UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Departamento de Señales y Comunicaciones

IDeTIC®
Instituto para el Desarrollo Tecnológico y la Innovación en Comunicaciones

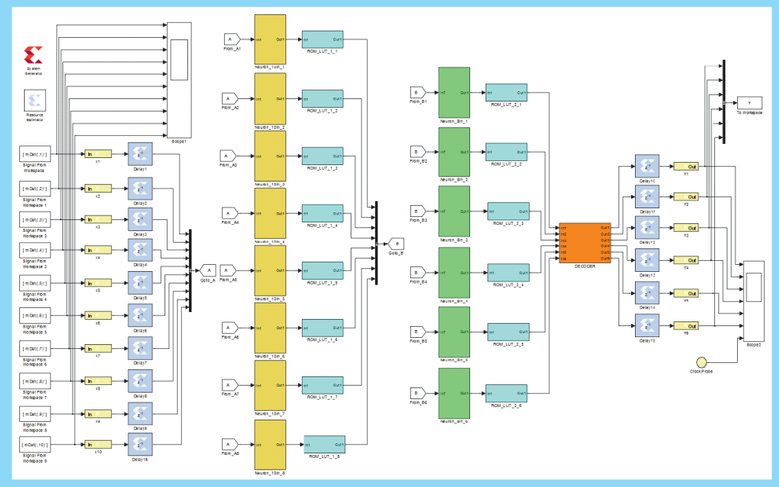
METODOLOGÍAS DE DISEÑO DE REDES NEURONALES SOBRE DISPOSITIVOS DIGITALES PROGRAMABLES PARA PROCESADO DE SEÑALES EN TIEMPO REAL *Santiago Tomás Pérez Suárez*

TESIS DOCTORAL

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
DEPARTAMENTO DE SEÑALES Y COMUNICACIONES

METODOLOGÍAS DE DISEÑO DE REDES NEURONALES SOBRE DISPOSITIVOS DIGITALES PROGRAMABLES PARA PROCESADO DE SEÑALES EN TIEMPO REAL

Santiago Tomás Pérez Suárez



Directores: Dr. Carlos Manuel Travieso González
Dr. Jesús Bernardino Alonso Hernández

Las Palmas de Gran Canaria, agosto de 2015

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Departamento de Señales y Comunicaciones

Anexo I

DON PEDRO JOSE QUINTANA MORALES SECRETARIO DEL DEPARTAMENTO DE SEÑALES Y COMUNICACIONES DE LA UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA,

CERTIFICA,

Que la Comisión de Investigación del Departamento (que tiene delegadas las competencias en materia de doctorado), en su sesión de fecha diecisiete de julio de dos mil quince, tomó el acuerdo de dar el consentimiento para su tramitación, a la tesis doctoral titulada "METODOLOGÍAS DE DISEÑO DE REDES NEURONALES SOBRE DISPOSITIVOS DIGITALES PROGRAMABLES PARA PROCESADO DE SEÑALES EN TIEMPO REAL" presentada por el doctorando Don Santiago Tomás Pérez Suárez y dirigida por los Doctores Don Carlos Manuel Travieso González y Don Jesús Bernardino Alonso Hernández.

Y para que así conste, y a efectos de lo previsto en el Artº 6 del Reglamento para la elaboración, defensa, tribunal y evaluación de tesis doctorales de la Universidad de Las Palmas de Gran Canaria, firmo la presente en Las Palmas de Gran Canaria a, diecisiete de julio de dos mil quince.

t +34 928 451 265
f +34 928 451 279
e-mail secretaria@dsc.ulpgc.es
www.dsc.ulpgc.es
Edificio de Electrónica y Telecomunicaciones
Campus de Tafira
35017 Las Palmas de Gran Canaria

FIRMADO POR	FECHA FIRMA	ID. FIRMA
	17/07/2015 11:37:03	

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

DEPARTAMENTO DE SEÑALES Y COMUNICACIONES

PROGRAMA DE DOCTORADO

CIBERNÉTICA Y TELECOMUNICACIÓN

TESIS DOCTORAL

Metodologías de diseño de redes neuronales sobre dispositivos digitales programables para procesado de señales en tiempo real

AUTOR: Santiago Tomás Pérez Suárez

DIRECTORES: Dr. D. Carlos Manuel Travieso González
Dr. D. Jesús Bernardino Alonso Hernández

El Director **El Codirector** **El Doctorando**

Las Palmas de Gran Canaria, a 31 de agosto de 2015

A Gloria y Mary Paz por llenar nuestras vidas.

A Mame por convertirse en indispensable compañera de esta travesía.

A abuelo Paco, que ya está descansando.

A abuela Carmen.

A toda mi familia.

A todos mis amigos.

AGRADECIMIENTOS

Agradezco a mis tutores Carlos M. Travieso y Jesús B. Alonso por invitarme a trabajar con ellos, por sus aportaciones, por escuchar aventuras y desventuras; y sobre todo, por la confianza que depositan en mí.

También agradezco a José Luis Vásquez, de la Universidad de Costa Rica, el trabajo que realizamos juntos en los primeros diseños. Este agradecimiento lo hago extensivo a todo el personal de las universidades de Costa Rica, por su hospitalidad y apoyo: Juan Carlos Briceño, Daniel Briceño, Federico Bolaños y Javier Vásquez.

Debo agradecer al Departamento de Señales y Comunicaciones de esta universidad el apoyo humano, financiero e institucional. Este agradecieminto es también extensivo al Instituto para el Desarrollo Tecnológico y la Innovación en Comunicaciones.

Agradezco al técnico de laboratorio, José Cruz, la ayuda recibida para instalar un sistema operativo virtual, necesario para evitar un error en las herramientas de diseño.

Debo agradecer al profesor Luis Gómez su asesoramiento en la búsqueda de publicaciones y gestiones con la administración.

Expreso mi agradecimiento a la biblioteca de Telecomunicación, especialmente a Alfonso Canella, la diligencia en la compra de libros.

Debe agradecerse a las empresas Xilinx y Altera la donación de licencias para el uso de sus programas de diseño, la respuesta de sus gestores ante la aparición de errores y el permiso de uso de sus gráficos para la redacción de la memoria de esta tesis. En particular se agradece a Altera la donación de diferentes placas con dispositivos digitales programables.

A la empresa MathWorks, propietaria de Matlab, se agradece el permiso para usar las figuras de su documentación para la redacción de la memoria de la tesis.

Por último, que no menos importante, a Bhavish Chandnani le agradezo la ayuda para elegir la regla de oro en la clasificación de pulsos electrocardiográficos.

PRÓLOGO

"… y sé breve en tus razonamientos, que ninguno hay gustoso si es largo".

El Ingenioso Hidalgo Don Quijote de la Mancha
Miguel de Cervantes

ÍNDICE

1. INTRODUCCIÓN 1

1.1 Motivación y justificación 2
1.2 Antecedentes y estado del arte 5
1.3 Objetivos de la tesis 16
1.4 Metodología 16
1.5 Contribuciones y resultados 19
1.6 Estructura de la memoria 24

2. IMPLEMENTACIÓN DE REDES NEURONALES EN TIEMPO REAL 27

2.1 Las redes neuronales 28
2.2 La variabilidad en el diseño de las redes neuronales 33
2.3 Los tipos de aritmética binaria 34
2.3.1 Paso del modelo de punto flotante a punto fijo: la regla de oro 36
2.4 Las tecnologías disponibles 37
2.5 Los métodos de diseño para FPGA 41
2.5.1 Edición de esquemáticos 41
2.5.2 Los lenguajes de descripción hardware 42
2.5.3 Lenguajes de alto nivel 44
2.5.4 Los entornos de los fabricantes 45
2.5.5 Xilinx versus Altera 47
2.5.6 Otras empresas y entornos académicos 49
2.5.7 Matlab para FPGA 50
2.6 Parámetros de los métodos de diseño 52
2.7 Factor de calidad de un diseño: área, velocidad y potencia 55
2.8 Las herejías metodológicas 60

3. METODOLOGÍA EXPERIMENTAL 63

3.1 Esquema general de la metodología experimental 64

3.2 Las bases de datos 66

3.2.1 La palmera pejibaye 67

3.2.2 Pulsos electrocardiográficos 68

3.2.3 Temperatura 71

3.2.4 Señal binaria con ruido 71

3.2.5 Características de las bases de datos 72

3.3 Preprocesado y parametrización 73

3.4 Modelado en punto flotante 73

3.5 Modelado en punto fijo 76

3.6 Flujo de diseño para la FPGA 78

3.7 Herramienta y flujo de diseño 81

3.8 El entorno de diseño de Xilinx 84

3.8.1 System Generator 85

3.8.2 Integrated System Environment 86

3.9 El entorno de diseño de Altera 87

4. EXPERIMENTOS Y RESULTADOS 89

4.1 Introducción 90

4.2 La clasificación de la palmera pejibaye 90

4.2.1 Modelado en punto flotante 90

4.2.2 Diseño en punto fijo 92

4.2.2.1 Diseño con System Generator 92

4.2.2.2 Implementación con Integrated System Environment 105

4.3 La clasificación de los pulsos electrocardiográficos 112

4.3.1 Modelado en punto flotante 112

4.3.2 Diseño en punto fijo 113

4.3.2.1 Diseño con System Generator 113

4.3.2.2 Implementación con Integrated System Environment 126

4.4 La predicción de temperatura 132

4.4.1 Modelado en punto flotante 132
4.4.2 Diseño en punto fijo 135
4.4.2.1 Diseño con System Generator 135
4.4.2.2 Implementación con Integrated System Environment 140
4.5 El ecualizador para señal binaria 144
4.5.1 Modelado en punto flotante 144
4.5.2 Diseño en punto fijo 144
4.5.2.1 Diseño con System Generator 144
4.5.2.2 Implementación con Integrated System Environment 149
4.6 Comparación con el estado del arte 152

5. CONCLUSIONES Y LÍNEAS FUTURAS 155

5.1 Conclusiones 156
5.2 Líneas futuras 158

ANEXO. Modelado en punto flotante del ecualizador 161

A.1 Introducción 161
A.2 Modelado en punto flotante de la TDNN ecualizadora 162

BIBLIOGRAFÍA 181

ÍNDICE DE FIGURAS

Figura 1.1. Apariciones por año en el Web of Science de: a)“filter”, b) “neural network”, c) “filter” and (“design” or “implementation” or “hardware”), d) “neural network” and (“design” or “implementation” or “hardware”). 4

Figura 1.2. Modelado del sistema en punto flotante. 18

Figura 1.3. Modelado del sistema en punto fijo. 19

Figura 1.4. Total de aportaciones anuales. 20

Figura 2.1. Estructura de una neurona nerviosa. 28

Figura 2.2. Estructura de una neurona artificial. 29

Figura 2.3. Red neuronal con R entradas conectadas a una capa de S neuronas. 31

Figura 2.4. Red neuronal con R^1 entradas, una capa intermedia de S^1 neuronas y una capa de salida de S^2 neuronas. 31

Figura 2.5. Red neuronal realimentada o recurrente. 32

Figura 2.6. Obtención del algoritmo en punto fijo a partir del algoritmo en punto flotante. 37

Figura 2.7. Arquitectura de una FPGA. 38

Figura 2.8. Prestaciones frente al coste económico para las diferentes tecnologías en el desarrollo de un prototipo. 41

Figura 2.9. Representación en semiejes del área, potencia y retardo. 58

Figura 2.10. Representación en un prisma del área, potencia y retardo. 59

Figura 2.11. Representación en un diagrama en tela de araña del área, potencia y retardo. 59

Figura 2.12. Las herejías metodológicas y el proceso apropiado. 61

Figura 3.1. Esquema general de la metodología experimental. 65

Figura 3.2. Confluencia de conceptos en la metodología experimental. 66

Figura 3.3. Tipos de pulso elegidos para la detección. 69

Figura 3.4. Temperatura de la primera semana de la base de datos y su valor medio. 71

Figura 3.5. a) Señal binaria original, b) señal con una relación señal a ruido de +10 dB. 72

Figura 3.6. Modelado detallado de la NN en punto flotante. 75

Figura 3.7. Modelado detallado de la NN en punto fijo. 77

Figura 3.8. Flujo de diseño para la FPGA. 79

Figura 3.9. El escenario de diseño de Xilinx para System Generator. 84

Figura 3.10. Interacción entre System Generator, Simulink, Matlab y el ISE de Xilinx. . 86

Figura 3.11. El escenario de diseño de Altera para DSP Builder. 87

Figura 4.1. Ventana obtenida al finalizar el entrenamiento de la NN para pejibaye..... 91

Figura 4.2. Matriz de confusión obtenida en la etapa de testeo de la NN para pejibaye. 92

Figura 4.3. Etapa de entrada de la NN para pejibaye. 93

Figura 4.4. Ventana de configuración de un Gateway In. 94

Figura 4.5. Capa intermedia de la NN para pejibaye. 95

Figura 4.6. Primera fase de la primera neurona de la capa oculta para pejibaye. 96

Figura 4.7. Implementación de la función de transferencia logsig mediante una ROM.97

Figura 4.8. a) Función logsig, b) salida de la función implementada, c) error y d) error relativo. 98

Figura 4.9. Capa de salida de la NN para pejibaye. 99

Figura 4.10. Primera fase de la primera neurona de la capa de salida para pejibaye. 100

Figura 4.11. Bloque decodificador de las neuronas de salida para pejibaye. 100

Figura 4.12. Conexión de las salidas del decodificador a los registros de salida y a los pines de salida de la FPGA para pejibaye. 101

Figura 4.13. Diseño en Simulink del sistema final para pejibaye. 102

Figura 4.14. Señales de entrada en la NN para pejibaye..... 103

Figura 4.15. Señales de salida en la NN para pejibaye. 103

Figura 4.16. Bloque System Generator y su ventana de configuración. 104

Figura 4.17. Ventana que se muestra al finalizar la compilación con System Generator. 104

Figura 4.18. Simulación de la implementación para pejibaye en la FPGA, 8% de error y 4 palabras en las ROM. 106

Figura 4.19. Etapa de entrada de la NN para ECG. 113

Figura 4.20. Capa intermedia de la NN para ECG. 115

Figura 4.21. Agrupación de diez neuronas en la capa oculta de la NN para ECG. 116

Figura 4.22. Primera fase de la primera neurona de la capa oculta para ECG..... 117

Figura 4.23. Implementación de la función de transferencia tansig mediante una ROM. 118

Figura 4.24. a) Función tansig, b) salida de la función implementada, c) error y d) error relativo. 118

Figura 4.25. Capa de salida de la NN para ECG. 119

Figura 4.26. Primera fase de la primera neurona de la capa de salida para ECG. 120

Figura 4.27. Primer agrupamiento de diez multiplicadores de la primera neurona de la capa de salida para ECG..... 121

Figura 4.28. Bloque decodificador de las neuronas de salida para ECG. 122

Figura 4.29. Conexión de las salidas del decodificador a los registros de salida y a los pines de salida de la FPGA para ECG. 122

Figura 4.30. Diseño en Simulink del sistema final para ECG. 123

Figura 4.31. Señales de entrada en la NN para ECG. 124

Figura 4.32. Señales de salida en la NN para ECG. 125

Figura 4.33. Simulación para la fase final de la implementación en la FPGA para 4,2% de error y 64 palabras en la ROM para ECG. 127

Figura 4.34. Detalle de los primeros 2.500 ns de la simulación en la FPGA para 4,2% de error y 64 palabras en la ROM para ECG. 128

Figura 4.35. Red neuronal con línea de retardo. 133

Figura 4.36. Ventana obtenida en el entrenamiento con el Neural Network Time Series Tool. 134

Figura 4.37. Implementación de la TDNN para predicción de la temperatura en la FPGA. 136

Figura 4.38. Neurona de salida en el predictor de temperatura con resolución completa. 137

Figura 4.39. Neurona de salida en el predictor de temperatura con resolución ajustada de forma manual. 138

Figura 4.40. Simulación de los siete primeros días de 2009 con System Generator en el predictor de temperatura: muestras en la entrada (azul), estimación en la salida (rojo) y señal de error (negro). 139

Figura 4.41. Simulación del predictor de temperatura después del colocado y conexionado de los componentes en la FPGA para 0,52% de error y 1024 palabras en la ROM. 140

Figura 4.42. Simulación del predictor de temperatura después del colocado y conexionado de los componentes en la FPGA para 0,56% de error y 2048 palabras en la ROM. 142

Figura 4.43. Ecualizador diseñado con System Generator. 145

Figura 4.44. Diseño de la función satlin con el uso de un multiplexor. 146

Figura 4.45. Formas de onda de Simulink en el ecualizador para los primeros 40 bits: a) señal de datos original, b) entrada en la FPGA, c) salida de la FPGA. 149

Figura 4.46. Formas de onda de las primeras quince muestras en el ecualizador después del colocado y conexionado de los componentes en la FPGA. 149

Figura A.1. Modelo propuesto para el ecualizador. 163

Figura A.2. Señal de datos original binaria (a) y señal con ruido muestreada (b)..... 163

Figura A.3. Señal de datos original (a), la señal con ruido muestreada (b) y la salida de la TDNN (c). 164

Figura A.4. Curva de SNR en la salida de la TDNN frente a la SNR en la entrada para el entrenamiento con +10 dB. 165

Figura A.5. Efecto del número de neuronas en la capa intermedia para el ecualizador. 166

Figura A.6. Ventana de entrenamiento en Matlab para el ecualizador. 169

Figura A.7. Ventana de entrenamiento en Matlab, con el algoritmo de entrenamiento finalmente usado, para el ecualizador. 173

Figura A.8. Efecto del tamaño de la ventana de observación en el ecualizador. 173

Figura A.9. Efecto de la SNR en el entrenamiento del ecualizador (de -5 dB a +20 dB). 174

Figura A.10. Efecto de la SNR en el entrenamiento del ecualizador (de +6 dB a +10 dB). 176

Figura A.11. Testeo de la SNR de salida frente a la SNR de entrada para una SNR de entrenamiento de +7 dB..... 177

Figura A.12. Curvas de entrenamientos existosos (a), curva promedio y límites de la banda de estabilidad (b). 179

ÍNDICE DE TABLAS

Tabla 1.1. Cantidad de resultados obtenidos en una búsqueda en Web of Science de "filter", "neural network" y "equalizer". 3

Tabla 1.2. Cantidad de resultados obtenidos en una búsqueda en Web of Science de "filter", "neural network" y "equalizer"; añadiendo "design or implementation or hardware". 4

Tabla 1.3. Contribuciones alcanzadas durante el desarrollo de esta tesis..... 20

Tabla 1.4. Participación en Proyectos de Investigación. 23

Tabla 1.5. Trabajos Fin de Título desarrollados. 24

Tabla 2.1. Funciones de transferencia típicas. 30

Tabla 3.1. Estructura de la base de datos para la palmera pejibaye..... 67

Tabla 3.2. Matriz de confusión obtenida en el testeo de punto flotante para ECG. 70

Tabla 3.3. Características de las bases de datos..... 72

Tabla 3.4. Uso del preprocesado y parametrización en los distintos escenarios. 73

Tabla 3.5. Forma de operar la NN y de evaluar la funcionalidad. 76

Tabla 4.1. Funcionalidad del sistema frente al error en la representación y el número de palabras en las ROM para pejibaye……105

Tabla 4.2. Recursos hardware para 8% de error y 4 palabras en la ROM, aplicado a pejibaye y compilado en VHDL. 106

Tabla 4.3. Potencias para 8% de error y 4 palabras en la ROM, aplicado a pejibaye y compilado en VHDL. 107

Tabla 4.4. Recursos hardware para 8% de error y 4 palabras en la ROM, aplicado a pejibaye y compilado en Verilog. 107

