merging Sequential Loops*" y "*Flattening Nested Loops*". Ambas técnicas están enfocadas a reducir el número de ciclos de reloj necesarios para llevar a cabo las operaciones ejecutadas dentro de lazos (Xilinx - UG902, 2017).

Loop Flattening

Loop flattening convierte dos o más lazos anidados en uno solo automáticamente (Xilinx - UG902, 2017). Esta optimización permite eliminar los ciclos de reloj necesarios para entrar y salir de lazos interiores. Para aplicar *loop flattening* se requiere que no existan instrucciones en medio de los lazos. En la Figura 30 a) se presenta un ejemplo de dos lazos anidados, L1 y L2. El lazo L1 debe entrar y salir de L2 en cinco ocasiones, lo cual requerirá diez ciclos de reloj. En la Figura 30 b) se muestra la misma función implementando *loop flattening*. Gracias a esta optimización, los lazos se convierten en uno solo, disminuyendo la latencia de ejecución en diez ciclos.

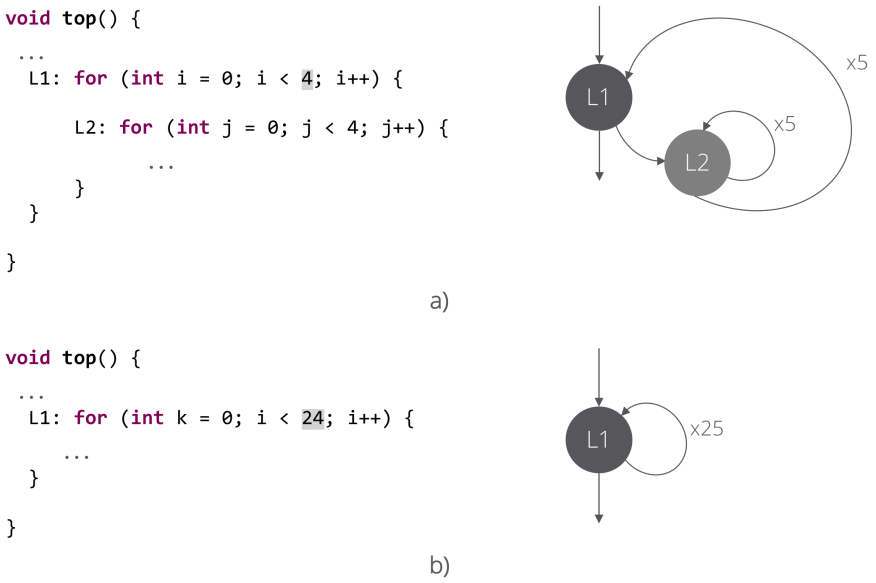


Figura 30. Loop flattening en lazos anidados. La aplicación de esta optimización reduce los ciclos de reloj necesarios para entrar y salir de lazos interiores.

3.3.5.2. Optimizaciones de Área

Todos los diseños de Vivado HLS buscan optimizar por defecto el número de recursos de hardware o área utilizada. Sin embargo, si se requiere reducir aún más el área de un diseño, existen otras directivas de optimización para este propósito. Por ejemplo, en la Sección 3.2.3.2 se habló sobre los tipos de datos eficientes para hardware. Estos tienen un gran impacto sobre

el número de recursos utilizados para una implementación. Las directivas de optimización de recursos como `ARRAY_MAP` y `ARRAY_RESHAPE`, buscan reducir la cantidad de bloques RAM usados mediante la organización de varios arreglos pequeños en arreglos más grandes (Xilinx - UG902, 2017). Otras optimizaciones de área importantes son `ALLOCATION` e `INLINE`, las cuales reducen el número de recursos mediante la compartición de los mismos por operaciones, funciones, lazos o regiones de código (Xilinx - UG902, 2017).

Capítulo 4

Diseño de Sistemas SoC con IP-Cores Personalizados Usando Vivado

En el presente capítulo se detalla la creación de sistemas SoC con núcleos IP en hardware, para acelerar algoritmos, desarrollados mediante Vivado HLS. El capítulo tiene como objetivo explicar los cinco algoritmos implementados en hardware, el proceso de desarrollo mediante HLS y la integración de los aceleradores a SoCs. La creación de software para el *hard-processor* de los SoCs en Xilinx Software Development Kit (SDK) no es uno de los objetivos principales de la tesis, pero al ser una parte constituyente del SoC se explican sus conceptos fundamentales.

Selección de benchmarks

La selección de los *bechmarks* se ha realizado tomando en cuenta que aplicaciones del futuro requerirían de sistemas embebidos y de control que serán cada vez más complejos, demandando procesamiento de datos provenientes de múltiples sensores, análisis de grandes cantidades de datos en tiempo real, técnicas de control inteligente, manejo de comunicaciones y además un consumo energético reducido. Por ejemplo, sistemas de navegación autónoma con necesidad de algoritmos avanzados, como *deep learning*, de alta demanda computacional

serán imposibles de ejecutarse en arquitecturas tradicionales. Es por ello que, en esta tesis se seleccionarán algoritmos relacionados a estas aplicaciones para explorar su desempeño en arquitecturas heterogéneas basadas en FPGA.

El primer benchmark seleccionado, una multiplicación de matrices, es un algoritmo sencillo, que sin embargo es ampliamente usado en infinidad de aplicaciones que requieren realizar cálculos numéricos utilizando algebra lineal. Este ejemplo se enfoca en cubrir los conceptos principales acerca del proceso de desarrollo mediante Vivado HLS. El segundo *benchmark*, FFT es un ejemplo típico para la evaluación de sistemas embebidos, y tiene aplicaciones en muchos campos de la ingeniería, sobre todo en aplicaciones relacionadas a comunicaciones y sistemas militares como radares. El tercer algoritmo, encriptación/des-encriptación AES, se ha seleccionado en base a su importancia para aplicaciones de seguridad y defensa, siendo uno de los estándares con mayor acogida para la protección de información. El cuarto algoritmo, retropropagación, es el método de entrenamiento supervisado más popular para redes neuronales, las cuales cubren un amplio rango de aplicaciones, incluyendo robótica, sistemas de control modernos, etc. El último algoritmo, una red neuronal, se ha desarrollado en base a una aplicación real en el ámbito de *machine learning*, el cual es uno de los campos de investigación actuales de sistemas basados en FPGA. Un resumen de los algoritmos y las suites seleccionadas se presenta en la Tabla 8.

Tabla 8
Benchmarks seleccionados para el desarrollo de aceleradores en hardware

Benchmark	Fuente	Descripción del algoritmo
Matrixmul	Xilinx UG871	Multiplicación de matrices
FFT	MachSuite	Transforma rápida de Fourier de 1024 puntos
AES	CHStone	Encriptación y des-encriptación AES
Backprop	MachSuite	Entrenamiento de redes neuronales por retropropagación
ANN	Desarrollo propio	Red neuronal para el reconocimiento de número manuscritos en imágenes de 20x20 pixeles

4.1. Multiplicador de Matrices

El primer acelerador en hardware es un multiplicador de matrices, el cual es un problema típico usado por muchos *benchmark suites*. Se lo ha escogido como primer ejemplo gracias a que su estructura matemática simple facilita la comprensión de la lógica del algoritmo, y permite enfocarse en la explicación de conceptos acerca de la implementación de IP-cores mediante Vivado HLS. En esta sección se explican conceptos importantes como la creación de un *test bench* y el manejo de interfaces AXI. Además, se explica de forma breve la integración del acelerador a un SoC usando Vivado IP Integrator, y los conceptos de la creación de software embebido para el SoC usando Xilinx SDK.

4.1.1. Algoritmo

Dadas las matrices $A = (a_{ik})$ de orden $m \times n$ y $B = (b_{kj})$ de orden $p \times n$, se define el producto de A y B, como una matriz $C = (c_{ij})$ tal que $i = 1, ...m$ y $j = 1, ...n$. Los elementos c_{ij} se definen como

$$c_{ij} = \sum_{k=1}^{p} a_{ik}.b_{kj} \tag{4.1}$$

Un ejemplo de la obtención del elemento $c_{1,1}$ de la matriz C para una multiplicación de matrices de orden 2×2 se muestra en la Figura 31. Para este caso, la matriz C se define como:

$$C = A.B = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

$$C = \begin{pmatrix} a_{11}.b_{11} + a_{12}.b_{21} & a_{11}.b_{12} + a_{12}.b_{22} \\ a_{21}.b_{11} + a_{22}.b_{21} & a_{21}.b_{12} + a_{22}.b_{22} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

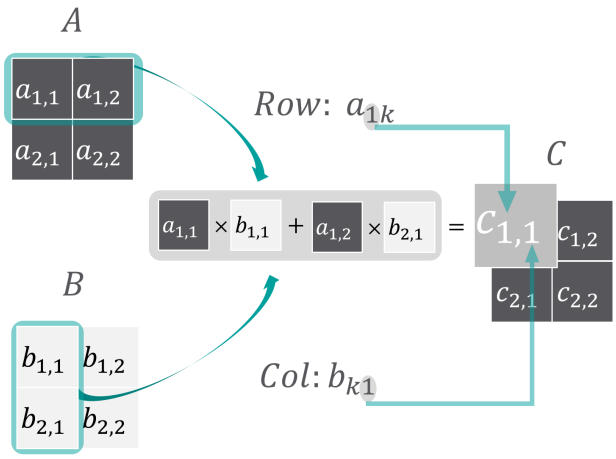


Figura 31. Proceso de obtención del elemento $c_{1,1}$ de la matriz para la multiplicación de las matrices A y B de orden 2×2.

El Código 8, muestra la función "matrixmul" desarrollada en C++, la cual se obtuvo de "*Vivado Design Suite Tutorial - High-Level Synthesis*" (Xilinx - UG871, 2017). Esta función tiene tres argumentos: las dos matrices a ser multiplicadas: `a[MAT_A_ROWS][MAT_A_COLS]` y `b[MAT_B_ROWS][MAT_B_COLS]`, y la matriz `res[MAT_A_ROWS][MAT_B_COLS]` que almacena el resultado. El algoritmo para multiplicar matrices consta de tres lazos `for` anidados. El primer lazo (`Row`) itera sobre las filas de la matriz a, el segundo lazo (`Col`) itera sobre las columnas de la matriz b, y finalmente el lazo interior (`Inner`) realiza el producto de las filas de la matriz a por las columnas de la matriz b como se define en la Ecuación 4.1. Para ello, se implementa una operación conocida como multiplicación-acumulación, la cual calcula el producto de dos números y lo suma a un acumulador. En sistemas computacionales, la unidad que realiza esta operación se denomina multiplier-accumulator (MAC), y es un componente importante en todos los procesadores digitales de señales (Malik & Dhall, 2012).

Código 8
Algoritmo para multiplicación de matrices en C++

```
#include "matrixmul.h"
void matrixmul(mat_a_t a[MAT_A_ROWS][MAT_A_COLS],
               mat_b_t b[MAT_B_ROWS][MAT_B_COLS],
               result_t res[MAT_A_ROWS][MAT_B_COLS]) {
        Row: for (int i = 0; i < MAT_A_ROWS; i++) {
               Col: for (int j = 0; j < MAT_B_COLS; j++) {
```

```
7                               res[i][j] = 0;
8                               Inner: for (int k = 0; k < MAT_B_ROWS; k++) {
9                                       res[i][j] += a[i][k] * b[k][j];
10                              }
11                      }
12              }
13      }
```

4.1.2. Implementación en Vivado HLS

El desarrollo de este ejemplo se enfoca en explicar aspectos prácticos en el proceso de implementación de un IP-core en Vivado HLS. Los conceptos fundamentales acerca del flujo de diseño de Vivado HLS se pueden consultar en el Capítulo 3, sin embargo, para facilidad del lector se resume a continuación en cinco etapas y se indica las secciones en las cuales se desarrollan estos temas.

1. **Creación de una especificación C:** Vivado HLS permite la creación de una nueva especificación o la importación de una existente. Es importante señalar que una función o jerarquía de sub-funciones desarrolladas en C/C++ puede ser directamente trasladado a Vivado HLS siempre que cumpla con las siguientes condiciones: 1) no contener operaciones típicas de OS no sintetizables a hardware (Sección 3.2.2), 2) definir las entradas/salidas del IP como argumentos en la función *top-level*.

2. **Simulación C:** Compilación de la especificación C y verificación de resultados a nivel de software mediante un *test bench* C apropiado (Sección 3.2.1.2).

3. **Especificación de directivas de síntesis:** Especificación de directivas que permiten ejercer control sobre el proceso de síntesis. De esta forma el diseñador puede especificar las interfaces para el IP (Sección 3.2.4), o realizar optimizaciones de recursos, latencia, desempeño (Sección 3.3.4).

4. **Síntesis:** Vivado HLS infiere la lógica de la especificación C y la sintetiza a hardware (Sección 3.1.2).

5. **Co-simulación C/RTL:** El *test bench* C se reutiliza para la creación de una simulación a nivel RTL, la cual verifica que los resultados de la implementación coincidan con los resultados esperados (Sección 3.3.1).

4.1.2.1. Creación de una Especificación C

Para este primer ejemplo, la especificación C es el algoritmo en C++ mostrado en el Código 8 con su correspondiente archivo de cabecera *matrixmul.h*[1]. En este archivo de cabecera se define los tipos de datos (`mat_a_t`, `mat_b_t`, `result_t`) y la dimensión de las matrices (`MAT_A_ROWS`, `MAT_A_COLS`, `MAT_B_ROWS`, `MAT_B_COLS`). Ambos archivos deben ser incluidos el momento de la creación de un nuevo proyecto en Vivado HLS. Además, se debe seleccionar la función *top-level* de la especificación, *matrixmul*, como se muestra en la Figura 32. Para mayor información acerca de la creación de proyectos puede referirse al documento "Vivado Design Suite Tutorial - High-Level Synthesis" (Xilinx - UG871, 2017).

4.1.2.2. Simulación C

Una vez se tiene lista la especificación C en Vivado HLS, el siguiente paso es ejecutar la simulación C. Sin embargo, esta no puede ser llevada a cabo sin la previa creación de un *test bench*. El *test bench* es necesario para llamar a la función *top-level* de la especificación para que esta sea ejecutada en software. Por la importancia del *test bench* en esta etapa y en la co-simulación C/RTL, se presenta a continuación una descripción que resume los puntos principales de su creación.

Test Bench C

El test bench C debe cumplir con ciertas directrices: i) Tener una función principal de

[1]Se considera una buena práctica en programación C/C++ crear archivos de cabecera (.h) donde se declaren los tipos de variables e identificadores usados por los archivos fuente. En Vivado HLS es necesaria una cabecera en la cual se defina la función top-level del DUT para que esta pueda ser llamada desde el testbench C.

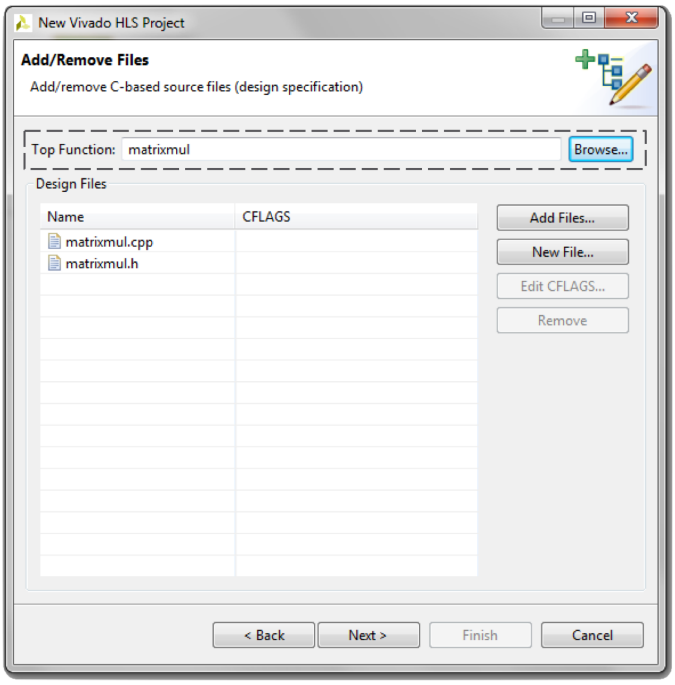


Figura 32. Importación de los archivos que forman parte de la especificación C. La función *matrixmul* ha sido seleccionado como *top-level*.

tipo entero, la cual llame a la función *top-level* del DUT. ii) Pasar datos al DUT en forma de valores, arreglos o matrices de prueba, según corresponda, para que sean procesados y se genere un resultado. iii) Los resultados entregados por el DUT deben ser comparados con resultados esperados, los cuales son obtenidos de ejecuciones de software previamente verificadas[2]. Para efectos de demostración en este ejemplo, los resultados esperados son generados dentro del mismo *test bench* C. iii) Si los resultados coinciden, el *test bench* debe retornar un valor de cero, caso contrario retornará un valor diferente de cero. En la Figura 33 se muestra la obtención de los *golden results* y su posterior comparación con los resultados del DUT durante la Simulación C en Vivado HLS.

[2]Se recomienda que los resultados esperados sean obtenidos de ejecuciones de software en compiladores C/C++. Esto asegura que la precisión de datos de punto flotante no difiera con respecto a Vivado HLS. Sin embargo, los resultados pueden ser obtenidos de otras fuentes como Matlab, tomando en cuenta que pueden presentarse diferencias significativas dependiendo del tipo de operaciones realizadas.

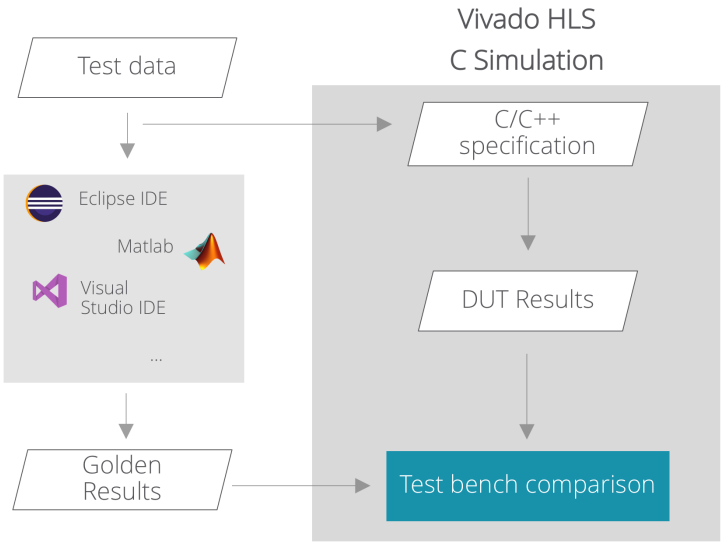


Figura 33. Comparación de resultados esperados y resultados del DUT en la simulación C de Vivado HLS. Los resultados esperados o Golden results son obtenidos de implementaciones de software.

En el Código 9 se puede observar el test bench C implementado para la función *top-level matrixmul*, el cual cumple con las directrices antes mencionadas. En primer lugar, se define una función principal y matrices de prueba. Se generan los resultados esperados y se llama a la función *matrixmul*. Los resultados esperados y resultados del DUT son finalmente comparados. El test bench realiza la comparación elemento a elemento, y acumula el número de errores encontrados en la variable err_cnt. Finalmente, dicha variable es retornada por la función main.

Código 9
Test Bench C para la multiplicación de matrices del Código 8

```
#include <iostream>
#include <stdio.h>
#include "matrixmul.h"
using namespace std;

int main() {
        mat_a_t in_mat_a[2][2] = { { 0, 0 }, { 0, 1 } };
        mat_b_t in_mat_b[2][2] = { { 1, 1 }, { 0, 1 } };
        result_t hw_result[2][2], sw_result[2][2];
        int err_cnt = 0;
        for (int i = 0; i < MAT_A_ROWS; i++) {
                for (int j = 0; j < MAT_B_COLS; j++) {
                        sw_result[i][j] = (result_t)0;
```

```
14                          for (int k = 0; k < MAT_B_ROWS; k++) {
15                                  sw_result[i][j] += (result_t)in_mat_a[i
                                       ][k] * (result_t)in_mat_b[k][j];
16                          }
17                  }
18          }
19          matrixmul(in_mat_a, in_mat_b, hw_result);
20          for (int i = 0; i < MAT_A_ROWS; i++) {
21                  for (int j = 0; j < MAT_B_COLS; j++) {
22                          if (hw_result[i][j] != sw_result[i][j]) {
23                                  err_cnt++;
24                          }
25                  }
26          }
27          if (err_cnt)
28                  cout << "ERROR: " << err_cnt << " mismatches detected!"
                        << endl;
29          else
30                  cout << "Test passes." << endl;
31          return err_cnt;
32  }
```

4.1.2.3. Especificación de Directivas de Síntesis

Como se explicó anteriormente, el uso de directivas de síntesis puede tener diferentes enfoques. Uno de estos enfoques es optimizar el desempeño de la implementación, lo cual será abordado en la sección 4.1.5. Mientras tanto, en esta sección se tratará únicamente acerca de las directivas de síntesis para el manejo de interfaces. El objetivo de la tesis es integrar los IP-cores generados en Vivado HLS a un SoC con un procesador *hard-core*. Por ello, a continuación, se detalla el manejo de las interfaces AXI destinadas a este propósito.

AXI4 en Zynq-7000

Para que los IP-cores implementados usando Vivado HLS puedan comunicarse con otros subsistemas, estos deben contar con interfaces que permitan su comunicación. Para la integración del IP-core, implementado en el PL, con el PS del Zynq-7000 es necesario utilizar interfaces AXI. La Figura 34 muestra la arquitectura de Zynq, compuesta por el PL y el PS. En esta figura se observa que el PS incluye un Advanced Microcontroller Bus Architecture

(AMBA) conectado al procesador ARM Cortex. A su vez el bus AMBA cuenta con interfaces AXI4 conectadas al PL, permitiendo comunicar IP-cores implementados en hardware con el software embebido en el procesador.

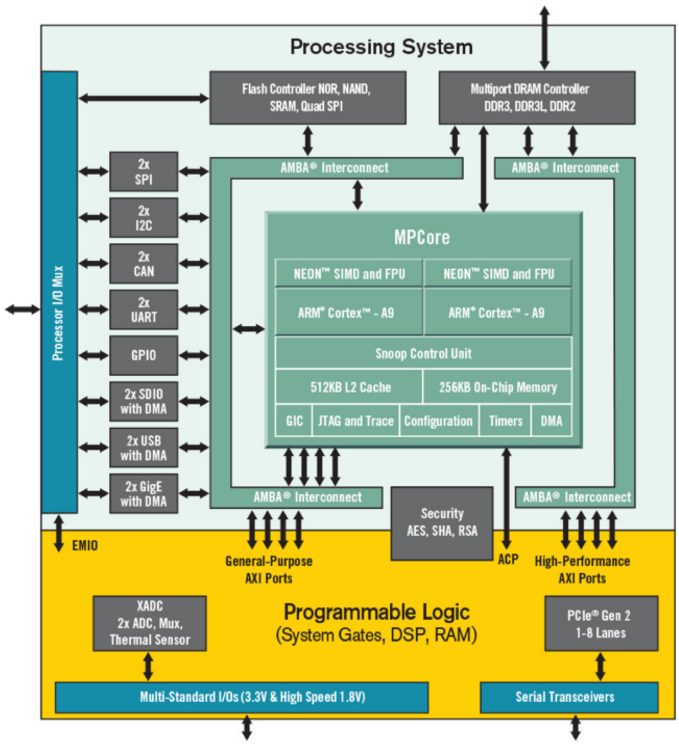


Figura 34. Arquitectura de un Zynq-7000 SoC. En esta figura se observa el PS y el PL comunicados mediante interfaces AXI.
Fuente: (Xilinx Inc., 2018)

Tipo de interfaces AXI4

El estándar AXI define tres tipos de interfaces: **(1) AXI4-Stream:** para transmisión de datos *streaming* a alta velocidad, **(2) AXI4-Lite:** para mapeo de memoria sin transmisiones *burst*[3] de datos, y **(3) AXI4 master:** igual que AXI4-Lite pero con soporte para transmisiones *burst* de datos. Durante el desarrollo de la tesis se utiliza específicamente la interfaz AXI4-Lite.

[3]Transmisión de datos de alta velocidad usada para datos secuenciales.

Funcionamiento de AXI4

AXI4 define un conjunto de transacciones mediante las cuales se comunican dos IP-cores, un maestro y un esclavo. Estas transacciones se realizan mediante canales de lectura y escritura independientes, los cuales permiten transacciones bidireccionales simultaneas de 32, 64, 128 o 256 bits (Xilinx - UG761, 2011). Estos canales son representados en la Figura 35. A continuación, se describe la función de los canales para transacciones de lectura y escritura.

El proceso de lectura se lleva a cabo mediante: **i) Canal de dirección y control de lectura:** el maestro escribe la dirección del esclavo que se requiere leer y genera las señales de control necesarias para ejecutar la transacción. **ii) Canal de datos de lectura:** transmite los datos desde el esclavo al maestro.

El proceso de escritura se lleva a cabo mediante: **i) Canal de dirección y control de escritura:** el maestro coloca la dirección del esclavo en la cual se desea escribir y genera las señales de control para ejecutar la transacción. **ii) Canal de datos de escritura:** transmite los datos desde el maestro al esclavo. **iii) Canal de respuesta de escritura:** el esclavo confirma la escritura de los datos al maestro.

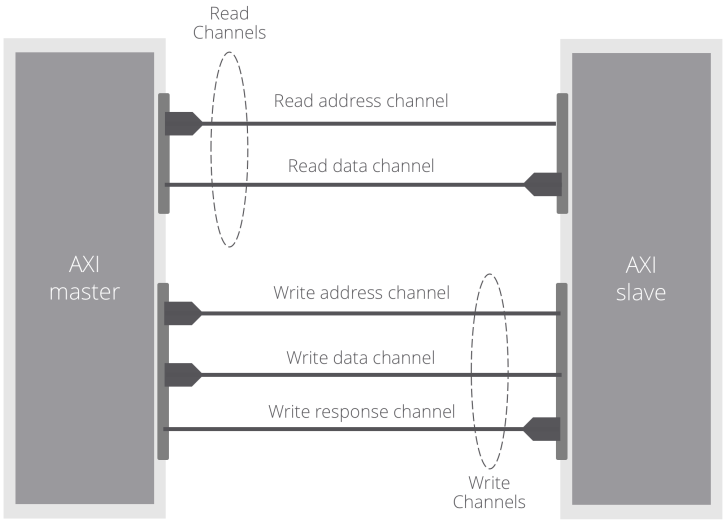


Figura 35. Canales de escritura y lectura de la interfaz AXI4. Estos canales permiten realizar operaciones de lectura y escritura simultaneas.

Los maestros y esclavos pueden ser conectados mediante un bloque de interconexión. Esto

facilita la comunicación entre IP-cores, permitiendo encaminar las transacciones entre varios maestros y esclavos. Xilinx ha creado el AXI Interconnect IP, el cual en un núcleo de licencia libre incluido en el catálogo de IP-cores de Vivado Design Suite. Este IP permite conectar un máximo de dieciséis maestros a dieciséis esclavos. En la Figura 36 se representa una topología de conexión denominada *Shared-Address Multiple-Data* (SAMD), en el cual se observa arbitradores de transacciones de escritura y lectura que definen el acceso de un maestro a uno de los esclavos seleccionado por un enrutador. Los arbitradores de transacciones definen el acceso a un único maestro a la vez por cada ciclo de arbitraje. Esto se realiza mediante la definición de una prioridad estática o un método de arbitraje denominado *Round-robin*. Para mayor información acerca de AXI puede consultar el documento "*AXI Reference Guide*" (Xilinx - UG761, 2011).

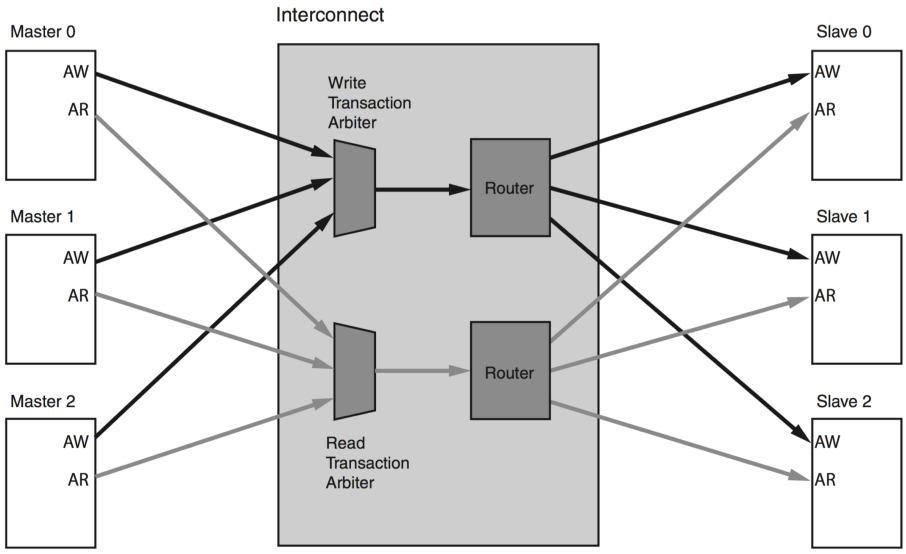


Figura 36. AXI Interconnect con múltiples maestros y esclavos.
Fuente (Xilinx - UG761, 2011)

Manejo de interfaces AXI

Vivado HLS define dos tipos de interfaces: i) interfaces de datos para los argumentos de la función *top-level* (a nivel de puertos) y ii) interfaces que coordinen la comunicación

mediante señales de control (a nivel de bloque) (Sección 3.2.4). Estas interfaces son especificadas mediante sentencias especiales `#pragma HLS INTERFACE`[4]. El Código 10. muestra la especificación de interfaces para el presente ejemplo. En este código, se indica el uso de AXI4-Lite tanto para cada puerto de datos (i.e., `port=a`, `b`, `res`), como a nivel de bloque, el cual Vivado asocia a la variable de retorno de la función *top-level* (`port=return`). Cada puerto de datos e interfaz de bloque puede ser creada como una interfaz AXI4 independiente. Sin embargo, para crear una única interfaz AXI4 consolidada, la cual agrupe las interfaces a nivel de puertos con la interfaz a nivel de bloque, se especifica la creación de un bundle (`bundle=MATRIXMUL_PERIPH_BUS`) (Crockett, Elliot, Enderwitz, & Stewart, 2015). Vivado se encarga de la creación automática del hardware y drivers necesarios para su implementación (Sección 3.2.4). Finalmente, Vivado creará señales de *clock*, *reset* e *interrupt* para todos los diseños. El reporte de interfaces se puede consultar en Figura 37.

Código 10
Directivas para la especificación de interfaces AXI4-Lite a nivel de puertos y a nivel de bloque para la función matrixmul

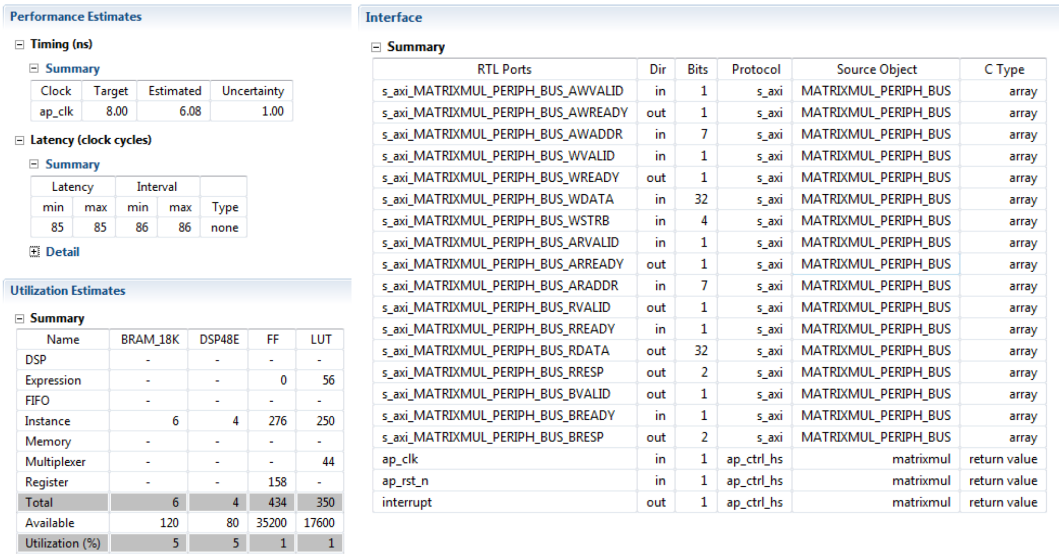
```
1  void matrixmul(mat_a_t a[MAT_A_ROWS][MAT_A_COLS],
2                 mat_b_t b[MAT_B_ROWS][MAT_B_COLS],
3                 result_t res[MAT_A_ROWS][MAT_B_COLS]) {
4     #pragma HLS INTERFACE s_axilite port=return bundle=
          MATRIXMUL_PERIPH_BUS
5     #pragma HLS INTERFACE s_axilite port=res bundle=MATRIXMUL_PERIPH_BUS
6     #pragma HLS INTERFACE s_axilite port=b bundle=MATRIXMUL_PERIPH_BUS
7     #pragma HLS INTERFACE s_axilite port=a bundle=MATRIXMUL_PERIPH_BUS
8  }
```

4.1.2.4. Síntesis HLS

Vivado llevará a cabo la síntesis de alto-nivel tomando en cuenta la especificación de directivas y restricciones. Como se mencionó previamente, las directivas son usadas para especificar interfaces, optimizaciones de desempeño, latencia y recursos. Por otro lado, las restricciones se usan para definir el periodo de reloj, incertidumbre de reloj y el dispositivo

[4]El uso de sentencias especiales #pragma no afecta a la lógica de la especificación C. Estas sentencias influyen únicamente en la forma en la que la síntesis implementa el hardware.

FPGA de destino. Vivado realiza la síntesis mediante las etapas de *Allocation*, *Scheduling* y *Binding*. Además, Vivado tomará en cuenta sus directrices propias, las cuales se mencionaron en la Sección 3.1.2. Una vez que la síntesis finalice, se generará un reporte detallado donde se presenta un estimado de performance, uso de recursos e interfaces sintetizadas (Sección 3.3.2). Para el presente ejemplo, *matrixmul*, el reporte de síntesis para un dispositivo Zynq Z-7010 se muestra en la Figura 37.