Tabla 4.5. Potencias para 8% de error y 4 palabras en la ROM, aplicado a pejibaye y compilado en Verilog. 108

Tabla 4.6. Recursos hardware necesarios para 12% de error y 8 palabras en la ROM, aplicado a pejibaye y compilado en VHDL. 108

Tabla 4.7. Potencias para 12% de error y 8 palabras en la ROM, aplicado a pejibaye y compilado en VHDL. 108

Tabla 4.8. Recursos hardware necesarios para 12% de error y 8 palabras en la ROM, aplicado a pejibaye y compilado en Verilog. 109

Tabla 4.9. Potencias para 12% de error y 8 palabras en la ROM, aplicado a pejibaye y compilado en Verilog. 109

Tabla 4.10. Recursos hardware necesarios para 13% de error y 128 palabras en la ROM, aplicado a pejibaye y compilado en VHDL. 109

Tabla 4.11. Potencias para 12% de error y 8 palabras en la ROM, aplicado a pejibaye y compilado en VHDL. 110

Tabla 4.12. Recursos hardware necesarios para 13% de error y 128 palabras en la ROM, aplicado a pejibaye y compilado en Verilog. 110

Tabla 4.13. Potencias para 13% de error y 128 palabras en la ROM, aplicado a pejibaye y compilado en Verilog. 111

Tabla 4.14. Resumen de los casos estudiados para pejibaye. 111

Tabla 4.15. Comparación en velocidad entre un ordenador personal y la FPGA para pejibaye. 112

Tabla 4.16. Matriz de confusión obtenida en el modelo en punto fijo para 4,2% de error y 64 palabras en las ROM para ECG. 125

Tabla 4.17. Funcionalidad del sistema frente al error en la representación y el número de palabras en la ROM para ECG. 126

Tabla 4.18. Recursos hardware necesarios para 4,2% de error y 64 palabras en la ROM, aplicado a ECG y compilado en VHDL. 128

Tabla 4.19. Potencias para 4,2% de error y 64 palabras en la ROM, aplicado a ECG y compilado en VHDL. 129

Tabla 4.20. Recursos hardware necesarios para 4,2% de error y 64 palabras en la ROM, aplicado a ECG y compilado en Verilog. 129

Tabla 4.21. Potencias para 4,2% de error y 64 palabras en la ROM, aplicado a ECG y compilado en Verilog. 129

Tabla 4.22. Recursos hardware necesarios para 4,7% de error y 512 palabras en la ROM ,aplicado a ECG y compilado en VHDL. 130

Tabla 4.23. Potencias para 4,7% de error y 512 palabras en la ROM, aplicado a ECG y compilado en VHDL. 130

Tabla 4.24. Recursos hardware necesarios para 4,7% de error y 512 palabras en la ROM, aplicado a ECG y compilado en Verilog. 130

Tabla 4.25. Potencias para 4,7% de error y 512 palabras en la ROM, aplicado a ECG y compilado en Verilog. 131

Tabla 4.26. Resumen de los casos estudiados para ECG. 131

Tabla 4.27. Comparación en velocidad entre un ordenador personal y la FPGA usada para ECG. 132

Tabla 4.28. Funcionalidad del predictor de temperatura en función del error permitido en la representación y el número de palabras en las memorias ROM. 139

Tabla 4.29. Recursos hardware necesarios para 0,52% de error y 1024 palabras en la ROM, predicción de temperatura y compilado en VHDL. 141

Tabla 4.30. Potencias para 0,52% de error y 1024 palabras en la ROM, , predicción de temperatura y compilado en VHDL. 141

Tabla 4.31. Recursos hardware necesarios para 0,52% de error y 1024 palabras en la ROM, predicción de temperatura y compilado en Verilog. 141

Tabla 4.32. Potencias para 0,52% de error y 1024 palabras en la ROM, predicción de temperatura y compilado en Verilog. 141

Tabla 4.33. Recursos hardware necesarios para 0,56% de error y 2048 palabras en la ROM, predicción de temperatura y compilado en VHDL. 142

Tabla 4.34. Potencias para 0,56% de error y 2048 palabras en la ROM, predicción de temperatura y compilado en VHDL. 142

Tabla 4.35. Recursos hardware necesarios para 0,56% de error y 2048 palabras en la ROM, predicción de temperatura y compilado en Verilog. 142

Tabla 4.36. Potencias para 0,56% de error y 2048 palabras en la ROM, predicción de temperatura y compilado en Verilog. 143

Tabla 4.37. Resumen de los casos estudiados para el predictor de temperatura. 143

Tabla 4.38. Comparación en velocidad entre un ordenador personal y la FPGA usada en la predicción de temperatura. 144

Tabla 4.39. Efecto del error de representación en el ecualizador. 148

Tabla 4.40. Recursos hardware necesarios en el ecualizador compilado en VHDL. 150

Tabla 4.41. Potencias en el ecualizador compilado en VHDL. 150

Tabla 4.42. Recursos hardware necesarios en el ecualizador compilado en Verilog. . 150

Tabla 4.43. Potencias en el ecualizador compilado en Verilog. 151

Tabla 4.44. Resumen de los casos estudiados para el ecualizador. 151

Tabla 4.45. Comparación en velocidad entre un ordenador personal y la FPGA usada para el ecualizador. 152

Tabla A.1. Dependencia de la curva SNRs-SNRe frente a las funciones de transferencia usadas en el ecualizador. 168

Tabla A.2. Curvas obtenidas en función del algoritmo de entrenamiento para el ecualizador. 170

Tabla A.3. Curvas obtenidas dependiendo de la función de error usada para el ecualizador. 172

ACRÓNIMOS

AHDL *Altera Hardware Description Language,* Lenguaje de Descripción Hardware de Altera.

ANN *Artificial Neural Network,* Red Neuronal Artificial.

ASIC *Application Specific Integrated Circuit,* Circuito Integrado de Aplicación Específica.

AWGN *Additive White Gaussian Noise,* Ruido Gaussiano Blanco Aditivo.

BIH *Beth Israel Hospital*, Hospital Beth Israel.

CMOS *Complementary Metal-Oxide-Semiconductor*, Metal-Óxido-Semiconductor en Simetría Complementaria.

CORDIC *Coordinate Rotation Digital Computer*, Rotación de coordenadas en computador digital.

CPU *Central Processing Unit,* Unidad Central de Procesamiento.

DNA *Deoxyribonucleic acid,* Ácido desoxirribonucleico.

DSP *Digital Signal Processor*, Procesador Digital de Señal.

DWT *Discrete Wavelet Transform,* Transformada discreta Wavelet.

ECG *Electrocardiogram*, Electrocardiograma.

EDIF *Electronic Design Interchange Format,* Formato de Intercambio de Diseño Electrónico.

FPAA *Field Programmable Analog Array,* Matriz analógica programable por el diseñador.

FPGA *Field Programmable Gate Array*, Matriz de puertas programables por el diseñador.

GPU *Graphics Processing Unit,* Unidad de Procesamiento Gráfico.

HDL *Hardware Description Language,* Lenguaje de Descripción Hardware.

HTM *Hypertext Markup Language*, Lenguaje de hipertexto marcado.

ICA *Independent Components Analisys*, Análisis de componentes independientes.

IEEE *Institute of Electrical and Electronics Engineers,* Instituto de Ingenieros Eléctricos y Electrónicos.

IP *Intellectual Property,* Módulo con Propiedad Intelectual.

ISE *Integrated System Environment,* Entorno de sistema integrado.

JHDL *Java Hardware Description Language,* Lenguaje de Descripción Hardware con Java.

LUT *Lookup-Table*, Tabla de búsqueda.

MAC *Multiplier-Accumulator*, Multiplicador-Acumulador.

mae *Mean absolute error,* Error absoluto medio.

MIT *Massachusetts Institute of Technology,* Instituto Tecnológico de Massachusetts.

MIT-BIH *Massachusetts Institute of Technology-Beth Israel Hospital*, Instituto Tecnológico de Massachusetts-Hospital Beth Israel.

mse *Mean squared error,* Error cuadrático medio.

NGC *Native Generic Database,* Base de Datos Genérica Nativa.

NN *Neural Network,* Red Neuronal.

NRZ *Non Return to Zero,* No retorno a cero.

OTP *One-time Programmable*, Programable una sola vez.

PCA *Principal Components Analysis*, Análisis de Componentes Principales.

PLD *Programmable Logic Device,* Dispositivo digital programable.

RAPD *Random Amplification of Polymorphic Deoxyribonucleic acid*, Amplificación aleatoria del ácido desoxirribonucleico polimórfico.

ROM *Read Only Memory*, Memoria de solo lectura.

sae *Sum absolute error,* Suma del error absoluto.

SNR *Signal to Noise Relation*, Relación señal a ruido.

SoC *System on Chip,* Sistema en un chip.

SOM *Self Organizing Map*, Mapa autoorganizado.

sse *Sum squared error*, Suma del error al cuadrado.

TDNN *Time Delay Neural Network,* Red Neuronal con Línea de Retardo.

USB *Universal Serial Bus*, Bus Universal Serie.

VHDL *Very High Speed Integrated Circuit Hardware Description Language,* Lenguaje muy rápido de descripción hardware para circuitos integrados.

VLSI *Very Large Scale Integration*, Sistema de muy alta escala de integración.

XST *Xilinx Synthesis Tool*, Herramienta de Síntesis de Xilinx.

CAPÍTULO 1

INTRODUCCIÓN

"Yo tengo un sueño …"

Martin Luther King

En este capítulo se justifican los motivos para la realización de esta tesis. En ella también se describen los antecedentes que existen del tema a tratar. Posteriormente se plantean los objetivos, el teorema propuesto, la metodología usada, y las contribuciones y resultados obtenidos. Finalmente, se expone la estructura de la memoria.

1.1 Motivación y justificación

La sociedad actual ha vivido en los últimos años la revolución tecnológica. En el año 1969 el hombre llegó a la Luna, este se ha convertido en un hito único en la historia. Este evento se basó en los avances tecnológicos desarrollados durante la Segunda Guerra Mundial y en el pulso mantenido por las potencias involucradas en la Guerra Fría. La nave Apollo 11 llevaba un computador a bordo con ocho operadores de cálculo y 64 kbytes de memoria. Su frecuencia de reloj era de unos 2 MHz, consumía 55 W, pesaba casi 32 kgr y ocupaba 33 dm^3. Casi cincuenta años después los computadores han mejorado enormemente sus prestaciones de cálculo, memoria, velocidad, potencia y dimensiones físicas. Además, no solo se han convertido en un electrodoméstico más del hogar, sino que han pasado a ser dispositivos portátiles y personales con multitud de funciones. En esta línea los dispositivos deben disminuir su volumen y peso, tanto de su circuitería como de su batería. Además deben mejorar su potencia y velocidad de cálculo para tratar eventos en tiempo real. Finalmente deben minimizar el consumo de potencia, para que la misma batería otorgue más autonomía; o que para la misma autonomía se pueda usar una batería de menor tamaño. En resumen, es clara y necesaria la mejora de las prestaciones físicas de área, velocidad y potencia.

La presente tesis pretende hacer aportaciones a las metodologías de diseño de redes neuronales (NN, *Neural Network*) para dispositivos digitales programables. Estas mejoras permitirán desarrollar NN capaces de operar en tiempo real para procesado digital de la señal. Los métodos de diseño permitirán la descripción rápida y flexible de estos sistemas. Del sistema final se extraerán sus prestaciones físicas: área, potencia y velocidad. Los dispositivos óptimos, para la realización de prototipos, son las matrices de puertas programables por el diseñador (FPGA, *Field Programmable Gate Array*). El ahorro de área y potencia consumida mejorará la portabilidad física del prototipo final.

Esta tesis se apoya en tres conceptos: las redes neuronales artificiales (ANN, *Artificial Neural Network*), los dispositivos digitales programables (PLD, *Programmable Logic Devices*) y el procesado de señal en tiempo real. El concepto "metodología de diseño" es el punto de unión para que converjan y se facilite la realización de prototipos. El concepto PLD incluye dispositivos antiguos, con pocos recursos

hardware; los actuales dispositivos han pasado a denominarse matrices de puertas digitales programables por el diseñador (FPGA, *Field Programmable Gate Array*). En el resto del documento, por simplificación, a las ANN se les llamará redes neuronales (NN, *Neural Networks*).

Antes de entrar en cuestiones técnicas y metodológicas, conviene resaltar un fenómeno observado en la base de datos del *Web of Science* [Web of Science, 2014]. Si se buscan por título los conceptos de la columna izquierda de la tabla 1.1, se obtienen las cantidades que indica la columna de la derecha. Esta búsqueda se realizó en los títulos de las aportaciones en las bases de datos relacionadas con las tecnologías de la electrónica, la computación y las comunicaciones. Los conceptos "*filter"* y "*equalizer"* son bloques ampliamente usados en procesado y comunicaciones. Solo el concepto "*filter"* supera ampliamente a "*neural network"*, no debe olvidarse que los filtros son bloques de uso obligado en casi todos los sistemas. Esto pone de manifiesto la importancia relativa que el concepto de red neuronal tiene para la comunidad científica.

Tabla 1.1. Cantidad de resultados obtenidos en una búsqueda en Web of Science de "filter", "neural network" y "equalizer".

Filter	138.939
Neural Network	84.868
Equalizer	3.419

Si a la búsqueda se añade las palabras *"design"*, *"implementation"* y *"hardware"* se obtienen los valores de la tabla 1.2. En proporción se ha estudiado y consolidado mucho más el diseño de filtros y ecualizadores que de redes neuronales. Por ejemplo, para el diseño de filtros se han desarrollado extensamente metodologías y herramientas de diseño. Esto no sucede para las redes neuronales por las características intrínsecas de estos sistemas, que se describirán posteriormente.

Tabla 1.2. Cantidad de resultados obtenidos en una búsqueda en Web of Science de "filter", "neural network" y "equalizer"; añadiendo "design or implementation or hardware".

Filter *and* (design *or* implementation *or* hardware)	13.999	10,1%
Neural network *and* (design *or* implementation *or* hardware)	3.937	4,6%
Equalizer *and* (design *or* implementation *or* hardware)	457	13,4%

En la figura 1.1 (izquierda) se muestra el número de apariciones en la búsqueda del *Web of Science* en función del año, para *"filter"* (a) y *"neural network"* (b); con trazo fino y grueso respectivamente. En ella se observa que los estudios sobre redes neuronales aparecen mucho más tarde que para los filtros. Además, justo antes de 1990 se produce el inicio de los estudios para redes neuronales, y alcanzan en 10 años similares niveles de producción. En la figura 1.1 (derecha) se observan los resultados de la búsqueda cuando se añaden las palabras *"design"*, *"implementation"* y *"hardware"*; para *"filter"* con trazo fino (c), y para *"neural network"* con trazo grueso (d). En esta figura se observa que el concepto de diseño e implementación de redes neuronales arranca justo antes de 1990 pero no alcanza los niveles de los diseños de filtros.

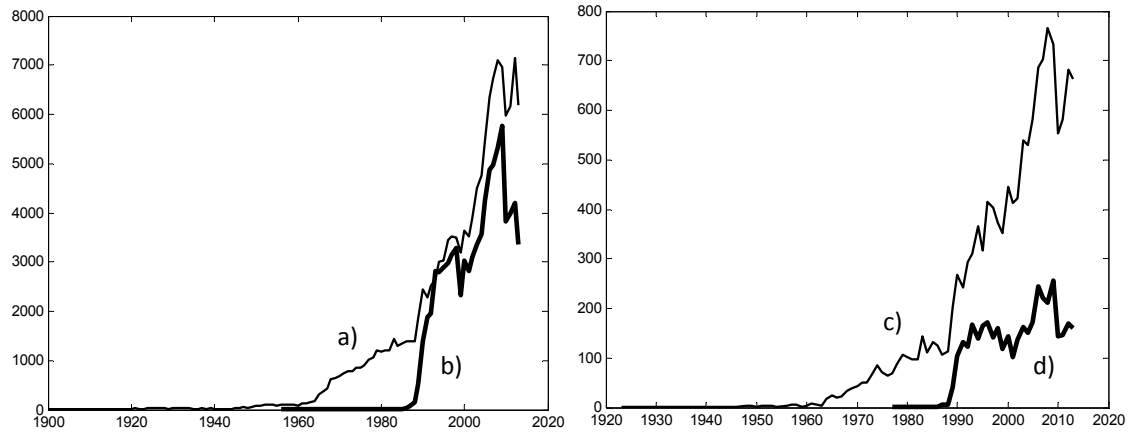


Figura 1.1. Apariciones por año en el Web of Science de: a)"filter", b) "neural network", c) "filter" and ("design" or "implementation" or "hardware"), d) "neural network" and ("design" or "implementation" or "hardware").

Se puede llegar a la conclusión, observando los datos de las búsquedas anteriores, de que existe un retraso en la implementación de las redes neuronales frente al diseño de filtros. Esto puede deberse a la escasa necesidad de altas prestaciones físicas en la

operación de las NN, a la dificultad intrínseca de las NN y a la improvisación al realizar estos diseños sin el uso de métodos apropiados.

La necesidad de mejorar las prestaciones depende del escenario de la aplicación; y de las restricciones de velocidad, consumo de potencia y de tamaño en la implementación. La dificultad intrínseca de las NN y su variabilidad debe ser analizada por el diseñador, quien debe tomar las decisiones apropiadas. Estas decisiones se basan en la experiencia del diseñador o en un proceso de prueba y error. Por último, debe resaltarse que los métodos existentes son múltiples y con diferentes características. El diseñador elegirá el método apropiado en función de sus características. Todo lo anterior se analizará en capítulos posteriores.

1.2 Antecedentes y estado del arte

Es preciso por tanto analizar el estado actual de las implementaciones de las NN desde diferentes puntos de vista. Para ello se describen las aportaciones más significativas, desde la más antigua hasta la más reciente. Los diseños pueden no incluir el entrenamiento, que se hace previamente, normalmente en una computadora usando aritmética de punto flotante; se dicen que tienen un entrenamiento *offline*. Cuando los circuitos incluyen las estructuras de entrenamiento se dicen que es de tipo *online*, estos últimos se reentrenan cada cierto tiempo y reconfiguran, adaptándose a circunstancias cambiantes. Se entenderá, por defecto, que las aportaciones son de tipo *offline*; cuando sean de tipo *online* se indicará expresamente. En los casos descritos a continuación los desarrollos se llevaron a cabo para FPGA, en caso contrario se indica la tecnología que se usó. Como regla general, si no se especifica otra cosa, la función de transferencia usada es de tipo sigmoidea, que es la usada habitualmente como función de transferencia en las neuronas artificiales.

Como se observa en la figura 1.1 (d), existen diseños de NN sobre dispositivos electrónicos desde justo antes de 1990. En [Myers y Hutchinson, 1989] se plantea un método de síntesis para la función sigmoidea. Estos autores usan una aproximación lineal por tramos y plantean la posibilidad de diseñarla como un sistema digital con muy alta escala de integración (VLSI, *Very Large Scale Integration*). Se usó aritmética

en punto fijo de complemento a dos y se diseñó usando circuitos esquemáticos. El sistema fijó 16 bits de entrada y 8 bits de salida, pero no se estudió el efecto de este número de bits sobre la cantidad de hardware requerido. Los autores no muestran tampoco estimación de velocidad ni de potencia consumida.

En [Satyanarayana et al., 1992] se diseñó una NN de tres capas en un sistema VLSI analógico. El sistema se diseñó en forma de esquemático y los autores presentan el hardware estimado y la velocidad, pero no la estimación de potencia consumida.