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	8.00	6.08	1.00

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
85	85	86	86	none

Detail

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	56
FIFO	-	-	-	-
Instance	6	4	276	250
Memory	-	-	-	-
Multiplexer	-	-	-	44
Register	-	-	158	-
Total	6	4	434	350
Available	120	80	35200	17600
Utilization (%)	5	5	1	1

Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
s_axi_MATRIXMUL_PERIPH_BUS_AWVALID	in	1	s_axi	MATRIXMUL_PERIPH_BUS	array
s_axi_MATRIXMUL_PERIPH_BUS_AWREADY	out	1	s_axi	MATRIXMUL_PERIPH_BUS	array
s_axi_MATRIXMUL_PERIPH_BUS_AWADDR	in	7	s_axi	MATRIXMUL_PERIPH_BUS	array
s_axi_MATRIXMUL_PERIPH_BUS_WVALID	in	1	s_axi	MATRIXMUL_PERIPH_BUS	array
s_axi_MATRIXMUL_PERIPH_BUS_WREADY	out	1	s_axi	MATRIXMUL_PERIPH_BUS	array
s_axi_MATRIXMUL_PERIPH_BUS_WDATA	in	32	s_axi	MATRIXMUL_PERIPH_BUS	array
s_axi_MATRIXMUL_PERIPH_BUS_WSTRB	in	4	s_axi	MATRIXMUL_PERIPH_BUS	array
s_axi_MATRIXMUL_PERIPH_BUS_ARVALID	in	1	s_axi	MATRIXMUL_PERIPH_BUS	array
s_axi_MATRIXMUL_PERIPH_BUS_ARREADY	out	1	s_axi	MATRIXMUL_PERIPH_BUS	array
s_axi_MATRIXMUL_PERIPH_BUS_ARADDR	in	7	s_axi	MATRIXMUL_PERIPH_BUS	array
s_axi_MATRIXMUL_PERIPH_BUS_RVALID	out	1	s_axi	MATRIXMUL_PERIPH_BUS	array
s_axi_MATRIXMUL_PERIPH_BUS_RREADY	in	1	s_axi	MATRIXMUL_PERIPH_BUS	array
s_axi_MATRIXMUL_PERIPH_BUS_RDATA	out	32	s_axi	MATRIXMUL_PERIPH_BUS	array
s_axi_MATRIXMUL_PERIPH_BUS_RRESP	out	2	s_axi	MATRIXMUL_PERIPH_BUS	array
s_axi_MATRIXMUL_PERIPH_BUS_BVALID	out	1	s_axi	MATRIXMUL_PERIPH_BUS	array
s_axi_MATRIXMUL_PERIPH_BUS_BREADY	in	1	s_axi	MATRIXMUL_PERIPH_BUS	array
s_axi_MATRIXMUL_PERIPH_BUS_BRESP	out	2	s_axi	MATRIXMUL_PERIPH_BUS	array
ap_clk	in	1	ap_ctrl_hs	matrixmul	return value
ap_rst_n	in	1	ap_ctrl_hs	matrixmul	return value
interrupt	out	1	ap_ctrl_hs	matrixmul	return value

Figura 37. Reporte de síntesis del IP *matrixmul* para un dispositivo Zynq Z-7010. En el reporte se muestra las métricas de desempeño estimadas, la utilización de los recursos y el reporte de interfaces.

4.1.2.5. Co-Simulación C/RTL

El último paso en la implantación de IP-cores usando Vivado HLS es la co-simulación C/RTL. Para poder ejecutar esta simulación es necesario disponer de i) el *test bench* C y ii) la implementación sintetizada del DUT. El proceso llevado a cabo en la co-simulación se explica en la Sección 3.3.1. Vivado se encargará de crear una simulación a nivel RTL, la cual se realiza por defecto en el simulador RTL de Vivado utilizando la descripción generada en Verilog[5].

[5]Es posible seleccionar otros simuladores (ModelSim simulator, VCS simulator, NC-Sim simulator, Riviera simulator), y la descripción VHDL en el panel de preferencias de co-simulación.

Para este ejemplo, el reporte de co-simulación se muestra en la Figura 38, en el cual se observa que la simulación ha sido exitosa (*Pass*), es decir, el *test bench* ha comprobado que los resultados obtenidos en software y hardware coinciden. También se muestran valores de latencia, los cuales son calculados en base a la duración de la simulación que involucra procesos adicionales a la ejecución del IP, y por lo tanto no representan a los estimados reales de latencia (Figura 37). Además, los valores en la estimación de intervalo son cero debido a que las entradas están siendo aplicadas una sola vez al IP durante el *test bench* y por lo cual no hay una medición de intervalo durante esta co-simulación.

Cosimulation Report for 'matrixmul'

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	134	134	134	0	0	0

Figura 38. Reporte de co-simulación exitoso para el IP matrixmul. La co-simulación se ha ejecutado en el simulador RTL de Vivado para la descripción Verilog (*Pass*). La co-simulación para la descripción VHDL no se ha realizado, por lo cual el reporte no está disponible (NA).

4.1.3. Integración del IP-core en el SoC

La integración del IP al SoC se la realiza en Vivado IP Integrator, la cual es parte del conjunto de herramientas de Vivado Design Suite. El proceso básico de creación de un SoC se describe de forma breve a continuación. Para un detalle completo acerca de la integración de IP-cores a SoCs puede referirse al documento "*Vivado Design Suite User Guide - Designing IP Subsystems Using IP Integrator*" (Xilinx - UG994, 2017).

1. El primer paso es crear un nuevo proyecto en Vivado Design Suite e importar el IP generado en HLS al catálogo de IP-cores. En la Figura 39 se muestra la importación del IP *matrixmul* para este ejemplo. Los IP-cores exportados desde Vivado HLS están

listos para ser importados y cuentan con soporte para automatizar su conexión al PS de SoCs mediante interfaces AXI4.

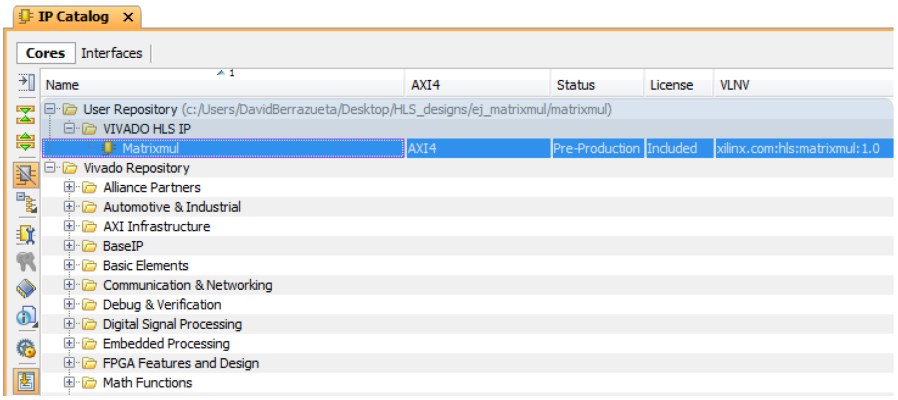


Figura 39. IP Matrixmul importado al catálogo de IP-cores de Vivado IP Integrator.

2. El siguiente paso es crear un diseño de bloques en Vivado IP Integrator. En el diseño se debe agregar el PS del Zynq SoC, el cual está basado en un procesador ARM Cortex[6], cache, controladores y otros componentes que permiten la ejecución de software y su interacción con el PL y periféricos. La asistencia para automatizar la configuración del PS establece los parámetros del mismo en base a la tarjeta de desarrollo seleccionada en la creación del proyecto. Sin embargo, ciertos parámetros como la frecuencia de reloj generada para el PL, puerto de interrupciones, y configuraciones del Multiplexed I/O[7] (MIO) deben realizarse manualmente de acuerdo al diseño. El PS y su respectiva ventana Re-customize IP se muestra en la Figura 40.

[6]El modelo del procesador ARM depende del dispositivo Zynq utilizado. Los Zynq-7000 integran procesadores ARM Cortex-A9 *dual-core*, mientras que los Zynq UltraScale+ utilizan procesadores ARM Cortex-A53 y Cortex-R5 *dual-core*.

[7]Subsistema del PS de Zynq que multiplexa las entradas y salidas de varios controladores de periféricos.

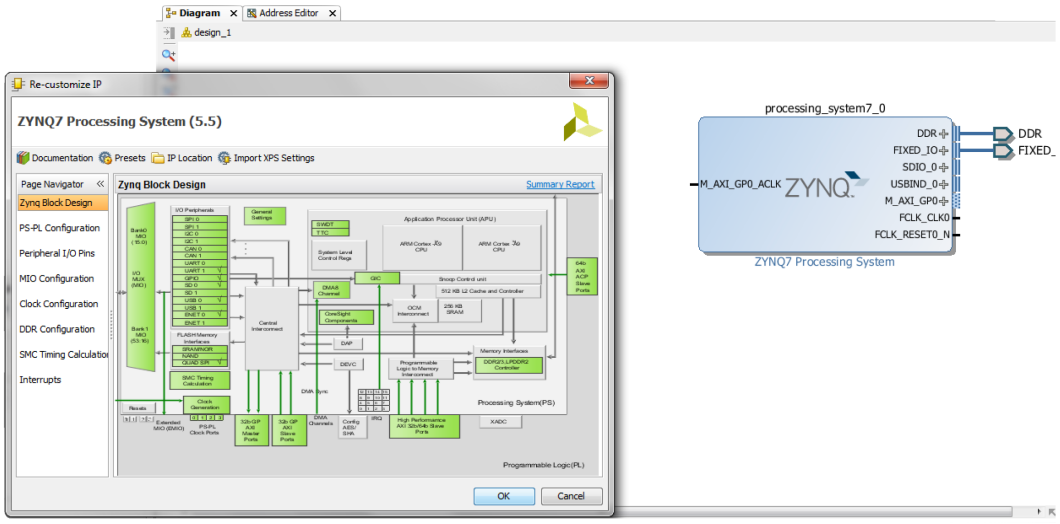


Figura 40. Zynq PS y su ventana *Re-customize IP* en el diseño de bloques de Vivado IP Integrator.

3. Posteriormente se debe añadir el IP importado previamente al diseño de bloques. Para el presente ejemplo se incluye el IP *matriximul*, el cuál es el acelerador para la multiplicación de matrices de orden 2×2 implementado en hardware. Vivado mostrará el asistente para la conexión automática mediante AXI4, como se muestra en la Figura 41.

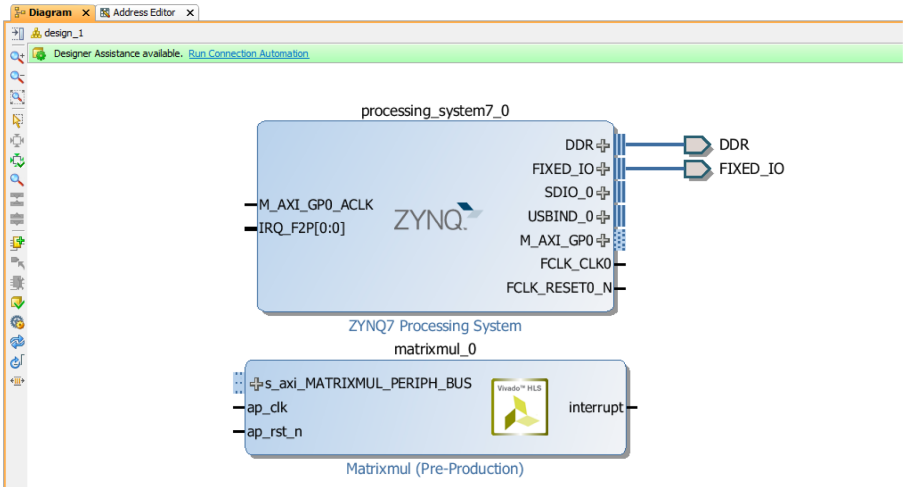


Figura 41. Zynq PS e IP matrixmul en el diseño de bloques de Vivado IP Integrator.

4. Ahora se debe ejecutar el asistente de conexión automática, el cual permite generar

el diagrama de bloques mostrado en la Figura 42. En esta figura se pueden distinguir dos nuevos bloques: **1) AXI Interconnect:** este IP permite conectar varios esclavos a varios maestros AXI. En este ejemplo, el bloque de interconexión AXI permite conectar el IP *matrixmul* con interfaces *slave* AXI al PS de Zynq con interfaz *master* AXI (Sección 4.1.2.3). Es importante notar que AXI Interconnect es visto como una interfaz *slave* para el PS y como un *master* para el IP *matrixmul*. **2) Processor System Reset:** este IP se encarga del reset del sistema, el cual actúa tanto sobre el AXI Interconnect como sobre los IP-cores conectados a él.

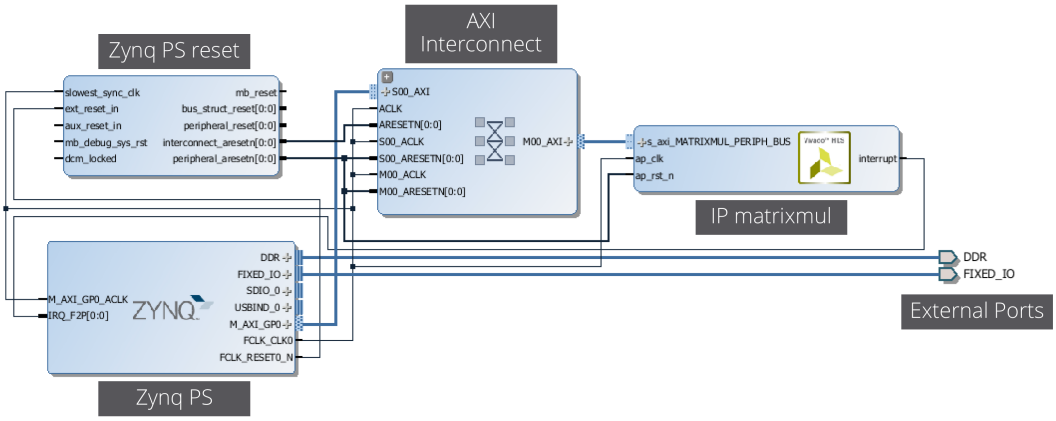


Figura 42. Diagrama de bloques de la integración de IP *matrixmul* en el SoC.

5. Posteriormente, en el panel *Sources* de Vivado Design Suite se selecciona el diseño de bloques creado (*Zynq_design.bd*) y se ejecuta la creación de un wrapper HDL, como se muestra en la Figura 43. Esto generara un *wrapper* para el diseño de bloques.

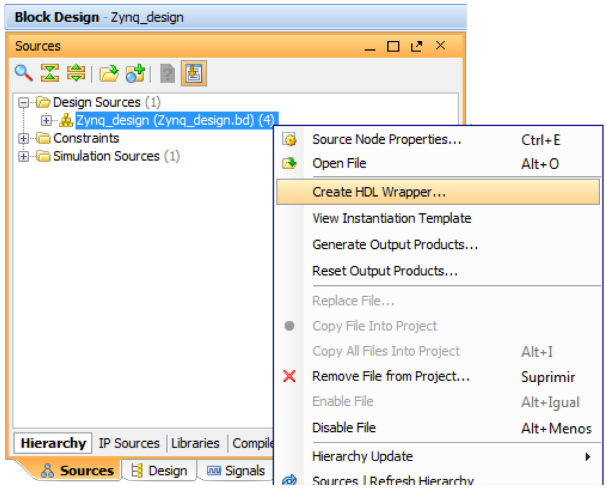


Figura 43. Creación de un wrapper HDL de un diseño de bloques en Vivado Design Suite.

6. Ahora, en el panel de navegación de Vivado Design Suite se selecciona la opción *Generate Bitstream*. El *bitstrem* es el archivo que contiene la información para la configuración del dispositivo FPGA (Xilinx - UG1291, 2018). En la ventana de dígalo resultante se confirma la ejecución de la síntesis e implementación de hardware, como se muestra en la Figura 44.

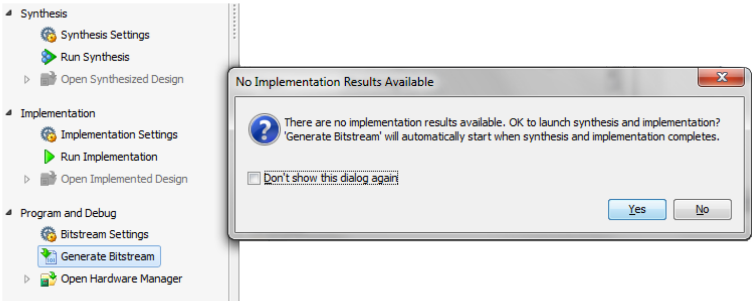


Figura 44. Generación de bitstream en Vivado Design Suite.

7. Finalmente, la especificación de hardware (*hardware platform*) creada debe ser exportada a Xilinx SDK. Para ello, en la barra de herramientas se selecciona *File>Export>Export Hardware*, y posteriormente *File>Launch SDK*. Esto permite ejecutar la aplicación Xilinx SDK, la cual importará la especificación de hardware automáticamente.

4.1.4. Software Embebido en Xilinx SDK

El diseño de un SoC comprende la creación de un subsistema de hardware y un subsistema de software que interactúan entre sí. El hardware personalizado desarrollado y exportado en Vivado representa al subsistema de hardware. Por otro lado, el subsistema de software comprende una aplicación ejecutada sobre un OS corriendo en un procesador. El desarrollo de software para SoCs de Xilinx se realiza en SDK.

Una forma de entender la estructura de un SoC es mediante un modelo que consta de capas de hardware y software. En la tesis se utiliza el modelo basado en la jerarquía de capas mostrada en la Figura 45. Cada una de las capas se describe a continuación:

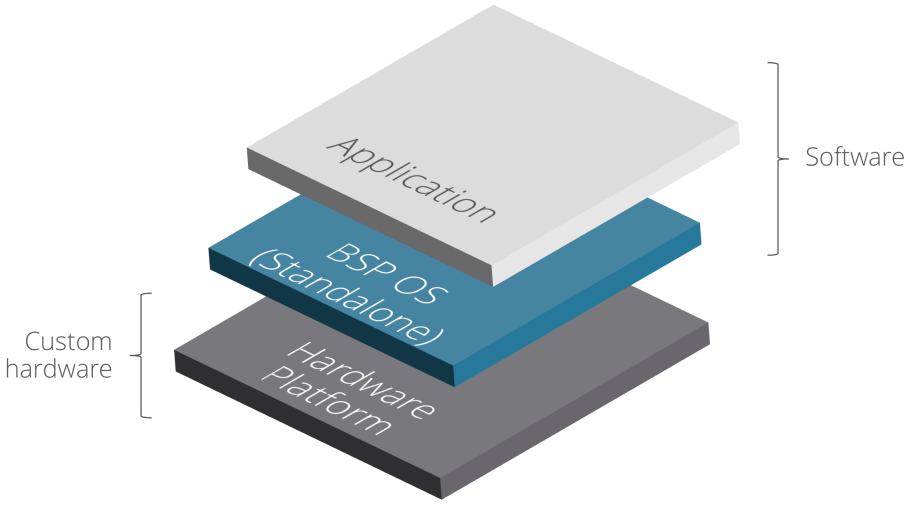


Figura 45. Modelo de capas de hardware y software para el diseño de SoC en Xilinx Zynq utilizado en la tesis. El hardware personalizado se representa en la capa inferior, el BPS OS (*standalone*) y la aplicación de software representan al subsistema de software.

1. **Hardware Platform:** incluye toda la información de hardware creado en Vivado IP Integrator (Figura 46). El principal archivo que representa a esta capa se denomina *system.hdf*. Este archivo contiene la especificación de la plataforma de hardware con información sobre el dispositivo de destino, el mapa de direcciones del procesador seleccionado, los bloques IP presentes en el diseño y las hojas de datos de todos los periféricos del sistema (Peter Thorwartl, 2017). Esta capa permite al software embebido

de las capas superiores tener acceso al hardware del SoC.

2. **Board Support Package (BSP):** es un sistema operativo básico denominado standalone. Este OS es una capa de software de bajo nivel, la cual se encarga de proveer el acceso a las funciones básicas del PS. El archivo principal del BSP se denomina *microprocessor software specification system.mss*. Este archivo contiene bibliotecas y controladores para el control del PS, permitiendo manejar caches, interrupciones, excepciones, *timers*, y otras configuraciones (Xilinx Wiki, 2018).

3. **Aplicación:** es la capa superior de software de alto-nivel. La aplicación es especificada en C/C++ y puede ser estar compuesta por una jerarquía de funciones y archivos. La aplicación se comunica con el hardware del SoC a través de las capas inferiores.

   Xilinx SDK permite importar la especificación de hardware desde Vivado Design Suite (Sección 4.1.3). En base a esta especificación, se crea una aplicación para uno de los sistemas operativos que pueda ejecutar el *hard-procesador* del SoC[8]. Para esta tesis, se selecciona un OS *standalone* a ser ejecutado en uno de los núcleos del procesador ARM Cortex-A9. SDK creará un proyecto conforme a la jerarquía de capas mostrada en la Figura 45. Esta jerarquía se representa en el árbol de proyecto de SDK, el cual se muestra en la Figura 46. En esta figura se observa tres directorios principales, los cuales se corresponden a cada una de las capas del modelo: i) *hardware platfrom*, ii) BSP OS y iii) *software application*.

Si se requiere mayor información acerca de la creación de aplicaciones de software para SoCs se puede consultar el documento "*Xilinx Software Development Kit (SDK)*" (Xilinx - UG1145, 2018).

[8]Los OSs que es capaz de ejecutar un Zynq PS son: Linux, FreeRTOS y standalone.

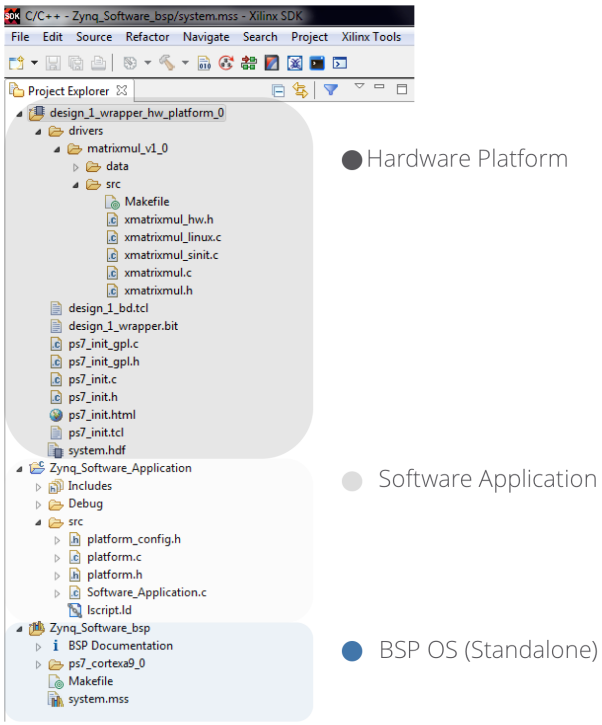


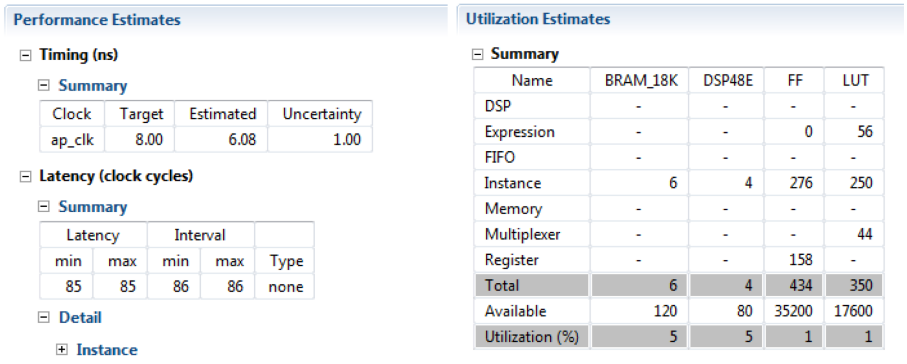
Figura 46. Árbol de proyecto en Xilinx SDK para un SoC basado en el modelo de capas de la Figura 45. En la presente figura se muestran los directorios correspondientes a la plataforma de hardware, aplicación de software y el sistema operativo *standalone* para el SoC con el acelerador *matrixmul*.

4.1.5. Optimizaciones

Una de las grandes ventajas de Vivado HLS es la posibilidad de influenciar la síntesis permitiendo al diseñador alcanzar las métricas de diseño requeridas. A continuación, se explica cómo analizar e implementar optimizaciones para mejorar el desempeño del acelerador, reduciendo al mínimo posible la latencia e intervalo de ejecución (Sección 3.1.2). Para este propósito, se han creado cuatro diferentes *soluciones*, las cuales implementan la misma especificación, pero logran diferentes resultados mediante la influencia de directivas de la síntesis.

4.1.5.1. Solución 1

La primera solución corresponde al DUT original (Código 8) sin la aplicación de ninguna directiva de optimización. Una vez sintetizada la especificación, el reporte de síntesis entrega la estimación de desempeño y recursos mostrada en la Figura 47. Es importante notar que la latencia e intervalo son demasiado grandes para lo que se esperaría de una multiplicación de matrices de orden 2×2. Esto se debe a que Vivado HLS busca por defecto priorizar el uso de recursos y no el desempeño. Cada una de las siguientes soluciones está enfocada a reducir la latencia e intervalo, lo cual afecta también al número de recursos utilizados.



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	8.00	6.08	1.00

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
85	85	86	86	none

Detail

Instance

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	56
FIFO	-	-	-	-
Instance	6	4	276	250
Memory	-	-	-	-
Multiplexer	-	-	-	44
Register	-	-	158	-
Total	6	4	434	350
Available	120	80	35200	17600
Utilization (%)	5	5	1	1

Figura 47. Estimación de desempeño y recursos de la Solución 1 del IP *matrixmul*. Esta solución no incluye optimizaciones de desempeño. La latencia de ejecución estimada es de 85 ciclos de reloj a una frecuencia de 125MHz para un Zynq Z-7010.

Para mejorar las métricas de desempeño, se debe analizar la implementación usando la vista de análisis de Vivado HLS, la cual se presenta en la Figura 48. Esta vista representa los lazos en color amarillo, mientras que las operaciones en su interior se representan en color morado. Esta vista muestra únicamente una iteración por cada uno de los lazos cuando estos se encuentran implementados en la forma denominada *rolled* (Sección 3.1.2). Por ejemplo, la implementación de la Solución 1 tiene una latencia de 85 ciclos, sin embargo, la perspectiva de análisis muestra apenas 12 ciclos.

A continuación, se explica cómo relacionar la vista de análisis de la Figura 48 con los ciclos de reloj presentados en el reporte de síntesis de la Figura 47. El lazo interior (`Col.1`)

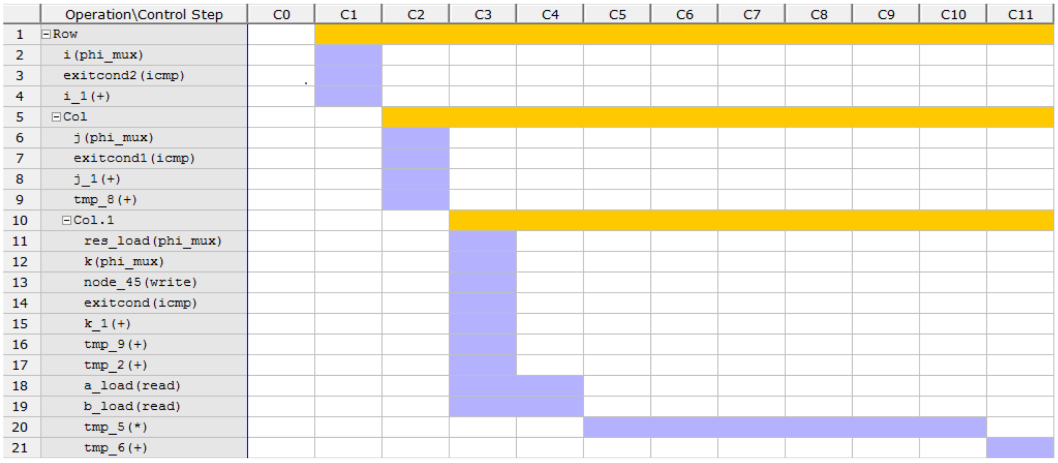


Figura 48. Perspectiva de análisis de la Solución 1 para el IP *matrixmul*. Esta solución no incluye optimizaciones de desempeño. En la figura se observa la ejecución secuencial de las operaciones.

ocupa nueve ciclos de reloj para completar cada iteración, en dos iteraciones ocupara 18 ciclos de reloj. El lazo `Col` contiene al lazo `Col.1`. Entrar y salir de `Col.1` toma dos ciclos de reloj, por lo cual cada iteración de `Col` ocupa en total 20 ciclos, 18 que toma `Col.1` más 2 ciclos en entrar y salir. Como `Col` es iterado dos veces, este lazo ocupa un total de 40 ciclos. El lazo superior `ROW` contiene a su vez al lazo `Col`, por lo que cada iteración tomara 42 ciclos. Como `ROW` se itera dos veces, se tiene 84 ciclos de reloj en total para esta multiplicación de matrices. Si a estos 84 ciclos se le suma un ciclo inicial (C0) que toma entrar al primer lazo `ROW` se tienen 85 ciclos de latencia.

4.1.5.2. Solución 2

Del análisis de la Solución 1 se estableció que los lazos se han implementado en la forma denominada *rolled*. Es decir, se crea el hardware necesario para una sola iteración por lazo, lo cual conlleva que cada iteración comparta los mismos recursos, debiendo esperar a que la anterior finalice para empezar con la siguiente. Para mejorar el desempeño en esta solución se utiliza la directiva `PIPELINING`, con la cual se consigue: i) aplicar *unrolled* a los lazos, creando el hardware necesario para cada iteración, y ii) paralelizar la ejecución de operaciones.

Vivado HLS permite aplicar *pipelining* tanto a nivel de lazos como a nivel de funciones. Para demostrar la diferencia entre ambas, primero se aplica la directiva `PIPELINING` en el lazo exterior `ROW` (Código 8). El reporte de síntesis de esta segunda implementación se muestra en la Figura 49.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	8.00	6.08	1.00

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
15	16	8	8	loop rewind

Detail

Instance

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	70
FIFO	-	-	-	-
Instance	6	16	276	250
Memory	-	-	-	-
Multiplexer	-	-	-	877
Register	-	-	1038	3
Total	6	16	1314	1200
Available	120	80	35200	17600
Utilization (%)	5	20	3	6

Figura 49. Estimación de desempeño y recursos de la Solución 2 del IP *matrixmul*. Esta solución utiliza pipelining a nivel del lazo exterior del Código 8. La latencia de ejecución estimada es de 15 ciclos de reloj a una frecuencia de 125MHz para un Zynq Z-7010.

El importante realizar un análisis comparativo de cada nueva solución respecto a soluciones previas. De la comparación de los reportes de síntesis de las soluciones 1 y 2, se puede destacar dos puntos importantes: i) la reducción significativa de la latencia de ejecución, y ii) el incremento de recursos de hardware. Es inherente que casi toda optimización de desempeño requiere incrementar el hardware necesario para reducir el número de ciclos de reloj que toma completar las operaciones.

Para analizar los resultados de esta nueva solución, se presenta la perspectiva de análisis de la Figura 50. La aplicación de *pipelinig* en el lazo superior (`ROW`) realiza *unrolled* de todos los lazos inferiores (en este caso los lazos `Col` y `Col.1`) mas no del lazo al que se aplica la directiva. Vivado crea el hardware necesario para cada uno de las iteraciones de los dos lazos interiores. Como la obtención de cada elemento en una multiplicación de matrices es independiente de los demás elementos, se puede observar que las multiplicaciones son llevadas

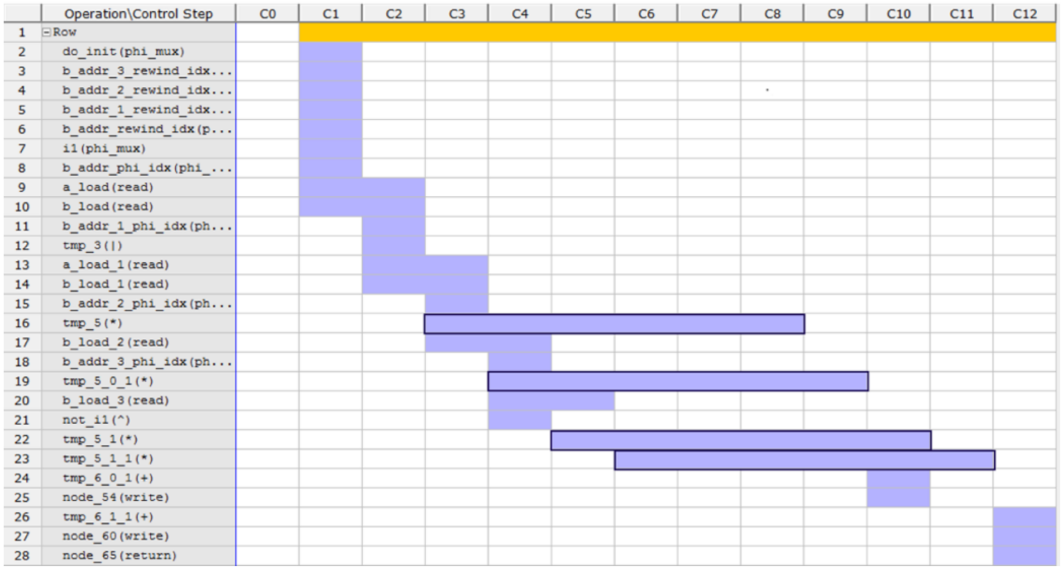


Figura 50. Perspectiva de análisis de la Solución 2 para el IP *matrixmul*. Esta solución utiliza *pipelining* a nivel del lazo exterior del Código 8. En la figura se observa como las operaciones dentro del lazo son ejecutadas de forma paralela. Además, se resaltan las multiplicaciones ejecutadas en cada iteración.

a cabo en paralelo (resaltados en la Figura 50). Estas multiplicaciones inician su ejecución con diferencia de un ciclo de reloj cada una, lo cual se debe a la limitación en el acceso a memoria y no a dependencias de datos de operaciones previas.

4.1.5.3. Solución 3

En esta nueva solución se utiliza también la directiva `PIPELINING` pero esta vez a nivel de la función *matrixmul*. Con esta optimización se consigue que se aplique unrolled en todos los lazos dentro de la función. En la Figura 51 se muestra el reporte de síntesis de esta solución. Si se comparan los reportes de las soluciones 2 y 3, se puede notar que la latencia de ejecución se ha reducido en dos ciclos de reloj.

En la Figura 52 se muestra la perspectiva de análisis de esta solución. En esta figura, se observa cómo se ha implementado *unrolled* en todos los lazos, con lo cual ahora se puede observar todas las operaciones que realiza la implementación. En la figura, se han resaltado las ocho multiplicaciones necesarias en la ejecución de una multiplicación de matrices de

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	8.00	6.08	1.00

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
13	13	4	4	function

Detail

Instance

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	128
FIFO	-	-	-	-
Instance	6	32	276	250
Memory	-	-	-	-
Multiplexer	-	-	-	41
Register	-	-	584	-
Total	6	32	860	419
Available	120	80	35200	17600
Utilization (%)	5	40	2	2

Figura 51. Estimación de desempeño y recursos de la Solución 3 del IP *matrixmul*. Esta solución utiliza pipelining a nivel de la función *top-level* del Código 8. La latencia de ejecución estimada es de 13 ciclos de reloj a una frecuencia de 125MHz para un Zynq Z-7010.

orden . Si se observa la Figura 50, correspondiente a la Solución 2, solo son mostradas cuatro multiplicaciones.