En agosto del año 1994 se desclasifica el informe [Tebbe, 1994], que se realizó entre septiembre de 1992 y mayo de 1994. Este informe fue realizado por *Harris Corporation* para el *Government Communications Systems Division* de los Estados Unidos; trata de las NN para procesado de señal en comunicaciones, en particular para la ecualización y demodulación de señales digitales.

En [Costa et al., 1999] se diseñó una NN con tecnología VLSI analógica. Se describió en forma de esquemático, se da la estimación de área y potencia; pero no la de velocidad.

En el trabajo de [Vassiliadis et al., 2000] se proponen varios métodos para la implementación de funciones no lineales, se incluyen las funciones sigmoideas. La aritmética usada es en punto fijo en complemento a uno y complemento a dos. Los métodos usados son por tramos lineales y por tramos de segundo orden. Este estudio se centra en el valor medio del error en valor absoluto y el error máximo de la representación; considera que los valores máximos deben ser 10^{-3} y 10^{-2} respectivamente. A partir de estas referencias compara el área necesaria y el retardo en ciclos de reloj. No se da estimación de potencia. El número de bits es fijo, cuatro para la parte entera y diez para la parte decimal. Las soluciones propuestas no se implementaron; solo se bosquejó el área y los ciclos de reloj necesarios. Obviamente, esto impide la estimación de potencia y la máxima frecuencia de funcionamiento. Además, se propuso la arquitectura para la derivada primera de la función de transferencia, con el objeto de poder realizar un entrenamiento *online* de la NN.

En [Krips, 2002] se diseñó una NN con una sola capa intermedia, se implementó sobre una FPGA para seguimiento de la imagen de una mano en tiempo real. La

aritmética usada es de punto fijo y como función de transferencia se usó la tangente hiperbólica, que se implementó almacenando 15 niveles de cuantificación en una memoria usada como tabla de búsqueda (LUT, *Lookup-Table*). Se diseñó usando el lenguaje muy rápido de descripción hardware para circuitos integrados (VHDL, *Very High Speed Integrated Circuit Hardware Description Language*). Los autores indican la tasa de acierto obtenida, la velocidad y el área necesaria; pero no la potencia consumida.

En [Lee y Burgess, 2003] se aproxima la función de transferencia usando la serie de Taylor sobre una FPGA. Se empleó aritmética de punto fijo y se describió con VHDL. Los autores presentan el efecto del número de bits sobre el área ocupada y el retardo; pero no presentan el efecto sobre la potencia consumida.

Otros autores, como [Manjunath y Gurumurthy, 2003], plantean el diseño de una NN sobre una matriz analógica programable por el diseñador (FPAA, *Field Programmable Analog Array*). Para la función de transferencia se propone un circuito con amplificador operacional. Este trabajo no presenta estimaciones de área, velocidad o consumo de potencia.

En [Tommiska, 2003] se estudian implementaciones eficientes de la función de transferencia sigmoidea usando aritmética de punto fijo. El autor diseñó diferentes formas de la aproximación lineal por tramos, aproximación por tramos con polinomios de segundo orden, usando circuitos combinacionales y LUT. Los diseños se describieron con VHDL. En esta aportación se estima el área y la velocidad de los diferentes diseños, pero no se estimó la potencia. La funcionalidad de las implementaciones se analizó con el error máximo y el valor medio del error absoluto. El autor propuso la fórmula 1.1 como factor de calidad, en ella se mezcla la funcionalidad con las prestaciones físicas. Este factor de calidad, aunque válido, compara sistemas con diferente error en la representación; además no incluye la potencia. Sería mejor comparar las prestaciones físicas (área, velocidad y potencia) en sistemas de similar funcionalidad.

$$Q = \frac{f_{max}}{\text{Área} \cdot \text{Error}_{\text{máximo}} \cdot \text{Error}_{\text{medio}}} \qquad (1.1)$$

En [Zhu y Sutton, 2003] se hace una revisión de la última década en el diseño de NN sobre FPGA. En este trabajo los autores inciden que en el futuro se debe optimizar la precisión binaria; además, deberá usarse métodos de diseños apropiados para comparar diferentes implementaciones.

Se aproximó la función sigmoidea y su primera derivada en [Basterretxea et al., 2004]. Esto se hizo con una aproximación por tramos lineales y un algoritmo iterativo. Se implementó con VHDL en aritmética de punto fijo. Los autores dan el error del sistema en función de los parámetros del algoritmo usado; pero no dan las prestaciones físicas de área, velocidad o potencia.

Los autores [Dias et al., 2004] revisaron los circuitos integrados específicos que se vendían hasta ese año como NN. Estos tenían una utilidad restringida por las limitaciones en su arquitectura. Por un lado, aparece limitado el número de neuronas, de entradas, y de salidas; por otro, la resolución binaria usada es fija.

En [Oh y Jung, 2004] se implementó una NN sobre una Unidad de Procesamiento Gráfico (GPU, *Graphics Processing Unit*). Los autores hacen uso de las operaciones con matrices, principalmente la multiplicación. En este desarrollo se compara la velocidad obtenida con la GPU frente a una unidad central de procesamiento (CPU, *Central Processing Unit*), el diseño en la GPU resultó ser unas 20 veces más rápido. Conviene recordar que las GPU son dispositivos que usan aritmética en punto flotante.

El diseño de un ecualizador sobre FPGA se expone en [Yen et al., 2004]. Se probaron dos arquitecturas: una sola neurona y una configuración con solo las capas de entrada y salida. Se diseñó para una señal digital modulada en fase de cuatro símbolos. Los autores analizaron la tasa de error de símbolo frente a la relación señal a ruido (SNR, *Signal to Noise Relation*), y frente al número de bits usado. Se usó aritmética de punto fijo y las funciones de transferencia se diseñaron usando una LUT. El sistema se describió usando el lenguaje de descripción hardware Verilog. Los autores muestran estimaciones de área y retardo; pero no de potencia consumida. Los circuitos incluyen el entrenamiento *online*.

En el trabajo fin de grado de [Lange, 2005] se bosquejan diferentes arquitecturas para las NN según el nivel de secuenciamiento en el cálculo. Se plantea: la NN

totalmente paralelizada, una NN donde para cada neurona se dedica un multiplicador-acumulador (MAC, *Multiplier-Accumulator*), un MAC por capa y un solo MAC para toda la NN. El autor finalmente elige la solución de un MAC por capa. Se usó aritmética de punto fijo; pero no se estudió el efecto del número de bits, se estableció el número de bits siguiendo una estrategia conservadora. Las funciones de transferencia se diseñaron mediante LUT. Para el diseño se usó una herramienta automática que genera el código VHDL. El autor da estimaciones de área y velocidad; pero no de potencia consumida.

En [Chen et al., 2006] se propone el diseño de la función sigmoidea usando el algoritmo CORDIC (*Coordinate Rotation Digital Computer*). Se usó aritmética de punto flotante con 32 bits de entrada y 64 de salida. Para el diseño se usó VHDL. Los autores estimaron la velocidad, pero no el área ni la potencia consumida.

Una NN se diseñó en [Dong et al., 2006] usando varias FPAA. La función de transferencia se aproximó por tramos lineales. Los autores estimaron la velocidad y el área ocupada, pero no la potencia consumida.

Los autores en [Larkin et .al, 2006] generaron la función de activación usando aproximaciones polinómicas por tramos usando aritmética en punto fijo. Como formato de la entrada se usó cuatro bits para la parte entera y diez para la parte decimal. Igualmente para la salida se usaron dos bits para la parte entera y diez para la parte decimal. Se diseñó la propuesta usando Verilog, y se compiló el diseño para un circuito integrado de aplicación específica (ASIC, *Application Specific Integrated Circuit*) y una FPGA. Los autores extrajeron las prestaciones en área, velocidad y potencia consumida.

En [Ferreira et al., 2007] se presentó un método de diseño para una NN en aritmética de punto flotante usando VHDL. Las funciones de transferencia se aproximaron por tramos lineales. Los autores dan el área ocupada, pero no la potencia ni la máxima velocidad del circuito.

En la aportación de [Himavathi et al., 2007] se diseñó la capa de mayor tamaño de una NN; es decir, la que tenía el mayor número de neuronas. Esta capa se usa recursivamente para computar las salidas de todas las capas. Se usó aritmética de

punto fijo, no se estudió el efecto del número de bits porque se usaron cantidades prefijadas. Las funciones de transferencia se implementaron usando LUT. El sistema se diseñó con el lenguaje Verilog. Los autores estimaron el área ocupada y la velocidad, pero no la potencia.

En [Mishra et al., 2007] se diseñó una neurona en aritmética de punto fijo, pero no se estudió el efecto del número de bits. La función de transferencia se aproximó por tramos lineales. El sistema se describió usando VHDL. Los autores no muestran estimación de área, potencia o velocidad.

Se implementó una NN para reconocimiento de posturas de la mano en [Oniga et al., 2007]. Se diseñó con *System Generator* de Xilinx usando aritmética de punto fijo. En este caso el diseño de las funciones de transferencia es trivial, se usaron funciones identidad y competitiva. Los autores no dan estimaciones de área, velocidad ni potencia.

En [Savich et al., 2007] se comparan implementaciones de NN usando punto fijo y punto flotante. Los autores llegan a la conclusión, de que para la misma funcionalidad, el diseño en punto fijo mejora las prestaciones de área y velocidad. La función de transferencia se aproximó por tramos lineales. Los diseños se realizaron usando VHDL. No se da estimaciones de la potencia consumida.

Se implementó una NN para procesado de imagen en [Ho et al., 2008] usando una GPU. La función de transferencia usada era lineal en el tramo central y saturada en los extremos. Se usó el lenguaje C++ para su diseño. El diseño en la GPU es entre 8 y 17 veces más rápido que el de una CPU.

Los autores [Lin y Wang, 2008] diseñaron por tramos lineales la función tangente hiperbólica. Se diseñó en forma de esquemático con aritmética de punto fijo. Se hicieron estimaciones de área y velocidad, pero no de potencia.

En [Ogrenci, 2008] se plantea una metodología de diseño para una NN. Se diseñó en punto fijo con VHDL. El autor da estimación de área y velocidad, pero no de potencia.

Una NN, de una o dos capas, se diseñó en [Oniga et al., 2008]. Los autores no indican cuál es la función de transferencia usada. Se diseñó con *System Generator* de

Xilinx en aritmética de punto fijo. Los autores estudiaron el efecto del número de bits sobre el área ocupada y la frecuencia máxima. En esta aportación no se estudió la potencia consumida.

Los autores [Saichand et al., 2008] implementaron la función de transferencia sigmoidea. La primera versión se diseñó con una LUT en punto fijo. Para el segundo modelo se usó una aproximación lineal por tramos en punto flotante. Los autores no indican el método o lenguaje de diseño empleado. Se hizo la estimación del área y del retardo, pero no de la potencia consumida.

Una utilidad para generar funciones de transferencias se desarrolló por [Namin et al., 2009]. El entorno se basa en Matlab y genera la descripción en VHDL en punto fijo. Se puede sintetizar la función sigmoidea y la tangente hiperbólica. Son posibles tres arquitecturas basadas en LUT. Los autores muestran la estimación de área y retardo, pero no de potencia.

En [Oniga et al., 2009] se diseñó una NN con una sola capa oculta. El cálculo en cada neurona se realizó de forma secuencial con un MAC en punto fijo. Aunque no se indica expresamente, parece que la función usada era tipo sigmoidea y se realizó con una LUT. Para el diseño se usó *System Generator* de Xilinx. Los autores estudiaron el efecto del número de bits sobre la funcionalidad del sistema. En esta aportación se indicó la máxima frecuencia de funcionamiento, hay estudios parciales del área y no existen estimaciones de potencia.

Una NN con tecnología mixta, digital y analógica, se diseñó en [Maliuk et al., 2010]. Los coeficientes se almacenaban en punto fijo con 6 bits para la parte decimal. La multiplicación-suma y la función de transferencia se realizaron con transistores metal-óxido-semiconductor en simetría complementaria (CMOS, *Complementary metal-oxide-semiconductor*). Los autores dan estimación de área, pero no de velocidad ni de potencia. El entrenamiento se establece con el método *chip-in-the-loop*; esto es, se realiza fuera del diseño, en un ordenador personal, pero el sistema está conectado directamente al ordenador para actualizar los coeficientes.

En [Misra y Saha, 2010] se da un repaso de las implementaciones hardware de las NN desde veinte años atrás; tanto de circuitos comerciales, como fruto de la investigación.

Los autores [Armato et al., 2011] implementaron las funciones sigmoideas y sus derivadas en punto fijo. Usaron aproximación lineal por tramos, el método de optimización en cada intervalo fue minimizar el máximo error relativo en el intervalo. No se realizó ninguna implementación; obviamente no se obtuvieron prestaciones físicas. Se bosquejó el efecto del número de bits sobre el error producido.

Una pequeña NN se diseñó en [Bahoura y Park, 2011], tenía una entrada y una salida, y dos neuronas en la capa intermedia. Se diseñó en punto fijo usando *System Generator* de Xilinx, no se estudió el efecto del número de bits sobre las prestaciones de la NN. En la capa oculta se usó una función sigmoidea implementada con LUT; en la salida se usó la función identidad. Los autores estimaron el área y la máxima frecuencia de reloj, pero no la potencia consumida. El diseño incluye los circuitos para un entrenamiento *online*.

Los autores [Cavuslu et al., 2011] diseñaron una NN de tres capas en aritmética de punto flotante. La función de transferencia sigmoidea se aproximó con una función racional. El diseño se realizó mediante VHDL. En este artículo se estima el área y la frecuencia máxima, pero no la potencia.

En [Gomperts et al., 2011] se generó una herramienta para sintetizar una NN en VHDL con número de capas variable. Se usó aritmética de punto fijo y las funciones de transferencia se generaron con LUT. Los autores estudiaron el efecto del número de bits. También dan estimación de área y velocidad, pero no de potencia. El sistema permite entrenamiento *online*.

La velocidad de un motor se consiguió estimar en [Orlowska y Kaminski, 2011] mediante una NN. Se diseñó con LabView de National Instruments en aritmética de punto fijo [LabView, 2015]. La NN tiene dos capas intermedias. Para el diseño de las funciones sigmoides se usaron LUT. En el artículo se muestra la estimación de área y velocidad, pero no de potencia.

Los autores en [Reis et al., 2011] implementaron una herramienta de generación automática de NN con una capa oculta. Se usó aritmética de punto fijo; las funciones de transferencia usadas son la identidad y la tangente hiperbólica, la segunda se implementó mediante LUT. La herramienta genera el diagrama de bloques totalmente paralelizado para *System Generator* de Xilinx, lo que es una gran ventaja. Los autores muestran su aplicación en un determinado escenario, analizan la funcionalidad y el área del diseño; no muestran su velocidad ni potencia, aunque podrían ser extraídas usando las correspondientes herramientas de Xilinx. Todas las entradas tienen el mismo formato binario. Los coeficientes de la capa oculta usan el mismo formato binario, lo mismo sucede con la capa de salida. El sistema tiene una única salida. La uniformidad del formato binario, en un sistema paralelizado; y el uso de una sola salida, son claras desventajas.

En [Basterretxea, 2012] se realizó la implementación de la función de transferencia descrita en [Basterretxea et al., 2004]. Esto se realizó en aritmética de punto fijo con *System Generator* de Xilinx. Solo se comprobó su funcionalidad, se mostraron las estimaciones de área aproximadas de *System Generator* y el número de ciclos de reloj necesarios. No se implementó con la herramienta estándar de Xilinx para FPGA; por este motivo no se obtuvo la frecuencia máxima de funcionamiento, ni la potencia consumida, ni la estimación exacta de área.

En la tesis doctoral [Martínez, 2012] el autor realiza una NN con las neuronas extendidas en dos dimensiones para emular una retina artificial. Se usó aritmética de punto fijo y la función de transferencia usada es lineal en el tramo central con saturación en los extremos. Se implementó sobre una FPGA usando procesadores empotrados por la gran cantidad de neuronas existentes. El sistema se diseñó usando VHDL. La tesis muestra estimaciones de área y velocidad, pero no de potencia.

En la aportación de [Nascimento et al., 2013] se da una nueva solución para la función tangente hiperbólica. Esta se basa en la descomposición del exponente en la parte entera y fraccionaria; la correspondiente a la parte fraccionaria se calcula mediante un polinomio, pero la correspondiente a la parte entera se almacena en una LUT. El diseño se realizó en punto fijo en VHDL y Verilog. Las prestaciones se compararon con la aproximación lineal por tramos y el algoritmo CORDIC. El trabajo

muestra estimaciones de área, pero no de potencia. Los autores comparan el retardo de las diferentes implementaciones atendiendo al número de ciclos de reloj necesarios; sería conveniente evaluar la máxima frecuencia de cada implementación, sobre el mismo dispositivo, para comparar las velocidades.

En [Tisan y Cirstea, 2013] los autores diseñaron una NN con mapa autoorganizado (SOM, *Self Organizing Map*). Se usó aritmética de punto fijo y se probaron con neuronas de 16 y 32 bits de salida. El sistema se diseñó con *System Generator* de Xilinx. Los autores dieron estimaciones de área, velocidad y potencia. El circuito permite entrenamiento *online*.

Una NN bidimensional se desarrolló en [Nambiar et al., 2014], donde las neuronas se ordenan en filas y columnas formando una matriz, y cada neurona se comunica con sus adyacentes de la misma fila y columna. El sistema se implementó en una FPGA; una parte descrita en Verilog, y otra mediante un microprocesador empotrado. Se usó aritmética binaria en punto fijo y complemento a dos; no se varió el número de bits, se usó una cantidad fija según estudios previos para este tipo de NN. Se hicieron estimaciones de área y velocidad, pero no de potencia. Lo más relevante fue la forma de implementar la función de transferencia, que se basó en [Kwan, 1992], esto consistió en una aproximación para la tangente hiperbólica. Este método genera la zona de transición mediante dos funciones cuadráticas, con saturación en los extremos. Como mucho, para esta implementación, se necesitan dos multiplicaciones y una suma; el circuito es relativamente sencillo. En [Nambiar et al., 2014] se entrenó directamente con la aproximación descrita; aunque, en general, es posible entrenar con la tangente hiperbólica y sustituirla en la implementación por la aproximación propuesta en [Kwan, 1992].

En [Zamanlooy y Mirhassani, 2014] se propuso un diseño para la función tangente hiperbólica para dispositivos ASIC, usando aritmética de punto fijo. Como en muchos otros diseños se aprovecha la simetría impar de la función; además, se divide en tres tramos: la zona casi lineal cercana al origen, la zona de transición y la zona de saturación. En función del máximo error se determinaron los puntos que separan estas zonas, la zona de transición se aproximó con la técnica del mapeo binario. El sistema se

describió con el lenguaje Verilog. Los autores dan estimaciones de área, velocidad y potencia.