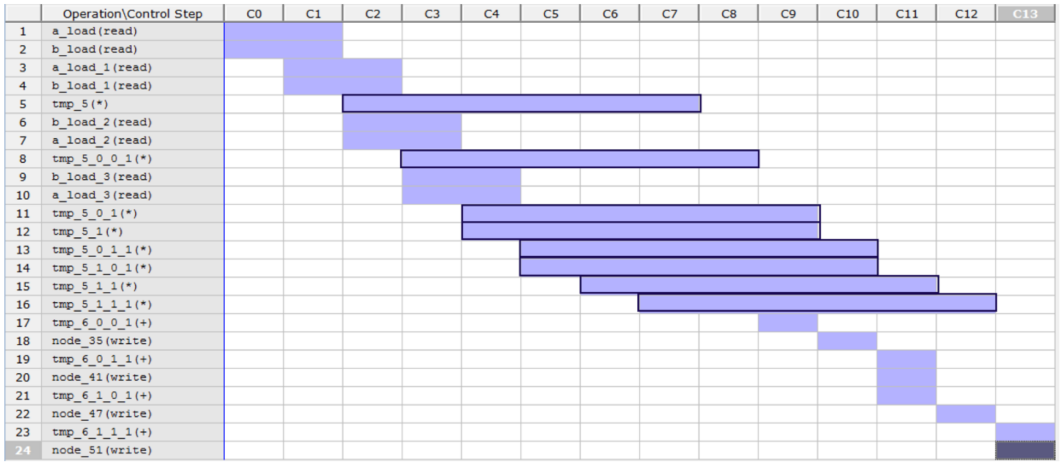


Figura 52. Perspectiva de análisis de la Solución 3 para el IP *matrixmul*. Esta solución pipelining a nivel de la función *top-level* del Código 8. En la figura se observa como el IP ejecuta todas sus operaciones usando *pipelining*, incluyendo las multiplicaciones.

La presenta solución se ejecuta con *pipeline*, gracias a lo cual la latencia e intervalo de ejecución han sido reducidos considerablemente comparados con la Solución 1. Sin embargo, la ejecución de las operaciones puede optimizarse más para alcanzar un máximo nivel de

paralelismo. Lo que impide incrementar el paralelismo en la solución actual es el acceso a los elementos de memoria que almacenan los datos. Esto se debe a que los datos son almacenados en Block RAMs, los cuales son implementados con un máximo de dos puertos de acceso en Vivado HLS, lo que restringe a dos accesos a memoria en el mismo ciclo de reloj.

4.1.5.4. Solución 4

Para resolver la limitante del acceso a memoria en Block RAMs discutida en la Solución 3, en esta solución se realiza un particionamiento de los elementos de memoria. Para ello, los Block RAMs son divididos en registros individuales de datos. Con esta mejora es posible leer y escribir en todos los registros a la vez. El reporte de síntesis de esta solución se muestra en la Figura 53, en el cual se observa una mejora considerable con respecto a la solución previa.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	8.00	6.08	1.00

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
7	7	1	1	function

Detail

Instance

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	128
FIFO	-	-	-	-
Instance	0	32	492	808
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	520	-
Total	0	32	1012	936
Available	120	80	35200	17600
Utilization (%)	0	40	2	5

Figura 53. Estimación de desempeño y recursos de la Solución 4 del IP *matrixmul*. Esta solución utiliza *pipeling* a nivel de la función *top-level* y *array partitioning* de las matrices del Código 8. La latencia de ejecución estimada es de 7 ciclos de reloj a una frecuencia de 125MHz para un Zynq Z-7010.

La reducción de ciclos de reloj puede apreciarse en la perspectiva de síntesis de la Figura 54. En esta solución, todos los registros pueden ser leídos y escritos a la vez, y así las operaciones llevarse a cabo en paralelo. Con estas optimizaciones se obtiene el mejor desempeño posible para el acelerador *matrixmul* usando Vivado HLS.

	Operation\Control Step	C0	C1	C2	C3	C4	C5	C6	C7
1	a_0_0_read(read)								
2	b_0_0_read(read)								
3	a_0_1_read(read)								
4	b_1_0_read(read)								
5	b_0_1_read(read)								
6	b_1_1_read(read)								
7	a_1_0_read(read)								
8	a_1_1_read(read)								
9	tmp_5(*)								
10	tmp_5_0_0_1(*)								
11	tmp_5_0_1(*)								
12	tmp_5_0_1_1(*)								
13	tmp_5_1(*)								
14	tmp_5_1_0_1(*)								
15	tmp_5_1_1(*)								
16	tmp_5_1_1_1(*)								
17	tmp_6_0_0_1(+)								
18	node_38(write)								
19	tmp_6_0_1_1(+)								
20	node_44(write)								
21	tmp_6_1_0_1(+)								
22	node_50(write)								
23	tmp_6_1_1_1(+)								
24	node_54(write)								

Figura 54. Perspectiva de análisis de la Solución 4 para el IP *matrixmul*. Esta solución utiliza *pipeling* a nivel de la función *top-level* y *array partitioning* de las matrices del Código 8. En la figura se observa como todas las multiplicaciones son ejecutadas en forma paralela.

4.1.5.5. Resumen de Soluciones

En la Tabla 9 se presenta un resumen mostrando las diferentes directivas de optimización y su influencia sobre la latencia, intervalo de ejecución y recursos. En la Solución 1 no se han aplicado directivas de optimización, siendo la solución con mayor latencia e intervalo. Por otro lado, la Solución 4 es la mejor implementación alcanzada para el diseño, con la mínima latencia e intervalo.

Tabla 9
Resumen de soluciones para el IP matrixmul

Solución	Directivas de Optimización	Desempeño		Recursos			
		Latencia	Intervalo	BRAM	DSP48	FF	LUT
1	-	85	86	6	4	434	350
2	Pipelinigal lazo exterior	15	8	6	16	1314	1200
3	Pipelinigen función top-level	13	4	6	32	860	419
4	Array partitioning completo de todos los bloques de memoria	7	1	0	32	1012	936

Nota: Las latencias, intervalos y recursos son estimados para un Zynq Z-7010.

4.2. Transformada Rápida de Fourier (FFT)

El segundo acelerador en hardware implementado en esta tesis es una Transformada Rápida de Fourier (FFT), la cual tiene muchas aplicaciones en varios campos de la ingeniería, y es uno de los *benchmarks* más usados en estudios relacionados a aceleradores de hardware (Reagen, Adolf, Shao, Wei, & Brooks, 2014). Se ha escogido el *benchmark* FFT/STRIDED proporcionado por MachSuite. La suite fue desarrollada por investigadores de la Universidad de Harvard y está enfocada a la evaluación de herramientas HLS para la implementación de aceleradores.

El desarrollo de este ejemplo se aprovechará para explicar conceptos particulares acerca de la implementación en Vivado HLS que pueden representar dificultades para el diseñador, de forma específica problemas relacionados con *variable loop bounds*.

4.2.1. Algoritmo

La FFT es una implementación eficiente para el cálculo de la DFT (Transformada Discreta de Fourier). La DFT de una señal x se define como:

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j2\pi kn/N} \qquad , k = 0, 1, \ldots N-1 \tag{4.2}$$

Donde N es el número de puntos muestreados. El coeficiente se denomina generalmente como factor de giro (*twiddle*) y se lo representa como

$$\omega_N = e^{-j2\pi/N} \tag{4.3}$$

Usando la Ecuación 4.3 se reescribe la Ecuación 4.2 como sigue

$$X[k] = \sum_{n=0}^{N-1} x[n] {\omega_N}^{kn} \qquad , k = 0, 1, \ldots N-1 \tag{4.4}$$

El cálculo de la DFT tal como se define en la Ecuación 4.4 tiene un gran costo computacional. El número de operaciones requeridas al calcular una DFT de esta manera es $N(N-1)$ de sumas y N^2 multiplicaciones. Es por esto que muchos algoritmos FFT han sido desarrollados para poder calcular eficientemente la DFT, usando menos recursos computacionales. Uno de los algoritmos más eficientes es el método de Cooley - Tukey (S.Arunachalam, S.M.Khairnar, & B.S.Desale, 2005), el cual es utilizado en el *benchmark* seleccionado.

4.2.1.1. Algoritmo Cooley – Tukey para FFT

Este método consiste en separar una DTF de un gran número de muestras en varias DFTs más pequeñas. Para ello se separa la sumatoria de la Ecuación 4.4 en dos sumatorias, una para índices pares $n = 2r$, y otra para índices impares $n = 2r + 1$. Reescribiendo la Ecuación

4.4 de esta forma se tiene

$$X[k] = \sum_{r=0}^{N/2-1} x[2r]{\omega_N}^{k(2r)} + \sum_{r=0}^{N/2-1} x[2r+1]{\omega_N}^{k(2r+1)} \tag{4.5}$$

El factor de giro tiene una propiedad denominada recursiva, la cual se define como

$${\omega_N}^{2rk} = {\omega_{N/2}}^{rk} \tag{4.6}$$

En base a la Ecuación 4.6 se reescribe la Ecuación 4.5 a continuación

$$X[k] = \sum_{r=0}^{N/2-1} x[2r]{\omega_{N/2}}^{kr} + {\omega_N}^{k} \sum_{r=0}^{N/2-1} x[2r+1]{\omega_N}^{kr} \tag{4.7}$$

La primera sumatoria de la Ecuación 4.7 corresponde a la DFT de las muestras pares $X_e[k]$ y la segunda sumatoria corresponde a la DFT de las muestras impares $X_o[k]$, cada una de $N/2$ puntos. Gracias a la periodicidad del exponencial complejo se puede expresar el resultado de la FFT como

$$\begin{cases} X[k] = X_e[k] + {\omega_N}^{k} X_o[k] \\ X[k+N/2] = X_e[k] - {\omega_N}^{k} X_o[k] \end{cases} , k = 0, 1, \ldots N/2-1 \tag{4.8}$$

Una de las formas en las que se implementa el algoritmo Cooley y Tukey es conocida como *radix-2*. La cual consiste en dividir recursivamente cada nueva DFT entre coeficientes pares e impares hasta conseguir pequeñas DFTs de dos puntos cada una. Para facilitar su comprensión, este algoritmo generalmente se representa mediante un diagrama conocido como mariposa. En la Figura 55 se representa el diagrama mariposa para una FFT de cuatro puntos.

En la Figura 55 a) se observa cómo son separadas las muestras por índices pares e impares, reduciendo la DFT de cuatro puntos a dos DFTs de dos puntos cada una. Los resultados de las mismas son combinados de acuerdo a la Ecuación 4.8. En la Figura 55 b) se muestra el

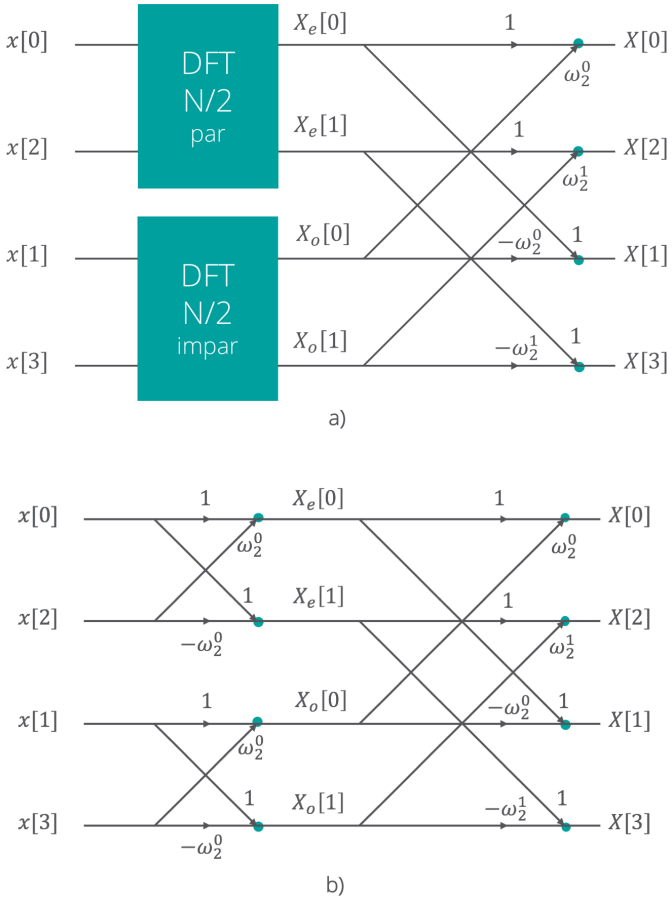


Figura 55. Diagramas de mariposa para una FFT de 4 puntos: a) Representación inicial con DFTs de dos puntos, b) diagrama completo. Las flechas representan el flujo de datos de las muestras, y los valores sobre las fechas representan los pesos por los cuales son multiplicados. Cada círculo verde representa un punto de suma entre las dos fechas que llegan al círculo.

diagrama de mariposa completo, cuyo resultado se resume en las siguientes ecuaciones:

$$X[0] = x[0] + \omega_4^0 x[2] + \omega_4^0(x[1] + \omega_4^0 x[3])$$
$$X[1] = x[0] + \omega_4^2 x[2] + \omega_4^1(x[1] + \omega_4^2 x[3])$$
$$X[2] = x[0] + \omega_4^2 x[2] + \omega_4^2(x[1] + \omega_4^0 x[3])$$
$$X[3] = x[0] + \omega_4^2 x[2] + \omega_4^3(x[1] + \omega_4^2 x[3])$$

Aplicando el método Cooley - Tukey *radix-2* se logra reducir el número de operaciones necesarias para su cálculo a $N \log_2 N$ sumas y $(N/2) \log_2 N$ multiplicaciones. En la Tabla 10

se muestra el número de operaciones necesarias para llevar a cabo tanto una DFT y una FFT de 1024 puntos.

Tabla 10
Número de operaciones para una DFT y FFT de 1024 puntos

Número de puntos	DFT		FFT	
N	Sumas $N(N-1)$	Multiplicaciones N^2	Sumas $N \log_2 N$	Multiplicaciones $(N/2) \log_2 N$
1024	1047552	1048576	10240	5120

Para la creación del acelerador de FFT, se emplea el algoritmo proporcionado por MachSuite. El algoritmo se encuentra especificado en lenguaje C para una FFT de 1024. Dicho algoritmo se ha trasladado a C++ y se muestra en el Código 11. Los argumentos de la función *fft* son: **i)** las muestras discretas reales (`real[FFT_SIZE]`), **ii)** muestras discretas imaginarias (codeimg[FFT_SIZE]), **iii)** parte real de los factores de giro (`real_twid[FFT_SIZE/2]`) y iv) parte imaginaria de los factores de giro (`img_twid[FFT_SIZE/2]`).

Código 11
Algoritmo para FFT con método Cooley - Tukey en C++

```
#include "fft.h"
#include <math.h>
void fft(double real[FFT_SIZE], double img[FFT_SIZE],
                 double real_twid[FFT_SIZE/2], double img_twid[FFT_SIZE
                    /2]) {
        int even, odd, span, log, rootindex;
        double temp;
        log = 0;
        for (span = FFT_SIZE >> 1; span; span >>= 1, log++) {
                for (odd = span; odd < FFT_SIZE; odd++) {
                        odd |= span;
                        even = odd ^ span;
                        temp = real[even] + real[odd];
                        real[odd] = real[even] - real[odd];
                        real[even] = temp;
                        temp = img[even] + img[odd];
                        img[odd] = img[even] - img[odd];
                        img[even] = temp;
                        rootindex = (even << log) & (FFT_SIZE - 1);
                        if (rootindex) {
                                temp = real_twid[rootindex] * real[odd]
                                                - img_twid[rootindex] *
                                                    img[odd];
```

```
22                                  img[odd] = real_twid[rootindex] * img[
                                        odd]
23                                                  + img_twid[rootindex] *
                                                        real[odd];
24                                  real[odd] = temp;
25                              }
26                          }
27                      }
28  }
```

El algoritmo FFT debe seleccionar los índices de muestras pares e impares para calcular cada DFT de dos puntos. En la Figura 56 se muestra un ejemplo para una DFT de 8. La selección de los índices se logra a través de la inversión de la representación binaria[9] de los índices originales, lo cual se muestra en la Tabla 11. Debido a que no existe una instrucción en C/C++ para realizar este tipo de inversión, el algoritmo obtiene estos índices a través de una serie de operaciones a nivel de bits.

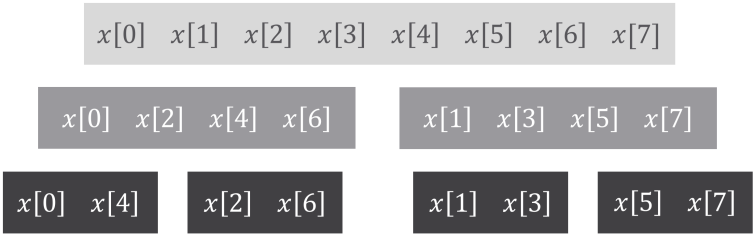


Figura 56. Selección de índices de las muestras para calcular pequeñas DFTs de dos puntos. Las muestras son divididas en índices pares e impares hasta conseguir grupos de dos muestras.

[9]La inversión binaria mencionada no hace referencia a una inversión del estado de cada bit, sino a una inversión del orden en la posición de los bits (ej. El número 110 se invierte como 011).

Tabla 11
Representación binaria para la selección de índices en una FFT

Índice original		Índice con inversión de bits	
Decimal	Binario	Decimal	Binario
0	000	0	000
1	001	4	100
2	010	2	010
3	011	6	110
4	100	1	001
5	101	5	101
6	110	3	011
7	111	7	111

4.2.2. Implementación en Vivado HLS

El código C++ mostrado en el Código 11 se traslada a Vivado HLS para sintetizar el acelerador en hardware. La explicación de los conceptos acerca de la creación del *test bench* y del uso de interfaces AXI de las Secciones 4.1.2.1 y 4.1.2.2 son aplicables también a este diseño. En esta sección, se aprovecha el algoritmo FFT, para explicar detalles acerca de la implementación de especificaciones C con variable loop bounds (*lazos de límites variables*).

4.2.2.1. Variable Loop Bounds

Variable loop bounds son una condición en la cual el número de iteraciones de un lazo no es fijo. Este tipo de lazos son comunes en implementaciones de software, sin embargo, puede limitar la optimización de implementaciones en hardware utilizando Vivado HLS. A continuación, se explica las causas que generan que los límites de un lazo sean variables y como solucionarlos.

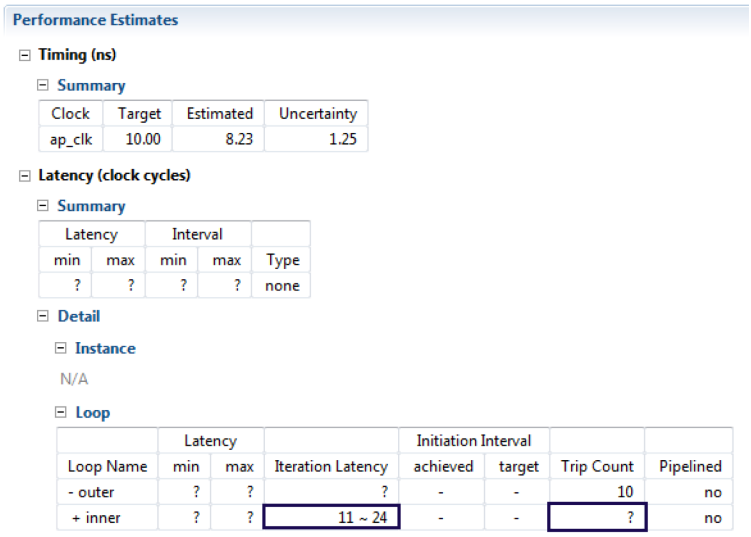
Se debe prestar especial cuidado al uso de estructuras condicionales (`If`, `For`, `While`, `Do while`, `Switch`) en una especificación C a ser sintetizada por Vivado HLS. Si la evaluación de la condición depende de argumentos de entrada, o del resultado de operaciones que Vivado no pueda inferir, se considerará que los lazos tienen límites variables. Esto se debe a que

la herramienta no puede determinar exactamente si las operaciones dentro de la estructura condicional se ejecutaran o no. Si esto sucede, Vivado HLS no pueda determinar la latencia e intervalo de una implementación ya que estos no son valores estáticos. Por lo tanto, los reportes de síntesis mostraran signos de interrogación en los estimados de desempeño. En la Figura 57 se muestra el reporte de síntesis del IP *fft*, el cual contiene límites de lazos variables.

Pese a que una especificación incluya lazos con *variable loop bounds*, esta especificación es sintetizable a hardware mediante Vivado HLS. Sin embargo, el diseñador encontrará restricciones al momento de aplicar optimizaciones. Por ejemplo, optimizaciones de desempeño que impliquen realizar *unrolled* de lazos no son posibles en la mayoría de los casos, ya que puede ser incierto cuantas copias del hardware deben ser creadas. Para más información acerca de *variable loop bounds* puede consultar el documento "*Vivado Design Suite User Guide - High-Level Synthesis*" (Xilinx - UG902, 2017).

Para este ejemplo, la función *fft* del Código 11 muestra dos ciclos for anidados. La condición del ciclo `for` interior depende de la variable `span`, sobre la cual se llevan a cabo operaciones a nivel de bits en el ciclo `for` exterior. Vivado no puede inferir cuantas veces será iterado el lazo interior debido a que no conoce el resultado de las operaciones. En la estimación de desempeño de la Figura 57 se ha resaltado el *trip count* del lazo interior (`inner`), el cual es remplazado por un signo de interrogación. Además, se resalta también la latencia por iteración del ciclo `for inner`, la cual también es variable. Esto se debe a que existe una sentencia condicional `if` en su interior, la misma que depende también de operaciones previas. Por lo tanto, Vivado no puede determinar cuántas veces será verdadera la evaluación de la condición de la estructura `if`, y por ello muestra una latencia máxima y mínima. La latencia máxima asume que la condición siempre será verdadera, y la latencia mínima asume que la condición siempre es falsa.

Para que Vivado sea capaz de mostrar las latencias estimadas es necesario establecer límites fijos en el ciclo `for` interior. Esto conlleva realizar cambios en el algoritmo, los cuales se muestran en el Código 12. En este código se observa que el ciclo `for` (`inner`) tiene límites



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.23	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
?	?	?	?	none

Detail

Instance

N/A

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- outer	?	?	?	-	-	10	no
+ inner	?	?	11 ~ 24	-	-	?	no

Figura 57. Estimación de desempeño para la función *fft* original del Código 11. Los estimados de desempeño no se muestran en el reporte debido a que los lazos tienen límites variables. Además, se resalta el trip count del lazo interior, el cual es desconocido y causa que la latencia de iteración sea variable.

fijos definidos, y las asignaciones `odd=span` y `odd++` se realizan fuera de la definición de la estructura condicional. Estos cambios no afectan al funcionamiento del algoritmo, únicamente ayudan a que Vivado infiera los límites del lazo y permitir la implementación de optimizaciones. Al sintetizar la función *fft* del Código 12 se obtiene el reporte de síntesis mostrado en la Figura 58, donde ya se observa el *trip count* y las latencias estimadas.

Código 12
Algoritmo para FFT con método Cooley - Tukey en C++ modificado para resolver problema de variable loop bounds en Vivado HLS

```
#include "fft.h"
void fft(double real[FFT_SIZE], double img[FFT_SIZE],
                 double real_twid[FFT_SIZE / 2], double img_twid[
                    FFT_SIZE / 2]) {
        int even, odd, span, log, rootindex,i;
        double temp;
        log = 0;
        outer:for (span = FFT_SIZE >> 1; span; span >>= 1, log++) {
                odd=span;
                inner:for (i = 0; i < FFT_SIZE/2; i++) {
                        odd |= span;
                        even = odd ^ span;
                        temp = real[even] + real[odd];
                        real[odd] = real[even] - real[odd];
```

```
14                              real[even] = temp;
15                              temp = img[even] + img[odd];
16                              img[odd] = img[even] - img[odd];
17                              img[even] = temp;
18                              rootindex = (even << log) & (FFT_SIZE - 1);
19                              if (rootindex) {
20                              temp = real_twid[rootindex] * real[odd]
21                                              - img_twid[rootindex] * img[odd
                                                    ];
22                              img[odd] = real_twid[rootindex] * img[odd]
23                                              + img_twid[rootindex] * real[
                                                    odd];
24                              real[odd] = temp;
25                              }
26                              odd++;
27                      }
28              }
29      }
```

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.23	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
56341	122901	56342	122902	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- outer	56340	122900	5634 ~ 12290	-	-	10	no
+ inner	5632	12288	11 ~ 24	-	-	512	no

Figura 58. Estimación de desempeño para la función *fft* modificada del Código 12. Los valores de latencia e intervalo se muestran variables debido a que la especificación C incluye una sentencia condicional if al interior de un lazo for.

4.2.3. Integración del IP-core en el SoC

La integración del acelerador de FFT en el SoC se la realiza en Vivado IP Integrator. En la Figura 59 se presenta el diagrama de bloques de la integración. En este diagrama se distinguen cuatro IP-cores: **1) Zynq PS:** IP que representa al procesador ARM, cache,

controladores y otros componentes que permiten la ejecución de software y su interacción con el PL y periféricos. **2) IP generado en HLS:** acelerador en hardware para calcular una FFT de 1024 puntos. **3) AXI Interconnect:** IP para la conexión del acelerador a través de su interfaz AXI *slave* a la interfaz AXI *master* del PS (Sección 4.1.2.3). **4) Processor System Reset:** IP encargado del *reset* del sistema, el cual actúa tanto sobre el AXI Interconnect como sobre los IP-cores conectados a él.

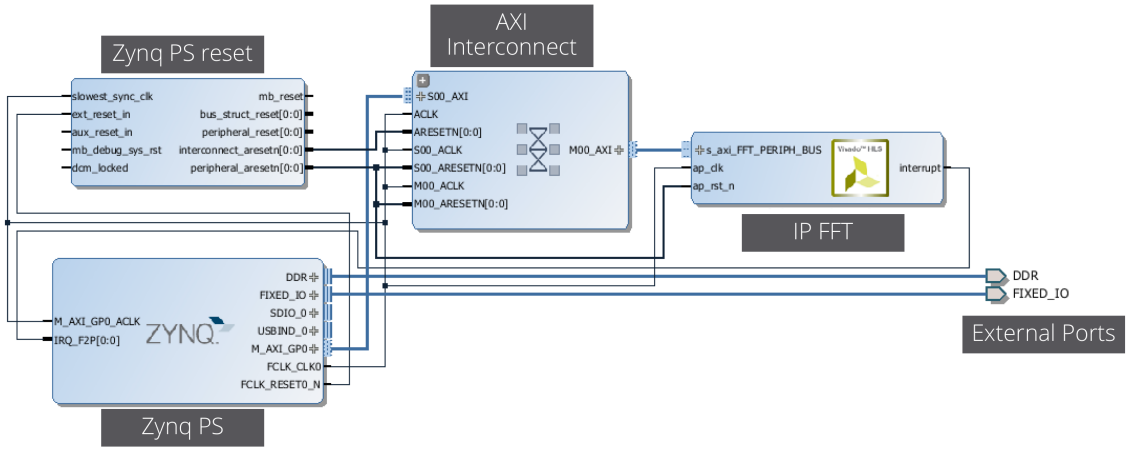


Figura 59. Diagrama de bloques de la integración de IP *fft* en el SoC.

4.2.4. Optimizaciones

En este diseño se ha aplicado directivas de optimización de desempeño como pipeline. Sin embargo, no se ha logrado mejores resultados comparados con la solución generada por defecto en Vivado HLS. Esta limitación se debe a dos razones: 1) el algoritmo para FFT usado tiene una alta dependencia de datos, y 2) el algoritmo tiene dos lazos anidados que no pueden ser ejecutados usando pipeline. Cada una de estas limitaciones se explican a continuación.

1. **Dependencias de datos:** este algoritmo tiene tres tipos de dependencias: i) lectura después de escritura (RAW): una instrucción depende del resultado de la anterior, ii) lectura después de escritura (WAR): una instrucción no puede actualizar una variable mientras esta no haya sido leída previamente, iii) escritura después de escritura (WAW):

una variable es escrita dos veces con una operación de lectura en medio, por lo cual la actualización de la misma debe realizarse en un orden determinado para no afectar a otras variables (Xilinx - UG902, 2017). Este tipo de dependencias se presentan en varias variables del algoritmo FFT del Código 12, lo cual obliga a que la implementación en hardware sea secuencial.

2. **Limitaciones de pipeline en lazos anidados:** para aplicar pipelining a lazos anidados se requiere que los mismos se implementen usando loop flattening (Sección 3.3.5.1). Esta optimización es posible únicamente en lazos denominados por Xilinx como perfectos o semi-perfectos. Las condiciones para la aplicación de pipelining a un lazo anidado son: i) no contener instrucciones en medio de los lazos anidados, ii) los límites de los lazos anidados deben ser constantes, y únicamente el lazo exterior puede tener límites variables (Xilinx - UG902, 2017). El algoritmo del Código 12 no cumple con ninguna de estas dos condiciones, y por lo tanto una implementación con pipeline no puede ser sintetizada usando Vivado HLS.

4.3. AES (Advanced Encryption Standard)

El tercer acelerador en hardware creado en la tesis es un sistema criptográfico AES (Advanced Encryption Standard). AES es el estándar de encriptación y des-encriptación del gobierno de los Estados Unidos desde el 2001, y actualmente es usado a nivel mundial en la encriptación de información financiera, de telecomunicaciones, militar y referente a datos de gobiernos (Azad & Pathan, 2014). Este algoritmo ha sido seleccionado de la suite CHStone, la cual plantea *benchmarks* para evaluar herramientas HLS basadas en lenguaje C (Hara, Tomiyama, Honda, Takada, & Ishii, 2008).

4.3.1. Algoritmo

AES es un algoritmo simétrico, lo cual significa que la misma clave es utilizada para la encriptación y des-encriptación de datos. Las longitudes de las claves pueden ser de 128, 192, o 256 bits, mientas que el tamaño de los bloques de datos a encriptase y des-encriptarse son de 128 bits. AES tiene la ventaja de ser un algoritmo que opera a nivel de bytes, lo cual lo hace más simple de entender, y además puede implementarse en software y hardware (Azad & Pathan, 2014).

AES está basado en permutaciones y sustituciones, las cuales son organizadas en pasos establecidos que se repiten múltiples veces; cada una de las repeticiones es llamada *round* (Azad & Pathan, 2014). El número de *rounds* a realizarse depende de la longitud de la clave. Para claves de 128, 192, 256 se ejecutan 10, 12 y 14 rounds respectivamente. Cada *round* está formado por cuatro tipos de operaciones, las cuales se describen a continuación.

4.3.1.1. Operaciones Usadas en la Encriptación AES

Los datos de entrada requeridos para una encriptación AES son: i) un vector de 128 bits denominado *State*, el cuál es el bloque de datos a ser encriptado, y ii) una clave de 128,192, o 256 bits denominada *Key*. El vector *State* es organizado en una matriz de orden 4×4 bytes. Por otro lado, la clave es organizada como un arreglo de orden $4 \times (b/32)$ bytes, donde b es el número de bits de *key*. Posteriormente, la clave es expandida dependiendo el número de rounds r a una matriz de 4 filas y $r + 1$ columnas en un proceso denominado *key schedule*.

- **SubByte:** En esta operación cada byte es remplazado por otro byte de acuerdo a una tabla de búsqueda predefinida, la cual se conoce como *S-box*. En la Figura 60 se muestra un ejemplo de la operación *SubByte*. En este ejemplo, el número hexadecimal 43 es remplazado por el número 1A correspondiente de la tabla de búsqueda. Este proceso es llevado a cabo para todos los bytes de la matriz *State*. De igual forma existe otra *S-box* definida para el proceso de des-encriptación.

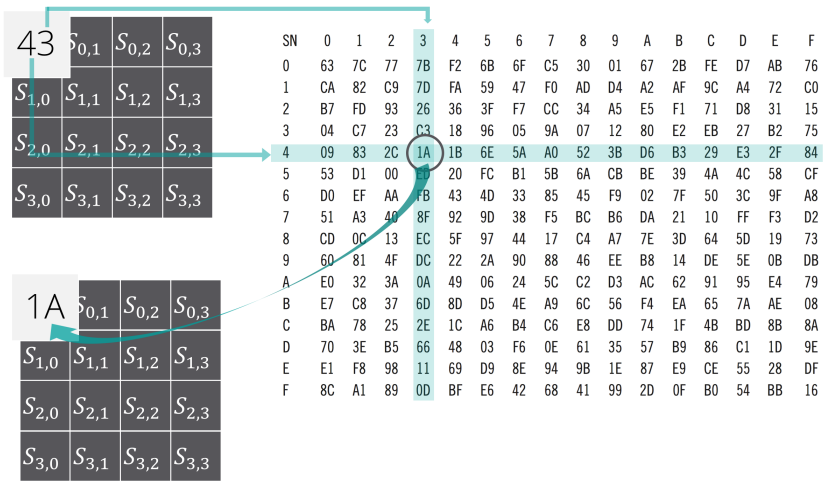


Figura 60. Operación *SubByte* de AES. Se utiliza representación hexadecimal para cada byte. El primer dígito de *State* índica la fila y el segundo dígito indica la columna de la tabla de búsqueda. El byte localizado en la tabla remplaza al byte original de la matriz *State*

- **ShiftRows:** En esta operación, los elementos de la matriz *State* se rotan ciertas posiciones hacia la derecha según la fila en la que se encuentran. Los elementos de la primera fila no son rotados, los de la segunda se rotan una posición, los de la tercera fila se rotan dos posiciones y finalmente los de la cuarta fila se rotan tres posiciones. En la Figura 61 se muestra un ejemplo de la operación *ShiftRows*. Para la des-encriptación la rotación se la realiza hacia la izquierda.

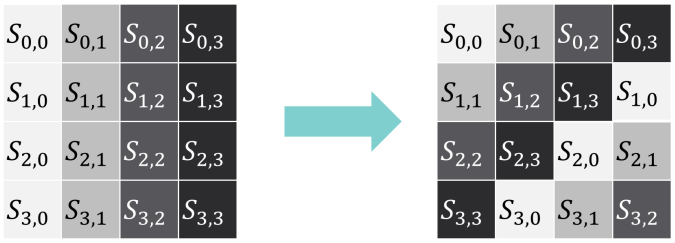


Figura 61. Operación *ShiftRows* de AES. Los elementos de la primera fila no son rotados, los de la segunda se rotan una posición, los de la tercera fila se rotan dos posiciones y finalmente los de la cuarta fila se rotan tres posiciones.

- **MixColumn:** En esta operación cada columna de la matriz *State* es multiplicada por una matriz de orden preestablecida en el estándar AES. Esta multiplicación resulta en

un vector columna el cual remplaza a la columna original de la matriz *State*. En la Figura 62 se muestra un ejemplo de la operación *MixColumn*.

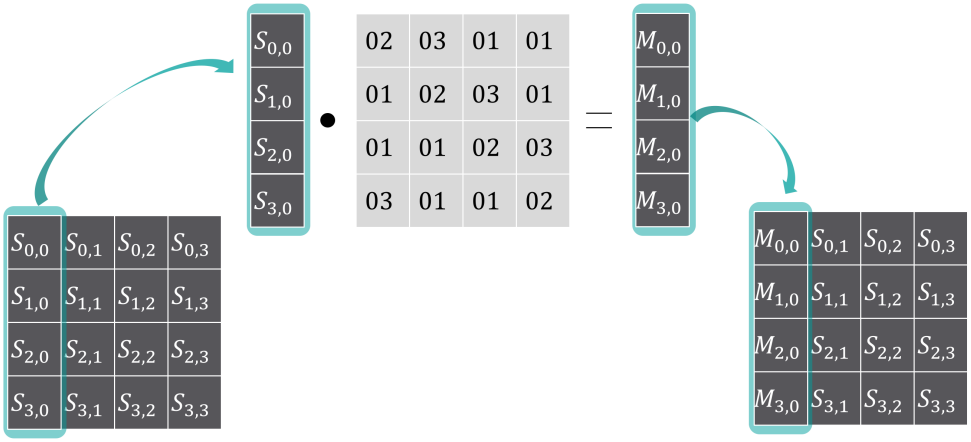


Figura 62. Operación *MixColumn* de AES. En esta operación cada columna es multiplicada por la matriz preestablecida (en color gris). El resultado remplaza a la columna original en la matriz *State*.

- **AddRoundKey:** En esta etapa se realiza una operación XOR entre cada byte de la matriz *State* y cada byte de la sub-clave correspondiente[10]. En la Figura 63 se muestra un ejemplo de la operación *AddRoundKey*.

Cada una de las cuatro operaciones usadas en AES son ejecutadas 10, 12 o 14 veces dependiendo de la longitud de la clave utilizada. Estas operaciones son organizadas dentro de tres fases. En la Figura 64 se muestra un ejemplo de encriptación AES considerando una clave de 128 bits de longitud. AES es un algoritmo denominado reversible, lo cual indica que los pasos ejecutados para la encriptación pueden ser aplicados en orden inverso para la des-encriptación. Por lo tanto, la explicación para la des-encriptación será omitida.

[10]El proceso de Key Schedule se explica en la parte final de esta sección, en ella se detalla la generación de las sub-claves a partir de una clave única.

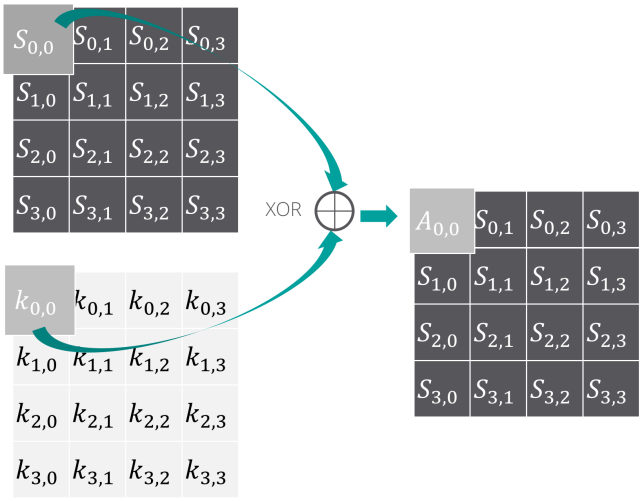


Figura 63. Operación *AddRoundKey* de AES. En esta operación realiza una XOR entre cada elemento de la matriz *State* y cada elemento de la subclave.

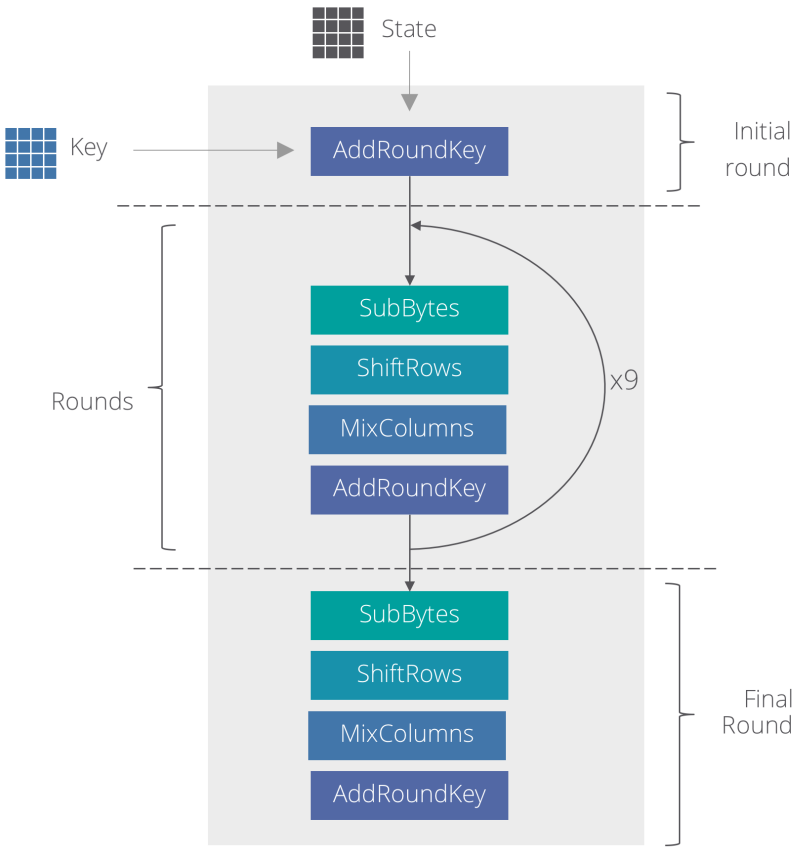


Figura 64. Proceso de encriptación AES para una clave de 128 bits. Se realizan un total de 10 rounds con las operaciones *SubBytes*, *ShiftRows*, *MixColumns* y *AddRoundKey*.

4.3.1.2. Key Schedule

La expansión de la clave a una matriz de 4 filas y $r + 1$ columnas tiene como objetivo crear una sub-clave para cada uno de los *rounds* r. Los tres procesos utilizados dentro de *key schedule* son: **i) RotWorld:** Rotación de un byte en cada columna similar al proceso explicado para la etapa *ShiftRow*. **ii) SubWord:** Remplazo de bytes usando la misma tabla de búsqueda usada en *SubByte*. **iii) Rcon:** Operación XOR entre los bytes obtenidos de las etapas anteriores, un vector constante y la columna W_{i-4} siendo W_i la columna buscada. En la Figura 65 se ha representado la generación de las sub-claves para una clave única de 128 bits.

4.3.2. Implementación en Vivado HLS

El presente ejemplo se aprovecha para explicar detalles acerca de la implementación de una especificación C usando una jerarquía de archivos con variables globales, cuyo uso es común en software, pero puede generar problemas al momento de ser sintetizado a hardware.

El algoritmo para AES desarrollado por la suite CHStone en lenguaje C incluye una jerarquía de archivos, los cuales realizan los procesos de encriptación, des-encriptación y *key schedule*. Para que estos archivos y funciones C puedan acceder a los arreglos *State* y *key*, entre otros, dichos arreglos deben ser definidos como variables globales. Una jerarquía de este tipo puede ser trasladada a Vivado HLS, simularse mediante un *test bench* para comprobar los resultados y sintetizarse. Sin embargo, es posible que la ejecución de la co-simulación falle. En la Figura 66 se muestra el reporte de co-simulación para el algoritmo AES de la suite CHStone, el cual indica que la simulación RTL ha fallado (*Fail*). Al parecer, la causa de este error se debe a que existen problemas en la forma en la que Vivado infiere la lógica RTL de escritura y lectura en variables globales. En Vivado HLS, un error en la co-simulación significa que la implementación RTL no obtiene los mismos resultados que los indicados en el *test bench*. Por lo tanto, pese a que el algoritmo se ejecute en software (Simulación C), en

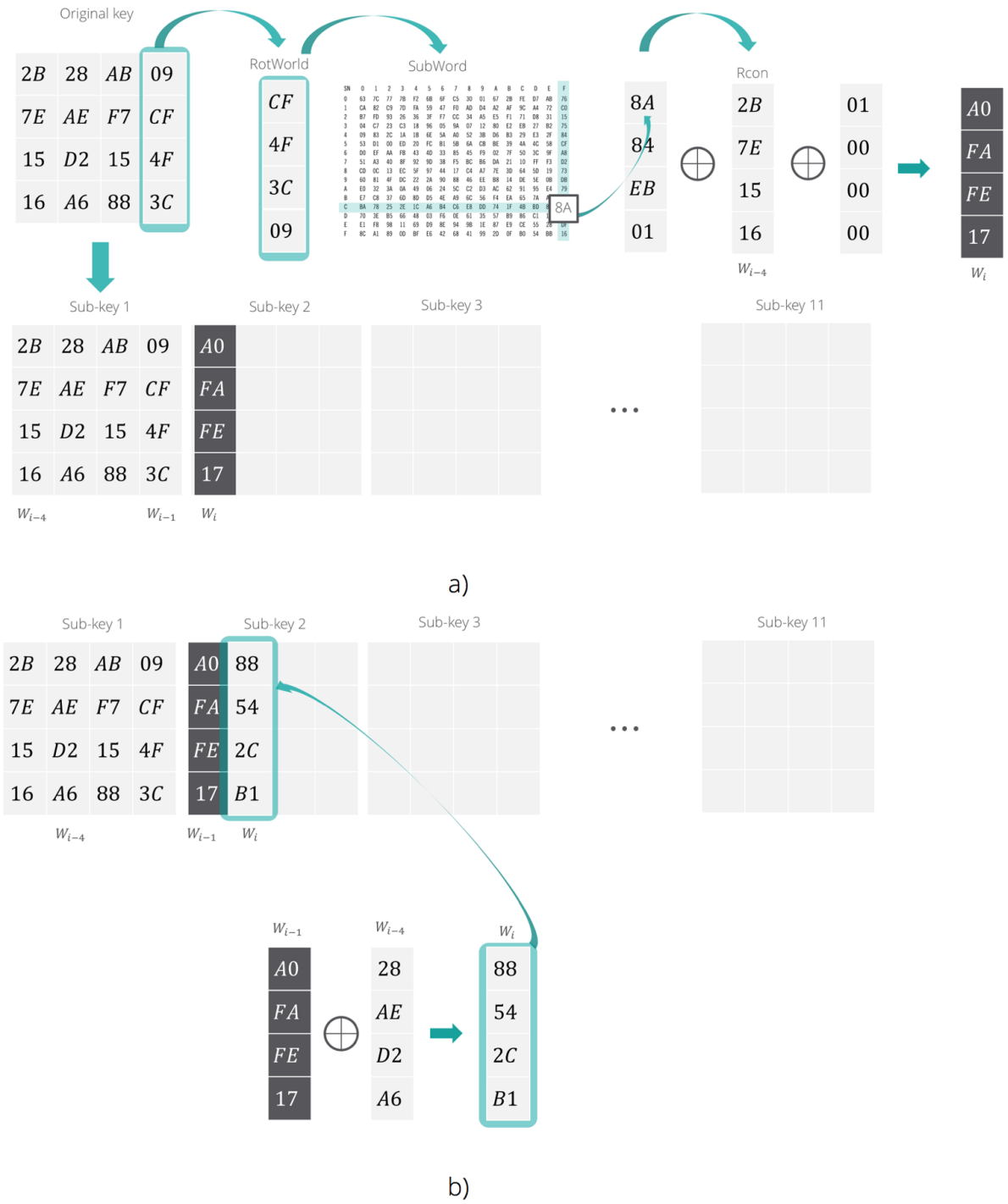


Figura 65. Proceso key schedule para obtención de sub-claves AES. a) Proceso de obtención de la primera columna de cada sub-clave. La primera sub-clave es una copia idéntica de la clave original. Para las siguientes sub-claves se toma la última columna de la anterior y se aplica los procesos RotWord, SubWord y Rcon. b) Proceso de obtención de las demás columnas de las sub-claves. Se realiza una operación XOR entre la columna anterior W_{i-4} y la columa W_i .

ciertos casos Vivado no podrá inferir la lógica correcta para su implementación en hardware.

Cosimulation Report for 'aes_main'

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Fail	3151	3151	3151	0	0	0

Figura 66. Reporte de co-simulación para el benchmark AES de la fuente CHStone.

Modificación de la especificación C del algoritmo AES

La solución que se ha plateado para la implementación del acelerador de AES consiste en: i) trasladar el algoritmo a lenguaje C++, ii) definir todas las variables globales como tipo extern, y iii) convertir las variables globales que sean leídas durante el *test bench* en argumentos de la función *top-level* del DUT. Esto permite que tanto la simulación C, síntesis y co-simulación se lleven a cabo sin problemas. En la Figura 67 se muestran los cambios realizados en el presente ejemplo. En esta figura se observa como se ha modificado el tipo de las variables globales a `extern`. Además, las variables `enc` y `dec`, que son leídas en el *test bench*, son convertidas en argumentos de las funciones `aes_main`, `decrypt` y `encrypt`.

```
#include <stdio.h>
int type;
int nb;
int round_val;
int key[32];
int statemt[32];
int word[4][120];
int main_result;
int enc[32];
int dec[32];
void aes_main(int statemt[32], int key[32]);
void decrypt(int statemt[32], int key[32], int type);
void encrypt(int statemt[32], int key[32], int type);
void KeySchedule(int type, int key[32]);
void ByteSub_ShiftRow(int statemt[32], int nb);
int SubByte(int in);
void InversShiftRow_ByteSub(int statemt[32], int nb);
void MixColumn_AddRoundKey(int statemt[32], int nb, int n);
void AddRoundKey_InversMixColumn(int statemt[32], int nb, int n);
void AddRoundKey(int statemt[32], int type, int n);
```

a)

```
#include <stdio.h>
extern int type;
extern int nb;
extern int round_val;
extern int key[32];
extern int statemt[32];
extern int word[4][120];
extern int main_result;
int aes_main(int statemt[32],int key[32],int enc[32],int dec[32]);
int decrypt(int statemt[32], int key[32], int type,int dec[32]);
int encrypt(int statemt[32], int key[32], int type,int enc[32]);
int KeySchedule(int type, int key[32]);
void ByteSub_ShiftRow(int statemt[32], int nb);
int SubByte(int in);
void InversShiftRow_ByteSub(int statemt[32], int nb);
int MixColumn_AddRoundKey(int statemt[32], int nb, int n);
int AddRoundKey_InversMixColumn(int statemt[32], int nb, int n);
int AddRoundKey(int statemt[32], int type, int n);
```

b)

Figura 67. Modificación del archivo de cabecera AES.h. Las variables de tipo global se definen como extern. Las variables leidas durante el test benchs son convertidas en argumentos de la función top-level.

Síntesis de variables globales

Cuando se utilizan variables externas pueden surgir inconvenientes en la inferencia de las interfaces. Por ejemplo, variables externas que no son argumentos de la función *top-level* pueden ser inferidas como puertos. Si fuese el caso, es conveniente que las variables no sean definidas como variables globales de tipo *extern* y sean convertidas en argumentos de las *sub-funciones*. Por el contrario, si el diseñador desea sintetizar como puertos aquellas variables externas que no sean inferidas como tales, puede realizarlo en la configuración de las soluciones de Vivado HLS mediante una directiva de síntesis. Para más información acerca de la síntesis de variables globales puede consultar el documento "*Vivado Design Suite User Guide, High-Level Synthesis*" (Xilinx - UG902, 2017).

4.3.3. Integración del IP-core en el SoC

La integración del acelerador de AES en el SoC se la realiza en Vivado IP Integrator. En la Figura 68 se presenta el diagrama de bloques de la integración. En este diagrama se distinguen cuatro IP-cores: **1) Zynq PS:** IP que representa al procesador ARM, cache, controladores y otros componentes que permiten la ejecución de software y su interacción con el PL y periféricos. **2) IP generado en HLS:** acelerador en hardware para realizar una encriptación y des-encriptación AES. **3) AXI Interconnect:** IP para la conexión del acelerador a través de su interfaz AXI *slave* a la interfaz AXI *master* del PS (Sección 4.1.2.3). **4) Processor System Reset:** IP encargado del *reset* del sistema, el cual actúa tanto sobre el AXI Interconnect como sobre los IP-cores conectados a él.

4.3.4. Optimizaciones

En el presente diseño se ha analizado cada una las etapas usadas por AES. Las operaciones dentro de *SubBytes*, *ShiftRows*, *MixColumns*, y *AddRoundKey* son altamente paralelizables. Las operaciones que realiza cada etapa sobre los elementos de la matriz State son indepen-

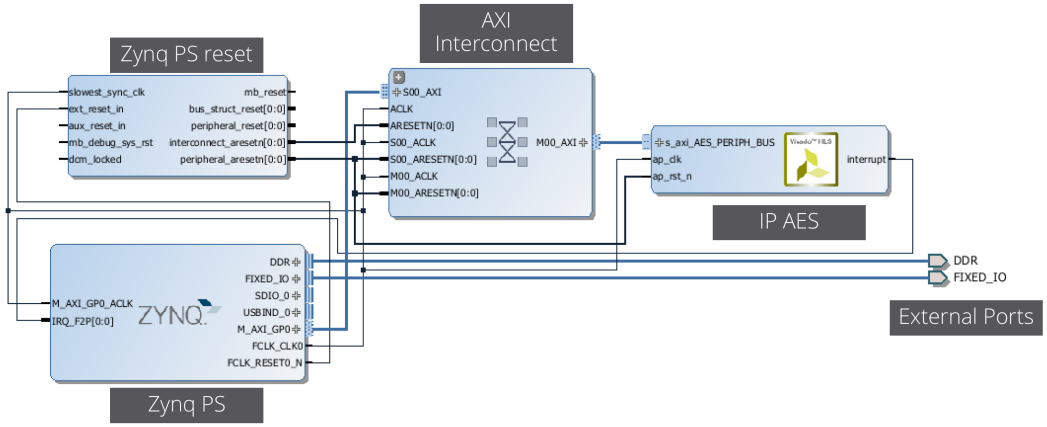


Figura 68. Diagrama de bloques de la integración de IP AES en el SoC.

dientes. Además, estas etapas pueden ser ejecutadas usando pipelining. Sin embargo, para iniciar con la encriptación AES, se requiere de las sub-claves obtenidas en el proceso *Key Schedule*. Este proceso se realiza previamente a la encriptación y cuenta con las etapas *RotWorld*, *SubWord* y *Rcon*, las cuales tienen una alta dependencia de datos entre ellas, por lo cual no se puede paralelizar. Adicionalmente, el proceso encriptación AES es dependiente del proceso *Key Schedule*, por lo cual tampoco se pueden ejecutar en paralelo. En la Figura 69 se representa las operaciones del acelerador de AES usando pipelining para las etapas *SubBytes*, *ShiftRows*, *MixColumns* y *AddRoundKey*.

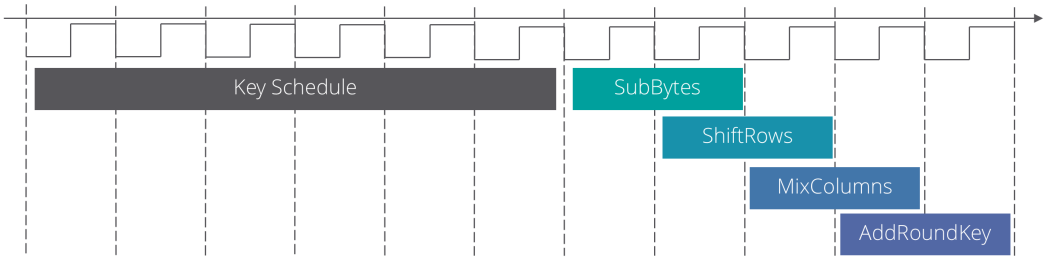


Figura 69. Ejecución de encriptación AES utilizando pipelining. Esta optimización se aplica a las etapas de SubBytes, ShiftRows, MixColumns y AddRoundKye. Las etapas dentro de Key Schedule no pueden ser paralelizadas.

En la Tabla 12 se resumen las dos soluciones creadas para este diseño. La primera sin la aplicación de optimizaciones, y la segunda utilizando pipelining en las etapas de encriptación

como se muestra en la Figura 69. Además, la Solución 2 implementa todos los argumentos de la función de *top-level* como registros individuales, eliminando el tiempo de espera para acceder a los bloques de memoria. Estos argumentos son: i) statemt: bloque de datos a encriptase, ii) key: clave de encriptación, iii) enc: bloque de datos encriptado, iv) dec: bloque de datos des-encriptado.

Tabla 12
Resumen de soluciones para el IP AES

Solución	Directivas de Optimización	Desempeño		Recursos			
		Latencia	Intervalo	BRAM	DSP48	FF	LUT
1	-	1085	1086	30	0	28353	45789
2	Pipelining en función top-level; Array partitioning en argumentos de la función top-level (state, key, enc, dec)	928	929	32	0	35165	53379

Nota: Las latencias, intervalos y recursos son estimados para un Zynq Z-7020 a una frecuencia de operación de 100MHz.

4.4. Backpropagation Neural Network

El cuarto acelerador en hardware creado en esta tesis realiza el entrenamiento de una red neuronal por el método *backpropagation* (retropropagación). Las redes neuronales son ampliamente usadas en técnicas de *machine learning*, las cuales se aplican en tareas de clasificación, reconocimiento de patrones, y teoría de control (Reagen et al., 2014). El entrenamiento de redes neuronales es un proceso computacionalmente costoso, ya que involucra procesos iterativos con un gran número de parámetros a ser calculados. Dentro de los varios métodos desarrollados para el entrenamiento de redes neuronales, uno de métodos más ampliamente usados es el método de *retropropagación*. En esta sección se presentan conceptos acerca de redes neuronales artificiales y se describe el método de *retropropagación* para su entrenamiento.

4.4.1. Algoritmo

4.4.1.1. Redes Neuronales Artificiales (ANN)

Las redes neuronales artificiales buscan emular el comportamiento del cerebro humano. Las redes neuronales biológicas constan de millones de interconexiones entre neuronas, las cuales son activadas dependiendo de impulsos recibidos desde otras neuronas formando así una gran red sensorial, capaz de aprender y razonar (Burns, 2001).

Al igual que en las redes neuronales biológicas, las redes neuronales artificiales tienen como elemento básico una neurona capaz de recibir impulsos y generar una salida a partir de ellos. El modelo básico de una neurona artificial se muestra en la Figura 70. A continuación, se detallan los elementos mostrados en el diagrama.

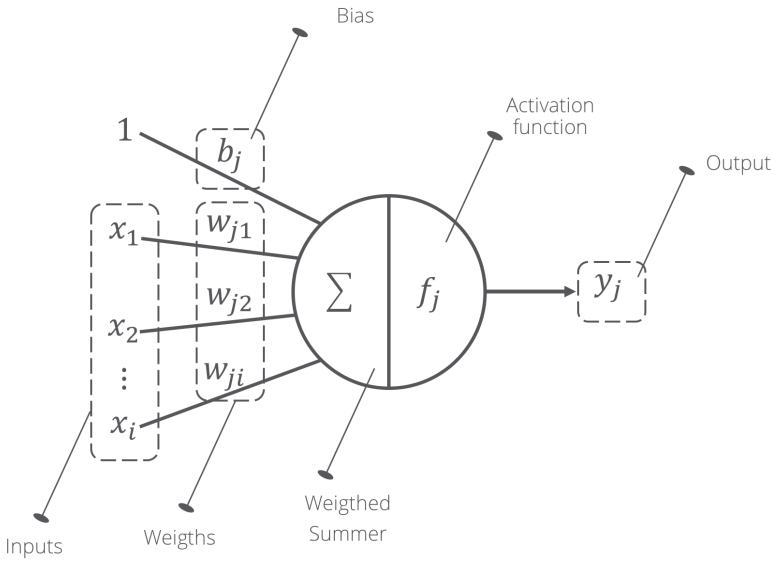


Figura 70. Modelo básico de una neurona artificial. El subíndice j representa a la j-ésima capa de neuronas.

- x_i: representan las entradas a la neurona, ya sean obtenidas del entorno o procedentes de otras neuronas.
- w_{ji}: representan los pesos por los cuales son multiplicadas cada una de las entradas.
- b_j: se denomina bias o sesgo, y permite fijar el punto de activación de la neurona.

- f_j: es la función de activación de la neurona. Existen varios tipos de funciones de activación, entre los más populares se encuentran: escalón unitario, identidad, tangente hiperbólica y sigmoide.

- y_j: es la salida generada por la neurona.

La neurona realiza una suma ponderada en base a las entradas, pesos y bias definida como

$$s_j = \sum_{i=1}^{N} w_{ij} x_i + b_j \tag{4.9}$$

donde N es el número de entradas a la neurona. En base a la Ecuación 4.9, la salida se calcula dependiendo de la función de activación. Para una función Sigmoide, mostrada en la Figura 71, la salida es igual a

$$y_j = f(s_j) = \frac{1}{1 + e^{-s_j}} \tag{4.10}$$

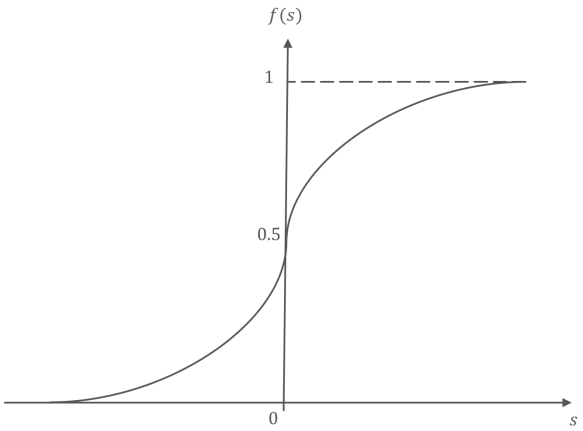


Figura 71. Función sigmoide. Esta función de activación es una de las funciones más usadas en el método de retropropagación gracias a que es una función continua derivable, lo cual es un requisito para este método.

Las redes neuronales están formadas por varios conjuntos de neuronas denominadas capas. Cada capa de neuronas está conectada a otras capas de neuronas por conexiones denominadas

sinápticas. Existen tres tipos de capas, de entrada, escondidas y de salida. En la Figura 72 se muestra una red neuronal artificial de tres capas, una capa de entrada (4 neuronas), una capa escondida (5 neuronas) y una capa de salida (3 neuronas). Es importante observar que no existe una restricción en el límite de capas escondidas o número de neuronas por capa.

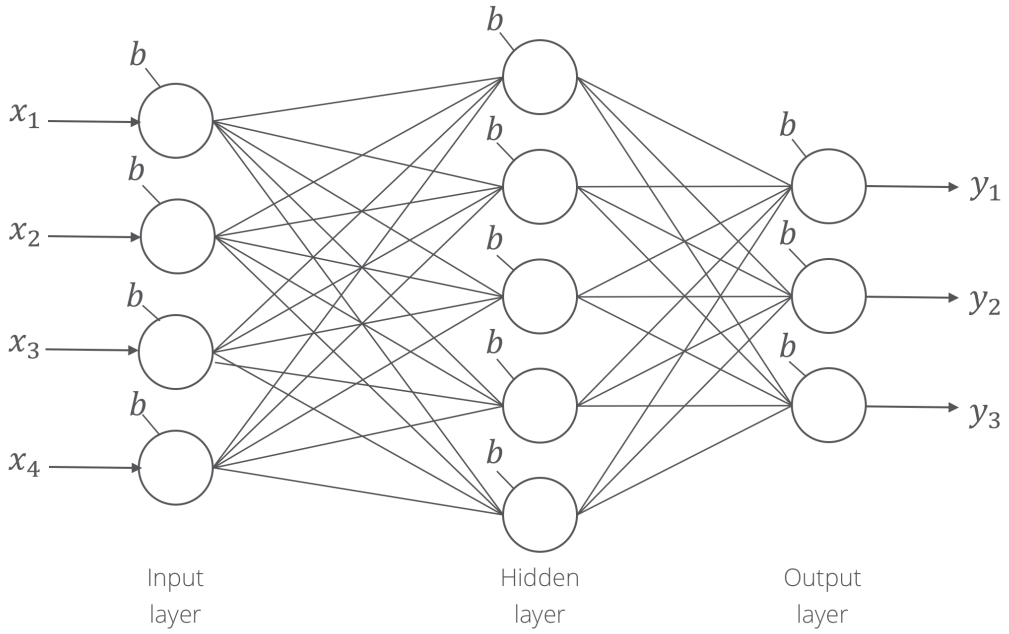


Figura 72. Red neuronal *feedforward* de tres capas. Las entradas a la red neuronal son representadas como x_i, las salidas como y_i y los bias (sesgos) como b.

4.4.1.2. Backpropagation

El proceso de entrenar una red neuronal involucra ajustar sus parámetros para lograr que esta se comporte de una manera deseada. Existen varios métodos para entrenar redes neuronales, siendo uno de los más populares el método de retropropagación. Este método forma parte de los algoritmos de aprendizaje de redes neuronales denominados supervisados. Estos algoritmos utilizan conjuntos de entradas y salidas (*targets*) denominados datos de entrenamiento. En base a estos datos los pesos y bias son ajustados de forma que la red pueda obtener el valor de las salidas deseadas con un mínimo error. Gracias al aprendizaje con los datos de entrenamiento, la red neuronal es capaz de identificar patrones en ejemplos

nunca antes presentados.

El método de retropropagación se basa en el método denominado descenso de gradiente. Este método busca que el error de la red converja a un valor mínimo a través de la actualización de los pesos y bias en base a un incremento proporcional a la pendiente de la función de error. En el algoritmo de retropropagación la función de error es remplazada por la función de coste (Ecuación 4.11), la cual representa una sumatoria de los errores cuadráticos medios de las salidas de la red respecto a las salidas deseadas. En la Figura 73 se representa el método de descenso de gradiente para retropropagación.

$$J = \frac{1}{2}\sum_{j=1}^{M}(d_j - y_j)^2 \tag{4.11}$$

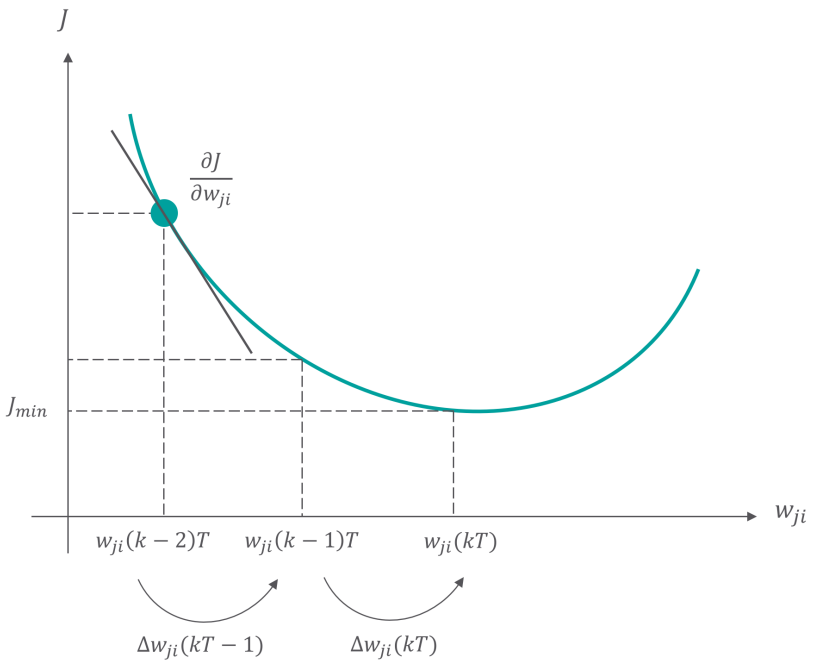


Figura 73. Se calcula la pendiente de la función de coste J respecto a cada peso, en base a ella se calcula un incremento Δw_{ji} que actualiza el valor de w_{ji}, forzando a la función a converger a un valor mínimo.

La derivada de la función de coste determina el valor de actualización de cada peso como

$$\Delta w_{ji}(kT) = -\mu \frac{\partial J}{\partial w_{ji}} \tag{4.12}$$

donde μ es una constante. Una vez determinado el incremento, los pesos se actualizan según la siguiente ecuación

$$w_{ji}(kT) = w_{ji}(k-1)T + \Delta w_{ji}(kT) \tag{4.13}$$

La Ecuación 4.12 puede ser reescrita como

$$\Delta w_{ji}(kT) = \eta \delta_j x_i \tag{4.14}$$

donde η se denomina factor de aprendizaje, y es una valor entre 0 y 1. El valor de δ_j para la capa de salida se define como

$$\delta_j = y_j(1 - y_j)(d_j - y_j) \tag{4.15}$$

Para el resto de capas, δ_j se obtiene como

$$[\delta_j]_\ell = [y_j(1 - y_j)]_\ell \left[\sum_{j=1}^{N} w_{ji}\delta_j \right]_{\ell+1} \tag{4.16}$$

donde ℓ es el índice de la capa, iniciando desde la capa de entrada. Por ejemplo, consideremos la red neuronal de tres capas $N = 3$ de la Figura 72. La capa de entrada será $\ell = 1$, la capa oculta $\ell = 2$ y la capa de salida $\ell = 3$. La actualización de los pesos de la capa de salida $\ell = N$ se la realiza con las Ecuaciones 4.14 y 4.15. Para las capas anterior $\ell = N - 1, \ldots 1$ se utiliza las Ecuaciones 4.14 y 4.16. La demostración matemática completa puede ser consultada en "*Advanced Control Engineering*"(Burns, 2001).

Es importante notar que los errores de la red neuronal tienen influencia sobre la actua-

lización de las capas anteriores, de donde se deriva el nombre de retropropagación. El error final calculado influencia la actualización de los pesos desde la capa de salida hacia las capas anteriores, ajustando cada peso según el nivel de incidencia de sí mismo en el error final.

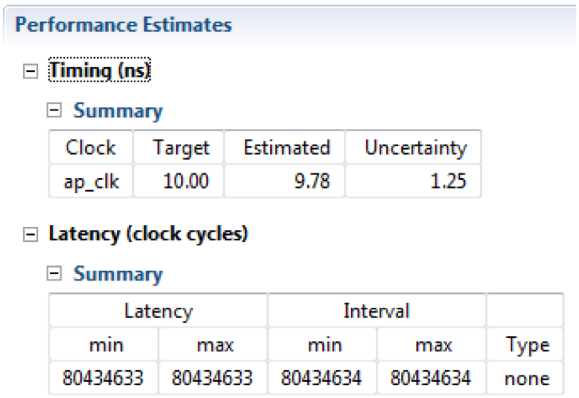
4.4.2. Implementación en Vivado HLS

El presente diseño, retropropagación, es el *benchmark* computacionalmente más intensivo del grupo de algoritmos seleccionado para esta tesis. Un entrenamiento de redes neuronales por retropropagación necesita una gran cantidad de ciclos de reloj para poder completar una ejecución. Esto se debe al gran número de iteraciones necesarias para calcular los pesos correspondientes a cada una de las neuronas que conforman la red neuronal. En la Tabla 6 se resume los parámetros del entrenamiento utilizados en este *benchmark*.

Tabla 13
Parámetros de entrenamiento del benchmark Backpropagation

Dimensión de las entradas	Número de salidas	Sets de entrenamiento	Nodos por capa	Capas	Épocas
10	3	15	10	2	1000

La simulación C y síntesis HLS se realizan sin problemas en este diseño, sin embargo, la co-simulación no se ha podido completar debido a la cantidad de ciclos de reloj a ser simulados. En la Figura 74 se observa el reporte de síntesis con los estimados de desempeño. En este reporte se puede observar que el DUT requiere de aproximadamente ochenta millones de ciclos de reloj en una ejecución, lo cual es extremadamente complejo para una simulación a nivel RTL y requiere demasiado tiempo y recursos computacionales. Es por ello que, en diseños complejos, como este ejemplo, es posible que la co-simulación no se pueda completar, siendo la implementación del IP la única forma de comprobar sus resultados.



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.78	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
80434633	80434633	80434634	80434634	none

Figura 74. Estimación de desempeño para el IP Backprop. Los valores de latencia e intervalo son demasiado altos para completar la co-simulación.