En resumen, en las diferentes aportaciones, los autores se centran en unos aspectos más que en otros. La aritmética de punto flotante es poco usada por la elevada cantidad de recursos que requiere, alto consumo de potencia y gran retardo. Por tanto, la mayor parte de las aportaciones se han desarrollado en punto fijo. El número de bits en las diferentes partes del sistema se determina por tanteo o con anchos de palabra prefijados, buscar el número idóneo de bits debe ser objeto de estudio. Esto permitirá mejorar las prestaciones físicas manteniendo la funcionalidad. A veces el formato se reduce a un número entero de octetos, quizás porque los diseños provienen de una versión previa sobre microprocesador, o porque los autores pertenecen al área de computación y se imponen esta restricción; el formato de los datos en ocasiones puede ser totalmente flexible. Los autores usan diferentes tecnologías para su implementación física, la mayor parte usa FPGA por sus características frente a las otras tecnologías disponibles. En cuanto a los métodos de diseño empleados el reparto es más heterogéneo. Todavía se presentan diseños usando un lenguaje de descripción hardware (HDL, *Hardware Description Language*), o incluso edición de esquemáticos; si bien ya existen herramientas de descripción más rápidas y flexibles. Las herramientas avanzadas que operan sobre *Simulink* se usan con moderación, no de forma habitual. Apenas si existen aportaciones que usan las utilidades propias de Matlab, si bien estas herramientas datan de hace solo un par de años. Casi todos los autores dan estimación de área ocupada y de máxima velocidad; pero pocos dan estimaciones de la potencia consumida, lo que afecta claramente a la portabilidad y autonomía del prototipo. En algunos diseños se incluyen las estructuras para el entrenamiento en conexión (*online*) de la NN. Esto permite readaptar la NN a un nuevo escenario, pero la inserción de los circuitos necesarios para el entrenamiento provocan un aumento del retardo en la operación de la NN; y obviamente un aumento del área y de la potencia consumida.

1.3 Objetivos de la tesis

En esta tesis doctoral se propone la búsqueda de métodos de diseño de NN sobre dispositivos digitales programables. El tipo de aritmética usada será en punto fijo y complemento a dos, por las ventajas que presenta frente a la implementación en punto flotante. El estudio estará enfocado a la mejora de las prestaciones del procesado digital de la señal en tiempo real. Como tecnología se usarán las FPGA por su idoneidad para realizar prototipos. De los sistemas diseñados se extraerán las prestaciones físicas de área, velocidad y potencia consumida. Por tanto, se pretende demostrar la hipótesis que se expone a continuación.

> *"Es posible encontrar métodos de diseño sobre dispositivos digitales programables para implementar redes neuronales que operen en tiempo real en procesado digital de la señal. Los métodos deben ser rápidos y flexibles; y permitir evaluar el efecto del número de bits sobre la arquitectura. Además, deben posibilitar la comprobación de la total funcionalidad del sistema y las prestaciones físicas de área, potencia y velocidad".*

La anterior hipótesis se pretende aplicar a escenarios reales; además, se persigue que el método sea extensivo y generalizable. Para comprobar su correcta aplicación se aplicó sobre cuatro escenarios distintos, con bases de datos de diferente naturaleza.

1.4 Metodología

Esta tesis se centra en la NN tipo perceptrón multicapa con conexión hacia adelante (*Feedforward Multilayer Perceptron*) [Bishop, 2005]. Como se expresó anteriormente, se enfocará al procesado de señal en tiempo real. En este ámbito la tasa de eventos a procesar puede crecer de manera indefinida. La máxima velocidad de proceso se consigue cuando el cálculo está totalmente paralelizado; dicho de otra forma, mínimamente secuencializado. Es decir, la máxima frecuencia de funcionamiento, o mínimo retardo, se alcanzará cuando se puedan desplegar de forma paralela todos los operadores. Este tipo de despliegue puede ser tolerado por las FPGA

porque algunas incluyen actualmente varios miles de multiplicadores. En esta tesis se hará énfasis en las NN totalmente paralelizadas.

La parte crítica de la implementación de las NN lo constituye la función de transferencia, típicamente no lineal. Estas funciones suelen incluir exponentes y divisiones, operaciones de costosa realización; por ello se suele recurrir a una implementación aproximada. Existen para esto diferentes técnicas. La estrategia con menor retardo es almacenar muestras de la función en una memoria; o sea, el uso de LUT. Por eso en la arquitectura planteada se usarán LUT, aunque como inconveniente cabe destacar que tiene una alta ocupación de recursos hardware. No obstante, en ocasiones se recurrirá a funciones de transferencia lineales por tramos; si con estas funciones se alcanzan la funcionalidad, entonces se permite mejorar el área, la velocidad y el consumo de potencia.

Tras un estudio de las diferentes herramientas de diseño se optó por un método sobre *Simulink* de Matlab. Esto permitirá una descripción rápida y flexible. Además facilitará la simulación del sistema; hasta el punto de comprobar su total funcionalidad. Finalmente se indicarán las prestaciones físicas conseguidas en los diseños: área, velocidad y potencia.

Para probar el método se diseñarán NN que funcionarán sobre diferentes escenarios. Un escenario es la clasificación de la palmera pejibaye atendiendo a marcadores moleculares de ácido desoxirribonucleico (*DNA*, *Deoxyribonucleic acid*). Esta base de datos proviene de Costa Rica. También se usará una NN para clasificar pulsos de electrocardiograma (ECG, *Electrocardiogram*), la base de datos está tomada del Hospital Beth Israel del Instituto Tecnológico de Massachusetts (MIT-BIH, *Massachusetts Institute of Technology-Beth Israel Hospital*). En una tercera situación se usó una NN para predecir temperatura. La base de datos disponible en este caso proviene también de Costa Rica. Finalmente, se presenta una metodología de diseño para una NN funcionando como ecualizador de una señal digital binaria unipolar en presencia de ruido. En este caso la base de datos es sintética.

Para cada escenario, en una primera fase, se realiza el entrenamiento y testeo, como indica la figura 1.2. Esto constituye el modelado del sistema en punto flotante. En esta etapa se fija la arquitectura de la NN. Si no se obtiene la funcionalidad

requerida de reentrena el sistema o se cambia la arquitectura. Cuando se alcanza la funcionalidad necesaria se obtiene la regla de oro. La regla de oro consiste en la arquitectura de la NN y los valores de los coeficientes obtenidos en el entrenamiento.

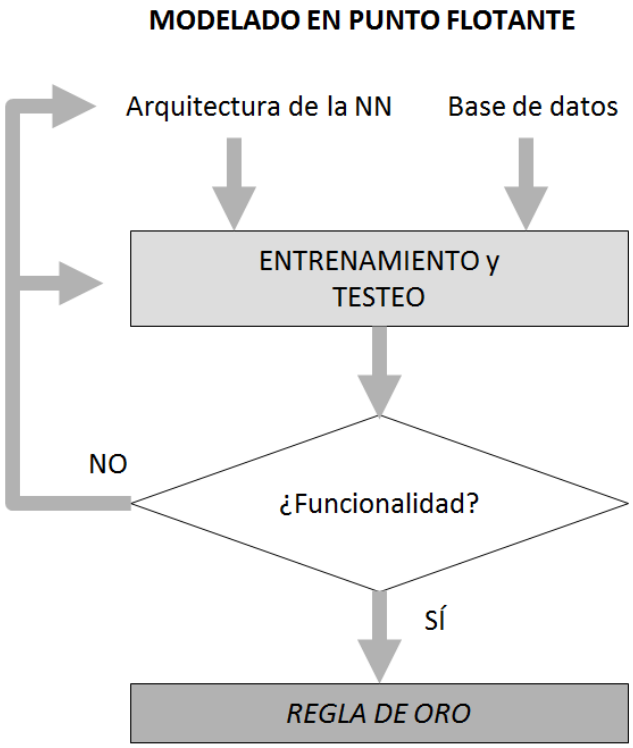


Figura 1.2. Modelado del sistema en punto flotante.

Una vez obtenida la regla de oro la NN está totalmente especificada. El objetivo de la segunda etapa es conseguir el diseño del circuito en punto fijo. Para ello, según la figura 1.3, se usa un entorno gráfico sobre *Simulink* de Matlab. Los bloques de *Simulink* quedan especificados con nombres de variables, entre ellos los nombres de los coeficientes de la NN. Con un programa de Matlab se fija el número de bits para la representación de los coeficientes; esto se hace teniendo como entrada los valores en punto flotante de la regla de oro y un error máximo permitido en la representación. El objetivo es determinar el mínimo número de bits que mantiene la funcionalidad de la NN; lo que minimiza el área y el consumo de potencia, y maximiza la velocidad.

Una vez que se ha fijado el número de bits es posible comprobar la completa funcionalidad de la NN. Los bloques circuitales usados en *Simulink* tienen un bajo nivel de detalle, esto permite simulaciones muy rápidas; por contra, la estimación de área es

aproximada, y no se obtiene estimaciones de velocidad ni de potencia consumida. Una vez garantizada la funcionalidad se compila el diseño, obteniendo la descripción en un HDL para la herramienta estándar de la FPGA. En este último entorno se tiene un alto nivel de detalle de los circuitos. Esto permite hacer estimaciones precisas de área, velocidad y potencia. Estas simulaciones son lentas, e imposibilitan comprobar la total funcionalidad, a menos que el diseño sea sencillo.

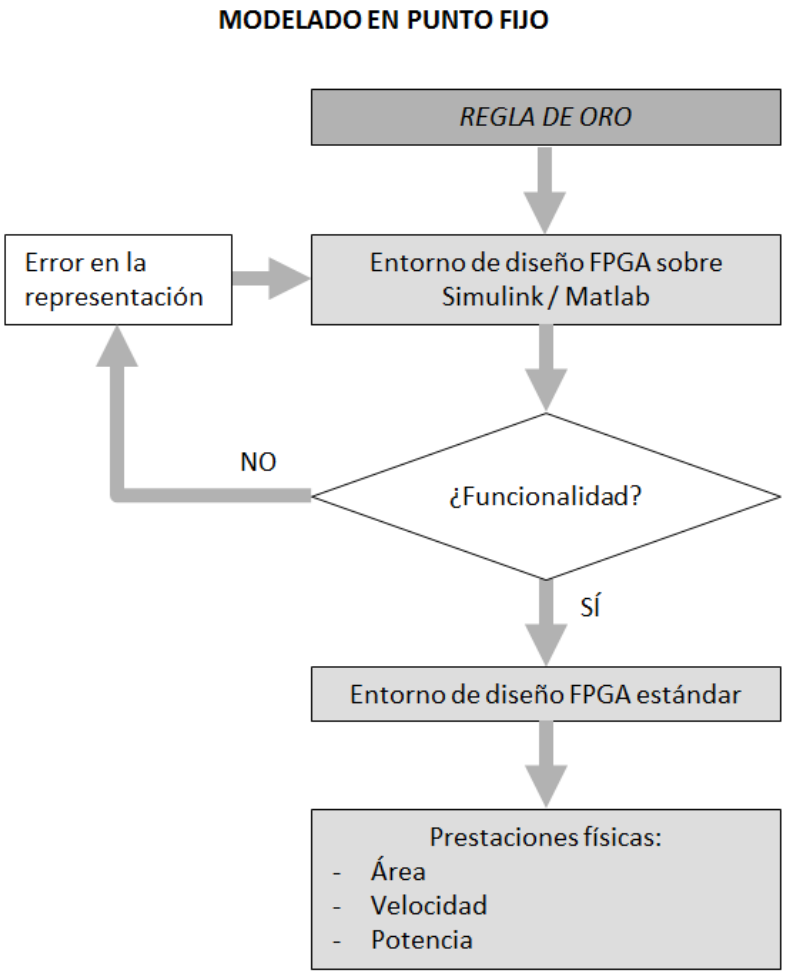


Figura 1.3. Modelado del sistema en punto fijo.

1.5 Contribuciones y resultados

Durante el desarrollo de la presente tesis se han conseguido diferentes contribuciones que se exponen en la tabla 1.3. Se puede afirmar que se han realizado aportaciones metodológicas para el diseño de NN sobre dispositivos digitales

programables; todo esto para diferentes escenarios con bases de datos de distinta naturaleza. En la figura 1.4 se observa las aportaciones anuales.

Tabla 1.3. Contribuciones alcanzadas durante el desarrollo de esta tesis.

Publicaciones	Cantidad	Referencias
Con JCR	2	[Pérez et al., 2013] [Vásquez et al., 2013]
Sin JCR	2	[Pérez et al., 2009a] [Travieso et al., 2013a]
Capítulos de libros	2	[Pérez et al., 2011b] [del Pozo et al., 2012]
Congresos internacionales	9	[Pérez et al., 2009b] [Pérez et al., 2011c] [Pérez et al., 2011d] [Ticay et al., 2011] [Vásquez et al., 2012] [Vázquez et al., 2012] [Travieso et al., 2013b] [Pérez et al., 2014a] [Pérez et al., 2014b]
Congresos nacionales	3	[Pérez et al., 2009c] [Pérez et al., 2011a] [Alonso et al., 2013]

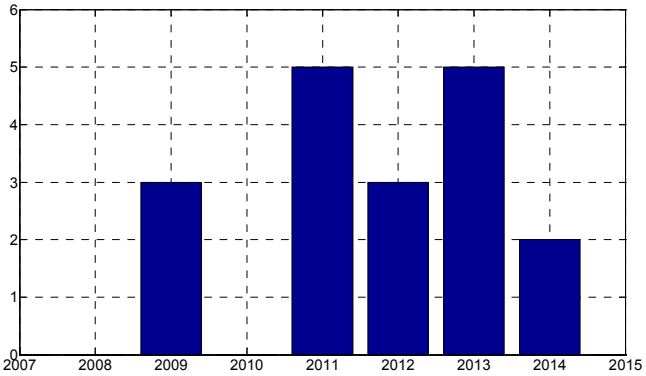


Figura 1.4. Total de aportaciones anuales.

A continuación se describen las principales aportaciones siguiendo la evolución temporal. En [Pérez et al., 2011c] los autores diseñaron un NN para clasificar especies de la palmera Pejibaye en función de marcadores moleculares. Se usó aritmética de punto fijo y la función de transferencia se diseñó usando una LUT. El sistema se describió sobre *Simulink* de Matlab, usando *System Generator* de Xilinx. Los autores estimaron el área, la potencia y la velocidad del sistema.

Una NN se usó en [Vázquez et al., 2012; Travieso et al., 2013a; Travieso et al., 2013b] para discriminar pulsos de electrocardiograma y detectar este tipo de patologías. Básicamente el método y el diseño de la NN coincide con [Pérez et al., 2011c], con la salvedad de que se hace un preprocesamiento previo usando la transformada discreta Wavelet (DWT, *Discrete Wavelet Transform*) y los datos son normalizados.

Una NN se usó en [Pérez et al., 2011d; Vásquez et al., 2012; Vásquez et al., 2013] para predicción de temperatura, en la entrada se incluyó una red de células de retardo en cascada; constituyendo una red neuronal con línea de retardo (TDNN, *Time Delay Neural Network*). En este trabajo se expusieron las prestaciones físicas logradas. Para alcanzar la funcionalidad se tuvo en cuenta el valor medio del error en valor absoluto.

Los autores en [Pérez et al., 2013] exponen una metodología de diseño de un ecualizador para una señal binaria unipolar sin retorno a cero (NRZ, *Non Return to Zero*) en presencia de ruido Gaussiano blanco aditivo (AWGN, *Additive White Gaussian Noise)*. Para ello se usó una TDNN empleando aritmética de punto fijo y optimizando el número de bits en los diferentes puntos del sistema. El diseño de las funciones de transferencia se simplificó, las prestaciones se alcanzaron con la función identidad y la función lineal en el tramo central con saturación en los extremos. Para el diseño se usó *System Generator* de Xilinx sobre *Simulink* de Matlab. Se hicieron estimaciones de área, potencia consumida y de la máxima tasa de muestras soportable en la entrada.

En [Pérez et al., 2014a] se presentó el método de diseño de NN sobre *System Generator* comprobado en los cuatro escenarios mencionados anteriormente: palmera pejibaye, pulsos electrocardiográficos, predicción de temperatura y ecualizador para señal binaria. En todos se usó aritmética de punto fijo. En los dos últimos escenarios se añade una línea de retardo para crear una TDNN; esta nueva arquitectura actúa como un predictor. En los tres primeros casos se usaron funciones sigmoides aproximadas con LUT. En el ecualizador se consiguió simplificar la implementación de las funciones al usar la función identidad y la función lineal en el tramo central con saturación en los extremos. En todos los casos se realizó entrenamiento *offline* y se obtuvieron las prestaciones de área, velocidad y potencia consumida.

Los métodos de diseño para dispositivos digitales programables por el diseñador se repasaron en [Pérez et al., 2014b] y se expusieron en un congreso sobre innovación educativa; además, en ese trabajo se hicieron propuestas al Grado en Ingeniería en Tecnologías de la Telecomunicación y al Máster Universitario en Ingeniería de Telecomunicación de la Universidad de Las Palmas de Gran Canaria.

A continuación se muestras otros méritos obtenidos durante el desarrollo de esta línea de trabajo.

- Participación en Proyectos de Investigación, se muestran en la tabla 1.4.
- Concesión de un sexenio de investigación para los años evaluados: 2004, 2005, 2006, 2011, 2012 y 2013. Concedido por la Comisión Nacional Evaluadora de la Actividad Investigadora.
- Charla invitada: "*CARDIODETECT. Detección y clasificación de arritmias en tiempo real*" en las "Jornadas de Innovación de Atención Sanitaria con las TIC". Celebradas en el año 2010 en Santa Cruz de Tenerife y organizadas por la Cátedra de Telefónica de la Universidad de Las Palmas de Gran Canaria.
- Colaboración en congresos internacionales:
  - Revisor en la *International Conference on ReConfigurable and FPGAs* (Reconfig 2011), Cancún, México.
  - Moderador y miembro del comité organizador en la *International Conference on NoLinear Speech Processing* (NoLISP 2011), Las Palmas de Gran Canaria, España.
  - Miembro del comité organizador en el *2nd Workshop on Bioinspired Intelligence* (WOBI 2012), San José, Costa Rica.
  - Miembro del comité organizador en el *3rd International Conference and Workshop on Bioinspired Intelligence* (IWOBI 2014), Liberia, Costa Rica.
  - Miembro del comité organizador en las I Jornadas Iberoamericanas de Innovación Educativa en el ámbito de las TIC (InnoEducaTIC 2014), Las Palmas de Gran Canaria, España.
  - Miembro del comité organizador en el *4th International Work Conference on Bioinspired Intelligence* (IWOBI 2015), San Sebastián, España.

- Miembro del comité organizador en las II Jornadas Iberoamericanas de Innovación Educativa en el ámbito de las TIC (InnoEducaTIC 2015), Las Palmas de Gran Canaria, España.

Tabla 1.4. Participación en Proyectos de Investigación.

NOMBRE	***Investigación, desarrollo e innovación en imágenes en el infrarrojo cercano y lejano (de 900 nm a 1700 nm) vinculadas a aplicaciones en biometría***
Investigador Principal	*Carlos M. Travieso González*
Cuantía	*15.245,93 €*
Entidad financiadora	*Gobierno Autónomo de Canarias*
Entidades participantes	*Universidad de Las Palmas de Gran Canaria*
Inicio	*2007*
Duración	*1 año*
NOMBRE	***Laboratorio de Sistemas y Autómatas Inteligentes en Biodiversidad***
Referencia	*D/027406/09, anualidad 2010; D/033858/10, anualidad 2011; A1/039531/11, anualidad 2012*
Investigador Principal	*Carlos M. Travieso González*
Cuantía	*515.372 €*
Entidad financiadora	*Agencia Española para la Cooperación Internacional y el desarrollo del Ministerio de Asuntos Exteriores y Cooperación del Gobierno de España*
Entidades participantes	*Universidad de Las Palmas de Gran Canaria, Universidad de Costa Rica*
Inicio	*2010*
Duración	*3 años*
NOMBRE	***Metodología de la evaluación acústica del sistema fonador***
Investigador Principal	*Jesús B. Alonso Hernández*
Cuantía	*6.000 €*
Entidad financiadora	*Innova Canarias 2020, Grupo Sedicana y Clubes de Leones de Gran Canaria*
Entidades participantes	*Universidad de Las Palmas de Gran Canaria*
Inicio	*2010*
Duración	*1 año*
NOMBRE	***e-VOICE : Sistema de Evaluación Remota del Sistema Fonador***
Investigador Principal	*Jesús B. Alonso Hernández*
Cuantía	*10.100 €*
Entidad financiadora	*Cátedra Telefónica*
Entidades participantes	*Universidad de Las Palmas de Gran Canaria*
Inicio	*2012*
Duración	*1 año*
NOMBRE	***Síntesis de muestras biométricas para aplicaciones de salud y seguridad***
Referencia	*TEC2012-38630-C04-02*
Investigador Principal	*Miguel Á. Ferrer Ballester*
Cuantía	*65.988 €*
Entidad financiadora	*Ministerio de Ciencia y Competitividad del Gobierno de España*
Entidades participantes	*Universidad de Las Palmas de Gran Canaria*
Inicio	*2013*
Duración	*3 años*

- Colaboración en congresos nacionales.
  - Revisor y moderador en las *XI Jornadas de Computación Reconfigurable y Aplicaciones* (JCRA 2011), San Cristóbal de La Laguna.
  - Miembro del comité organizador en las *VI Jornadas de Reconocimiento Biométrico de Personas* (JRBP 2012), Las Palmas de Gran Canaria.
  - Miembro del comité organizador en las *I Jornadas Multidisciplinares de Usuarios de la Voz, el Habla y el Canto* (JVHC 2013), Las Palmas de Gran Canaria.
- Revisor en la revista internacional *Advances in Research* (ISSN: 2348-0394), de *SCIENCEDOMAIN international* en el año 2014.
- Estancia en la Escuela de Ciencias de la Computación e Informática de la Universidad de Costa Rica, del 3 al 19 de junio de 2010, con cargo al proyecto financiado por la Agencia Española de Cooperación Internacional titulado "Laboratorio de Sistemas y Autómatas Inteligentes en Biodiversidad".
- Se citó la aportación [Pérez et al., 2009a] en la tesis doctoral *Synchronization Algorithms and Architectures for Wireless OFDM Systems* de la Universidad de Newcastle, Reino Unido, [Younis, 2012].
- Se han desarrollado los Trabajos Fin de Título de la tabla 1.5.

Tabla 1.5. Trabajos Fin de Título desarrollados.

Trabajos Fin de Título	Cantidad	Referencias
Proyectos Final de Carrera	2	[Vázquez, 2011] [Tarapuez, 2015]
Trabajos Fin de Master	2	[Chadnani, 2014] [Osorio, 2014]

1.6 Estructura de la memoria

La memoria de esta tesis se estructura en cinco capítulos con los contenidos que se indican a continuación. Además se incluye una lista de figuras, una lista de tablas, una lista de acrónimos y las referencias bibliográficas.

En el capítulo 1 se realiza la introducción de la tesis. En este capítulo se expone la motivación y justificación para su realización. Igualmente se detallan los antecedentes y el estado del arte del tema a tratar. Posteriormente se indican los objetivos de esta

tesis y la metodología para alcanzarlos. Por último, se muestran las contribuciones y resultados alcanzados durante su realización.

El capítulo 2 analiza el problema de la implementación de NN para funcionar en tiempo real. En principio se analizan los tipos posibles de aritmética binaria y se define el concepto de regla de oro. Posteriormente se exponen las tecnologías disponibles y los métodos de diseño para FPGA. Además, se analizan los parámetros de los diferentes métodos de diseño y se introduce el concepto de factor de calidad de un diseño. Finalmente se exponen herejías metodológicas; o dicho de otra forma, como no debe llevarse a cabo el proceso.

En el capítulo 3 se trata el esquema general de la metodología experimental, se exponen los escenarios y las bases de datos usadas. También se describen los problemas del modelado en punto flotante y en punto fijo. Además, se describe el flujo de diseño estándar para FPGA y las cualidades que debe tener la herramienta elegida y el flujo de diseño usado. Finalmente, se exponen de forma resumida las herramientas de Xilinx y Altera que funcionan sobre *Simulink* de Matlab.

El capítulo 4, para cada uno de escenarios descritos en el capítulo 3, muestra los experimentos realizados y los resultados obtenidos. Esto consiste en la revisión del modelado en punto flotante, la descripción de la regla de oro, y la implementación en punto fijo en la FPGA con las herramientas elegidas; en cada escenario se analizan los resultados. Finalmente se realiza una comparación con el estado del arte.

Finalmente, en el capítulo 5 se exponen las conclusiones y las posibles líneas futuras. En el anexo se expone en detalle el modelado en punto fijo del último escenario. El documento finaliza con la bibliografía.

CAPÍTULO 2

IMPLEMENTACIÓN DE REDES NEURONALES EN TIEMPO REAL

"Da el primer paso con fe. No tienes por qué ver toda la escalera. Basta con que subas el primer peldaño".

Martin Luther King

En este capítulo se describirá el escenario sobre el que desarrolla esta tesis. Primeramente se presentará el concepto de red neuronal artificial y su alto grado de variabilidad. Seguidamente se presentan los tipos de aritmética binaria y las diferencias entre las implementaciones en punto flotante y punto fijo.

Normalmente las NN artificiales se implementan en una computadora usando aritmética en punto flotante. Si se quiere mejorar las prestaciones de este tipo de ejecución se debe cambiar el soporte en el que se realiza el cálculo. Por este motivo se describen a continuación las tecnologías disponibles sobre las que se puede implementar una red neuronal. Una vez elegida la tecnología se describen los métodos de diseño existentes. Posteriormente se definen una serie de parámetros para caracterizar y comparar los métodos de diseño. Finalmente, una vez realizado el diseño, se propone medir su calidad en función de las prestaciones físicas: área, velocidad y potencia consumida. Para acabar, se enfatiza cómo no debería llevarse a cabo un diseño, cuando se realiza la implementación en punto fijo partiendo de su especificación en punto flotante.

2.1 Las redes neuronales

Las redes neuronales artificiales toman su nombre por las analogías que presentan con las redes neuronales del sistema nervioso [Graupe, 2013]. Una neurona nerviosa se muestra en la figura 2.1, donde las entradas que reciben los estímulos se llaman dendritas y la salida se llama axón. El axón de una neurona nerviosa está conectada a las dendritas de otras neuronas para formar una red en el sistema nervioso.

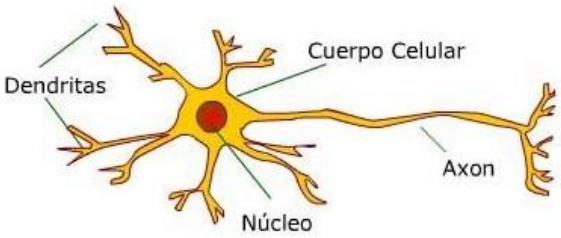


Figura 2.1. Estructura de una neurona nerviosa.

Las redes neuronales artificiales se basan en el concepto de neurona artificial [Graupe, 2013], cuya arquitectura típica se muestra en la figura 2.2. Diferentes autores usan diferentes nomenclaturas para las NN, en esta tesis se asume la nomenclatura de Matlab, el motivo es que este paquete de herramientas software es ampliamente usado en los ámbitos académico, científico e industrial [Matlab, 2014]; además,

también en él se basa el desarrollo de esta tesis. Una neurona artificial es una función real de variable real, con varias entradas (*R*) y una salida. Cada una de las entradas se multiplica por un coeficiente real (*W*) y se suman estos resultados. A los coeficientes *W* se les llama pesos o *weights*. Por otro lado, se suma un valor *b* real, llamado coeficiente de desplazamiento o polarización, en inglés *bias*. Se puede considerar que se tiene una entrada adicional de valor la unidad y se multiplica por el peso *bias*; o *bias* es una entrada que se multiplica por uno. Con este conjunto de multiplicaciones y sumas se obtiene el valor *n*. Finalmente el valor *n* se conecta a una función real, llamada *f* en la figura, que en general es no lineal. La neurona artificial, por tanto, es una función con múltiples entradas, en general no lineal. La salida de las neuronas artificiales se conectan a las entradas de otras neuronas para formar las redes neuronales artificiales. De la neurona artificial cabe destacar la función, llamada de transferencia, y en general no lineal. En la tabla 2.1 se muestran las más usadas y su nomenclatura en Matlab.

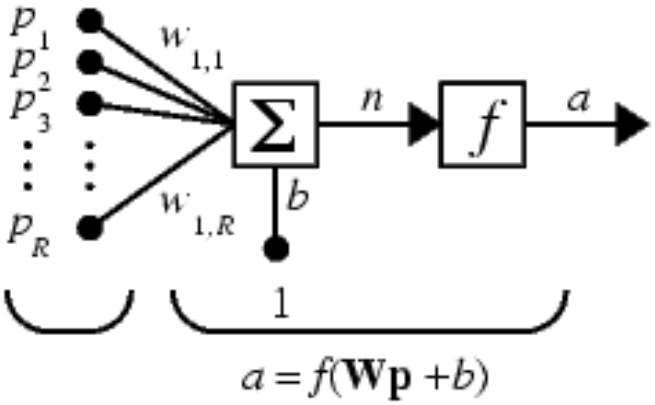
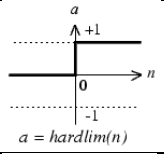
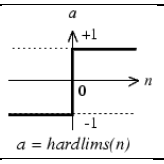
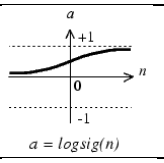
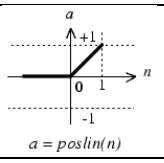
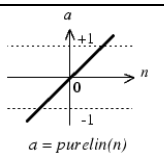
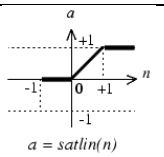
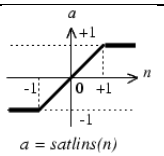
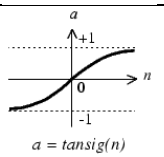


Figura 2.2. Estructura de una neurona artificial.

Tabla 2.1. Funciones de transferencia típicas.

Hard-limit transfer function		$a = \begin{cases} 0 & n < 0 \\ 1 & n > 0 \end{cases}$
Symmetric hard-limit transfer function		$a = \begin{cases} -1 & n < 0 \\ 1 & n > 0 \end{cases}$
Log-sigmoid transfer function		$a = \frac{1}{1+e^{-n}}$
Positive linear transfer function		$a = \begin{cases} 0 & n \leq 0 \\ n & n \geq 0 \end{cases}$
Linear transfer function		$a = n$
Saturating linear transfer function		$a = \begin{cases} 0 & n \leq 0 \\ n & 0 \leq n \leq 1 \\ 1 & n \geq 1 \end{cases}$
Symmetric saturating linear transfer function		$a = \begin{cases} -1 & n \leq -1 \\ n & -1 \leq n \leq 1 \\ 1 & n \geq 1 \end{cases}$
Hyperbolic tangent sigmoid transfer function		$a = \frac{1-e^{-2n}}{1+e^{-2n}}$

Cabe destacar que la neurona es un sistema no lineal, salvo cuando la función de transferencia sea lineal y el término *bias* es nulo. En la mayoría de las ocasiones se usan funciones no lineales; en particular las funciones *tansig* y *logsig* por ser continuas, monótonas crecientes y derivables [Chen et al., 2006; Lin y Wang, 2008]. En general la función de transferencia debe tender en su salida a uno de dos niveles claramente diferenciados.

Un conjunto de neuronas se puede combinar para formar la estructura de la figura 2.3, donde las *R* entradas se conectan a *S* neuronas. Normalmente se dice que las *S* neuronas forman una capa; y las *R* entradas forman otra capa, llamada de entrada.

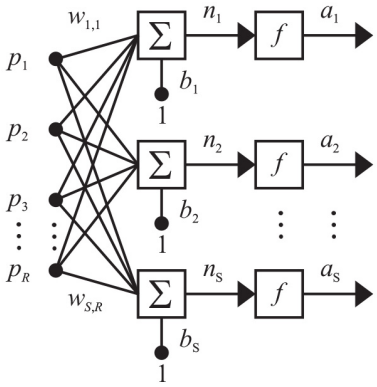


Figura 2.3. Red neuronal con R entradas conectadas a una capa de S neuronas.

Se puede añadir una nueva capa como en la figura 2.4; donde la capa de entrada está formada por las R^1 entradas, las S^1 neuronas constituyen la llamada capa intermedia u oculta, y S^2 es el número de neuronas de la capa de salida. La figura 2.4 forma una NN de tres capas y se denota como R^1-S^1-S^2. El número de salidas coincide con el número de neuronas en la capa de salida. Normalmente, las neuronas en una misma capa tienen el mismo tipo de función de transferencia; pero el tipo de función de transferencia puede ser distinto para cada capa. En la figura 2.4 se usa el tipo f_1 para la capa oculta y f_2 para la capa de salida. Así se puede construir una NN con cualquier número de capas. Este modelo de NN se dice que es del tipo perceptrón multicapa con conexión hacia adelante (*Feedforward Multilayer Perceptron*) [Bishop, 2005]. En este tipo de NN, las salidas en un instante dado, solo dependen del valor de las entradas en ese instante. La figura 2.4 muestra una NN de tres capas, la mayoría de los autores considera a las entradas como una capa [Duda et al., 2001]; otros autores la consideran de dos capas [Oniga et al., 2009].

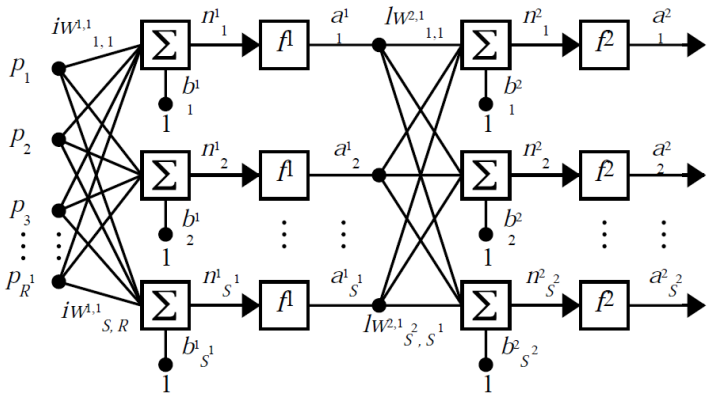


Figura 2.4. Red neuronal con R^1 entradas, una capa intermedia de S^1 neuronas y una capa de salida de S^2 neuronas.

También existen las NN realimentadas o recurrentes [Bishop, 2005], en ellas la salida de una neurona puede realimentarse a su propia entrada, o a neuronas de su propia capa o capas anteriores. Un ejemplo se muestra en la figura 2.5, donde se usa notación abreviada por el uso de matrices. Esto conlleva la existencia de una base de tiempo o reloj en el sistema. En las NN realimentadas la salida en un instante dado no solo dependen de las entradas en ese instante, sino también de las salidas de las neuronas en instantes anteriores; es decir, de la evolución temporal de la NN.

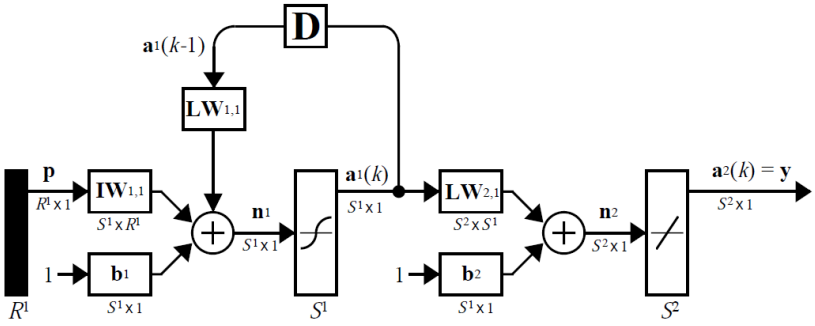


Figura 2.5. Red neuronal realimentada o recurrente.

Finalmente, debe destacarse que existen multitud de arquitecturas de NN; por ejemplo, las que se despliegan en tres dimensiones. Esta tesis se centra en el tipo perceptrón multicapa con conexión hacia adelante, dado que es una arquitectura relativamente sencilla, ampliamente usada y satisface muchas aplicaciones.

Las NN se usan básicamente como identificadores de patrones o clasificadores [Duda et al., 2001]. Por ejemplo, cuando se presenta en la entrada un determinado juego de valores, se activa una de las salidas que indica la clase a la que pertenece la entrada. Para que una NN como la de la figura 2.4 sirva como clasificador, se necesita determinar los valores de los coeficientes de peso y polarización. Obviamente debe fijarse el número de entradas, número de capas, número de neuronas en cada capa y los tipos de función de transferencia. Para determinar el valor de estos coeficientes se necesita una base de datos o conjunto de matrices; donde para cada entrada se indica la salida deseada. Con esta información un algoritmo de entrenamiento es el encargado de calcular los valores óptimos de los coeficientes, de acuerdo a una serie de parámetros (valores iniciales, error, criterio de parada, etc.). Cada vez que se

entrena, se obtiene un conjunto distinto de valores de los coeficientes, se debe elegir uno que cumpla con las prestaciones del clasificador; por ejemplo la tasa de acierto. Una vez que ha finalizado el entrenamiento se realizada el testeo de la NN. Esto consiste en introducir un conjunto de datos no usados en el entrenamiento para comprobar el valor de las salidas. En general las NN con tres capas consiguen alcanzar la funcionalidad necesaria en la mayoría de las aplicaciones. En resumen, debe disponerse de una base de datos, de la que una parte se usa para el entrenamiento y otra para el testeo. En este caso se dice que el entrenamiento es supervisado, porque para cada combinación en la entrada se indica la salida deseada. En esta tesis se usa este tipo de entrenamiento.

También las NN pueden usarse para agrupamiento (*clustering*) de los patrones de entrada [Duda et al., 2001]. En este caso el entrenamiento es no supervisado, no se indica la salida para cada entrada. Es la NN la encargada de agrupar y diferenciar las entradas. Además las NN se usan para: predicción de series temporales (predicción climática, ecualización de señales, etc.) [Connor et al., 1994], aproximación de funciones [Wu y Er, 2000], optimización [Narendra y Parthasarathy, 1991], como memoria asociativa [Cao, 2003] y en control de procesos [Rivals y Personnaz, 2000].

2.2 La variabilidad en el diseño de las redes neuronales

Cuando se pretende usar una NN para solucionar un problema, han de tomarse muchas decisiones. Estas decisiones se basan en la experiencia del diseñador o en un proceso de prueba y error. Antes que nada, debe elegirse el tipo de NN: sin realimentación, recurrente, etc.

Fijada la arquitectura, supóngase el tipo perceptrón multicapa, debe fijarse el número de capas y el número de neuronas en cada capa. El número de entradas viene dado por las condiciones del problema o fijadas tras el proceso de parametrización, pero diferentes parametrizaciones pueden tener distintas prestaciones sobre distintas arquitecturas. Estas prestaciones se refieren a la cantidad de cálculo requerido y a la tasa de acierto. El número de salidas viene dado por el problema que se quiere solucionar. Para el caso de un clasificador, se puede activar una salida por clase o

asignar a cada clase una codificación de las salidas; lo último vuelve a tener efecto sobre las prestaciones del clasificador. Todavía queda elegir el tipo de función de transferencia para cada capa, supuesta igual para todas las neuronas de una misma capa. El tipo de función tiene igualmente efecto en las prestaciones del clasificador. Si la NN fuese recurrente cabe decidir qué señales se realimentan y a qué neuronas, esto vuelve a afectar a las prestaciones.