4.4.3. Integración del IP-core en el SoC

La integración del acelerador de entrenamiento por retropropagación en el SoC se la realiza en Vivado IP Integrator. En la Figura 75 se presenta el diagrama de bloques de la integración. En este diagrama se distinguen cuatro IP-cores: **1) Zynq PS:** IP que representa al procesador ARM, cache, controladores y otros componentes que permiten la ejecución de software y su interacción con el PL y periféricos. **2) IP generado en HLS:** acelerador en hardware para realizar el entrenamiento de una red neuronal por el método de retropropagación. **3) AXI Interconnect:** IP para la conexión del acelerador a través de su interfaz AXI *slave* a la interfaz AXI *master* del PS (Sección 4.1.2.3). **4) Processor System Reset:** IP encargado del *reset* del sistema, el cual actúa tanto sobre el AXI Interconnect como sobre los IP-cores conectados a él.

4.4.4. Optimizaciones

Para este ejemplo se han analizado las optimizaciones de desempeño posibles. Sin embargo, directivas como *unrolled*, *pipelinig* aplicadas a nivel de funciones no han podido ser sintetizadas por Vivado HLS debido a la complejidad del diseño. Durante el proceso de síntesis de estas optimizaciones se muestran mensajes de advertencia relacionados con la cantidad

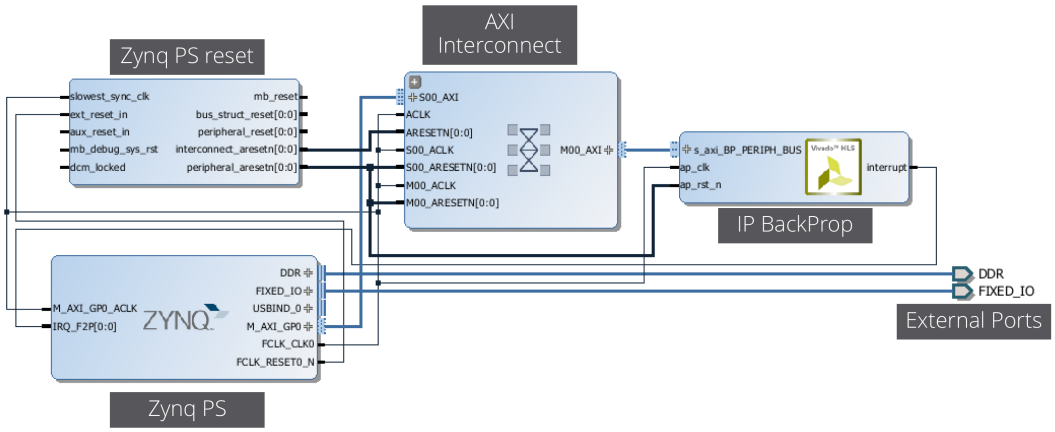


Figura 75. Diagrama de bloques de la integración de IP Backprop en el SoC.

de instrucciones de lectura y escritura a ser analizadas, lo cual se muestran en la Figura 76.

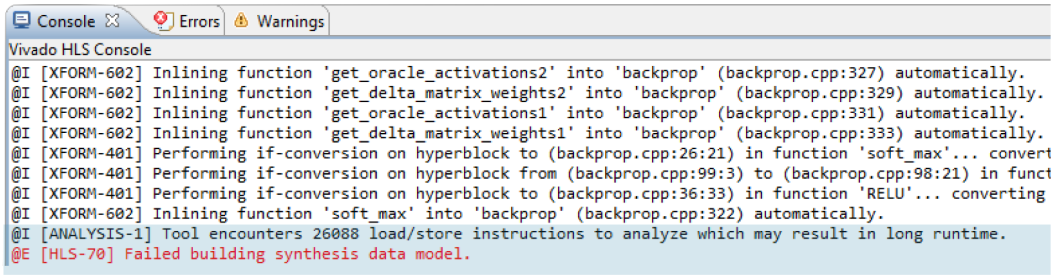


Figura 76. Mensajes de error en la síntesis con optimizaciones pipelining, unrolled para el IP Backprop.

Debido a los problemas antes mencionados, se han evaluado optimizaciones de desempeño a nivel de lazos. La menor latencia de ejecución se logra aplicando pipelining en el cálculo de las funciones de activación y del valor de actualización de los pesos de todas las capas de la red. En la Tabla 14 se presenta el resumen de soluciones para este diseño.

Tabla 14

Resumen de soluciones para el IP Backprop

Solución	Directivas de Optimización	Desempeño		Recursos			
		Latencia	Intervalo	BRAM	DSP48	FF	LUT
1	-	464387001	464387002	19	148	28561	40262
2	Pipelining a nivel de lazos	82890001	82890002	26	168	61546	53345

Nota: Las latencias, intervalos y recursos son estimados para un Zynq Z-7020 a una frecuencia de operación de 100MHz.

4.5. Red Neuronal Artificial (ANN) para Reconocimiento de Números Manuscritos

El quinto acelerador en hardware desarrollado en esta tesis realiza la ejecución de una red neuronal para el reconocimiento de números manuscritos en imágenes de 20×20 pixeles. El entrenamiento de la red neuronal se ha realizado previamente en software para obtener los parámetros de la red. Se ha seleccionado el set de datos "*MNIST Handwritten Digits*" proporcionado por la Universidad de Nueva York (Roweis, 2010). El set consta de cinco mil ejemplos, de los cuales quinientos han sido utilizados para el entrenamiento de la red y cuatro mil quinientos ejemplos para la evaluación de la ejecución del acelerador en hardware. El acelerador en hardware ha obtenido un 92 % de identificaciones correctas.

4.5.1. Algoritmo

El algoritmo para este ejemplo implementa una red *feedforward* de dos capas con función de activación Sigmoide. La teoría acerca de Redes Neuronales Artificiales (ANN) se puede consultar en la Sección 4.4.1. En la Figura 77 se muestra el esquema de la red neuronal implementada en este acelerador. En la figura se observa una imagen con un total de 400 pixeles, donde cada uno de ellos representa una entrada para la red. Estas entradas son conectadas a una primera capa de neuronas, que a su vez se conecta a una capa de salida de 10 neuronas. Cada salida de la red representa el porcentaje de coincidencia de la imagen de entrada con los patrones conocidos para los 10 dígitos. Finalmente, el algoritmo selecciona la salida de la red con mayor porcentaje de coincidencia.

4.5.2. Implementación en Vivado HLS

La implementación de redes neuronales en FPGA puede demandar muchos recursos dependiendo de la complejidad de cada red. En el presente ejemplo, una forma de incrementar

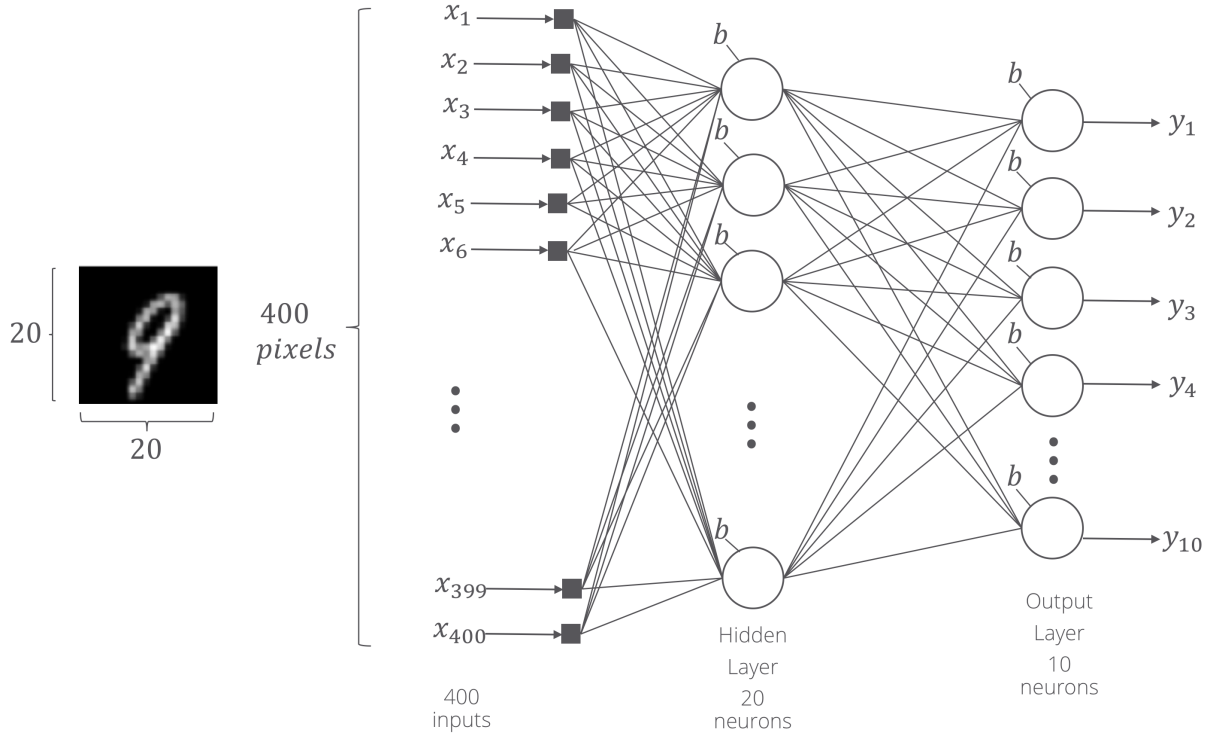


Figura 77. Red neuronal para el reconocimiento de dígitos manuscritos en imágenes de 20×20 pixeles. Cada pixel de la imagen se representa como x_i, las salidas de la red como y_i, y los sesgos como b.
Fuente: (Roweis, 2010)

el porcentaje de detecciones correctas es incrementar el número de neuronas en la capa escondida de la red neuronal. Sin embargo, esto tendría una gran incidencia sobre los recursos utilizados para su implementación. Este incremento de recursos se debe a la gran cantidad de pesos a almacenarse en el IP y el hardware adicional para el cálculo de las sumas ponderadas en cada neurona. Es importante notar que las imágenes utilizas se han limitado a dimensiones de 20×20 pixeles. El uso de imágenes de mayor tamaño incrementaría el tamaño de toda la red neuronal debido a que se necesitarían muchas más neuronas para un reconocimiento efectivo.

4.5.3. Integración del IP-core en el SoC

La integración del acelerador de ANN en el SoC se la realiza en Vivado IP Integrator. En la Figura 78 se presenta el diagrama de bloques de la integración. En este diagrama se distinguen cuatro IP-cores: **1) Zynq PS:** IP que representa al procesador ARM, cache, controladores y otros componentes que permiten la ejecución de software y su interacción con el PL y periféricos. **2) IP generado en HLS:** acelerador en hardware para la ejecución de una red neuronal para el reconocimiento de números manuscritos en imágenes de 20×20 pixeles. **3) AXI Interconnect:** IP para la conexión del acelerador a través de su interfaz AXI slave a la interfaz AXI master del PS (Sección 4.1.2.3). **4) Processor System Reset:** IP encargado del reset del sistema, el cual actúa tanto sobre el AXI Interconnect como sobre los IP-cores conectados a él.

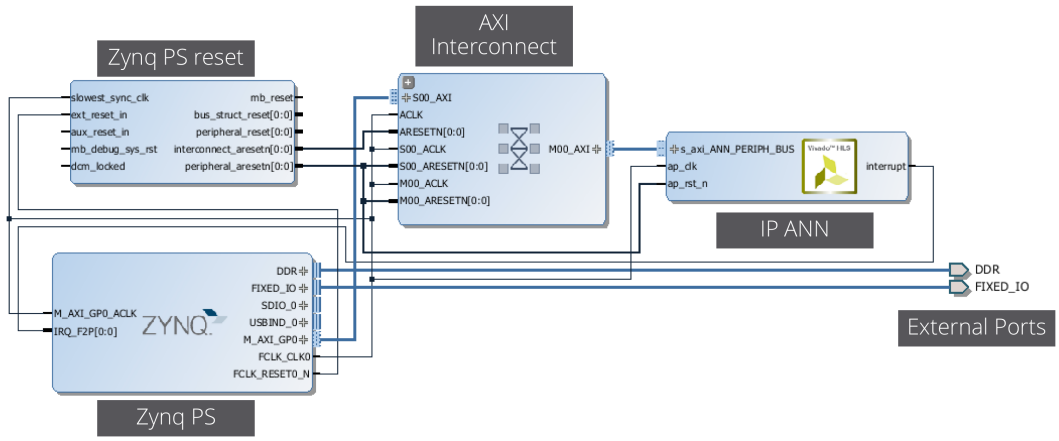


Figura 78. Diagrama de bloques de la integración de IP ANN en el SoC

4.5.4. Optimizaciones

El presente diseño, ANN, es altamente paralelizable. Las operaciones ejecutadas por la red neuronal se han analizado para poder establecer la mejor optimización posible con los recursos disponibles. Las sumas ponderadas de todas las neuronas de una misma capa en una red neuronal pueden ejecutarse en paralelo. La primera capa de esta red tiene 400 entradas. Esto significa que la suma ponderada de cada neurona de la primera capa requiere de 401

sumas y 400 multiplicaciones, como se muestra en la Ecuación 4.17.

$$s_j = \sum_{i=1}^{400} w_{ij} x_i + b_j \tag{4.17}$$

La mejor optimización para este diseño se ha alcanzado usando *pipelining* en el cálculo de la suma pondera de cada neurona, reduciendo de forma considerable la latencia de ejecución. La red neuronal se puede paralelizar aún más realizando *pipelining* a nivel de las capas de neuronas. Sin embargo, para alcanzar esta optimización se requeriría multiplicar en número de recursos utilizados para el cálculo de cada neurona por el número de neuronas en cada capa, lo cual excede por mucho a los recursos de los FPGAs disponibles (Z-7010 y Z-7020). Inclusive, los recursos requeridos excederían la capacidad de la mayoría de FPGAs del mercado. La Tabla 15 muestra un resumen de las soluciones creadas para este diseño.

Tabla 15
Resumen de soluciones para el IP ANN

Solución	Directivas de Optimización	Desempeño		Recursos			
		Latencia	Intervalo	BRAM	DSP48	FF	LUT
1	-	83993	83994	19	34	7017	9966
2	Pipelining parcial en la suma ponderada de las neuronas	32298	32298	18	132	16410	23705

Nota: Las latencias, intervalos y recursos son estimados para un Zynq Z-7020 a una frecuencia de operación de 100MHz.

Capítulo 5

Definición de un Flujo de Diseño Para SoCs e Incremento de Nivel de Automatización

En el presente capítulo se define un flujo de diseño confiable para la creación de SoCs con aceleradores en hardware (IP-cores) desarrollados mediante Vivado HLS. Además, se introducen conceptos básicos acerca del uso de scripts Tcl en las herramientas de Xilinx Vivado HLS y Vivado Design Suite. Finalmente, se demuestra la automatización parcial del flujo de diseño definido en este capítulo mediante la ejecución de scripts Tcl.

5.1. Definición de un Flujo de Diseño Confiable para la Aceleración Mediante IP-Cores en SoCs de Xilinx

Esta sección tiene como objetivo definir un flujo de diseño confiable para el desarrollo de aceleradores en hardware (IP cores) en SoCs de Xilinx, el cual esta basados en la experiencia obtenida en el desarrollo de esta tesis. La Figura 79 muestra los puntos fundamentales en

este flujo de diseño, y su explicación se presenta en el resto de la sección. La demostración de este flujo de diseño se realiza a través de un ejemplo práctico en la Sección 4.1.

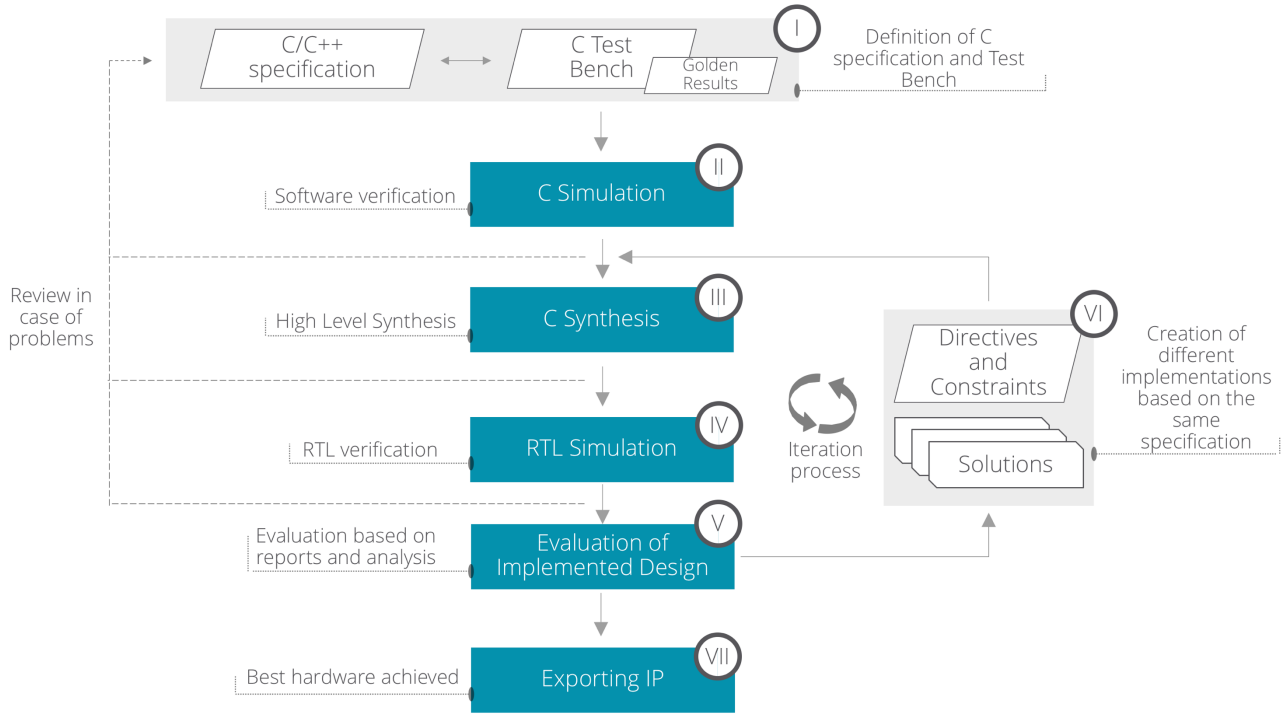


Figura 79. Flujo de diseño confiable para la creación de aceleradores en hardware en Vivado HLS

5.1.1. Definición de Especificación C y Test Bench

Muchas aplicaciones críticas y otras que procesan grandes cantidades de datos, son complejas, o requieren muchos recursos computacionales, pueden encontrar limitaciones de desempeño (e.g., tiempo de ejecución, *throughput*, consumo energético) en procesadores de software. Estas aplicaciones son candidatas a implementarse en sistemas SoCs basados en FPGA de Xilinx, trasladando secciones de sus algoritmos a aceleradores en hardware mediante Vivado HLS. Para ello, los algoritmos o secciones de algoritmos que sean destinados a hardware deberán convertirse en una *especificación* C. Esta especificación se define como una descripción de comportamiento escritas en C/C++, la cual debe incluir una función *top-level* y puede estar compuesta de una jerarquía de sub-funciones dentro de uno o varios archivos.

Una especificación C en Vivado HLS debe acompañarse de un *test bench* C (en un archivo independiente C/C++), el cual sirve para comprobar los resultados de la especificación mediante la comparación con resultados ideales conocidos como *"golden results"*. Estos resultados ideales pueden encontrarse dentro del mismo *test bench* o ser almacenados en un archivo de datos (.dat). La Figura 80 muestra un ejemplo de una especificación C acompañada de un *test bench*. En esta figura se muestran los tres archivos básicos requeridos en Vivado HLS, cuya descripción se presenta a continuación.

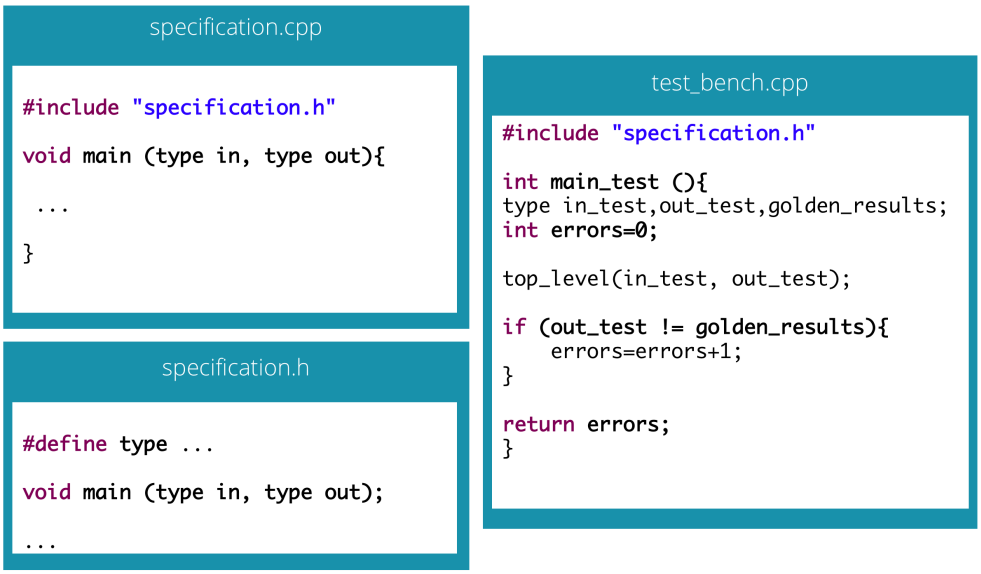


Figura 80. Estructura básica de una especificación C en Vivado HLS. En este ejemplo se incluyen los archivos: i) specification.cpp: archivo C++ que incluye la definición de la función top-level del DUT, ii) specification.h: archivo de cabecera en donde se definen variables y sus tipos, y se declara la función top-level, iii) test_bench.cpp: test bench con una función main independiente que llama a la función top-level del DUT y comprueba sus resultados.

- **specification.cpp**[1]: archivo principal de la especificación C, en el cual se *define* la función *top-level* y las sub-funciones requeridas. Toda función en C/C++ consta de: i) una cabecera de función "`void main(type in, type out)`" donde se definen el tipo de retorno, el nombre y argumentos, y ii) un cuerpo de la función, donde se declaran las

[1]Los nombres de este y los demás archivos mostrados en la Figura 2 han sido seleccionados únicamente para este ejemplo. El diseñador tiene total libertad para seleccionar los nombres que desee.

sentencias a ser ejecutadas. La *definición* de las funciones involucra especificar tanto la cabecera como el cuerpo de las mismas.

- **specification.h:** archivo de cabecera en el cual se *declara* la función *top-level* de la especificación. La *declaración* de una función requiere especificar únicamente la cabecera de la misma. El archivo de cabecera deberá ser incluido tanto en el archivo que contenga la definición de la función *top-level* (specification.cpp) como en el *test bench* (test_bench.cpp).

- **test_bench.cpp:** El *test bench* C realiza una verificación automática de la funcionalidad de la especificación y debe cumplir con: **i)** Tener una función *main* de tipo entero, la cual llame a la función *top-level*. **ii)** Pasar datos a la función *top-level* en forma de valores, arreglos o matrices de prueba, según corresponda, para que sean procesados y se genere un resultado. **iii)** Los resultados obtenidos del DUT deben ser comparados con los *golden results*. **iii)** Si estos resultados coinciden, el *test bench* debe retornar un valor de cero, caso contrario retornará un valor diferente de cero.

La función principal de Vivado HLS es realizar la síntesis de alto-nivel desde la especificación C al hardware del FPGA, mas no ser un entorno de desarrollo de especificaciones C. Por lo tanto, se recomienda que la especificación sea previamente desarrollada y comprobada en entornos de desarrollo de lenguaje C/C++ que utilicen los compiladores estándar ANSI-C (GCC 4.6), C++ (G++ 4.6), los cuales son soportados por Vivado HLS[2] (Xilinx - UG902, 2017). Las especificaciones C previamente verificadas en estos compiladores no debe presentar inconvenientes al diseñador cuando se utilizan en Vivado HLS. En el desarrollo de la tesis se ha utilizado Eclipse IDE para C/C++ versión Helios Service Release 2, el cual incluye el compilador GCC GNU v4.6.

[2]Vivado HLS soporta estos compiladores desde la versión v2013.2 hasta la versión actual v2018.3.

5.1.2. Simulación C

La simulación C es el primer paso a ejecutarse en Vivado HLS. Esta simulación es una verificación funcional a nivel de software previa a la síntesis C. Vivado ejecutará el *test bench* para contrastar los resultados del DUT y *golden results*. En caso de que la simulación C falle, significa que la especificación no representa fielmente el comportamiento del algoritmo original o contiene errores, por lo cual debe ser revisada y modificada.

5.1.3. Síntesis C

Si la simulación C ha sido exitosa, se procede a la síntesis de alto-nivel, denominada en Vivado HLS como síntesis C. Un requisito antes de continuar con el proceso de creación de aceleradores en hardware para SoCs es la definición de interfaces AXI, las cuales permiten posteriormente integrar un IP implementado en el PL de FPGA con el procesador del PS (Sección 4.1.2.3). La síntesis C es realizada de forma automática por Vivado HLS, donde se analiza la especificación C y se completan las etapas del flujo HLS (Sección 2.6.2). Si la síntesis falla, significa que la especificación contiene operaciones o lógica no trasladable al hardware de FPGA utilizando Vivado HLS, por lo cual debe ser revisada nuevamente. La información acerca de constructores C no soportados puede consultarse el documento *"Vivado Design Suite User Guide - High-Level Synthesis"* (Xilinx - UG902, 2017). El proceso de simulación C se explica mediante un ejemplo en la Sección 4.1.2.4.

5.1.4. Co-simulación C/RTL

Una vez que el diseño ha sido sintetizado exitosamente se procede a realizar una verificación funcional de la implementación a nivel RTL. Esta etapa es tan importante como la síntesis C, ya que permite verificar que el hardware creado puede obtener resultados idénticos a los obtenidos en software. Si la co-simulación falla se puede deber a cuatro principales razones: 1) existe una configuración incorrecta del entorno de simulación, 2) la precisión de los

resultados de hardware difiere de la precisión en software, 3) el análisis de la especificación C no ha inferido la lógica correcta para replicar los resultados, o 4) se han aplicado directivas de optimización de forma incorrecta (Xilinx, 2014). Para resolver estos problemas es necesario referirse a la guía *"Vivado HLS: Debug Guide for investigating C/RTL co-simulation issues"* (Xilinx, 2014). El proceso de co-simulación se explica mediante un ejemplo en la Sección 4.1.2.5.

5.1.5. Evaluación del Diseño Implementado

Una vez que la funcionalidad de la implementación en hardware ha sido comprobada, aun es necesario evaluarla con respecto a: **i)** indicadores de desempeño, como latencia e intervalos de iniciación, y **ii)** indicadores de uso de recursos, como la cantidad de LUTs, FFs, Block RAMs, DSPs utilizados. Estos indicadores pueden ser consultados en el reporte de síntesis de Vivado HLS. Además, Vivado dispone de una perspectiva de análisis donde se representa la programación o *scheduling* de las operaciones en cada ciclo de reloj. Tanto el reporte de síntesis como la perspectiva de análisis tienen como objetivo facilitar al diseñador identificar las optimizaciones en base a las cuales se realiza la aplicación de directivas.

5.1.6. Aplicación de Directivas de Optimización

En base al análisis previo se aplicarán las directivas[3] enfocadas a optimizar aspectos puntuales, acorde a los objetivos de diseño, como: **i)** desempeño, **ii)** latencia y **iii)** recursos utilizados. Este es un proceso iterativo, en donde se crean diversas implementaciones, llamadas *soluciones*, de una misma especificación. Cada nueva *solución* deberá ser sintetizada, verificada a nivel RTL y evaluada, como se muestra en el flujo de diseño de la Figura 1. La evaluación de una implementación y la aplicación de optimizaciones se explica mediante un ejemplo en la Sección 4.1.5.

[3]Las directivas son el medio por el cual el diseñador puede influenciar la forma en la que la síntesis C implementa la especificación.

5.1.7. Exportación del IP

En esta etapa se exporta el IP creado usando Vivado HLS, esto significa empaquetar (*package*) el IP en el formato IP-XACT[4] para poder importarlo en Vivado IP Catalog. Es importante que antes de exportar el IP final se obtenga una solución que cumpla con las métricas de diseño requeridas, es decir, la implementación cumple con el desempeño, latencia y recursos que satisfagan al diseñador. Esta recomendación es importante debido a que ciertas optimizaciones pueden cambiar la forma en la que se implementan los argumentos de la función *top-level*. Si la implementación de estos argumentos varía, será necesario cambiar la forma en la que los datos son enviados desde el procesador al IP, lo cual involucra a su vez cambiar parcialmente el software del SoC para diferentes soluciones.

5.1.8. Integración del IP-core en el SoC

La integración de un IP creado en Vivado HLS en un SoC se realiza en Vivado IP Integrator. Las interfaces AXI previamente definidas facilitan su conexión al PS del SoC, ya que son soportadas por el asistente de conexión automática. Finalmente, la plataforma de hardware es exportada a Xilinx SDK. El proceso de integración del IP en el SoC puede ser consultado en la Sección 4.1.3.

5.1.9. Creación de Software

Xilinx SDK importará la plataforma de hardware (hdf.) e incluirá todos los drivers necesarios para manejar el hardware del SoC, incluyendo los IP-cores creados en Vivado HLS. Los conceptos acerca de la creación de software embebido para un SoC en Xilinx SDK se explican en la Sección 4.1.4.

[4]Especificación para la documentación de IP-cores ampliamente adoptada y usada en el catálogo de IP-cores de Xilinx.

5.2. Generación de Sistemas Embebidos SoCs Basados en FPGA Mediante Scripts Tcl

Esta sección se enfoca en explicar conceptos acerca del uso de scripts Tcl para la automatización de procesos de diseño de sistemas embebidos SoCs de Xilinx. En la Sección 5.2.1 se presenta una breve introducción al uso de *scripts* Tcl en herramientas de Xilinx. En la Sección 5.2.2 se tratan conceptos acerca de la automatización en la generación de aceleradores en hardware (IP-cores) usando Vivado HLS, y en la Sección 5.2.3 se trata la automatización en la generación de hardware para SoCs usando Vivado Design Suite.

5.2.1. Lenguaje Tcl en Herramientas de Xilinx

Tcl es un lenguaje de programación con comandos, variables y estructuras de control utilizado como interfaz en muchas herramientas de diseño. Mayor información puede ser consultada en la página oficial de Tcl Core Team (Tcl Developer Xchange, 2018). Xilinx ha adoptado el lenguaje Tcl como extensión en sus aplicaciones, incluyendo Vivado Design Suite, Vivado HLS y SDK. Estas aplicaciones cuentan con un soporte completo para la ejecución de *scripts*, los cuales pueden acceder al control de la aplicación, propiedades y procesos de diseño (Xilinx - UG894, 2016).

El lenguaje Tcl, como cualquier otro, tiene un conjunto de comandos nativos predefinidos, los cuales son ejecutados por un intérprete. Xilinx ha dotado a sus herramientas de un intérprete propio, capaz de ejecutar tanto comandos nativos como comandos específicos para la ejecución de procesos propios de Xilinx. Una mayor información acerca de estos comandos puede ser consultado en el documento *"Vivado Design Suite Tcl Command Reference Guide"* (Xilinx - UG835, 2018).

Una de las ventajas de utilizar comandos Tcl es la posibilidad de ejecutar scripts. Estos scripts son archivos de texto plano con una serie de comandos Tcl que realizan una tarea o conjunto de tareas. La función principal de estos *scripts* en las herramientas de Xilinx es automatizar el proceso de diseño. Así, se puede consultar informes de análisis, aplicar restricciones, crear diseños y verificar el comportamiento de los mismos sin volver a ejecutar los pasos realizados en las interfaces GUI de las aplicaciones (Xilinx - UG835, 2018).

5.2.2. Scripts Tcl en Vivado HLS

El proceso de diseño de aceleradores en hardware usando Vivado HLS puede realizarse mediante un script Tcl. A continuación, se presenta un resumen de los comandos que pueden ser utilizados para replicar el primer ejemplo del Capítulo 4, *matrixmul*. Cabe destacar que los comandos discutidos a continuación pueden ser utilizados para replicar diseños anteriores o para crear diseños completamente nuevos.

5.2.2.1. Creación Automatizada de un IP en Vivado HLS Usando Comandos Tcl

La siguiente explicación asume que los archivos de la especificación C, *matrixmul*[5], se han colocado en un mismo directorio junto al script Tcl. Este script se ha dividido en: (1) creación de un nuevo proyecto, (2) creación de una solución, y (3) ejecución de simulación C, síntesis C, co-simulación y exportación del IP. Cada una de estas partes se explica en el resto de esta sección. El script utiliza un archivo Tcl independiente, el cual contiene las directivas de optimización de desempeño. En la Figura 81 se muestran los archivos necesarios para este ejemplo.

[5]La especificación C matrixmul cuenta con los archivos matrixmul.cpp, matrixmul.h y el test bench matrixmul_test.cpp.

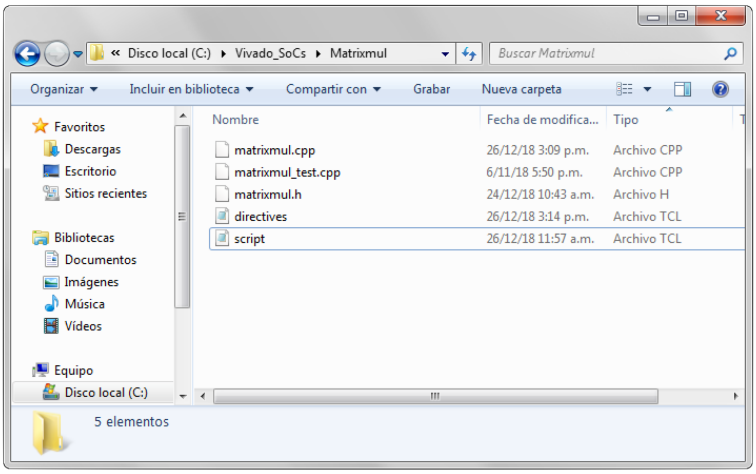


Figura 81. Archivos para la creación automatizada de un IP-core utilizando Vivado HLS para el ejemplo de la Sección 5.2.2.1. Para este ejemplo se incluyen: i) los archivos de la especificación C: "matrixmul.cpp", "matrixmul_test.cpp", "matrixmul.h", ii) el script "script.tcl" que contiene los comandos para la creación del IP-core, y iii) el archivo Tcl "directives.tcl" que incluye las directivas de optimización para la especificación C.

Creación de un nuevo proyecto

En el Código 13 se realiza la creación de un nuevo proyecto en Vivado HLS. En este código se define el nombre del proyecto y se agregan los archivos correspondientes a la especificación C y el *test bench*.

Código 13
Creación de un nuevo proyecto en Vivado HLS mediante comandos Tcl

```
1  open_project -reset Matrixmul_HLS_IP
2  set_top matrixmul
3  add_files matrixmul.h
4  add_files matrixmul.cpp
5  add_files -tb matrixmul_test.cpp
```

- "`open_project {reset`" crea un nuevo proyecto con el nombre "`Matrixmul_HLS_IP`" y restaura el proyecto borrando todos los archivos previos cada vez que se ejecute el Script.