En cuanto a la base de datos faltaría decidir qué parte se usa para el entrenamiento y cuál para testeo. Además, debe elegirse el algoritmo de entrenamiento y sus parámetros: valores iniciales, error, criterio de parada, etc. El tipo de entrenamiento interactúa, y se comporta de forma distinta, dependiendo de la arquitectura elegida. En resumen, aparecen una serie de decisiones que a un profano en el tema le puede hacer desistir de solucionar el problema mediante una NN.

Suponiendo que se ha conseguido dar con una solución satisfactoria, que no necesariamente óptima, todavía pueden quedar serios inconvenientes que se describen en los apartados siguientes.

2.3 Los tipos de aritmética binaria

Para calcular las salidas de la figura 2.4 se precisan realizar $R^1 \cdot S^1 + S^1 \cdot S^2$ multiplicaciones y sumas, además hay que añadir el coste computacional de las funciones de transferencia. El cálculo requerido para las funciones de transferencia puede ser muy elevado, dependiendo de la forma de implementación. Si la tasa de eventos que procesa la NN es suficientemente pequeña entonces el cálculo puede residir en un ordenador personal o sistema microprocesador. Pero si la tasa es alta, entonces la velocidad de respuesta puede ser lenta, habiendo que recurrir a plataformas más rápidas. También se puede recurrir a otros métodos de implementación, que no sea sobre un ordenador personal, si se quiere disminuir el volumen del prototipo o el consumo de energía.

Las NN pueden diseñarse con cualquier lenguaje de alto nivel, donde los algoritmos se desarrollan en aritmética de punto flotante. Por ejemplo, se usa extensamente el entorno Matlab [Raida, 2002; Strik et al., 2005]. El lenguaje de alto

nivel puede ser interpretado, que no necesita tiempo de compilación, pero presenta un retardo relativamente grande en la ejecución. Por otro lado puede ser compilado y traducido al lenguaje ensamblador del microprocesador, lo que supone una mejora en el tiempo de ejecución.

En la aritmética de punto flotante, el rango y la precisión se pueden ajustar con el número de bits del exponente y la mantisa. Es posible obtener un amplio rango y gran precisión en este tipo de representación [Gokhale y Graham, 2005]. Pero las operaciones con punto flotante necesitan muchos recursos hardware y son muy costosas computacionalmente; además, gastan gran cantidad de potencia y necesitan un número elevado número de ciclos de reloj [Belanovic y Leeser, 2002; Belanovic, 2002]. Por otro lado, la aritmética en punto fijo necesita menos recursos hardware, pero el rango y la precisión solo pueden mejorarse a consta de aumentar el número de bits; si se mantiene constante el número de bits la mejora de uno de ellos hace que el otro empeore [Hauck y Dehon, 2008]. Es posible usar en la mayoría de las aplicaciones la aritmética de punto fijo, cuando se conoce a priori el rango en amplitud de las señales o puede determinarse por métodos estadísticos [Hauck y Dehon, 2008].

Dada la complejidad de implementar operaciones de punto flotante es habitual recurrir a la aritmética de punto fijo. En particular la aritmética usada será de punto fijo en complemento a dos, esto se debe a la mejora que presenta en las operaciones frente al complemento a uno. Dicho de otra forma, antes de optar por la implementación en punto flotante en un dispositivo electrónico, es conveniente evaluar las prestaciones de la solución en punto fijo. El número de bits usado en punto fijo en cada punto del flujo de datos dependerá de rango y la resolución requerida. Si el número de bits es excesivo se está aumentando los recursos lógicos, la potencia consumida y el retardo, sin que el sistema aumente de forma significativa su calidad. Por el contrario, si se disminuye el número de bits, el comportamiento del sistema se puede deteriorar por el aumento del error de cuantificación.

El Instituto de Ingenieros Eléctricos y Electrónicos (IEEE, *Institute of Electrical and Electronics Engineers*) define un estándar para punto flotante [IEEE Std 754, 2008]. Este estándar define el formato binario con representaciones de 32, 64, y 128 bits; es decir, 4, 8 y 16 octetos respectivamente. Esta cantidad de bits es enorme, la mayoría

de las aplicaciones pueden operar en punto fijo con muchos menos bits, con el ahorro que esto supone. Este estándar también define el formato decimal, está pensado para la ejecución en procesadores en punto flotante; de hecho está patrocinado por el *Microprocessor Standards Committee*. Este estándar define: los formatos de representación, atributos y redondeos, las operaciones, los números no calculables (NaN*, Not a Number*), el cero, el infinito, etc. Puede usarse un formato no estándar para punto flotante [Belanovic, 2002], pero esto obliga a definir un estándar propio, cosa que es muy costosa.

2.3.1 Paso del modelo de punto flotante a punto fijo: la regla de oro

La elección del número de bits es un parámetro que debe optimizarse comparando la funcionalidad del sistema de punto fijo con el sistema de punto flotante. Por tanto, es conveniente diseñar previamente el sistema en punto flotante con un lenguaje de alto nivel. Una vez que se tenga el algoritmo definido en punto flotante se dice que se tiene la "regla de oro", lo que Xilinx llama *golden rule* [AccelDSP, 2008] y otros autores *golden reference* [Meeus et al., 2012]. La regla de oro incluye la arquitectura, el valor de los coeficientes y la funcionalidad del algoritmo. Por ejemplo si se tratase de un filtro incluye su arquitectura, el valor de los coeficientes y la respuesta en frecuencia. Si se trata de una NN incluye: el tipo y arquitectura de la NN, el valor de los coeficientes, el tipo de función de transferencia usada en cada neurona y el resultado del testeo obtenido.

Al desarrollar la implementación en punto fijo debe tomarse como referencia la regla de oro (figura 2.6). En la implementación en punto fijo debe mantenerse la arquitectura y los valores de los coeficientes. En el flujo de datos se debe buscar el mínimo número de bits en punto fijo que permite alcanzar la funcionalidad de la regla de oro.

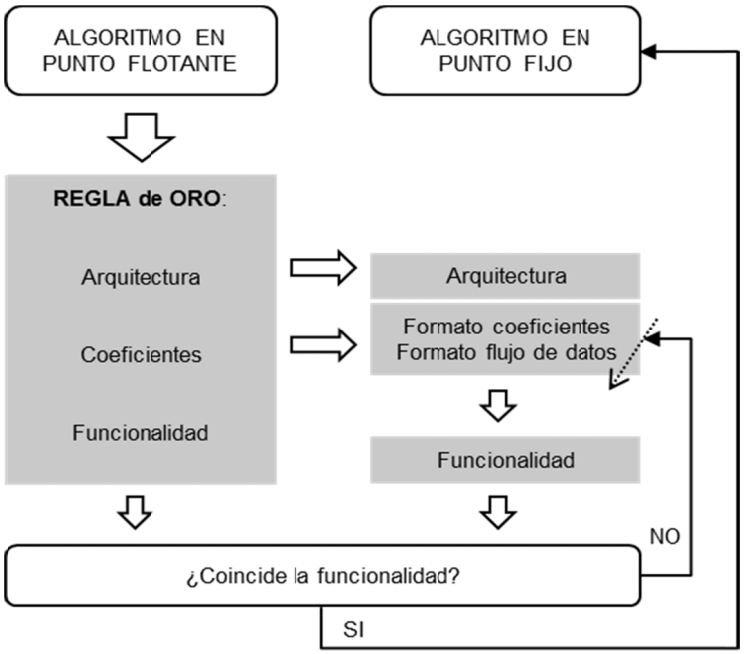


Figura 2.6. Obtención del algoritmo en punto fijo a partir del algoritmo en punto flotante.

2.4 Las tecnologías disponibles

Después de elegido el tipo de aritmética la cuestión es sobre qué dispositivo realizar la implementación. Hay varias alternativas al uso de un ordenador para la implementación de una NN. Una de las posibilidades es usar circuitos integrados de aplicación específica (ASIC, *Application Specific Integrated Circuit*). Los diseños ASIC deben ser enviados a la fábrica para la creación del dispositivo; como ventajas tienen: escasa área ocupada, bajo consumo de potencia y alta velocidad [Chandrasetty, 2011]. Como inconvenientes presentan: alto precio, dificultosa depuración y verificación, gran tiempo de acceso al mercado, no permite la reprogramación y gran coste de ingeniería no recurrente. Esto lo hace normalmente desaconsejable en el desarrollo de prototipos.

Por otro lado, se pueden usar procesadores digitales de señales (DSP, *Digital Signal Processor*) [Oniga et al., 2010; Wang y Ma, 2001], que son más baratos que los elementos ASIC. Los DSP alcanzan mayores frecuencias de reloj que los ASIC. Sin embargo, la tasa de datos que pueden procesar se ve limitada por que el paralelismo de los datos es limitado; el tamaño y formato de los datos es restringido; y el *pipeline* es fijo. Todo esto viene impuesto por la arquitectura fija de los DSP. También puede

usarse una unidad de procesamiento gráfico (GPU, *Graphics Processing Unit*), que es un coprocesador que usa aritmética en coma flotante especializado en el procesamiento de imágenes [Ho et al., 2008; Oh y Jung, 2004]. Las GPU tienen mayor capacidad de procesamiento que las CPU, esto se debe a que cuenta con multitud de unidades aritméticas y lógicas.

Finalmente, se pueden usar matrices de puertas digitales programables por el diseñador (FPGA, *Field Programmable Gate Array*). Un esquema típico de FPGA se muestra en la figura 2.7. Las FPGA constan de bloques de circuitos lógicos, líneas de conexión, matrices de interruptores y pines de entrada-salida. Los bloques de circuitos lógicos reciben diferentes nombres según el fabricante, en ellos se concentra la capacidad de computación de la FPGA. La arquitectura interna de los bloques lógicos depende del fabricante y del dispositivo. Las líneas de conexión son las encargadas de conectar los diferentes bloques y los pines; son largos conductores que atraviesan el integrado. Puede haber líneas específicas dedicadas, por ejemplo a señales de reloj o reinicio del sistema, que se extiende por toda la FPGA. Las matrices de interruptores conectan las líneas de conexión, es un sistema importante para el funcionamiento de la FPGA y en sí mismo es un tema de investigación [Schmit y Chandra, 2005]. La mayor parte de los pines son de entrada-salida de señal, pero también los hay exclusivos para alimentación, relojes, reinicio y programación de la FPGA.

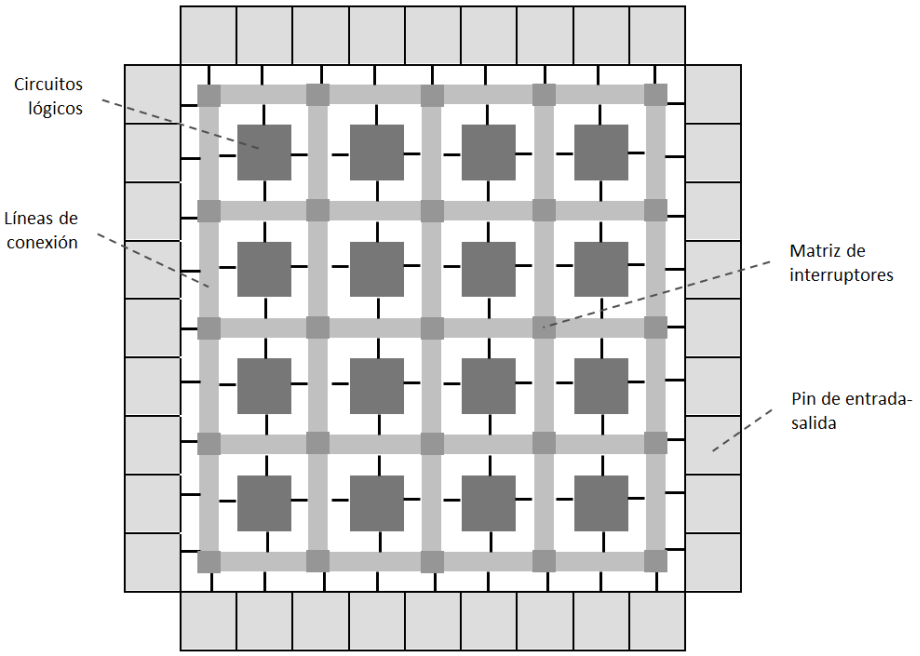


Figura 2.7. Arquitectura de una FPGA.

Las FPGA se pueden clasificar atendiendo a diferentes criterios. Uno de ellos es la forma en que se programan. Algunas FPGA no guardan su configuración, esta se almacena en una memoria externa, y cada vez que se inicia el sistema se carga la configuración en la FPGA. Por otro lado, las hay que son capaces de almacenar internamente su configuración, aún desconectadas de la alimentación. Entre las que almacenan su configuración cabe diferenciar; las que solo se pueden programar una sola vez (OTP, *One-time Programmable*), sin posibilidad de reprogramación; y las reprogramables.

Una FPGA puede quedarse limitada para un diseño por no tener suficientes circuitos lógicos, suficiente número de pines de entrada-salida o por agotamiento del conexionado interno. Por este motivo algunos fabricantes dan la posibilidad a las herramientas de enviar a su servidor la información estadística de los recursos usados en los diseño. Esta información se usa para dimensionar los recursos de las nuevas familias de FPGA a los sistemas generados por los diseñadores.

La gran ventaja de las FPGA frente a los ASIC es que son programables por el diseñador, sin necesidad de ser enviadas a una fábrica. Las FPGA presentan como ventajas: que no existe ingeniería no recurrente, mínimo tiempo de desarrollo, facilidad de depuración y verificación, menor tiempo de acceso al mercado, alto paralelismo de datos, tamaño y formato de datos flexible, y *pipelined* flexible [Cofer y Harding, 2006; Maxfield, 2004]. Aunque la frecuencia de reloj en las FPGA no es tan elevada como en los DSP, con las características anteriores se consigue procesar un mayor flujo de datos. Para pocas unidades, el precio de las FPGA es menor que los elementos ASIC, pero mayor que los microprocesadores. En general, el consumo de potencia en las FPGA es mayor que el caso de los ASIC, pero menor que los microprocesadores. Por todo lo anterior las FPGA son apropiadas para el desarrollo de prototipos cuando el flujo de datos a procesar es elevado, como es el caso del procesado digital de señales. Por otro lado muchas empresas ofrecen placas de circuito impreso que incluyen la FPGA y dispositivos externos. Estas placas contienen: convertidores analógicos-digitales y digitales-analógicos, memorias, dispositivos para entrada-salida de audio y video, puertos de comunicaciones, etc. El uso de estas placas

evita la tediosa tarea de su diseño y fabricación; y lo más importante, permiten la creación de sistemas completos.

Cabe destacar la aparición, a principio de este siglo, de las matrices analógicas programables por el diseñador (FPAA*, Field Programmable Analog Array*). Obviamente, estos dispositivos analógicos programables aparecen mucho más tarde que sus homólogos digitales por razones tecnológicas [Dong et al., 2006]. Muchos de estos dispositivos se han desarrollado con el propósito específico de solucionar ciertos problemas, otros se han desarrollado con carácter general. Entre estos últimos cabe destacar el fabricante Anadigm, [Anadigm, 2015]. Los fabricantes de FPAA ofrecen en sus herramientas de diseño librerías de sistemas analógicos que se pueden configurar e interconectar. Existen dos tipos de FPAA, las que funcionan como sistemas de tiempo discreto y las de tiempo continuo. Las primeras se basan en técnicas de capacidades conmutadas, las señales son muestreadas de acuerdo a un reloj, pero siguen siendo sistemas analógicos dado que las muestras no sufren cuantificación. En general, las FPAA pueden usarse para el diseño de una NN porque permiten hacer multiplicaciones por constantes y sumas; además, incluyen bloques que son funciones de transferencia configurables. El principal inconveniente que presentan es el coste de área; es decir, solo pueden implementarse NN pequeñas por que disponen de recursos muy limitados. Por otro lado, los sistemas FPAA presentan mucha menor velocidad y mayor consumo de potencia que las FPGA. El rango y resolución de ganancias de los sistemas en las FPAA es mucho más limitado que en las FPGA; por ejemplo, en el caso de amplificadores que operan como multiplicadores por una constante. Es decir, los sistemas en las FPGA consiguen ganancias mucho más pequeñas, o más grandes, que en las FPAA, y además con mayor resolución numérica. Finalmente el rango dinámico de la amplitud de las señales en las FPGA es mucho mayor que en las FPAA. En este sentido [Selow et al., 2009] analiza las prestaciones de una FPAA frente a una FPGA para algunas aplicaciones típicas de procesado de señal. De todas formas, a pesar de los inconvenientes, conviene no descartar el uso de las FPAA para el diseño de NN. El motivo es que es una solución integrada, reconfigurable y económica; sobre todo si se desean integrar en sistemas analógicos ya existentes.

Por los motivos expresados en los párrafos anteriores el desarrollo de esta tesis se centra en las FPGA. Como resumen, se muestra en la figura 2.8, las prestaciones de las diferentes tecnologías frente a su coste económico para un prototipo. Obviamente, si el volumen de producción aumenta para dispositivos ASIC, su coste unitario disminuye, manteniéndose las prestaciones. Además, bajo ciertas condiciones, es posible migrar los diseños realizados para FPGA a dispositivos ASIC.

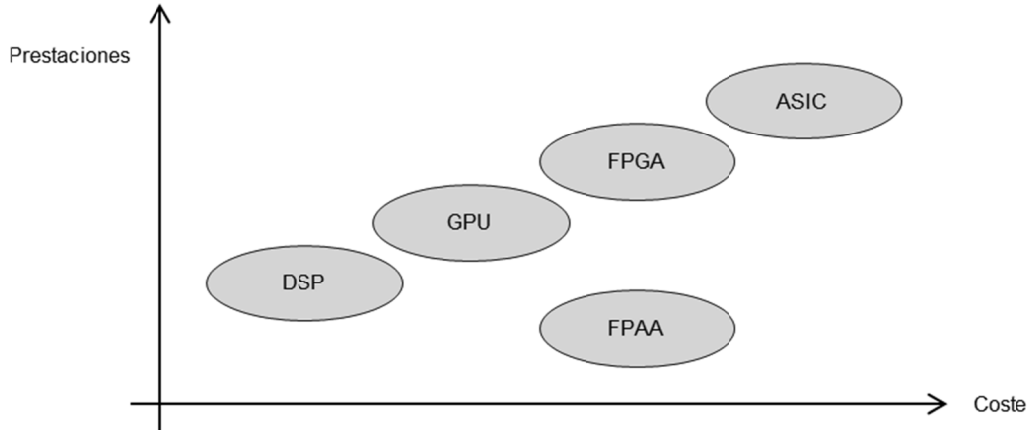


Figura 2.8. Prestaciones frente al coste económico para las diferentes tecnologías en el desarrollo de un prototipo.

2.5 Los métodos de diseño para FPGA

Una vez elegida la tecnología para desarrollar los prototipos, en este caso FPGA, debe elegirse el método de diseño. Existen diferentes métodos de descripción de un circuito en una FPGA, en todos ellos se diseña de forma independiente de la tecnología del dispositivo. Es decir, el diseñador no tiene que conocer la arquitectura interna de la FPGA, ni las características de los circuitos semiconductores. En este apartado se describen los diferentes métodos disponibles para este tipo de diseño.