- "`set_top`" indica el nombre de la función *top-level* de la especificación C (`matrixmul`).

- "`add_files`" agrega los archivos que formen parte de la especificación C (`matrixmul.cpp`, `matrixmul.h`).

- "`add_files –tb`" agrega los archivos que formen parte del *test bench* C (`matrixmul_test.cpp`).

Creación de una solución

Una vez el proyecto ha sido creado se procede a generar una nueva solución mediante los comandos mostrados en el Código 14. En este código se define el nombre de la solución y las restricciones para la misma.

Código 14
Creación de una solución en Vivado HLS mediante comandos Tcl

```
1 open_solution -reset "solution1"
2 set_part {xc7z010clg400-1}
3 create_clock -period 10 -name default
4 source "./directives.tcl"
```

- "`open_solution –reset`" crea una nueva solución con nombre "`solution1`".

- "`set_part`" selecciona el dispositivo de destino, en este caso un Zynq Z-7010 con el código de identificación `xc7z010clg400-1`. En la Tabla 16 se resume el código de los dispositivos Xilinx Zynq de tres de las tarjetas de desarrollo más usadas.

Tabla 16
Tarjetas de desarrollo con Xilinx Zynq más usadas

Tarjeta de desarrollo	Dispositivo	Identificador
Zynq-7000 SoC ZC702 Evaluation Kit	Zynq Z-7020	xc7z020clg484-1
ZedBoard	Zynq Z-7020	xc7z020clg484-1
ZYBO	Zynq Z-7010	xc7z010clg400-1

- "`create clock –period`" define el periodo de reloj en nanosegundos, en este caso 10 nanosegundos.

- "`source`" llama a otro script en el directorio especificado. El archivo `directives.tcl` contiene las directivas aplicadas a esta solución, las cuales se muestran en el Código 15.

La explicación de las directivas utilizadas en este ejemplo puede ser consultada en la Sección 4.1.5.

Código 15
Aplicación de directivas de optimización en Vivado HLS mediante comandos Tcl

```
set_directive_pipeline "matrixmul"
set_directive_array_partition -type complete -dim 1 "matrixmul" a
set_directive_array_partition -type complete -dim 1 "matrixmul" b
set_directive_array_partition -type complete -dim 2 "matrixmul" a
set_directive_array_partition -type complete -dim 2 "matrixmul" b
set_directive_array_partition -type complete -dim 1 "matrixmul" res
set_directive_array_partition -type complete -dim 2 "matrixmul" res
set_directive_loop_flatten "matrixmul/Col"
```

Ejecución de simulación C, síntesis C, co-simulación y exportación del IP

Finalmente se ejecutan las etapas de verificación, síntesis y exportación del IP generado. Para ello, se utilizan los comandos mostrados en el Código 16.

Código 16
Ejecución de simulación C síntesis C co-simulación y exportación del IP en Vivado HLS mediante comandos Tcl

```
csim_design
csynth_design
cosim_design
export_design -format ip_catalog
```

- "`csim_design`" ejecuta la simulación C del DUT
- "`csynth_design`" ejecuta la síntesis C
- "`cosim_design`" ejecuta la co-simulación C/RTL
- "`export_design –format ip_catalog`" exporta el IP en formato IP-XACT.

El script debe ser ejecutado en la consola Tcl de Vivado HLS, en la cual se utiliza el comando "`cd`" (*change directory*) para seleccionar el directorio donde se guardaron previamente los archivos. Una vez seleccionado el directorio se ejecuta el script mediante el comando

"`vivado_hls -f script.tcl`"[6]. Cuando la ejecución del script finalice, se creará una carpeta del proyecto dentro del mismo directorio del script. Esta carpeta contendrá a su vez subcarpetas correspondientes a los procesos de simulación C "*sim*", síntesis C "*syn*", cosimulación C/RTL "*csim*" e implementación "*impl*", la cual almacena el IP exportado. En la Figura 82 se muestra la carpeta de la solución creada en este ejemplo.

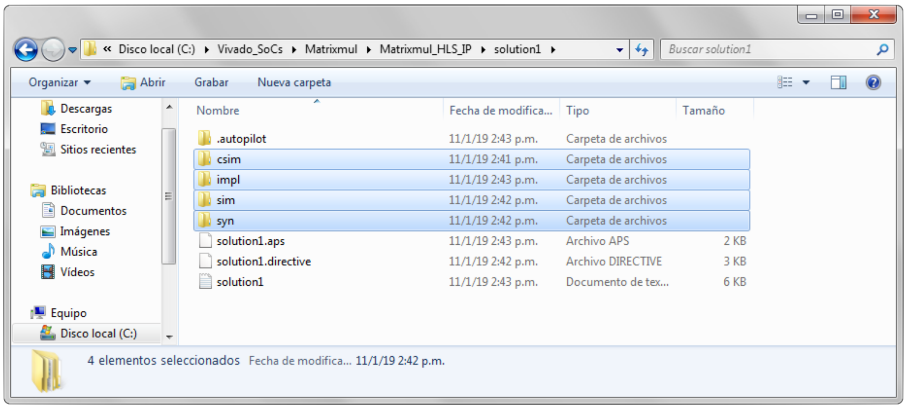


Figura 82. Se resaltan las subcarpetas correspondientes a los procesos de: i) co-simulación C/RTL "csim", ii) implementación "impl", iii) implementación "impl" y iv) síntesis C "syn".

Para consultar los reportes de la síntesis C sin usar la interfaz GUI debe dirigirse a la subcarpeta de la síntesis C, "syn", en la cual se almacena el reporte de síntesis en un archivo con extensión RPT, el cual se muestra en la Figura 83. Para el presente ejemplo la ruta del archivo de reporte es: "/Matrixmul_HLS_IP/solution1/syn/report/matrixmul_csynth.rpt".

5.2.3. Scripts Tcl en Vivado Design Suite

El proceso de generación de hardware del SoC en Vivado Design Suite puede ser altamente automatizado mediante un Script Tcl. A continuación, se presentan los comandos utilizados para replicar el ejemplo del Capítulo 4, *matrixmul*. En esta sección se asume que se ha creado previamente el IP *matrixmul* en Vivado HLS con el procedimiento mostrado en la Sección

[6]El nombre del script utilizado en este ejemplo es "script.tcl", y debe ser remplazado por el nombre que le asigne el diseñador.

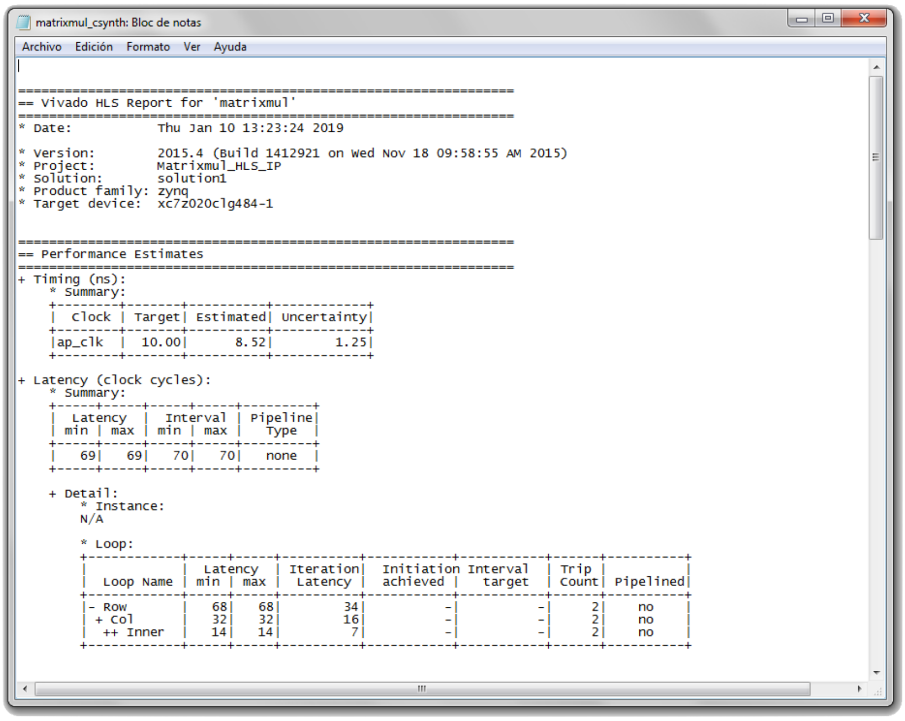


Figura 83. Reporte de síntesis RPT para el ejemplo matrixmul consultado utilizando un lector de texto plano.

5.2.2.

5.2.3.1. Creación Automatizada de un SoC en Vivado Design Suite Usando Comandos Tcl

Para la creación de un nuevo proyecto mediante Tcl en Vivado Desing Suite se debe ubicar el script Tcl en el mismo directorio donde fue creado el IP en Vivado HLS. En esta sección se asume que el directorio donde se encuentran los archivos es "C:/Vivado_SoCs/Matrixmul". En la Figura 84 se muestran los archivos necesarios para este ejemplo. El contenido del script se ha dividido en: (1) creación de un nuevo proyecto, (2) importación de IP creado usando Vivado HLS en catálogo de IP-cores, y (3) creación de un nuevo diseño de bloques. Cada una de estas partes se explica a continuación.

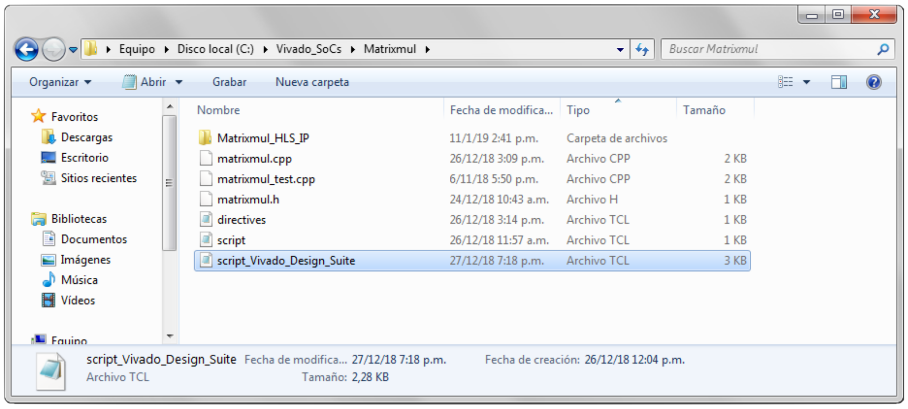


Figura 84. Archivos para la creación automatizada de un SoC integrando un IP-core generado en Vivado HLS. Para este ejemplo se incluye el script denominado "script_Vivado_Design_Suite.tcl".

Creación de un nuevo proyecto

En el Código 17 se muestra los comandos para la creación de un nuevo proyecto en Vivado Design Suite. En este código se especifica el nombre del proyecto, el dispositivo y la tarjeta de desarrollo de destino.

Código 17
Creación de un nuevo proyecto en Vivado Design Suite mediante comandos Tcl

```
1 create_project Matrixmul_SoC C:/Vivado_SoCs/Matrixmul/Matrixmul_SoC -
     part xc7z010clg400-1
2 set_property board_part digilentinc.com:zybo:part0:1.0 [current_project
     ]
```

- "`Create_project`" crea un nuevo proyecto denominado "`Matrixmul_SoC`" en el directorio especificado. El proyecto tiene como objetivo el dispositivo Zynq Z-7010 con el código de identificación `xc7z010clg400-1`.

- "`set_property board_part`" define la tarjeta de desarrollo de la cual forma parte el dispositivo antes seleccionado. En este caso la tarjeta de desarrollo seleccionada es una Zybo del fabricante Digilent. Se usa el comando "`current_project`", el cual devuelve el directorio del proyecto actual, y se usa para evitar reescribir este directorio nuevamente (Xilinx - UG894, 2016).

Importación de IP en catálogo de IP-cores

Una vez el proyecto ha sido creado se procede a importar el IP generado en HLS al catálogo de IP-cores de Vivado Design Suite como se muestra en el Código 18.

Código 18
Importación de IP en catálogo de IP-cores de Vivado Design Suite mediante comandos Tcl

```
1 set_property ip_repo_paths C:/Vivado_SoCs/Matrixmul [current_project]
2 update_ip_catalog
```

- "`set_property ip_repo_paths`" importa los IP-cores ubicados en el directorio especificado al proyecto actual "`current_project`".
- "`update_ip_catalog`" actualiza el catálogo de IP-cores de Vivado Design Suite.

Creación de un nuevo diseño de bloques

Finalmente se crea un diseño de bloques en Vivado IP Integrator en donde se integra el IP-core importado previamente a el Zynq PS como se detalla en el Código 19.

Código 19
Creación de un nuevo diseño de bloques en Vivado Design Suite mediante comandos Tcl

```
1 create_bd_design "Zynq_design"
2 create_bd_cell -type ip -vlnv xilinx.com:ip:processing_system7:5.5
     processing_system7_0
3 apply_bd_automation -rule xilinx.com:bd_rule:processing_system7 -config
      {make_external "FIXED_IO, DDR" apply_board_preset "1" Master "
     Disable" Slave "Disable" }  [get_bd_cells processing_system7_0]
4 create_bd_cell -type ip -vlnv xilinx.com:hls:matrixmul:1.0 matrixmul_0
5 apply_bd_automation -rule xilinx.com:bd_rule:axi4 -config {Master "/
     processing_system7_0/M_AXI_GP0" Clk "Auto" }  [get_bd_intf_pins
     matrixmul_0/s_axi_MATRIXMUL_PERIPH_BUS]
```

- "`create_bd_design`" crea un nuevo diseño de bloques con el nombre "`Zynq_design`".
- "`create_bd_cell –type ip`" agrega el IP del catálogo de Vivado Design Suite especificado al diseño de bloques. En el Código 19 se utiliza dos veces este comando, primero

se importa el IP-core Zynq PS y segundo el IP-core *matrixmul*. Para agregar un IP diferente creado en Vivado HLS, debe remplazar el nombre "*matrixmul*", que aparece dos veces en este comando, por el nombre del nuevo IP[7].

- "`apply_bd_automation -rule xilinx.com:bd_rule:processing_system7`" utiliza el asistente de configuración automática del PS. Este asistente aplica la configuración del dispositivo Zynq de la tarjeta de desarrollo seleccionada al IP-core Zynq PS.

- "`apply_bd_automation -rule xilinx.com:bd_rule:axi4`" utiliza el asistente de conexión automática de AXI, conectando la interfaz AXI slave del IP con la interfaz AXI master del PS. Para conectar un IP diferente, se debe remplazar el nombre "matrixmul" por el nombre de nuevo IP, y "`MATRIXMUL_PERIPH_BUS`" por el nombre del *bundle* AXI de la interfaz AXI slave del nuevo IP.

El script completo, incluyendo todas las partes descritas en esta sección, debe ser ejecutado en la consola Tcl de Vivado Design Suite. Para ello, el diseñador debe ubicarse en el directorio del proyecto mediante el comando "`cd C:/Vivado_SoCs/Matrixmul`". Una vez el directorio haya sido seleccionado se ejecuta el script con el comando "`source script_Vivado_Design_Suite.tcl`"[8]. En la Figura 85 se muestra el diseño de bloques del SoC generado mediante la ejecución del script de este ejemplo.

[7]Los IP-cores creados en Vivado HLS toman el nombre de la función top-level del DUT.

[8]El nombre del script utilizado en este ejemplo es "script_Vivado_Design_Suite.tcl", y debe ser remplazado por el nombre que le asigne el diseñador.

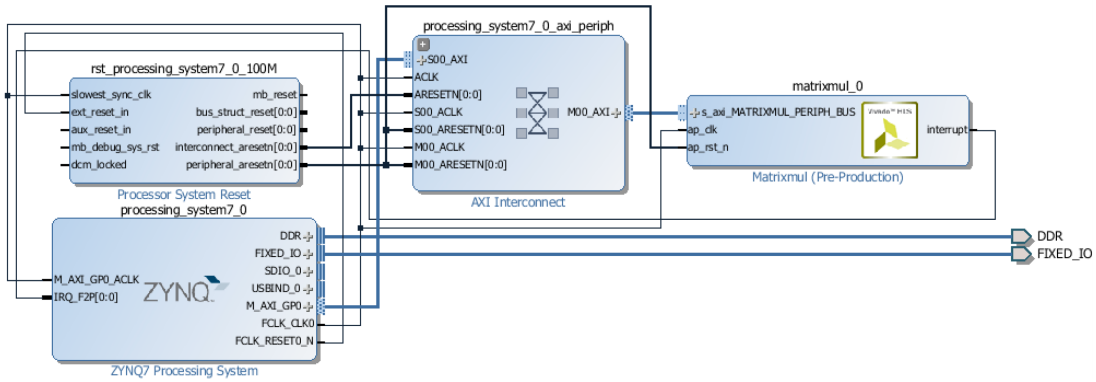


Figura 85. Arquitectura del SoC generado mediante el script Tcl del ejemplo de la Sección 5.2.3

5.3. Demostración y Verificación de un Flujo de Diseño Automatizado para la Creación de SoCs

En esta sección se demuestra el funcionamiento de un flujo de diseño automatizado para la creación de SoCs con aceleradores en hardware basado en el flujo planteado en la Sección 5.1. La automatización de este flujo se basa en la generación de *scripts* Tcl para las herramientas de diseño de Xilinx Vivado HLS y Vivado Design Suite. Este flujo toma como entrada una especificación C, en base a la cual se crea un IP en HLS y se integra en un SoC. El proceso se realiza de forma automática y finaliza con la exportación de la plataforma de hardware del SoC (hdf). El flujo de diseño automatizado se muestra en la Figura 86.

Para demostrar este flujo de diseño, se ha creado un script Tcl denominado IP_SoC_generator. Este script permite incrementar el nivel de automatización en la generación de IP-cores usando Vivado HLS e integrarlos a SoCs usando Vivado Design Suite. La explicación de la funcionalidad de este script y sus limitaciones se realiza en las siguientes secciones de este capítulo.

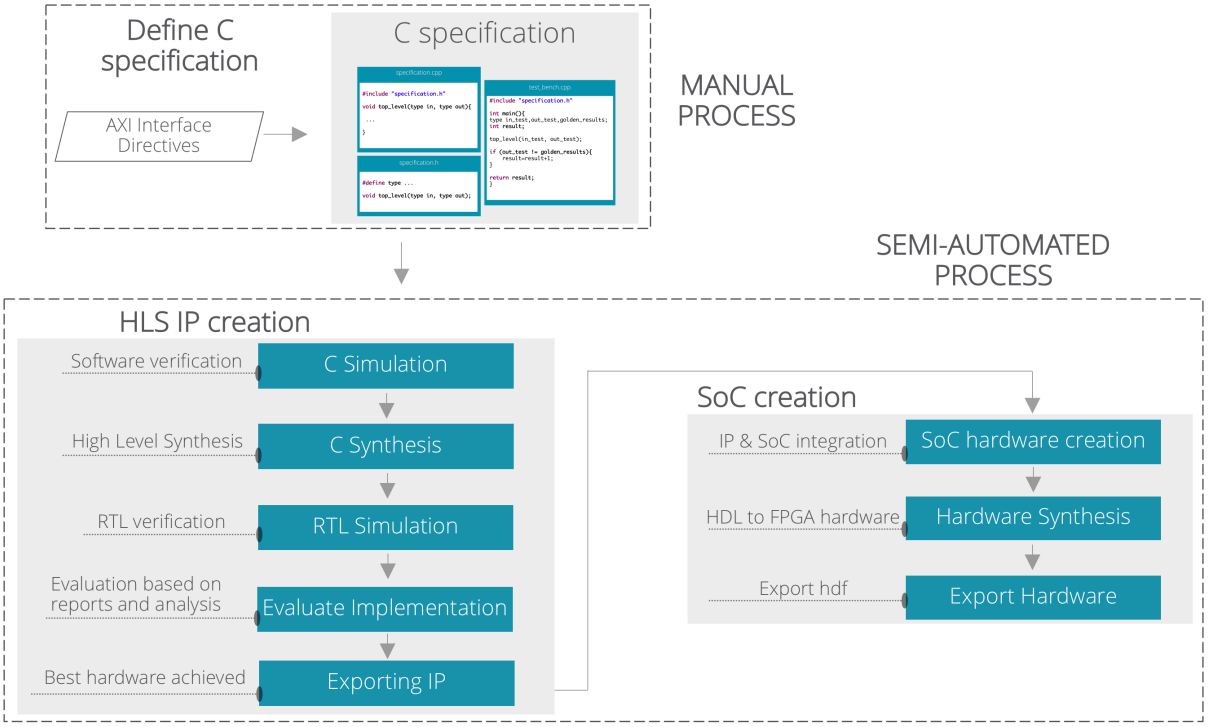


Figura 86. Flujo de diseño automatizado usando scripts Tcl para la creación de SoCs con aceleradores de hardware

5.3.1. IP SoC Generator

IP_SoC_generator tiene dos funciones principales: i) Replicar diseños de referencia: se puede replicar tanto los IP-cores en HLS como los SoCs generados en el Capítulo 4. Estos son (1) Matrixmul: multiplicador de matrices, (2) FFT: transformada rápida de Fourier de 1024 puntos, (3) AES: encriptación y des-encriptación AES de bloques de datos y claves de 128 bits, (4) Backprop: entrenamiento por retro-propagación de una red neuronal, (5) ANN: red neuronal artificial para el reconocimiento de números manuscritos en imágenes de 20x20 pixeles. ii) Crear nuevos IP-cores y SoC: crear nuevos IP-cores en Vivado HLS y SoCs basados en Zynq en Vivado Design Suite de forma automática en base a una especificación C (lenguaje C/C++) proporcionada por el diseñador. El uso de IP_SoC_generator tiene limitaciones, las cuales son explicadas en la Sección 5.3.6.

IP_SoC_generator se basa en la generación de scripts que automaticen los procesos de di-

seño en las herramientas de Xilinx. Las siguientes secciones de este capítulo tienen como objetivo demostrar el funcionamiento de este flujo de diseño automatizado a través de tutoriales prácticos. Para ello, se proporciona una carpeta de archivos denominada "IP_SoC_generator" que contienen las especificaciones en C++ de cada diseño y el script IP_SoC_generator.tcl, como se muestra en la Figura 87. La carpeta principal puede ser colocada en un directorio indistinto de un PC Windows donde se encuentre instalada una distribución de las herramientas de desarrollo de Xilinx[9]. En el resto de la sección se asume que el directorio de los archivos es "C:/IP_SoC_generator" para facilitar la explicación.

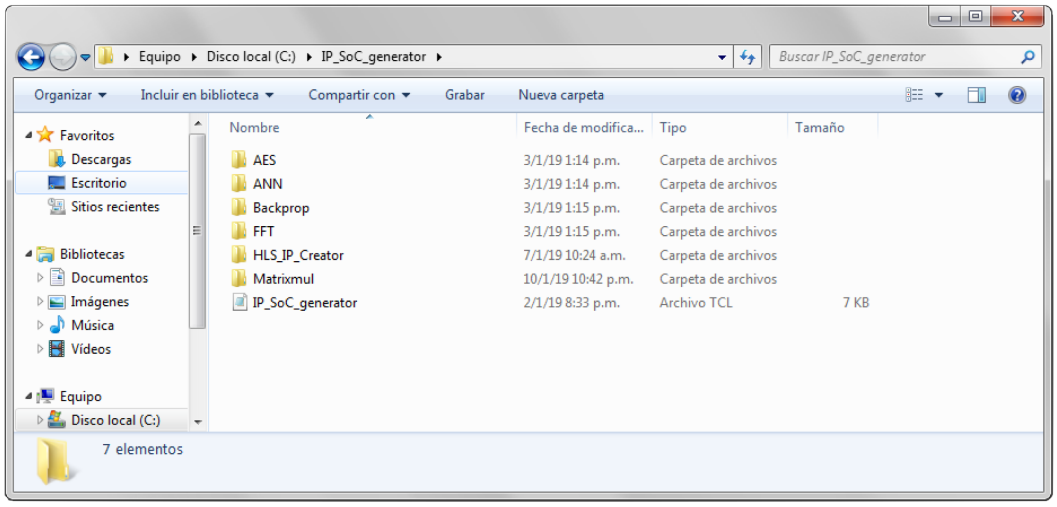


Figura 87. Archivos de IP_SoC_generator

5.3.2. Creación Automática de IP-cores de Referencia Mediante IP SoC Generator

El script IP_SoC_generator.tcl debe ser ejecutado en Vivado HLS Command Prompt, mostrado en la Figura 88. Por defecto el Command Prompt toma como ruta inicial el directorio de instalación de Vivado HLS. Para cambiar al directorio donde se colocaron los archivos de IP_SoC_generator se debe ejecutar el comando "cd C:/IP_SoC_generator".

[9]IP_SoC_generator ha sido probado en las distribuciones de Vivado 2015.4 y 2017.4 instaladas en Windows 7 Service Pack 1 ejecutándose en una máquina virtual en Mac OS X.

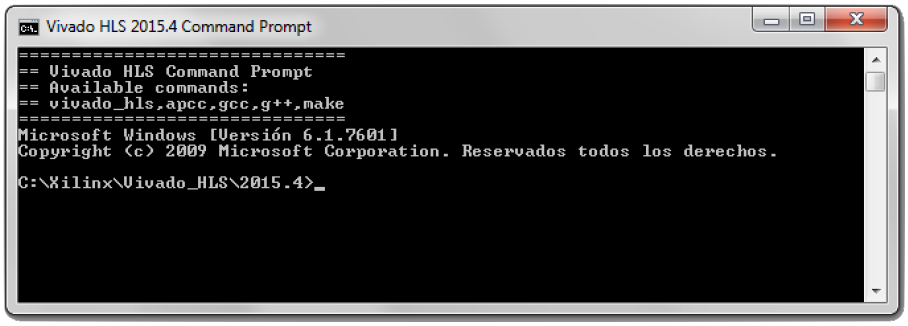


Figura 88. Interfaz Vivado HLS Command Prompt

Una vez en el directorio de IP_SoC_generator, se procede a ejecutar el script Tcl mediante el comando "vivado_hls -f IP_SoC_generator.tcl". El command prompt mostrará las opciones: 1) Matrixmul, 2) FFT, 3) AES, 4) Backprop, 5) ANN, 6) New IP[10] como se observa en la Figura 89. Una de estas opciones debe ser seleccionada mediante teclado. Como ejemplo, se selecciona la opción 1) Matrixmul.

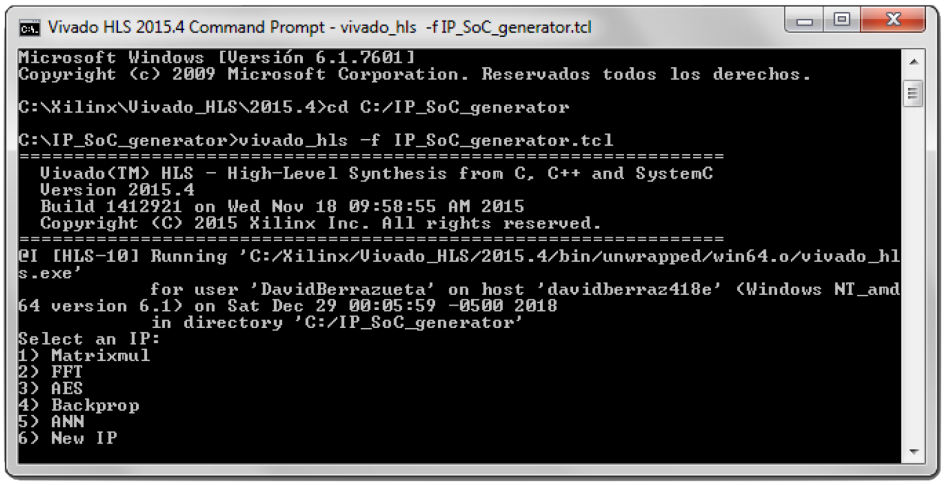


Figura 89. IP_SoC_generator script ejecutandose en consola

A continuación, el programa preguntará al usuario para que dispositivo de destino se crea el proyecto. El programa mostrará tres opciones de las tarjetas de desarrollo Zynq más utilizadas, 1) Zynq-7000 SoC ZC702 Evaluation Kit, 2) ZedBoard, 3) Zybo y una cuarta opción

[10]Revisar la Sección 5.3.4 "Creación Automática de nuevos IP-cores mediante IP_SoC_generator" para mayor información de esta opción.

que permite ingresar el identificador de otro dispositivo Zynq (ver Figura 90). En este caso se selecciona 1) Zynq-7000 SoC ZC702 Evaluation Kit. A continuación, IP_SoC_generator realizará de forma automática la simulación C, síntesis C, co-simulación C/RTL y la exportación del IP seleccionado.

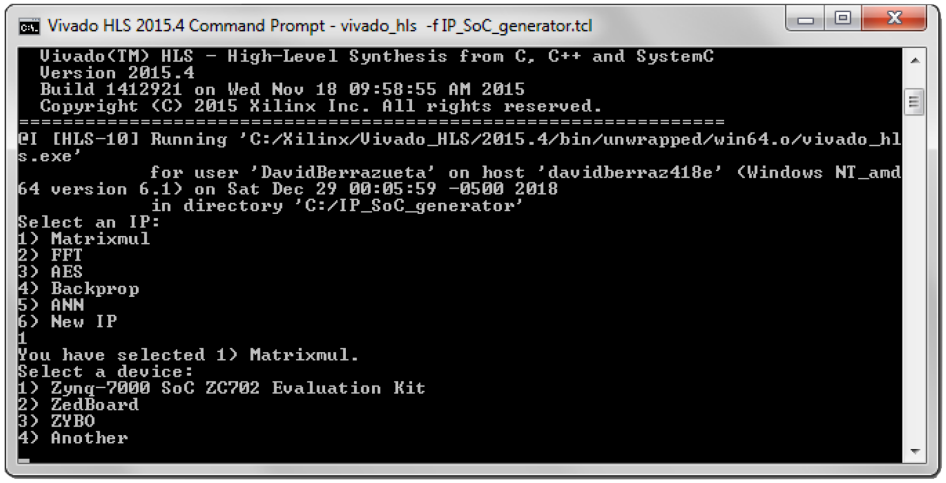


Figura 90. IP_SoC_generator selección de dispositivo

Una vez que el script finalice su ejecución se podrá observar que en la carpeta "C:/IP_SoC_generator/Matrixmul" se han creado nuevos archivos, como se muestra en la Figura 91. La función de estos archivos se explica a continuación: i) la carpeta "Matrixmul_HLS_IP" contendrá todos los archivos del proyecto generado en Vivado HLS, incluyendo el IP final. ii) "script_HLS.tcl": es el script creado de forma automática para la creación del proyecto en Vivado HLS. iii) "script_SoC_pre.tcl" y iv) "script_SoC_post.tcl" son los scripts que permitirán posteriormente automatizar la creación de SoC en Vivado Design suite.

5.3.3. Creación Automática de SoCs de Referencia Mediante IP SoC Generator

La creación automática de SoCs de referencia se realiza en la consola Tcl de Vivado Design Suite mostrada en la Figura 92. En esta consola Tcl se selecciona el directorio del diseño a

Matrixmul_HLS_IP	29/12/18 12:16 a.m.	Carpeta de archivos	
matrixmul.cpp	26/12/18 3:09 p.m.	Archivo CPP	2 KB
matrixmul.h	24/12/18 10:43 a.m.	Archivo H	1 KB
matrixmul_test.cpp	6/11/18 5:50 p.m.	Archivo CPP	2 KB
script_HLS	29/12/18 12:16 a.m.	Archivo TCL	1 KB
script_SoC_post	29/12/18 12:17 a.m.	Archivo TCL	1 KB
script_SoC_pre	29/12/18 12:17 a.m.	Archivo TCL	2 KB

Figura 91. Archivos generados automáticamente por IP_SoC_generator durante la recreación del proyecto de referencia matrixmul

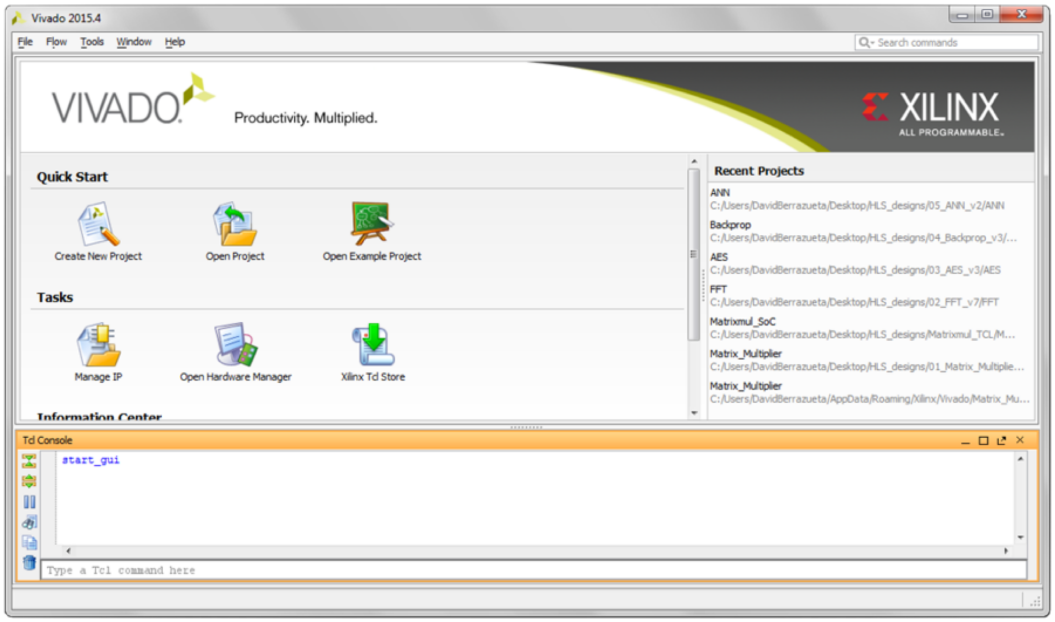


Figura 92. Ventana GUI de Vivado Design Suite. En la parte inferior se observa la ventana Tcl Console donde se ejecutan los scripts de IP_SoC_generator para la creación de SoCs de referencia.

ser replicado usando el comando "cd C:/IP_SoC_generator/Matrixmul". Si desea seleccionar otro diseño, debe colocar el directorio según se muestra en la Tabla 17[11].

[11]Para la creación de SoCs de referencia se requiere que el IP correspondiente se haya generado previamente siguiendo el procedimiento de la Sección 5.3.2.

Tabla 17
Directorios para los diseños de referencia

Diseños de referencia	Dispositivo
Matrixmul	.../IP_SoC_generator/Matrixmul
FTT	.../IP_SoC_generator/FFT
AES	.../IP_SoC_generator/AES
Backprop	.../IP_SoC_generator/Backprop
ANN	.../IP_SoC_generator/ANN

Una vez que el directorio se ha seleccionado se ejecutará el script para la automatización del proceso mediante el comando "source script_SoC_pre.tcl". Esto iniciará el proceso automático de creación del SoC. Este script finaliza con la ejecución de la síntesis, implementación y exportación del bitstream del hardware[12].

Cuando la ejecución del script "script_SoC_pre.tcl" haya finalizado, se mostrará la ventana de la Figura 93. En ella se debe seleccionar la opción Open Implemented Design[13]. Una vez que la ventana Implemented Design se haya abierto se debe ejecutar el comando "source script_SoC_post.tcl". Vivado exporta el diseño de hardware (hdf) y se importa en Xilinx SDK. En este punto concluye la automatización en la creación del hardware del SoC.

5.3.4. Creación Automática de Nuevos IP-cores Mediante IP SoC Generator

El script IP_SoC_generator también permite la creación de IP-cores que no fueron creados en esta tesis. Para la demostración de este flujo de diseño automatizado, se ha seleccionado una especificación C de un filtro FIR, obtenida de uno de los tutoriales de "Vivado Design Suite Tutorial - High-Level Synthesis" (Xilinx - UG871, 2017). El uso de IP_SoC_generator para la creación de nuevos IP-cores tiene limitaciones, las cuales son explicadas en la Sección 5.3.6.

[12]Vivado Design Suite permite ejecutar un script hasta el proceso de síntesis e implementación. Para realizar los procesos posteriores se debe ejecutar otro script. Es por esta limitación que IP_SoC_generator requiere dos scripts, script_SoC_pre.tcl y script_SoC_post.tcl.

[13]Si esta ventana no se despliega, se debe ejecutar el comando "open_run impl_1" en la consola Tcl de Vivado Design Suite.

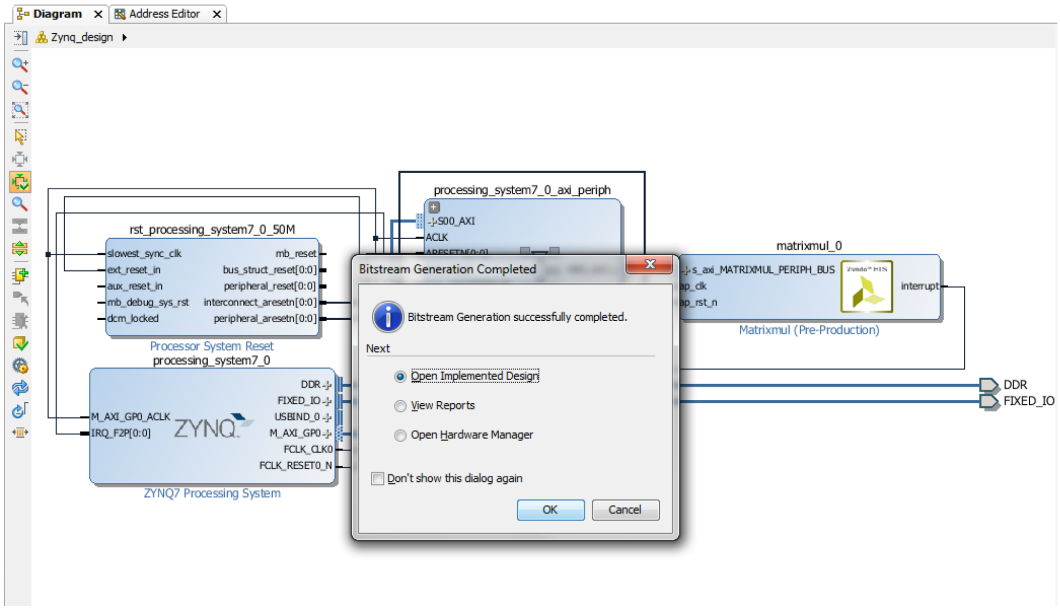


Figura 93. Mensaje de finalización del proceso de generación del bitstream.

Requisitos de la especificación C para la creación de nuevos IP-cores en IP_SoC_generator

IP_SoC_generator requiere que la especificación C proporcionada cumpla con ciertos requisitos para poder crear un nuevo IP, los mismos que se describen a continuación.

1. La especificación C y los resultados ideales del test bench deben haber sido verificados previamente en software.

2. La especificación no debe contener operaciones típicas de OS no sintetizables a hardware (Sección 3.2.2).

3. Se deben definir las entradas/salidas del IP como argumentos en la función top-level.

4. La especificación debe contar con un test bench que llame a la función top-level y compruebe sus resultados. El test bench deberá retornar un valor entero cero en caso de que los resultados sean correctos.

5. Se deben especificar las interfaces AXI en la función top-level en un único bundle para que posteriormente el IP generado pueda ser integrado en un SoC. Para ello, se debe agregar de forma manual la directiva para cada argumento de la función top-level como:

Código 20
Directiva para especificar interfaz AXI a nivel de puertos

```
#pragma HLS INTERFACE s_axilite port=nombre_arg bundle=nombre_bundle
```

- "nombre_arg" debe ser remplazado por el nombre del argumento de la función top-level.
- "nombre_bundle" debe ser remplazado por un nombre designado por el diseñador para el bundle. El mismo nombre del bundle debe colocarse en todas las directivas de interfaz de un mismo diseño.

Finalmente, se debe especificar la interfaz a nivel de bloque mediante la directiva:

Código 21
Directiva para especificar interfaz AXI a nivel de bloque

```
#pragma HLS INTERFACE s_axilite port=return bundle=nombre_bundle
```

Un ejemplo de esto se muestra en el Código 22 para la función top-level "fir". En donde se ha agregado puertos AXI para los argumentos y, c ,x, y además se agrega una interfaz AXI a nivel de bloque, todos bajo el bundle "FIR_PERIPH_BUS".

Código 22
Función top-level fir con especificación de interfaces AXI Lite

```
void fir(data_t y, coef_t c[N], data_t x) {
#pragma HLS INTERFACE s_axilite port=y bundle=FIR_PERIPH_BUS
#pragma HLS INTERFACE s_axilite port=c bundle=FIR_PERIPH_BUS
#pragma HLS INTERFACE s_axilite port=x bundle=FIR_PERIPH_BUS
#pragma HLS INTERFACE s_axilite port=return bundle=FIR_PERIPH_BUS
}
```

IP_SoC_generator para la creación de nuevos IP-cores

La especificación del filtro FIR seleccionada para el presente ejemplo consta de: i) fir.c, la cual contiene la función top-level "fir", y ii) fir.h, donde se define la función top-level.

Además, se incluye los archivos: iii) fir_test.c, el cual contiene el test bench para el DUT, y iv) out.gold.dat donde se almacenan los resultados esperados. Estos archivos deben ser colocados en el directorio de IP_SoC_generator en la carpeta "HLS_IP_Creator", como se muestra en la Figura 94.

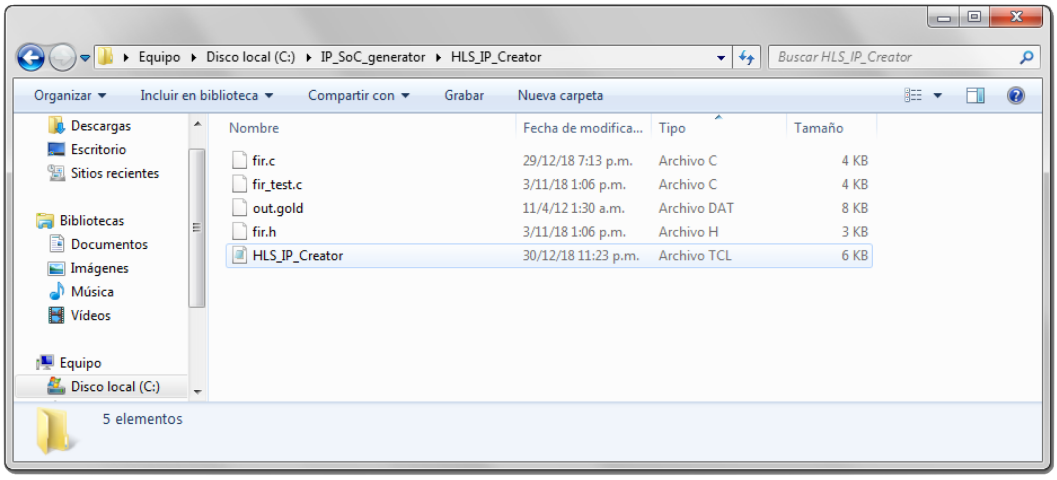


Figura 94. Archivos de la especificación C FIR para la creación de un nuevo IP-core mediante IP_SoC_generator

Ahora, la especificación C se encuentra lista para ser sintetizada a hardware. Al ejecutar IP_SoC_generator.tcl en Vivado HLS Command Prompt, se mostrarán las opciones presentadas en la Figura 89. Para generar un nuevo IP se selecciona la opción 6) New IP. IP_SoC_generator pedirá al usuario ingresar por teclado i) el nombre del proyecto para Vivado HLS, ii) el nombre de la función top-level del DUT, iii) el nombre de los archivos que forman parte de la especificación C, iv) el nombre de los archivos que forman parte del test bench C, y v) el dispositivo de destino. Este procedimiento se observa en la Figura 95 para el presente ejemplo.

Una vez ingresados estos datos, IP_SoC_generator realizará automáticamente todo el proceso de creación del IP en Vivado HLS. Cuando este proceso termine, se preguntará al usuario si desea crear los scripts necesarios para automatizar el proceso de creación de un SoC en Vivado Design Suite. Si la respuesta es afirmativa, IP_SoC_generator requiere que el usuario ingrese los siguientes datos: 1) el nombre del proyecto para Vivado Design Suite, 2) el nombre

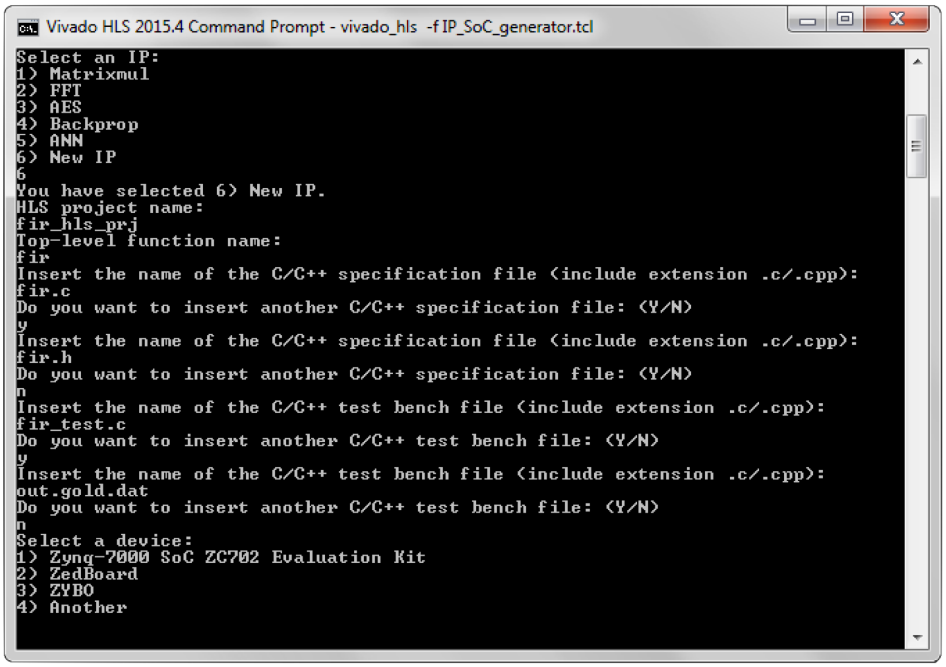


Figura 95. Creación de un nuevo IP a partir de la especificación C de un filtro FIR en IP_SoC_generator

del bundle AXI previamente definido. Este procedimiento se puede observar en la Figura 96.

Figura 96. Creación de scripts para la automatización del proceso de diseño de SoC en IP_SoC_generator

Cuando el script finalice su ejecución se podrá observar que en la carpeta "C:/IP_SoC _generator/HLS_IP_Creator" se han creado nuevos archivos, como se muestra en la Figura 97. La función de estos archivos se explica a continuación: i) la carpeta "fir_hls_prj" contendrá todos los archivos del proyecto generado en Vivado HLS, incluyendo el IP final. ii) "script_HLS.tcl": es el script creado de forma automática para la creación del proyecto en Vivado HLS. iii) "script_SoC_pre.tcl" y iv) "script_SoC_post.tcl" son los scripts que permitirán posteriormente automatizar la creación de SoC en Vivado Design Suite.

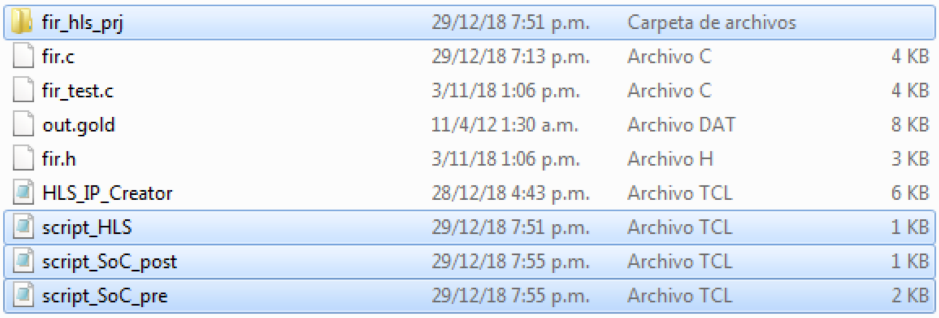


Figura 97. Archivos generados automáticamente por IP_SoC_generator durante la recreación de un nuevo IP

5.3.5. Creación Automática de Nuevos SoCs Mediante IP SoC Generator

En la Sección 5.3.4 se mostró como IP_SoC_generator es capaz de generar los scripts necesarios para automatizar el proceso de creación de SoCs que integren un acelerador en hardware desarrollado en Vivado HLS. Ahora, el usuario puede ejecutar el proceso en la consola Tcl de Vivado Design Suite, tal como se explicó en la Sección 5.3.3 "Creación Automática de SoCs de referencia mediante IP_SoC_generator". La única variante a tomar en cuenta con respecto a la explicación previa es el directorio origen a ser seleccionado. Para la creación de nuevos SoCs se debe seleccionar el directorio "C:/IP_SoC_generator/HLS_IP_Creator". Este directorio contendrá los scripts script_SoC_pre.tcl y script_SoC_post.tcl, los cuales permiten realizar la creación del SoC de forma automática.

5.3.6. Limitaciones de IP SoC Generator

IP_SoC_generator puede ser utilizado para la creación de IP-cores en Vivado HLS y su posterior integración en SoCs Zynq en Vivado Design Suite. Sin embargo, existen limitaciones en su funcionalidad que deben ser tomadas en cuenta. Estas limitaciones son descritas a continuación:

1. La recreación de los diseños realizados en esta tesis genera IP-cores sintetizados con in-

terfaces AXI4-Lite para su posterior integración en SoCs. Si el diseñador requiere variar el tipo de interfaz, puede hacerlo cambiando la directiva "#pragma HLS INTERFACE" de las especificaciones incluidas. Si se varia el tipo de interfaz de las especificaciones, la posterior recreación del SoC en Vivado Design Suite mediante IP_SoC_generator no es posible.

2. IP_SoC_generator no muestra reportes de síntesis y no permite un análisis de desempeño o recursos de las implementaciones del IP-core. Por lo tanto, IP_SoC_generator está limitado a la creación de una única solución. Sin embargo, los reportes de síntesis pueden ser consultados de forma manual (Sección 5.2.2.1). Si el diseñador desea añadir optimizaciones puede agregar las directivas necesarias directamente en los archivos de la especificación C y ejecutar nuevamente el proceso de creación del IP-core mediante IP_SoC_generator.

3. Los SoCs creados tienen una arquitectura fija, la cual integra el IP generado mediante Vivado HLS a un SoC Zynq mediante interfaces AXI4-Lite. La Figura 98 muestra esta arquitectura cuya descripción puede ser consultada en la Sección 4.1.3. En caso de que el diseñador necesite modificar esta arquitectura puede hacerlo abriendo el proyecto generado en la interfaz GUI de Vivado Design Suite y modificando el diseño de bloques.

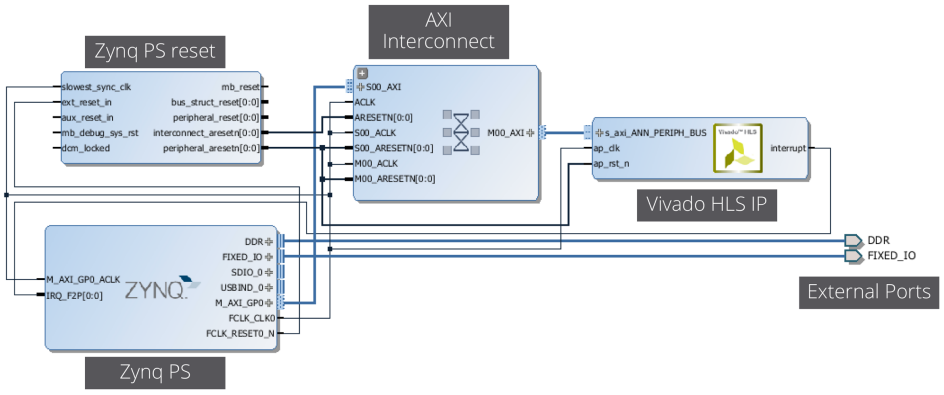


Figura 98. Arquitectura de SoCs generados mediante IP_SoC_generator. El SoC está basado en Zynq e integra un acelerador en hardware (IP-core) generados en Vivado HLS mediante interfaces AXI4-Lite.

Capítulo 6

Exploración de Desempeño

En este capítulo se describen los resultados obtenidos de las implementaciones en software y hardware de los cinco *benchmarks* seleccionados en esta tesis. Primero, se presentan los resultados obtenidos de las ejecuciones de software embebido utilizando *hard-core processors*. Después, se analizan las soluciones originales y optimizadas de los aceleradores en hardware creados usando Vivado HLS. Finalmente, se presenta un análisis comparativo de desempeño en base al tiempo de ejecución de los *benchmarks* en SoCs basados en FPGA con aceleradores en hardware y en los *hard-core processors*.

6.1. Implementación en Hard-Core Processor y Exploración de Desempeño

En esta sección se describen las implementaciones en software de los cinco *benchmarks*. Para estas implementaciones se ha utilizado procesadores ARM Cortex-A9 de Zynq, cuya descripción se presenta en la Sección 6.1.1. Mientras tanto, en la Sección 6.1.2 se presentan los resultados obtenidos de las ejecuciones en los procesadores.

6.1.1. Implementaciones en ARM Cortex-A9 de Xilinx Zynq

El procesador ARM Cortex-A9 de Xilinx Zynq PS ha sido usado de forma independiente del PL para las implementaciones de software. Este procesador tiene una arquitectura ARM v7A *dual-core*. En la Figura 99 se muestra la arquitectura del PS de un Zynq-7000, en la cual se resalta la APU (*Application Processing Unit*) que contiene dos núcleos de procesamiento ARM.

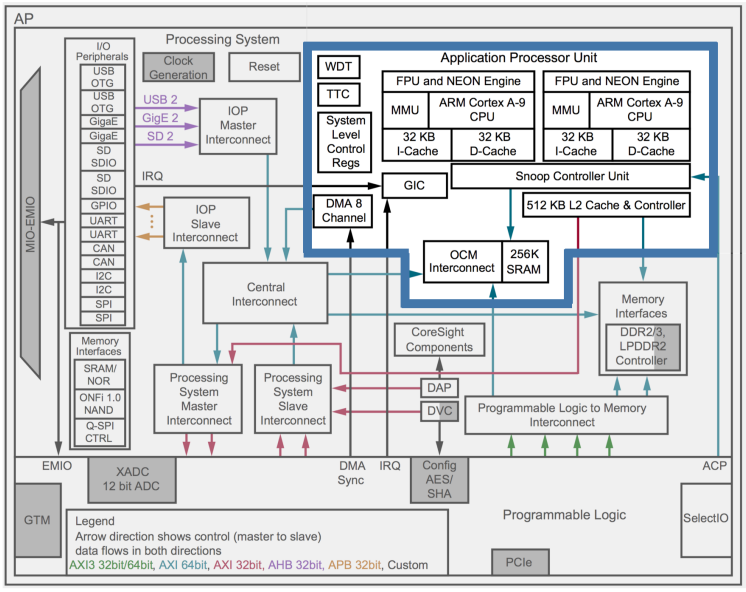


Figura 99. En la figura se resalta la APU (Application Processing Unit).

Para cada uno de los cinco *benchmarks* se ha creado una aplicación de software embebido escrita en lenguaje C++ y ejecutada sobre un BSP OS (*standalone*). En las pruebas realizadas se ha utilizado solo uno de los núcleos del ARM Cortex-A9 y se ha desactivado la memoria cache del sistema. La evaluación del procesador se ha realizado en las dos tarjetas de desarrollo basadas en Zynq usadas en esta tesis, Digilent Zybo y Xilinx ZC702 Evaluation Kit. En la Tabla 18 se presenta un resumen de la configuración de estas dos arquitecturas a las que se denomina como *A1-ARM* y *A2-ARM*, respectivamente.

Tabla 18
Resumen de configuración de las arquitecturas A1-ARM y A2-ARM

Arquitectura	Hardware	Software
A1-ARM	Tarjeta de desarrollo: Digilent Zybo Dispositivo: Zynq Z-7010 CPU: ARM Cortex-A9 650MHz Cores ARM utilizados: 1 Cache: ninguno Acelerador en PL: ninguno	Sistema operativo: BSP standalone Aplicación: C++ Compilador: GCC CNU 4.4
A2-ARM	Tarjeta de desarrollo: XilinxZC702 Evaluation Kit Dispositivo: Zynq Z-7020 CPU: ARM Cortex-A9 667MHz Cores ARM utilizados: 1 Cache: ninguno Acelerador en PL: ninguno	

Nota: Estas arquitecturas se basan en un núcleo ARM Cortex-A9 de Zynq.

6.1.2. Exploración de Desempeño de Implementaciones de Software en ARM Cortex-A9

El tiempo de ejecución de las implementaciones de esta tesis se ha medido utilizando el timer global del ARM Cortex, cuyo acceso se define en el archivo de cabecera "xtime_l.h" creado por Xilinx (Xilinx UG1145, 2018). Antes y después de llamar a la función *main* de cada *benchmark* se ejecuta la función "XTime_GetTime()", la cual entrega el número de ciclos de reloj del procesador desde el inicio de la ejecución de la aplicación. La diferencia entre los dos valores obtenidos es convertida a unidades de tiempo utilizando el número de ciclos de reloj por segundo definido para el procesador. Considerando que un sistema requerirá ejecutar un mismo algoritmo decenas de veces en una aplicación, cada *benchmark* ha sido ejecutado cien veces y se presenta la media de su latencia.

En la Tabla 19 se muestra el tiempo de ejecución de los benchamarks implementados en las arquitecturas A1-ARM y A2-ARM. Los resultados presentados en esta tabla no reflejan diferencias significativas en tiempo de ejecución. Sin embargo, la arquitectura A1-ARM pre-

senta un desempeño menor para todos los benchmarks, por lo que se ha decidido seleccionarla como sistema base para calcular los factores de aceleración en la Sección 6.3.

Tabla 19
Resultados de desempeño de las arquitecturas A1-ARM y A2-ARM

Benchmark	A1-ARM	A2-ARM
	Tiempo (us)	Tiempo (us)
Matrixmul	15,18	15,11
FFT	20372,95	20040,96
AES	2142,68	2094,99
Backprop	20046265,67	17389123,38
ANN	8452,2	8291,66

Nota: Estas arquitecturas están basadas en un núcleo ARM Cortex-A9 de Zynq.

6.2. Aceleración en IP-cores y Exploración de Desempeño

En esta sección se revisa el desempeño obtenido por los aceleradores (IP-cores) en hardware de FPGA. En la Sección 6.2.1 se realiza un análisis comparativo de desempeño entre las implementaciones generadas por defecto y optimizadas de los aceleradores creados en Vivado HLS. En la Sección 6.2.2. se definen las arquitecturas de SoCs que incluyen los aceleradores de hardware optimizados.

Métricas de prueba para los aceleradores en hardware de FPGA

Los resultados de desempeño de los aceleradores se analizan en base a tres métricas: **i) ciclos de latencia:** número de ciclos de reloj requeridos para completar una ejecución del acelerador, **ii) frecuencia del reloj:** frecuencia de reloj a la que se ejecuta el acelerador en el PL, **iii) tiempo de ejecución en microsegundos:** tiempo total de ejecución del acelerador medido desde el PS.

Factor de aceleración

Los factores de aceleración o *speedup* presentados en este capítulo se calculan en base a la Ley de Amdahl. Esta ley define la aceleración de un sistema que ha sido mejorado con respecto a un sistema previo como

$$S = \frac{T_{exec1}}{T_{exec2}} \tag{6.1}$$

donde T_{exec1} representa el tiempo de ejecución del sistema previo y T_{exec2} representa el tiempo de ejecución del sistema mejorado (Pankratius, Adl-Tabatabai, & Tichy, 2012).

6.2.1. Exploración de Desempeño de Aceleradores Creados en Vivado HLS

Vivado HLS sintetiza por defecto implementaciones en las cuales se prioriza una mínima utilización de recursos. Por lo tanto, cada implementación ha sido analizada para mejorar su desempeño. En base a este análisis, se han aplicado optimizaciones orientadas a reducir los ciclos de latencia. Los detalles acerca de la creación de cada uno de los aceleradores y sus optimizaciones pueden ser consultados en el Capítulo 4 "*Diseño de Sistemas SoC con IP-Cores Personalizados Usando Vivado*".

En la Tabla 20 se presenta una comparación entre las soluciones por defecto generadas en Vivado HLS y las soluciones optimizadas. Los factores de aceleración mostrados en esta tabla se calculan en base a la Ecuación 6.1, donde T_{exec1} y T_{exec2} representan el tiempo de ejecución[1] de los aceleradores originales y optimizados, respectivamente. En la Figura 100 se muestra gráficamente el factor de aceleración entre IP-cores. Además, en la Tabla 21 se comparan los recursos de lógica programable utilizados por cada una de estas soluciones. A continuación, se realiza un análisis sobre el desempeño de cada diseño comparando la solución

[1]El tiempo de ejecución para la comparación entre aceleradores en esta sección se calcula en base a los ciclos de latencia y frecuencias de reloj de cada uno.

generada por defecto con la mejor implementación obtenida.

Tabla 20
Resultados de desempeño de aceleradores por defecto y optimizados

Benchmark	Desempeño por defecto		Desempeño con optimizaciones		Aceleración
	Ciclos	Frec. (MHz)	Ciclos	Frec. (MHz)	
Matrixmul	85	125	7	125	12.14×
FFT	122901	100	122901	100	N/A
AES	1085	100	928	100	1.17×
Backprop	464387001	100	82890001	100	5.60×
ANN	83993	100	32298	100	2.60×

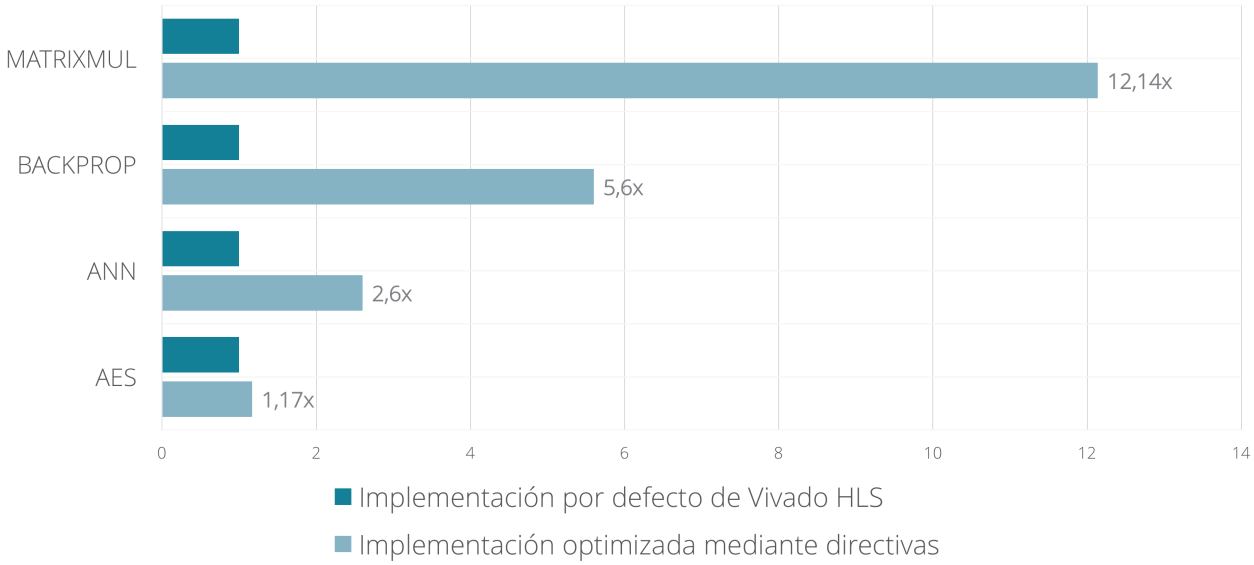


Figura 100. Factor de aceleración de los IP-cores optimizados en relación a los diseños generados por defecto en Vivado HLS.

1. **Matrixmul.** Este diseño ha sido optimizado hasta alcanzar una ejecución conocida como fully pipelined, lo que quiere decir que el IP tiene un intervalo de iniciación de uno. La aceleración lograda es de 12.14× en comparación a la solución original. En términos de porcentaje, los ciclos de reloj requeridos para una ejecución se han disminuido en un 91.76 %. Por otro lado, los recursos para la solución optimizada se han incrementado de forma considerable: DSP48 en 800 %, FFs en un 233.18 %, y LUTs en 267.43 %. (Véase Sección 4.1.5)

2. **FFT.** En este acelerador se aplicaron varias técnicas para mejorar el performance de la optimización por defecto de Vivado. Los resultados de estas nuevas soluciones no varían con respecto a la solución original. Esta exploración fue importante para descubrir que diseños con altas dependencias de datos no pueden ser paralelizados en Vivado HLS, lo cual se describe más en la Sección 4.2.4.

3. **AES.** Este acelerador se ha optimizado con pipelining parcial, es decir, solo ciertos lazos se ejecutan de forma paralela. Con esta optimización se logra una aceleración de 1.17× con respecto a la solución original. Esta aceleración representa una reducción del 14.47 % de los ciclos de reloj requeridos para completar una ejecución. Los recursos de esta solución que se han incrementado son: FFs en 24.03 % y LUTs en 16.58 %. (Véase Sección 4.3.4)

4. **Backprop.** En términos de recursos computacionales, el algoritmo de retropropagación es el benchmark más demandante entre el grupo de algoritmos seleccionados en esta tesis. Este algoritmo realiza cálculos y operaciones de lectura/escritura con una gran cantidad de datos, lo cual representa una limitante para aplicar pipelining a nivel de funciones utilizando Vivado HLS. Sin embargo, se ha utilizado pipelining a nivel de lazos, logrando una aceleración de 5.60×, lo cual representa un 82.15 % menos de ciclos de latencia. Los recursos de esta solución que se han incrementado son: FF en 115.49 % y LUTs en 32.49 %. (Véase Sección 4.4.4)

5. **ANN.** La ejecución de esta red neuronal se ha mejorado considerablemente mediante la aplicación de un pipelining parcial. Los ciclos de reloj necesarios para su ejecución se han reducido en 61.54 %. Esta disminución de ciclos de latencia representa una aceleración de 2.60×. Los recursos de esta implementación que se han incrementado son: DSP48 en 288.24 %, FFs en 133.86 %, y las LUTs en 137.86 %. (Véase Sección 4.5.4)

Tabla 21
Resumen de utilización de recursos de FPGA de aceleradores en hardware

IP	Área solución por defecto				Área con optimizaciones			
	BRAM	DSP48E	FF	LUT	BRAM	DSP48E	FF	LUT
Matrixmul	6	4	434	350	0	32	1012	936
FFT	16	56	4533	7890	16	56	4533	7890
AES	30	0	28353	45789	32	0	35165	53379
Backprop	19	148	28561	40262	26	168	61546	53345
ANN	19	34	7017	9966	18	132	16410	23705

Nota: Los recursos mostrados corresponden a las implementaciones por defecto y optimizadas en Vivado HLS.

6.2.2. Aceleradores en SoCs Basados en FPGA

Los aceleradores optimizados de los benchmarks se han implementado en un SoC basado en FPGA. El SoC integra un solo acelerador comunicándose a través de interfaces AXI al PS. El procesador controla el acelerador y mide el tiempo total de ejecución. La evaluación de los aceleradores se ha realizado en las tarjetas de desarrollo Digilent Zybo y Xilinx ZC702 Evaluation Kit. En la Tabal 22 se presenta el resumen de configuración de estas arquitecturas heterogéneas, las cuales se denominan como *A1-ARM+IPacc*[2] y *A2-ARM+IPacc*, respectivamente.

6.3. Análisis Comparativo entre Diferentes Arquitecturas

En esta sección se realiza la comparación de desempeño en base al tiempo de ejecución de los benchmarks en las arquitecturas basadas en hard-core processor: *A1-ARM*, *A2-ARM*, y las arquitecturas basadas en SoCs heterogéneos: *A1-ARM+IPacc* y *A2-ARM+IPacc*. En la Figura 101 se muestra una representación abstracta de estos dos tipos principales de arquitecturas. Para esta comparación se ha desactivado la memoria cache de datos e instrucciones

[2]IPacc (IP accelerator)

Tabla 22
Resumen de configuración de las arquitecturas A1-ARM+IPacc y A2-ARM+IPacc

Arquitectura	Hardware	Software
A1-ARM+IPacc	Tarjeta de desarrollo: Digilent Zybo Dispositivo: Zynq Z-7010 CPU: ARM Cortex-A9 650MHz Cores ARM utilizados: 1 Cache: ninguno Acelerador en PL: sí	Sistema operativo: BSP standalone Aplicación: C Compilador: GCC CNU 4.4
A2-ARM+IPacc	Tarjeta de desarrollo: Xilinx ZC702 Evaluation Kit Dispositivo: Zynq Z-7020 CPU: ARM Cortex-A9 667MHz Cores ARM utilizados: 1 Cache: ninguno Acelerador en PL: sí	

Nota: Estas arquitecturas se basan en un núcleo ARM Cortex-A9 y uno de los aceleradores en hardware de Zynq desarrollados en esta tesis.

del procesador ARM debido a que se desea evaluar el desempeño computacional de los dos sistemas de procesamiento, sin considerar ventajas y desventajas producidas por el desempeño de memoria. De hecho, la activación de memoria cache para el procesador y para el acelerador, a través de DMA (*Direct Memory Access*), mejorarían el desempeño de los dos sistemas.