2.5.1 Edición de esquemáticos

El método más intuitivo, usado desde hace decenios, es la edición de esquemáticos. Con este método el diseñador dibuja y describe los sistemas digitales usando puertas lógicas, registros, flip-flops, etc; básicamente colocándolos y conectándolos. Esto se llama “captura de esquemáticos” y en el proceso se describe el

sistema completo, pudiendo usarse niveles de jerarquía o sistemas ya creados. Obviamente, para evitar el dibujo manual de los sistemas, pronto se hizo necesario entornos de diseño gráfico sobre computador. Este método puede usarse para sistemas pequeños o bien conocidos por el diseñador, donde se pueden describir rápidamente y son fácilmente modificables. Para esta técnica se deben elaborar, con o sin ayuda de herramientas auxiliares, las tablas de Karnaugh de los sistemas combinacionales y secuenciales [Floyd, 2006]. Cuando los sistemas aumentan en tamaño la elaboración de estas tablas y circuitos puede llegar a ser inviable. En este caso esta técnica deja de ser aconsejable. Por otro lado, existe un formato estándar de intercambio de diseño electrónico (EDIF, *Electronic Design Interchange Format*) independiente de los fabricantes [EDIF, 2000]. El formato EDIF puede usarse para el intercambio de esquemáticos, pero los sistemas están descritos en modo de texto. Tanto Xilinx como Altera permiten introducir diseños en formato EDIF. Altera genera el formato EDIF desde otras formas de descripción de su propio entorno. En cambio, Xilinx solo genera el formato propio NGC (*Native Generic Database*); esto es así para mejora el rendimiento del flujo de diseño y optimizar el uso de sus circuitos. El formato estándar EDIF es de uso limitado en el intercambio de esquemáticos, y Xilinx obliga a diseñar los esquemáticos con su propio editor de circuitos. Esto hace que un circuito esquemático editado sobre una herramienta de diseño de un fabricante de FPGA no sea exportable a la herramienta de otro fabricante; a menos que se genere el formato EDIF, que no siempre es posible. En ese caso, si se cambiase de fabricante de FPGA se debe rediseñar el sistema en la nueva herramienta.

Finalmente, pueden usarse herramientas de diseño de empresas, que no son fabricantes de FPGA. Estas desarrollan entornos que soportan la edición de esquemáticos para diversos fabricantes de FPGA; pero estas herramientas no son gratuitas [Active-HDL, 2014; Synplify Pro, 2014].

2.5.2 Los lenguajes de descripción hardware

En la década de los años ochenta, aparecen los lenguajes de descripción hardware (HDL, *Hardware Description Language*). Estos lenguajes permiten describir sistemas digitales usando texto, cosa que facilita su diseño y modificación. Por medio de una

compilación se genera la arquitectura del circuito. De esta forma se solventan las dificultades del diseño usando esquemáticos. En los lenguajes de programación ordinarios las sentencias se ejecutan de forma secuencial, y se mantiene la dependencia entre ellas aunque haya una ejecución paralela. En contraposición, en los HDL las sentencias son concurrentes; es decir, se ejecutan de forma simultánea, tal y como se comportan un grupo de puertas lógicas en un circuito digital. Para ejecutar un grupo de sentencias de forma secuencial se han de incluir dentro de lo que se conoce con el nombre de proceso. Un proceso es la descripción de un sistema secuencial, que pasa por una serie de estados. Los procesos son concurrentes entre ellos; es decir, se ejecutan de forma simultánea.

Existen dos HDL estándar y ampliamente usados. Uno es VHDL [Pedroni, 2004], que es un doble acrónimo (*Very High Speed Integrated Circuit Hardware Description Language,* VHSIC-HDL). La traducción correcta de este acrónimo sería "lenguaje muy rápido de descripción hardware para circuitos integrados". El otro HDL estándar es Verilog [Palnitkar, 2003].

El VHDL fue inicialmente desarrollado por el Departamento de Defensa de los Estados Unidos. Actualmente VHDL es un lenguaje estándar definido por el IEEE y ha sufrido diferentes revisiones [IEEE Std 1076, 2009]. Por otro lado, Verilog se empezó a desarrollar en *Automated Integrated Design Systems* y también es un estándar definido por IEEE a lo largo de una serie de revisiones [IEEE Std 1364, 2006]. Verilog es más compacto que VHDL, y más fácil de aprender y usar. Esta facilidad se debe a su parecido con el lenguaje C. Los dos lenguajes se usan para el diseño en FPGA. Verilog es preferido para el diseño en sistemas ASIC por acceder a niveles más bajos en hardware, pero VHDL permite el uso de tipos de datos más complejos. Debe destacarse, que la descripción de un sistema usando uno de estos dos HDL estándar, hace posible la portabilidad a cualquier dispositivo FPGA.

Aparte de los dos HDL estándar cabe destacar AHDL (*Altera Hardware Description Language*), que es específico de Altera Corporation para sus dispositivos y herramientas [AHDL, 2014]. La sintaxis de este lenguaje es parecida a la del lenguaje Ada. Una ventaja de este lenguaje es que aprovecha mejor las características de los

dispositivos de Altera, pero como desventaja presenta que no es portable a otros fabricantes.

2.5.3 Lenguajes de alto nivel

Con la aparición de VHDL y Verilog aparece un curioso fenómeno que se describe a continuación. Los programadores siguen usando los lenguajes de programación de alto nivel (C, Java, etc.) para ejecutar programas secuenciales. Estos programas se ejecutan sobre computadoras, normalmente en punto flotante. Algunos de estos programadores, mayormente formados en el ámbito de la computación y no en el diseño electrónico, se preguntan por qué no pueden diseñar sistemas digitales sobre dispositivos programables. Este deseo se basa en el hecho de que la descripción del circuito se hace en forma de texto, igual que en los lenguajes de alto nivel. A partir de aquí empiezan a desarrollarse entornos de descripción de circuitos basados en esos lenguajes de alto nivel, que no fueron concebidos para la descripción hardware. Estos entornos aparecen a partir de los años noventa y se basan sobre todo en Java y C. Muchos de estos sistemas son abiertos y disponibles de forma libre, su origen está en foros y universidades. Otros entornos son propietarios de empresas. Como inconveniente presentan el hecho de que el diseño solo es portable a pocos fabricantes; y dentro de cada fabricante, a unas pocas familias de dispositivos. Además, los entornos gratuitos pueden quedar restringidos; bien por no incluir los dispositivos más recientes de un fabricante o por no dar suficiente soporte a los diseñadores.

En el lenguaje Java se basa JHDL (*Java Hardware Description Language*), este entorno es gratuito y solo soporta dispositivos de Xilinx, en el año 2006 se realizó su última actualización [JHDL, 2006]. El entorno C-to-Verilog traduce código C a Verilog, es gratuito, en el año 2009 se hizo su última actualización [C-to-Verilog, 2009]. El entorno SystemC se basa en C++, y es un estándar en código abierto de IEEE desde el año 2005 [IEEE Std 1666, 2011]. Además, SystemC está soportado por más de veinte desarrolladores de herramientas de diseño.

También pueden describirse sistemas con editores de forma de onda, que si bien para circuitos combinacionales pueden ser útiles, para circuitos secuenciales pueden ser de uso complicado. Algunas herramientas incluyen un editor de diagrama de flujo o de estados. Con estos entornos se edita el gráfico de los estados y las transiciones; son muy útiles para el diseño para sistemas secuenciales. Las diferentes formas de descripción de un circuito se pueden combinar; es decir, un sistema se puede componer de diferentes subsistemas donde cada uno de ellos se describe con algún método de los anteriormente descritos.

2.5.4 Los entornos de los fabricantes

Actualmente los fabricantes de FPGA ponen a disposición de los usuarios entornos de diseño estándar, en ellos se pueden describir los circuitos de las formas descritas anteriormente. Estas herramientas no son en principio gratuitas, aunque se pueden pedir como donación a instituciones académicas. También tienen versiones de demostración, bien por limitación en el tiempo de uso o por el tamaño del diseño. Estas herramientas permiten compilar y simular el diseño, para finalmente generar el fichero de programación para la FPGA del fabricante. Estos entornos permiten gestionar el conjunto de ficheros generados en el diseño, asignar los pines de entrada-salida, seleccionar opciones para la compilación, etc. Finalmente dan información de los recursos hardware requeridos, de la velocidad máxima del circuito y de la potencia consumida.

Los fabricantes de FPGA también incluyen librerías de componentes parametrizables, por ejemplo: memorias, unidades aritméticas, etc. También disponen de módulos con propiedad intelectual (IP, *Intellectual Property*) disponibles después del pago del canon correspondiente; a veces se ofrece de forma gratuita, con limitaciones en el tiempo de uso. El código HDL generado en estos casos solo es válido para los dispositivos del propio fabricante. En el dispositivo FPGA pueden integrarse otros módulos empotrados que incluyen software, como por ejemplo microprocesadores, funciones de comunicaciones, etc. Así se constituye lo que se conoce por sistema en un chip (SoC, *System on Chip*).

Los módulos integrables, por ejemplo los microprocesadores, pueden ser de tipo *hard,* son los que se incluyen junto a los recursos programables de la FPGA. Es decir, el microprocesador tiene un área fija asignada dentro de la FPGA. También pueden ser de tipo *soft,* en este caso el entorno de diseño los difunde en la FPGA aprovechando los recursos lógicos y de conexionado disponibles. En medio se encuentran los dispositivos *firm*, estos ya tienen la descripción para ser incluidos en la FPGA aprovechando sus recursos programables, solo están definidos para ciertos dispositivos. Además, los dispositivos *firm* no se pueden introducir en cualquier FPGA, ni se pueden recompilar. Estas tres clases de procesadores presentan diferentes características durante el diseño y diferentes prestaciones físicas en el funcionamiento.

Por otro lado los principales fabricantes de FPGA ofrecen entornos de diseño sobre *Simulink* de Matlab [Simulink, 2014]. Estas herramientas suponen una ampliación del entorno estándar, están disponibles en las mismas condiciones y solo sirven para los dispositivos del fabricante. Tras la instalación de estas herramientas aparecen uno o varios *Blocksets* nuevos en *Simulink*, que incluyen los bloques del fabricante. Con *Simulink* se diseña de forma rápida y flexible, es un entorno de diseño gráfico que usa diagrama de bloques. Los bloques son configurables mediante sus ventanas de diálogo. Con estas utilidades se diseña el sistema usando aritmética en punto fijo, aunque en los últimos años también se incluye unidades aritméticas en punto flotante. Una vez diseñado el sistema puede simularse aprovechando las ventajas de *Simulink*. La primera ventaja es el acceso directo al espacio de variables de Matlab, lo que facilita la generación de las señales de entrada y el análisis de las salidas. Por un lado, se pueden usar las diferentes fuentes de señal de *Simulink*, estas se encuentran en el *Blockset Sources*. Por otro lado, se pueden usar sus visualizadores y destinos de señal, que se encuentran en el *Blockset Sinks.* Las señales obtenidas se muestran como si fueran de punto flotante, lo que facilita el análisis de las formas de onda. Estas simulaciones son muy rápidas porque tienen un nivel pobre de detalle de los circuitos de la FPGA, esto hace que la estimación de área ocupada sea aproximada. Por otro lado no aporta estimaciones de máxima velocidad o potencia consumida. Como ventaja es posible la comprobación de la total funcionalidad del sistema. Esto es posible por ser rápidas sus simulaciones y Matlab poder manejar gran cantidad de datos. Al disminuir el tiempo

de diseño, y ser entornos más versátiles, se facilita al diseñador la comprobación de diferentes arquitecturas y configuraciones.

Una vez simulado el sistema se realiza una compilación que genera el proyecto para el entorno estándar del fabricante. El sistema queda descrito de forma estructural en VHDL o Verilog. Aquí los HDL son usados como una etapa intermedia. El código HDL generado no es portable a otros fabricantes, por que aprovecha las peculiaridades y arquitectura de la FPGA; dicho de otra forma, usa lo que se llaman primitivas, que son librerías de bloques específicos del fabricante. De todas formas, este tipo de herramientas pueden ser usadas para comprobar la arquitectura y prestaciones numéricas de un sistema digital; por ejemplo, el efecto del número de bits en diferentes partes de un circuito.

La comparación entre dispositivos de diferentes fabricantes es una tarea compleja. Existen multitud de familias de dispositivos; además, las diferentes tecnologías cambian rápidamente, así como sus prestaciones. En cuanto a la comparación de las herramientas de diseño equivalentes entre fabricantes la tarea es igualmente complicada. Estas herramientas van cambiando rápidamente, y los fabricantes mantienen en sus páginas web las diferentes versiones, incluso con diferentes simuladores. A esto hay que añadir la dificultad de que cambian la estructura de las páginas web. Debe destacarse que la forma de adquirir la licencia y de habilitarla para las herramientas también sufre cambios. Los fabricantes tienen las herramientas para diferentes versiones de sistemas operativos, principalmente Windows y Linux; las utilidades disponibles pueden depender del sistema operativo elegido. Cuando el software de diseño se apoya en *Simulink* solo son válidas unas determinadas versiones de Matlab. Resumiendo, si alguien quiere instalar un paquete de estas herramientas tendrá que comprobar la compatibilidad con su sistema operativo, incluso si es de 32 o 64 bits. Por otro lado, debe disponer de la versión de Matlab necesaria.

2.5.5 Xilinx versus Altera

La empresa Xilinx es la que acapara la mayor parte del mercado de FPGA a nivel mundial [Xilinx, 2014]. Aunque es líder en la investigación y desarrollo de estos

dispositivos debe destacarse que es una empresa tipo *fabless* (*fabrication less*). Este tipo de empresa diseña, testea y vende los circuitos integrados; pero no los fabrica. La fabricación es encargada a una empresa de circuitos integrados. Esto produce ahorro económico porque se encargan en países donde los costes son más baratos, y le permite concentrarse en el desarrollo de los dispositivos y de las herramientas de diseño. Actualmente Xilinx dispone de dos entornos de diseño; por un lado ISE (*Integrated System Environment*), con versiones hasta octubre de 2013 [ISE, 2013]; y por otro Vivado, con versiones desde 2012 hasta la actualidad [Vivado, 2014]. Ambos entornos incluyen *System Generator*, que es la utilidad de diseño sobre *Simulink* [System Generator, 2014].

El segundo suministrador en importancia de FPGA es Altera, también es una compañía tipo *fabless* [Altera, 2014]. El entorno de diseño de Altera se llama Quartus II [Quartus, 2014] y la utilidad para *Simulink* es *DSP Builder* [DSP Builder, 2014]. Es posible obtener donación de placas, por parte de Altera, dentro de su programa universitario. Estas placas incluyen sus FPGA, y son desarrolladas por otras empresas. Existen otros fabricantes que se reparten el resto del mercado de las FPGA, cabe destacar: Lattice, Atmel, Microsemi, Achronix y Tabula.

El software de diseño *AccelDSP* fue ofertado entre los años 2006 y 2010 por Xilinx [AccelDSP, 2014]. Este software era de pago, aunque Xilinx permitía usarlo durante un periodo de tiempo de forma gratuita como evaluación. Esta herramienta se basaba en *System Generator*; y por tanto en Matlab y *Simulink*. Con fuertes restricciones en el código en punto flotante *AccelDSP* automatizaba la conversión de este a punto fijo. El código de Matlab se convertía a C++ en punto fijo o a una descripción de bloques en *System Generator*. Finalmente se generaba el sistema descrito en VHDL o Verilog, no portable a otros fabricantes. Curiosamente *AccelDSP* de Xilinx nunca tuvo su herramienta equivalente en el competidor Altera, y siendo *AccelDSP* una herramienta novedosa desapareció tan rápido como apareció. Los motivos pueden ser la falta de rentabilidad económica, quizás los diseñadores se conformaban con *System Generator* y la interactuación que puede tener con Matlab en punto fijo, y no consideraron necesario pagar el canon.

En cuanto a las herramientas de diseño estándar de Xilinx y Altera se puede hacer una comparación cualitativa. La herramienta de diseño de Xilinx es tradicionalmente más compleja y difícil de manejar, pero más versátil y potente; tiene tiempos de compilación mayores y dispone de utilidades que permiten una precisa estimación del consumo de potencia. Por otro lado, su equivalente de Altera es más fácil de manejar y de uso intuitivo, pero menos potente y flexible. Cualquier proyecto puede realizarse con cualquiera de los dos principales fabricantes, pero se puede afirmar que Altera es más apropiado para el uso docente o académico; mientras que Xilinx es preferido en tareas de desarrollo e investigación.

2.5.6 Otras empresas y entornos académicos

Existen compañías que crean software no gratuito que puede usarse para diseñar en FPGA para diferentes fabricantes. La empresa Synopsys ofrece la utilidad Synplify en diferentes modalidades [Synplify, 2014]. Este entorno de diseño permite elegir entre los fabricantes de FPGA: Xilinx, Altera, Lattice, Atmel, Microsemi y Achronix. Para usar esta herramienta se necesita pagar un canon, aunque puede ser usada en forma de evaluación. Este sistema, aunque más caro, permite la portabilidad del diseño para los fabricantes que tiene habilitado. Como se puede compilar el diseño para cada uno de los fabricantes es posible comparar las prestaciones; sin necesidad de usar el entorno de cada fabricante por separado.

La compañía National Instruments ha desarrollado LabView (*Laboratory Virtual Instrumentation Engineering Workbench*) que es un entorno donde se diseña de forma visual y gráfica [LabView, 2015]. Para su uso hay que pagar una licencia, aunque se puede solicitar su evaluación. Se emplea para diseñar sistemas de instrumentación, control, procesado de señal y comunicaciones; admite gran cantidad de equipamiento externo y puertos de entrada-salida. Esta herramienta sobrepasa el diseño para FPGA porque cubre una gran cantidad de equipamiento de las áreas anteriores. Solo soporta las FPGA de Xilinx. Los diseños se pueden introducir usando VHDL, Verilog y de forma esquemática. Este fabricante muestra como una ventaja el evitar el uso de lenguajes tipo HDL en modo de texto; esto no es necesariamente una ventaja, más bien puede ser un inconveniente. La descripción gráfica de un circuito puede ser más difícil de

crear y modificar que su equivalente en código HDL. Si bien el profano debe aprender el HDL elegido, también debe aprender a describir de forma gráfica los circuitos en LabView. Se debe insistir en que la descripción gráfica para LabView no coincide con la edición del esquemático, aunque se parece. Además el código HDL es estándar y portable a otros fabricantes, mientras que la descripción gráfica es particular para National Instruments, y solo válida para los dispositivos de Xilinx que soporte LabView. Entonces puede ser conveniente usar el software de Xilinx, que puede llegar a ser gratis, tanto su herramienta estándar de diseño, como su entorno gráfico de diseño *System Generator* sobre *Simulink*. El uso de LabView puede justificarse si el diseño de la FPGA se integra en un sistema mayor e interdisciplinar que aprovecha las ventajas globales de LabView. Una particularidad de LabView al compilar el diseño de la FPGA es poder usar un compilador: tipo local, tipo servidor o en la "nube" a través de la red. En cualquier caso el compilador no hace otra cosa que usar el propio de Xilinx, en la última forma se evita tener que tener actualizado el compilador de Xilinx, pero una compilación a través de la red puede ser muy lenta por la transferencia de los ficheros; un diseño mediano puede generar tras la compilación varios cientos de Megabytes.

Se han desarrollado muchas otras herramientas de ayuda para el diseño en FPGA, algunas gratuitas y otras de pago. Las primeras provienen de entornos académicos o grupos de investigación, las segundas han sido generadas por empresas. Entre las gratuitas cabe destacar *Floating-Point to Fixed-Point Toolbox* para Matlab [flpfxptoolbox, 2006], que permite el paso de código de punto flotante a punto fijo, y analiza las prestaciones del sistema frente a un error establecido para la representación de punto fijo. Este *Toolbox* es un ejemplo de herramienta pionera en este campo, aunque ya está en desuso y ha sido ampliamente superado. Su página web no se actualiza desde 2006.