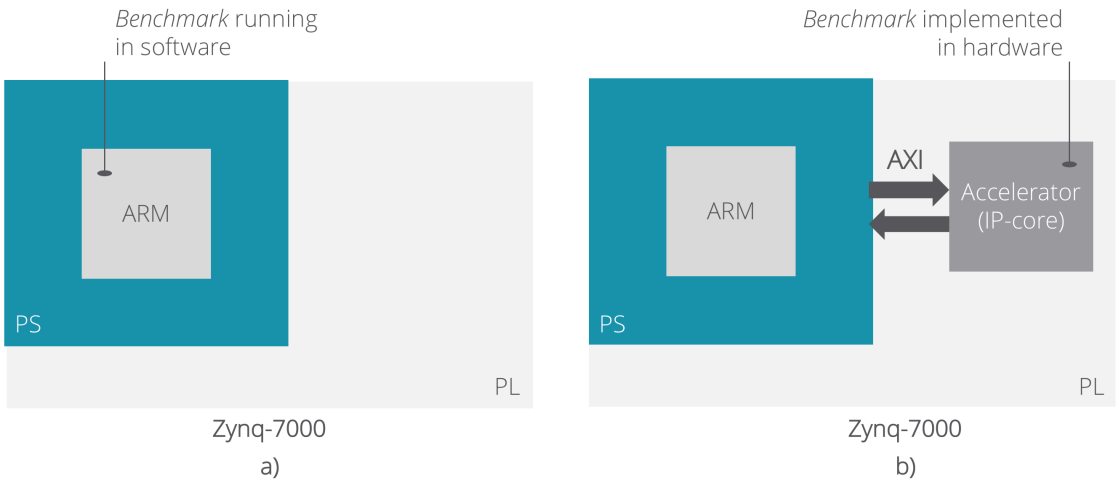


Figura 101. Representación abstracta de las arquitecturas basadas en Zynq. a) procesador ARM del PS ejecutando software y b) ARM ejecutando software con funciones aceleradas en IP-cores

En la Tabla 23 se presenta la comparación de desempeño basado en el tiempo de ejecución de los *benchmarks* en cada arquitectura. Las implementaciones de los aceleradores AES y ANN en las arquitecturas A1-ARM y A2-ARM difieren en las optimizaciones aplicadas debido a la diferencia de recursos de FPGA disponibles en cada una. En esta tabla se presentan los tiempos de ejecución y el speedup respecto a la arquitectura base *A1-ARM* de acuerdo a la Ecuación 6.1. En la Figura 102 se muestra gráficamente los factores de aceleración de cada *benchmark* entre la arquitectura base y la arquitectura con mejor desempeño, *A2-ARM+IPacc*.

Tabla 23
Resultados de desempeño de benchmarks en diferentes arquitecturas

Benchmark	A1-ARM	A2-ARM		A1-ARM+IPacc		A2-ARM+IPacc	
	T (us)	T (us)	S	T (us)	S	T (us)	S
Matrixmul	15,18	15,11	N/A	2,97	5,11×	2,92	5,20×
FFT	20372,95	20040,96	1,02×	1232,18	16,53×	1232,17	16,53×
AES	2142,68	2094,99	1,02×	44,62	48,02×	12,63	169,65×
Backprop	20046265,67	17389123,38	1,15×	N/A*	N/A*	828902,79	24,18×
ANN	8452,2	8291,66	1,02×	332,83	25,39×	325,75	25,95×

Nota: T=tiempo; S=speedup.
*El diseño requiere más recursos de FPGA de los disponibles en A1-ARM+IPacc

El factor de aceleración alcanzado en hardware de FPGA con respecto a las implementaciones de software del *hard-core processsor* puede diferir significativamente para cada *benchmark*. En general, en computación paralela, no todos los algoritmos son paralelizables, y el nivel de paralelizacion que se puede alcanzar es una característica de cada uno. A continuación, se analiza cada *benchmarks* explicando las limitaciones de su ejecución en hardware de FPGA. En la Figura 103 se presenta una gráfica comparativa del tiempo de ejecución de las dos arquitecturas utilizadas.

1. **Matrixmul.** La multiplicación de matrices es un algoritmo altamente paralelizable. El valor de cada elemento de la matriz resultado es independiente del cálculo de los demás elementos. Este inherente paralelismo del algoritmo ha permitido alcanzar la

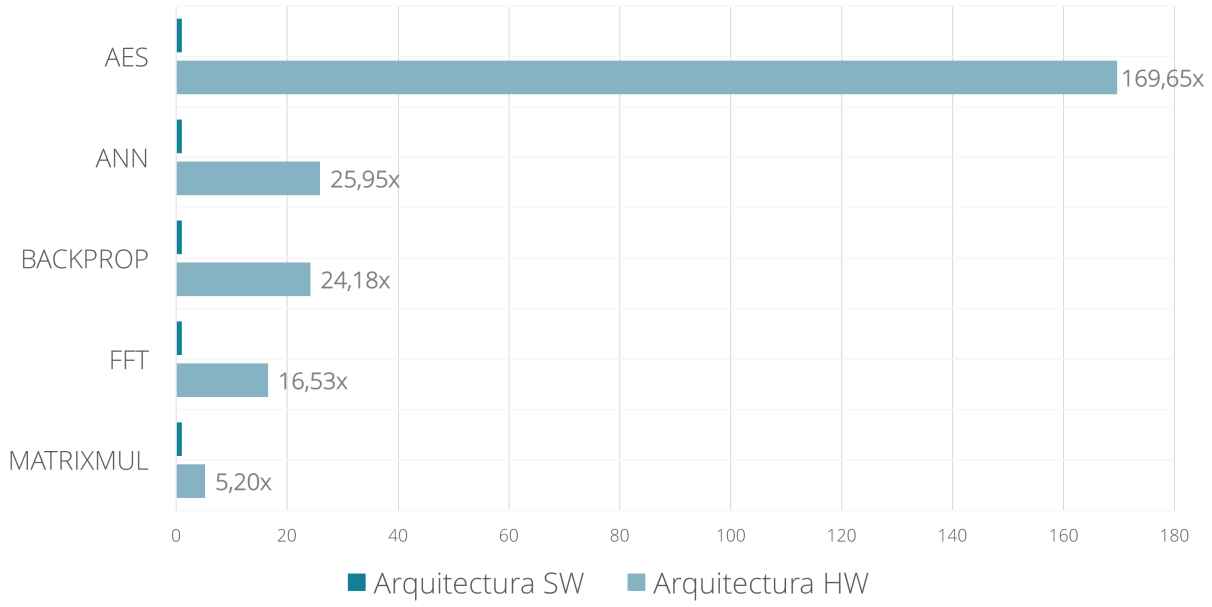


Figura 102. Factor de aceleración de la arquitectura A2-ARM+IPacc en relación a A1-ARM

mejor optimización de desempeño posible utilizando Vivado HLS. En comparación con procesador ARM, el acelerador reduce el tiempo de ejecución en 80.76 %.

2. **FFT.** El algoritmo FFT utilizado tiene un gran número dependencias de datos, lo que dificulta la implementación de paralelización. Para este diseño se han sintetizado soluciones con pipelining sin conseguir que el desempeño mejore. Pese a esta limitación, el acelerador reduce el tiempo de ejecución en 93,95 %.

3. **AES.** La ejecución de las operaciones de encriptación y des-encriptación AES (SubBytes, MixColumns, ShiftRows, AddRoundKey) son altamente paralelizables. Por otro lado, las etapas del proceso KeySchedule son dependientes del cálculo de valores previos. Los resultados de este proceso son usados en la encriptación y des-encriptación, lo que dificulta que este algoritmo sea paralelizado en su totalidad. Sin embargo, la aplicación de pipelining en las etapas de la encriptación AES, las cuales son realizadas varias veces en una ejecución (rounds), representa una ventaja considerable respecto a implementaciones de software. Este acelerador reduce el tiempo de ejecución del algoritmo

en un 99,41 % con respecto al procesador ARM.

4. **Backprop.** El algoritmo de retropropagación es paralelizable a nivel de lazos. El cálculo del error y actualización de los pesos de distintas capas de una red neuronal puede ser llevada a cabo aplicando pipelining. Además, la ejecución repetitiva por épocas del algoritmo usando paralelismo en FPGA representa una ventaja significativa respecto al software que realiza cada iteración secuencialmente. El acelerador reduce el tiempo de ejecución en 95.87 %.

5. **ANN.** La ejecución de redes neuronales es altamente paralelizable. El cálculo de la suma ponderada en las neuronas de una misma capa es independiente de las demás, por lo cual cada una de estas sumas puede ser realizada en paralelo. Sin embargo, las redes neuronales tienen que almacenar y realizar cálculos con grandes cantidades de datos, por lo cual un incremento de desempeño requeriría un significativo impacto a los recursos. Sin embargo, con este acelerador se reduce el tiempo de ejecución del procesador ARM en un 96.15 %.

6.4. Discusión de Resultados

Los aceleradores en hardware de FPGA desarrollados ofrecen un rendimiento mucho mayor al de un procesador de software embebido como el ARM Cortex-A9. Si bien Vivado HLS permite obtener resultados óptimos, el desempeño de los aceleradores comparado con implementaciones de software dependerá del nivel de paralelización de cada algoritmo, y de la capacidad de la herramienta HLS para explotarlo. El principal factor a tomar en cuenta en el desarrollo de aceleradores en hardware de FPGA es que el proceso de optimización se encuentra limitado por el nivel de paralelismo propio de cada algoritmo y por el número de recursos disponibles en el FPGA utilizado. La mayoría de algoritmos implementados en esta tesis tiene limitaciones propias, es decir, secciones del algoritmo no son paralelizables. Por

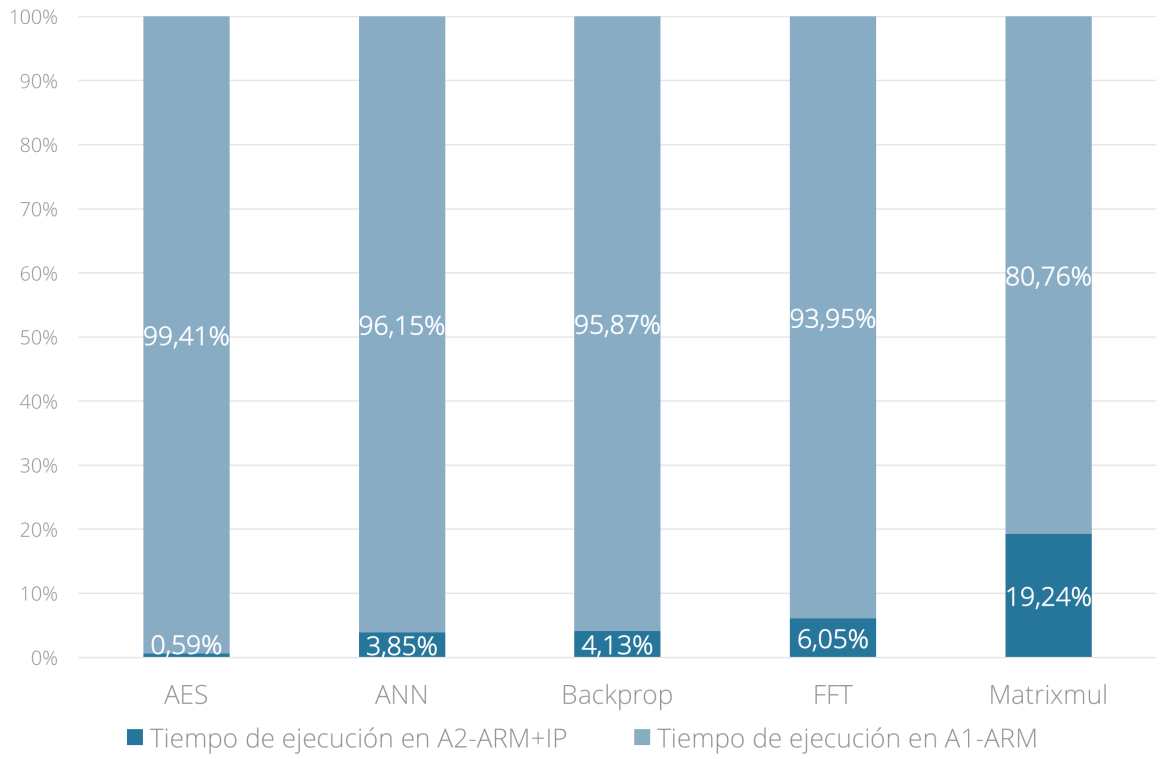


Figura 103. Reducción del tiempo de ejecución de algoritmos implementados en aceleradores en hardware. El total de la barra representa el total del tiempo de ejecución de la arquitectura A1-ARM. En color celeste se muestra la reducción del tiempo total de ejecución y en color azul el tiempo de ejecución con la arquitectura A2-ARM+IPacc, donde se obtuvo las mayores aceleraciones.

otro lado, algoritmos como ANN podrían implementarse con un nivel de pipelining mayor. Sin embargo, esto significaría incrementar los recursos de forma considerable. Por lo tanto, una optimización global de algoritmos complejos no es alcanzable en la mayoría de los casos. El diseñador debe enfocarse en optimizar el desempeño de secciones del algoritmo que puedan a su vez mejorar el desempeño general del acelerador.

Recomendaciones en la selección de algoritmos para aceleradores en hardware

Existen algoritmos que pueden ser más adecuados para acelerarse en FPGA usando Vivado HLS. En base a la experiencia de diseño y los resultados obtenidos se presentan pautas para seleccionar los algoritmos que pueden incrementar significativamente su desempeño eje-

cutándose en hardware.

- **Tipo de datos:** Algoritmos que realizan operaciones a nivel de bytes o enteros son altamente recomendados para ser optimizados en hardware. Por otro lado, algoritmos con operaciones de tipo flotante demandan demasiados recursos para almacenar datos y realizar operaciones con ellos.

- **Dependencias de datos:** Algoritmos con altas dependencias de datos no son buenos candidatos a acelerarse en hardware. Estas dependencias fuerzan a los algoritmos a ejecutarse de forma secuencial, lo cual no es adecuado para FPGA y no representa una ventaja considerable respecto a implementaciones de software.

- **Varias iteraciones o rounds:** Algoritmos o secciones de algoritmos que sean iterados una gran cantidad de veces son adecuados para implementarse en hardware, en especial si tienen pocas dependencias de datos. Por el contrario, algoritmos secuenciales con pocas o ninguna iteración no incrementaran el desempeño significativamente.

Capítulo 7

Conclusiones y Recomendaciones

En este capítulo se presentan las conclusiones y recomendaciones más significativas basadas en la investigación y experiencia adquirida en el desarrollo de esta tesis, de manera que aporten a trabajos futuros.

7.1. Conclusiones

Los métodos HLS para FPGA representan un significativo avance para el desarrollo de los sistemas embebidos. No obstante, se observan retos futuros para mejorar las herramientas HLS actuales. La acogida masiva de estas herramientas depende aún de la disminución del nivel de complejidad en el proceso de diseño. Este proceso requiere en gran parte ser guiado por el diseñador, el cual necesita involucrase con ciertos aspectos a nivel de hardware, como el uso de interfaces, especificación del tipo de recursos, y análisis de flujos de datos y optimizaciones a nivel de hardware. Esta limitante puede representar una brecha para el acercamiento de desarrolladores de software al uso de FPGAs.

Vivado HLS permite generar implementaciones RTL a partir de descripciones de alto-nivel alcanzando resultados óptimos con la misma exactitud de implementaciones de software, y, además, reduciendo los tiempos de diseño y verificación de aceleradores de forma considerable.

Sin embargo, Vivado HLS no automatiza el análisis de optimización de desempeño de las implementaciones. Para que los diseños puedan alcanzar un máximo desempeño se requiere que el desarrollador conozca en profundidad los algoritmos, en caso de requerir modificaciones, y evalúe el flujo de datos y sus dependencias para aplicar directivas que influencien la síntesis.

El proceso de análisis y optimización de desempeño de cada uno de los aceleradores implementados en esta tesis ha permitido explorar las limitaciones de Vivado HLS y exponer las dificultades presentadas durante el proceso de diseño. De esta forma, la tesis aporta con estos aceleradores y su documentación como puntos de referencia para nuevos diseños que, además, puedan ser fácilmente reutilizados en investigaciones futuras.

Es necesario un análisis para determinar que algoritmos justifican el uso de hardware de FPGA. En el desarrollo de SoCs heterogéneos con aceleradores de funciones, el sistema debe alcanzar un balance entre la mejora en el desempeño y el impacto en los recursos disponibles. El factor de éxito depende de la selección de algoritmos que pueden ser más apropiados para su traslado a FPGA. Por ejemplo, aceleradores de algoritmos desarrollados usando Vivado HLS que contengan operaciones paralelizables con bytes o datos de tipo entero, y que sean iterados una gran cantidad de veces ofrecen un desempeño mucho mayor en hardware. Por el contrario, aceleradores de algoritmos con muchas dependencias de datos, lazos de limites variables, datos de tipo flotante y pocas o ninguna iteración pueden tener un desempeño limitado en FPGA.

La exploración y comparación de desempeño de las arquitecturas basadas en procesadores de software embebido, A1-ARM y A2-ARM, con respecto a arquitecturas heterogéneas basadas en FPGA, A1-ARM+IPacc y A2-ARM+IPacc, muestran que el traslado a hardware de algoritmos de interés y tendencia actual como métodos de entrenamiento supervisados y ejecución de redes neuronales artificiales pueden incrementar su desempeño considerablemente, con aceleraciones de hasta 25x. Además, el sistema criptográfico AES, una aplicación de interés en seguridad y defensa, área de investigación de la Universidad de las Fuerzas Armadas ESPE, demuestra un desempeño muy superior al ejecutarse en hardware de FPGA,

con una aceleración de hasta 169x.

Se planteó un flujo de diseño confiable para la creación de SoCs basados en FPGA con funciones implementadas en IP-cores fundamentado en la investigación y experiencia obtenida en el desarrollo de esta tesis. Este flujo permite a diseñadores utilizar las herramientas de Xilinx para obtener resultados confiables aplicando los procesos de verificación disponibles. Además, se plantea un incremento en la automatización para la creación del hardware en este flujo de diseño mediante scripts Tcl. Estos scripts permiten generar IP-cores usando Vivado HLS e implementarlos en SoCs Zynq usando Vivado Desing Suite con un esfuerzo manual mínimo y sin requerir de un conocimiento profundo acerca del uso de las herramientas de diseño. Además, los scripts generados permiten replicar los diseños de referencia, facilitando la recreación de los trabajos realizados en esta tesis. Estos scripts están limitados a la creación del hardware de SoCs con una arquitectura fija y puede ser mejorado en trabajos futuros para incrementar el nivel de automatización alcanzado.

7.2. Recomendaciones

Para la creación exitosa de aceleradores en hardware de FPGA usando Vivado HLS se recomienda:

- Realizar todos los pasos de verificación disponibles en la herramienta, los cuales aseguran la funcionalidad de los IP-cores creados. Además, una verificación previa del algoritmo en software ahorra tiempo de diseño y la necesidad de correcciones posteriores.

- Seleccionar algoritmos que sean adecuados para su ejecución en hardware de FPGA, especialmente algoritmos con posibilidad de implementarse usando paralelismo masivo.

- Verificar que los algoritmos en software se encuentren previamente optimizados para que la síntesis HLS genere un hardware igualmente óptimo.

- Usar scripts para la creación de SoCs que evitan realizar tareas repetitivas en herramientas de diseño de Xilinx, lo cual disminuye aún más el tiempo requerido para la creación de aceleradores.

Por último, se presentan recomendaciones para trabajos futuros que aportarían de forma significativa a la investigación de sistemas embebidos heterogéneos de alto desempeño:

- Un análisis comparativo del consumo energético, parámetro trascendental en los sistemas embebidos, entre SoCs basados en FPGA y procesadores de propósito específico debería realizarse para, en conjunto con los resultados de desempeño de esta tesis, poder establecer ventajas y desventajas del uso de cada tipo de procesador en aplicaciones de interés para el DEEE.

- Es importante que investigaciones futuras puedan comparar implementaciones en FPGA con implementaciones en otros tipos de arquitecturas heterogéneas que usen coprocesadores para aceleración de algoritmos. Una comparación de desempeño en base a benchmarks similares, permitiría explorar un mayor espacio de diseño para sistemas embebidos de alto rendimiento, determinando el tipo de algoritmo adecuado para cada procesador, y poder así aprovechar las potencialidades de cada uno.

Referencias

Abdurohman, M., Kuspriyanto, Sutikno, S., y Sasongko, A. (2010). The New Embedded System Design Methodology For Improving Design Process Performance. *Int. J. Comput. Sci. Inf. Secur.*, *8*(1), 35–43. Descargado de `http://arxiv.org/abs/1005.0931`

Ananthanarayana, T., Lopez, S., y Lukowiak, M. (2017). Power analysis of HLS-designed customized instruction set architectures. En *Proc. - 2017 ieee 31st int. parallel distrib. process. symp. work. ipdpsw 2017.* doi: 10.1109/IPDPSW.2017.59

Arunachalam, S., Khairnar, S., y Desale, B. (2005). The Fast Fourier Transform Algorithm and Its Application in Digital Image Processing. *Int. Conf. Recent Trends Appl. Sci. with Eng. Appl.*, *3*(6), 267–274. doi: 10.1007/978-94-009-8543-8

Avelino, Á., Obac, V., Harb, N., Valderrama, C., Albuquerque, G., y Possa, P. (2017). *LP-P2IP: A low-power version of P2IP architecture using partial reconfiguration.* doi: 10.1007/978-3-319-56258-2_2

Azad, S., y Pathan, A. S. K. (2014). *Practical cryptography: Algorithms and implementations using C++.* doi: 10.1201/b17707

Bacon, D. F., Rabbah, R., y Shukla, S. (2013). FPGA programming for the masses. *Commun. ACM*. doi: 10.1145/2436256.2436271

Bobda, C., Mead, J., Whitaker, T. J., Kamhoua, C., y Kwiat, K. (2017). Hardware sandboxing: A novel defense paradigm against hardware trojans in systems on chip. En *Lect. notes comput. sci. (including subser. lect. notes artif. intell. lect. notes bioinformatics).* doi: 10.1007/978-3-319-56258-2_5

Brodtkorb, A. R., Dyken, C., Hagen, T. R., Hjelmervik, J. M., y Storaasli, O. O. (2010). State-of-the-art in heterogeneous computing. *Sci. Program.*. doi: 10.3233/SPR-2009-0296

Burns, R. S. (2001). *Advanced Control Engineering* (1st ed.). doi: 10.1016/B978-0-7506-5100-4.X5000-1

Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Anderson, J. H., ... Czajkowski, T. (2011). LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems. *Proc. 19th ACM/SIGDA Int. Symp. F. Program. gate arrays - FPGA '11*. doi: 10.1145/1950413.1950423

Choi, Y.-k., Cong, J., Fang, Z., Hao, Y., Reinman, G., y Wei, P. (2016). A quantitative analysis on microarchitectures of modern CPU-FPGA platforms. En *Proc. 53rd annu. des. autom. conf. - dac '16.* doi: 10.1145/2897937.2897972

Cong, J., Liu, B., Neuendorffer, S., Noguera, J., Vissers, K., y Zhang, Z. (2011). High-level synthesis for FPGAs: From prototyping to deployment. En *Ieee trans. comput. des. integr. circuits syst.* doi: 10.1109/TCAD.2011.2110592

Coussy, P., Gajski, D., Meredith, M., y Takach, A. (2009, jul). An Introduction to High-Level Synthesis. *IEEE Des. Test Comput.*, *26*(4), 8–17. Descargado de `http://ieeexplore.ieee.org/document/5209958/` doi: 10.1109/MDT.2009.69

Coussy, P., y Morawiec, A. (2016). *High-Level Synthesis* (1st ed.; P. Coussy y A. Morawiec, Eds.). Springer Netherlands. doi: 10.1007/978-1-4020-8588-8

Crockett, L. H., Elliot, R. A., Enderwitz, M. A., y Stewart, R. W. (2015). Zynq Book. *Aging (Albany. NY)*.. doi: 10.1017/CBO9781107415324.004

Del Sozzo, E., Baghdadi, R., Amarasinghe, S., y Santambrogio, M. D. (2017). A common backend for hardware acceleration on FPGA. En *Proc. - 35th ieee int. conf. comput. des. iccd 2017.* doi: 10.1109/ICCD.2017.75

de Oliveira, Á. B., Tambara, L. A., y Kastensmidt, F. L. (2017). Exploring performance overhead versus soft error detection in lockstep dual-core ARM cortex-A9 processor embedded into Xilinx Zynq APSoC. En *Lect. notes comput. sci. (including subser. lect.*

notes artif. intell. lect. notes bioinformatics). doi: 10.1007/978-3-319-56258-2_17

Domingo, R., Salvador, R., Fabelo, H., Madronal, D., Ortega, S., Lazcano, R., ... Sanz, C. (2017). High-level design using Intel FPGA OpenCL: A hyperspectral imaging spatial-spectral classifier. En *12th int. symp. reconfigurable commun. syst. recosoc 2017 - proc.* doi: 10.1109/ReCoSoC.2017.8016152

Farooq, U., Marrakchi, Z., y Mehrez, H. (2012). *Tree-based heterogeneous FPGA architectures: Application specific exploration and optimization*. doi: 10.1007/978-1-4614-3594-5

Fingeroff, M. (2010). *High-Level Synthesis, Introduction to Chip and System Design* (1st ed.; Xlibris, Ed.). Xlibris. Descargado de `http://link.springer.com/content/pdf/10.1007/978-1-4615-3636-9.pdf{%}5Cnhttp://scholar.google.com/scholar?hl=en{&}btnG=Search{&}q=intitle:High-Level+Synthesis+Blue+Book{#}0`

Fletcher, B. H. (2005). FPGA Embedded Processors Revealing True System Performance. *Embed. Syst. Conf.*.

Fort, B., Canis, A., Choi, J., Calagar, N., Lian, R., Hadjis, S., ... Anderson, J. (2014). Automating the Design of Processor/Accelerator Embedded Systems with LegUp High-Level Synthesis. En *Proc. - 2014 int. conf. embed. ubiquitous comput. euc 2014*. doi: 10.1109/EUC.2014.26

Göbel, M., Elhossini, A., Chi, C., Alvarez-Mesa, M., y Juurlink, B. (2017). *A quantitative analysis of the memory architecture of FPGA-SoCs*. doi: 10.1007/978-3-319-56258-2_21

Gort, M., y Anderson, J. (2015). Design re-use for compile time reduction in FPGA high-level synthesis flows. En *Proc. 2014 int. conf. field-programmable technol. fpt 2014.* doi: 10.1109/FPT.2014.7082746

Ha, S., y Teich, J. (2017). *Handbook of hardware/software codesign*. doi: 10.1007/978-94-017-7267-9

Hara, Y., Tomiyama, H., Honda, S., Takada, H., y Ishii, K. (2008). CHStone: A benchmark program suite for practical C-based high-level synthesis. En *Proc. - ieee int. symp. circuits syst.* doi: 10.1109/ISCAS.2008.4541637

Hoozemans, J., Heij, R., Straten, J. V., y Al-Ars, Z. (2017). VLIW-based FPGA computation fabric with streaming memory hierarchy for medical imaging applications. En *Lect. notes comput. sci. (including subser. lect. notes artif. intell. lect. notes bioinformatics).* doi: 10.1007/978-3-319-56258-2_4

Kazmierkowski, M. (2009). Embedded Systems Handbook, 2nd edition (Zurawski, R.; 2009) [Book News]. *IEEE Ind. Electron. Mag.*. doi: 10.1109/MIE.2009.933874

Koch, D., Hannig, F., y Ziener, D. (2016). *FPGAs for software programmers.* doi: 10.1007/978-3-319-26408-0

Kochan, S. G. (2014). *Programming in C* (4th editio ed.). doi: 10.1002/1521-3773(20010316)40:6⟨9823::AID-ANIE9823⟩3.3.CO;2-C

Lin, S. Y. L. (2006). *Essential issues in SOC design: Designing complex systems-on-chip.* doi: 10.1007/1-4020-5352-5

Malik, S., y Dhall, S. (2012). Implementation of MAC unit using booth multiplier and ripple carry adder. *Int. J. Appl. Eng. Res.*.

Martin, G., y Smith, G. (2009). High-Level Synthesis: Past, Present, and Future. *IEEE Des. Test Comput.*. doi: 10.1109/MDT.2009.83

Mathworks. (2018). *HDL Coder - MATLAB & Simulink.* Descargado 2019-01-19, de `https://www.mathworks.com/products/hdl-coder.html`

Meeus, W., Van Beeck, K., Goedemé, T., Meel, J., y Stroobandt, D. (2012). *An overview of today's high-level synthesis tools.* doi: 10.1007/s10617-012-9096-8

Moorthy, P., y Kapre, N. (2015). Zedwulf: Power-performance tradeoffs of a 32-node Zynq SoC cluster. En *Proc. - 2015 ieee 23rd annu. int. symp. field-programmable cust. comput. mach. fccm 2015.* doi: 10.1109/FCCM.2015.37

Nane, R., Sima, V. M., Pilato, C., Choi, J., Fort, B., Canis, A., ... Bertels, K. (2016). A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Trans. Comput. Des. Integr. Circuits Syst.*. doi: 10.1109/TCAD.2015.2513673

Navas, B., Öberg, J., y Sander, I. (2013). Towards the generic reconfigurable accelerator:

Algorithm development, core design, and performance analysis. *Reconfigurable Comput. FPGAs (ReConFig), Int. Conf.*, 1–6. doi: 10.1109/ReConFig.2013.6732334

Nikolov, H., Thompson, M., Stefanov, T., Pimentel, A., Polstra, S., Bose, R., ... Deprettere, E. (2008). Daedalus: Toward composable multimedia MP-SoC design. En *Proc. - des. autom. conf.* doi: 10.1109/DAC.2008.4555882

Pankratius, V., Adl-Tabatabai, A.-R., y Tichy, W. (2012). *Fundamentals of Multicore Software Development.* doi: S1568-7864(09)00102-5[pii]\r10.1016/j.dnarep.2009.04.023

Park, S. J., Shires, D. R., y Henz, B. J. (2008). Coprocessor computing with FPGA and GPU. En *2008 proc. dep. def. high perform. comput. mod. progr. users gr. conf. - solving hard probl.* doi: 10.1109/DoD.HPCMP.UGC.2008.69

Podobas, A., Zohouri, H. R., Maruyama, N., y Matsuoka, S. (2017). Evaluating high-level design strategies on FPGAs for high-performance computing. En *2017 27th int. conf. f. program. log. appl. fpl 2017.* doi: 10.23919/FPL.2017.8056760

Qin, S., y Berekovic, M. (2015). A Comparison of High-Level Design Tools for SoC-FPGA on Disparity Map Calculation Example. *Proc. 2nd Int. Work. FPGAs Softw. Program.*.

Reagen, B., Adolf, R., Shao, Y. S., Wei, G. Y., y Brooks, D. (2014). MachSuite: Benchmarks for accelerator design and customized architectures. En *Iiswc 2014 - ieee int. symp. workload charact.* doi: 10.1109/IISWC.2014.6983050

Rodríguez, A.; Valverde, J.; Portilla, J.; Otero, A.; Riesgo, T. d. l. T. (2018). E. FPGA-Based High-Performance Embedded Systems for Adaptive Edge Computing in Cyber-Physical Systems: The ARTICo3 Framework. *MDPI*. doi: 10.3390/s18061877

Roweis, S. (2010). *Data for MATLAB hackers.* Descargado 2019-01-06, de `https://cs.nyu.edu/{~}roweis/data.html`

Sinha, R., Roop, P., Salcic, Z., y Basu, S. (2014). Correct-by-construction multi-component SoC design. En *Proc. -design, autom. test eur. date.* doi: 10.1109/DATE.2012.6176551

Tcl Developer Xchange. (2018). *Tcl Developer Xchange.* Descargado 2019-01-10, de `https://www.tcl.tk/`

Thorwartl, P. (2017). *Basic Embedded System Design Tutorial: Creating the Software Platform Using SDK.* Descargado 2018-12-20, de https://www.so-logic.net/documents/knowledge/tutorial/Basic{_}ESD{_}Tutorial/sec2.html

Vinet, L., y Zhedanov, A. (2009). Electronic Design Automation: Synthesis, Verification, and Test. *Antimicrob. Agents Chemother.*. doi: 10.1088/1751-8113/44/8/085201

Weber, J. M., y Chin, M. J. (2006). Using FPGAs with Embedded Processors for Complete Hardware and Software Systems *. *Am. Inst. Phys.*, 187–192. doi: 10.1063/1.2401404

Windh, S., Ma, X., Halstead, R. J., Budhkar, P., Luna, Z., Hussaini, O., y Najjar, W. A. (2015). High-level language tools for reconfigurable computing. *Proc. IEEE*. doi: 10.1109/JPROC.2015.2399275

Wolf, W., Jerraya, A. A., y Martin, G. (2008). Multiprocessor system-on-chip (MPSoC) technology. *IEEE Trans. Comput. Des. Integr. Circuits Syst.*. doi: 10.1109/TCAD.2008.923415

Xilinx. (2014). *Vivado HLS: Debug Guide for investigating C/RTL co-simulation issues* (Inf. Téc.). Descargado de https://www.xilinx.com/Attachment/AR61063{_}VHLS{_}cosim{_}debug.pdf

Xilinx - DS190. (2017). *Zynq-7000 All Programmable SoC Data Sheet: Overview.* Descargado de www.xilinx.com

Xilinx - UG1145. (2018). *Xilinx Software Development Kit (SDK)* (Vol. 1145) (n.º 12). Descargado de http://www.xilinx.com/tools/sdk.htm

Xilinx - UG1291. (2018). *Vivado Isolation Verifier UG1291* (Vol. 1291). Descargado de www.xilinx.com

Xilinx - UG761. (2011). *AXI Reference Guide UG761* (Vol. 761). Descargado de www.xilinx.com

Xilinx - UG835. (2018). *Vivado Design Suite Reference Guide* (Vol. 958). Descargado de www.xilinx.com

Xilinx - UG871. (2017). *Vivado Design Suite Tutorial: HLS UG871* (Vol. 986). Descargado

de www.xilinx.com

Xilinx - UG894. (2016). *Vivado Design Suite User Guide, Using Tcl Scripting* (Vol. 2). Descargado de http://www.xilinx.com/support/documentation/sw{_}manuals/xilinx2015{_}4/ug903-vivado-using-constraints.pdf

Xilinx - UG902. (2017). *Vivado Design Suite User Guide: High-Level Synthesis.* Descargado de www.xilinx.com

Xilinx - UG994. (2017). *Designing IP Subsystems Using IP Integrator* (Vol. 4). Descargado de http://www.xilinx.com/support/documentation/sw{_}manuals/xilinx2015{_}4/ug903-vivado-using-constraints.pdf doi: 10.4103/2249-4863.174265

Xilinx Inc. (2018). *Zynq-7000 SoC.* Descargado 2018-12-20, de https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

Xilinx Wiki. (2018). *Build Device Tree Blob.* Descargado 2018-12-20, de https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842279/Build+Device+Tree+Blob{#}BuildDeviceTreeBlob-Microprocessorsoftwarespecification(MSS)filehttps://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842279/Build+Device+Tree+Blob

Zhou, B., Egele, M., y Joshi, A. (2017). High-performance low-energy implementation of cryptographic algorithms on a programmable SoC for IoT devices. En *2017 ieee high perform. extrem. comput. conf. hpec 2017.* doi: 10.1109/HPEC.2017.8091062

Zhou, S., Jiang, W., y Prasanna, V. (2014). A programmable and scalable openflow switch using heterogeneous soc platforms. *Proc. third Work. Hot Top. Softw. Defin. Netw. - HotSDN '14*. doi: 10.1145/2620728.2620767

Zohouri, H. R., Maruyamay, N., Smith, A., Matsuda, M., y Matsuoka, S. (2017). Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs. En *Int. conf. high perform. comput. networking, storage anal. sc.* doi: 10.1109/SC.2016.34

Zurawski, R. (2004). *Embedded Systems Handbook.* CRC Press, Inc.