2.5.7 Matlab para FPGA

El entorno Matlab se ha convertido en un estándar de hecho, tanto para la comunidad académica y científica, como para la industria. Si bien no es gratuita, en la práctica los diseñadores disponen de ella, entre otros motivos, por el uso de licencias institucionales. Es un entorno habitual para el área de procesado digital de la señal.

Primeramente cabe destacar *Filter Design HDL Coder* de Matlab que tiene la capacidad de generar el hardware necesario para el diseño de filtros [Filter Design HDL Coder, 2014]. Permite obtener la descripción del filtro en VHDL o Verilog en punto fijo, este código es portable a todos los fabricantes de FPGA. También genera las señales necesarias de entrada para la simulación y verificación del filtro.

La utilidad *Fixed-Point Designer* de Matlab permite manejar tipos de datos y herramientas para el desarrollo de algoritmos en punto fijo, usando código de Matlab, modelos de *Simulink* o el editor de diagramas de flujo *Stateflow* [Fixed-Point Designer, 2014]. Esta herramienta propone automáticamente el número de bits y el método de redondeo, que también se pueden especificar manualmente. Las simulaciones permiten observar el efecto del rango y la precisión.

En Matlab cabe destacar la utilidad *HDL Coder* donde los sistemas se describen usando funciones de Matlab, modelos de *Simulink* o el editor de diagramas de flujo *Stateflow* [HDL Coder, 2014]. El código generado puede ser en VHDL o Verilog, y es totalmente portable y sintetizable; además tiene un flujo de trabajo integrado con Altera y Xilinx, tanto para diseños en FPGA como en ASIC. Esta herramienta automatiza y facilita el flujo del diseño, permite control de la arquitectura y de los retardos críticos; además, da una estimación de los recursos hardware necesarios. Permite también comparar la respuesta del código HDL generado con el modelo inicial de *Simulink*. La herramienta *HDL Coder* es relativamente nueva, la primera versión data de marzo de 2012.

Finalmente Matlab ofrece un verificador del diseño realizado en VHDL o Verilog [HDL Verifier, 2014]. Este sistema permite comunicar las señales de Matlab con la placa que contiene la FPGA de Altera o Xilinx, hace funcionar la FPGA en tiempo real y compara las señales obtenidas con las del modelo de Matlab.

En resumen, con *HDL Coder* y *HDL Verifier* se facilita el diseño de sistemas en FPGA o ASIC; acortando el tiempo de la descripción, simulación y verificación del sistema. Esto se puede hacer generando un código portable a cualquier fabricante; o generando un código optimizado para dispositivos de Altera o Xilinx. En este último caso se puede programar los dispositivos desde Matlab y realizar la verificación del diseño.

En esta tesis se usa *System Generator*, que es la herramienta de Xilinx que funciona sobre *Simulink*. Finalizado el diseño con *System Generator* se usó el *Integrated System Environment* de Xilinx, que es la herramienta estándar para FPGA. Se eligieron estas herramientas por la donación de licencias que hace Xilinx a los entornos académicos, por ser herramientas que permiten el diseño rápido y flexible, por aprovechar las ventajas de Matlab y *Simulink*, por la continuidad de estas herramientas en la página web del fabricante en sus diferentes versiones; y finalmente, por el asesoramiento y gestión de errores que facilita a los diseñadores a través de su página web.

Igualmente se podían haber usado las herramientas equivalentes de Altera, pero inicialmente se descartaron por una serie de errores al ejecutar los tutoriales de este fabricante.

2.6 Parámetros de los métodos de diseño

Ya a mediados de los años noventa existían multitud de herramientas de diseño, "*Otra fuente de confusión es la plétora de herramientas disponibles hoy en día*" [Oldfield y Dorf, 1995]. Esto pone de manifiesto la importancia de las herramientas y métodos de diseño para los dispositivos programables. Esta multiplicidad de herramientas se debe a la complejidad de los diseños, y la dificultad para comprobar totalmente su funcionalidad y prestaciones. De hecho, si no se usa la herramienta apropiada, comprobar la funcionalidad de un sistema puede ser más complicado para el diseñador que el propio diseño del sistema.

Entonces caben varias preguntas: ¿cuál es el mejor método de diseño?, ¿cómo se pueden medir las prestaciones de un método de diseño? y ¿cómo se pueden comparar las prestaciones de diferentes métodos de diseño?

Todos los métodos de diseño pueden caracterizarse por los parámetros que se describen a continuación. Estos pueden servir para contestar a las preguntas anteriores.

El coste económico. La herramienta puede tener un coste económico o ser totalmente gratuita. A veces, aun siendo la herramienta de pago, puede conseguirse la

herramienta como versión de demostración; durante un periodo de tiempo limitado o con prestaciones limitadas.

El periodo de aprendizaje del diseñador. Debe considerarse si el diseñador, o equipo de diseño, necesitan un periodo para aprender el entorno; o por el contrario ya lo conocen y pueden empezar el diseño de forma inmediata.

El soporte de las herramientas. Se trata de determinar si el entorno de diseño está bien documentado. Para esto deben existir manuales y estar bien ordenados, ser claros y de fácil localización. También es deseable una herramienta de ayuda insertada en la propia herramienta.

La actualización del sistema. El entorno debe actualizarse periódicamente: ajustes con el sistema operativo, soporte, documentación, funcionalidades, fabricantes y dispositivos soportados. Si una herramienta de diseño no actualiza las FPGA que soporta pronto quedará con dispositivos antiguos y caerá en desuso.

Los sistemas operativos sobre los que funciona. Los sistemas operativos habituales en estos casos son Windows, Linux y Mac. Puede ser que la misma herramienta esté disponible para varios de ellos. Los continuos cambios y versiones de un sistema operativo obligan a la actualización de la herramienta, o puede caer igualmente en desuso.

La ayuda ante errores. Esto es especialmente importante ante errores inesperados, que el diseñador no sea capaz de solventar con la documentación existente. Por ejemplo, Altera y Xilinx tienen un sistema de ayuda en línea en sus páginas web. En este caso el diseñador describe y documenta la incidencia convenientemente, en el plazo de uno o dos días recibe una respuesta personal, y se abre una comunicación entre el diseñador y un asistente, hasta la resolución del problema. Al final el diseñador contesta una encuesta de satisfacción. También existen foros o listas de preguntas frecuentes, donde quizás pueda solventarse el error de forma rápida.

La portabilidad entre fabricantes y dispositivos. La máxima portabilidad se refiere a que el diseño puede llevarse a cualquier dispositivo de cualquier fabricante. Por el contrario puede estar restringido para uno o varios dispositivos de un único fabricante,

tratando de aprovechar sus peculiaridades. La portabilidad permite comparar las prestaciones del diseño para diferentes FPGA.

La flexibilidad del diseño. Un diseño es flexible si cuando se cambia alguno de sus parámetros o especificaciones se puede rediseñar fácilmente.

El tiempo de diseño. Se refiere a la facilidad para describir el diseño, realizando diferentes modificaciones, hasta conseguir el sistema final.

El tiempo de compilación. Se trata de evaluar si las diferentes compilaciones son rápidas, y si cuando se modifica solo una parte del diseño la recompilación es parcial y no del sistema completo.

El tiempo de simulación. Se trata de evaluar la existencia y prestaciones de diferentes tipos de simuladores, su facilidad de uso, rapidez y visualización de las formas de onda. Básicamente, la etapa de simulación debe comprobar la funcionalidad total del sistema y verificar la máxima velocidad de funcionamiento del diseño.

El tipo de reconfiguración del sistema. Una vez que se ha realizado la programación de la FPGA y se procede a cambiar una parte del diseño, la reconfiguración de la FPGA puede ser total o parcial. En el caso de reprogramación parcial la parte del diseño que no ha sufrido cambio puede seguir funcionando mientras se configura la parte modificada. Obviamente, está técnica está íntimamente ligada al tipo de dispositivo.

La seguridad y privacidad del diseño. Una vez que se ha programado la FPGA se trata de que su configuración no pueda ser leída y por tanto el diseño no pueda ser copiado. Las diferentes tecnologías permiten diferentes técnicas para proteger el diseño [Cofer y Harding, 2006; Maxfield, 2004].

Interactuación con otras herramientas. Se trata de determinar si la herramienta puede comunicarse con otros programas, por ejemplo: simuladores, editores de forma de onda, programas matemáticos, etc. Esto supone una facilidad para evaluar las prestaciones y funcionalidad del sistema.

En definitiva, es el diseñador o el equipo de diseño el que debe elegir la herramienta conveniente; en función de las características anteriores, de su experiencia previa y de las necesidades del diseño. Los parámetros enumerados

anteriormente se refieren a características que existen durante la realización del diseño, hasta justo el momento en el que entra en funcionamiento.

2.7 Factor de calidad de un diseño: área, velocidad y potencia

Una vez se ha finalizado el diseño, pueden proponerse parámetros que permitan evaluar la calidad del sistema. Estos parámetros también permiten compararlo con otras implementaciones, realizadas con diferentes métodos o diferentes dispositivos. Estos parámetros son las prestaciones físicas del diseño:

- el área,
- la potencia consumida,
- y la velocidad del sistema.

El concepto de área se refiere a la cantidad de recursos ocupados. Esto da idea del tamaño físico del diseño y del tamaño mínimo requerido para la FPGA. Un diseño será tanto mejor cuantos menos recursos hardware necesite, y así puede usarse una FPGA más pequeña y barata. Si el tipo de FPGA viene dado por otros factores, entonces el ahorro de área dejará recursos disponibles para otras funciones o futuras ampliaciones. Cuando se esté usando una determinada FPGA la comparación de área entre diferentes diseños es una tarea clara de realizar. El problema aparece cuando se quiere comparar diseños de fabricantes diferentes, donde las unidades de circuitos lógicos son distintas y reciben nombres distintos. En este caso, la comparación de área se puede reducir al coste económico de la compra de esos circuitos, si se usan distintas tecnologías. Para la misma tecnología la comparación de área se podría hacer estimando el número de transistores o de puertas equivalentes, pero este dato casi nunca está disponible, de hecho las utilidades que lo calculan han sido eliminadas de las herramientas en años recientes. Esta eliminación puede deberse a que en la industria no tiene interés su cálculo; y aunque puede tener interés científico y académico su estimación no es fácil.

Algunos autores estiman y comparan el coste de los recursos de conexión necesarios; es decir, cableado interno de la FPGA, pero las conexiones físicas no tienen coste computacional. El aumento del número de conexiones se traduce en un aumento

de la potencia, al tener que conectar las señales a más líneas conductoras y otros bloques de lógica. Por otro lado, se traduce en un aumento de los retardos del circuito. En resumen, el aumento del cableado interno se traduce en un aumento de la potencia y del retardo, claramente las otras dos prestaciones físicas.

Otro elemento del área, no típicamente usado, es el uso de pines de entrada-salida. Igual que los recursos de interconexión su uso se traduce en consumo de potencia y retardo, sin carga computacional. El número de pines de entrada-salida del sistema viene dada por la resolución numérica necesaria en el diseño, y su aumento se traduce en una mayor necesidad de recursos hardware para la computación. Un diseño en una FPGA puede colapsarse por falta de lógica programable, de recursos de conexión o falta de pines de entradas-salida.

Un concepto ligado al uso del área es el concepto de granularidad. La granularidad se refiere al tamaño de los circuitos lógicos de la FPGA. Así las FPGA pueden tener granularidad fina, media o gruesa. Una FPGA de grano fino tiene elementos lógicos pequeños; una de grano grueso tiene elementos lógicos grandes, con mayor número de bits de entrada-salida en sus operadores. A modo de ejemplo, puede pensarse en sumadores, de pocos o muchos bits respectivamente. En el primer caso si se quiere realizar una suma de muchos bits se usarán varios sumadores conectados convenientemente; en el segundo caso se puede realizar con solo un sumador. Las prestaciones de retardo y consumo en el segundo caso son mejores que en el primero. La primera solución es más flexible y solo ocupa los recursos estrictamente necesarios, los elementos sobrantes quedan libres para otros usos. En el segundo caso el sumador usado puede tener más bits de los necesarios, y quedan entradas-salidas sin usar; no se aprovecha toda la funcionalidad del sumador, y la parte no usada no puede ser reutilizada. Entonces dos diseños distintos de un mismo operador, y diferente número de bits, pueden ocupar la misma área si la granularidad de la FPGA es gruesa. Solo se puede comparar el área ocupada sobre las FPGA que tengan la misma granularidad. Si tienen granularidad distinta la comparación es difícil y hay que tener en cuenta el tamaño de los bloques.

En conclusión, comparar el área es trivial si diferentes versiones de un diseño se compilan sobre el mismo dispositivo; cambiar de fabricante o de dispositivo complica la comparación en términos de área.

El segundo parámetro es la potencia consumida. Básicamente hay dos tipos de potencia, la de corriente continua o estática, y la dinámica. La primera se produce por el solo hecho de conectar la FPGA a las fuentes de alimentación, se debe a las corrientes de fuga a través de la FPGA; esta potencia depende de la tecnología de la FPGA. En cuanto se conectan las señales de entrada aparece la componente dinámica, también depende de la tecnología, alguna tecnologías solo producen consumo de corriente cuando las señales cambian; es decir, en el flanco de subida o bajada. En este último caso la potencia dinámica depende directamente de la frecuencia de operación. Las herramientas de los principales fabricantes incluyen estimadores de potencia. Estos analizadores de potencia también estiman la temperatura de funcionamiento de la FPGA; pero para ello debe establecerse la temperatura del ambiente, y el tipo de disipación de que dispone la FPGA. La estimación de potencia es detallada para cada bloque del diseño y para los pines de entrada-salida; también se indica para cada valor de alimentación de la FPGA. Un diseño será tanto mejor cuanto menor sea la potencia consumida. La potencia se invierte en el propio funcionamiento de la FPGA y en pérdidas en forma de calor. Una disminución de la potencia consumida se traduce en una mayor autonomía si la FPGA se alimenta con baterías. Dicho de otra forma, si se disminuye la potencia consumida, para una determinada autonomía, se puede disminuir las dimensiones de la batería; esto hace disminuir el peso y mejora la portabilidad física.

El tercer parámetro físico es la velocidad a la que puede funcionar el diseño. Por ejemplo, si se trata de un filtro, sería la tasa máxima de muestras que puede soportar en la entrada (muestras por segundo), y vendría dada por el máximo valor de la frecuencia de reloj asociado. Normalmente la velocidad del diseño se indica con el valor de su frecuencia máxima, casi todos los diseños son sistemas secuenciales. También la velocidad se podría indicar por el periodo de la frecuencia máxima, que sería un valor mínimo. Si el sistema es totalmente combinacional; es decir, no existe frecuencia de reloj, la velocidad vendría dada por el retardo máximo entre las entradas-salidas; es deseable que este retardo sea lo más pequeño posible. En el caso de sistemas secuenciales puede proponerse para medir la velocidad el periodo mínimo. Entonces ambos casos, secuenciales y combinacionales, pueden compararse directamente por ser tiempos que se expresan en segundos.

En la etapa de diseño puede haber restricciones de área, potencia y velocidad. El diseño debe cumplir estas restricciones; y el diseñador debe procurar minimizar tanto las que estén restringidas, como las que no. La cuestión es cómo comparar dos diseños con la misma funcionalidad; que incluso pueden haber sido realizados con métodos distintos. Un primer método puede ser una representación cartesiana con tres semiejes positivos como en la figura 2.9, donde el mejor diseño será el que tenga la menor distancia al origen. Este tipo de representación puede dar una idea intuitiva de las prestaciones de diferentes diseños.

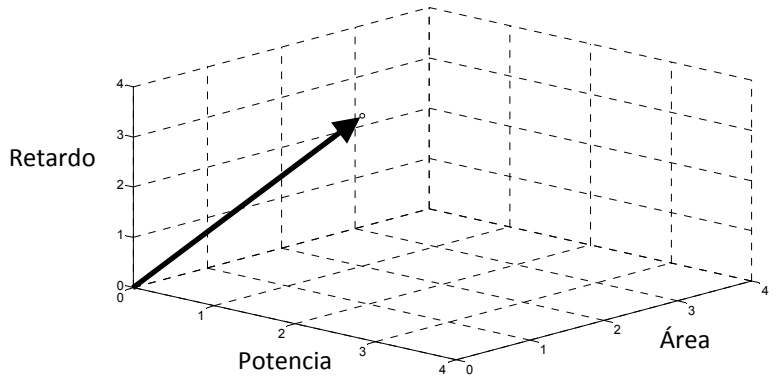


Figura 2.9. Representación en semiejes del área, potencia y retardo.

Otro método para comparar diseños podría ser mediante una expresión matemática, cómo la fórmula 2.1; el resultado es un factor de calidad que será tanto mayor cuanto mejor sea el diseño.

$$C=\frac{1}{\text{Área}\cdot\text{Potencia}\cdot\text{Retardo}} \tag{2.1}$$

Otra representación de los parámetros del diseño puede ser mediante un prisma triangular truncado como en la figura 2.10, donde con cada arista vertical se representa el valor de un parámetro. El diseño será tanto mejor cuanto menor sea el volumen de la figura o la altura de su punto medio. Otra forma de comparar prestaciones sería mediante el uso de un diagrama en tela de araña, como muestra la figura 2.11.

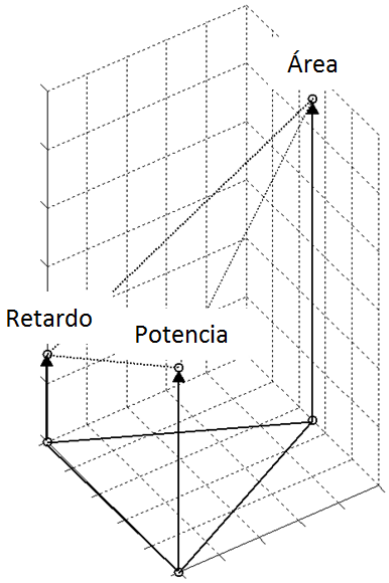


Figura 2.10. Representación en un prisma del área, potencia y retardo.

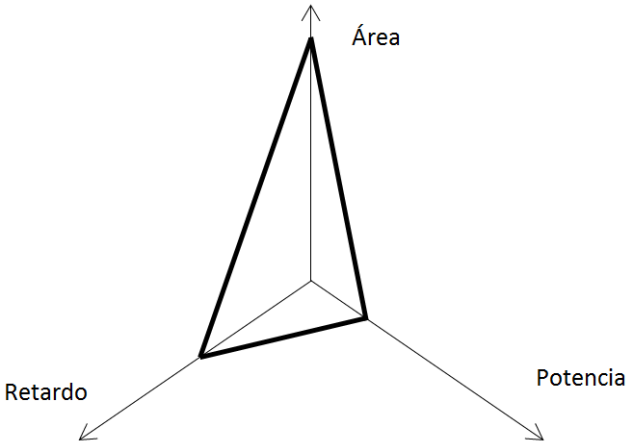


Figura 2.11. Representación en un diagrama en tela de araña del área, potencia y retardo.

En las representaciones anteriores se pueden añadir factores de penalización si los parámetros se consideran con diferente mérito. Algunos autores solo estudian el área y la velocidad, dejando de lado el consumo de potencia, lo que no es aconsejable [Gomperts et al., 2011; Himavathi et al., 2007; Savich et al., 2007; Yen et al., 2004]. Los parámetros son dependientes; en general, se consigue aumentar la velocidad de un diseño a consta de mayor área y potencia consumida [Deng et al., 2011].

2.8 Las herejías metodológicas

Hoy en día los diseños digitales han crecido en complejidad. Por este motivo es normal comprobar el algoritmo en un lenguaje de alto nivel en formato de punto flotante, habitualmente suele usarse el lenguaje C. También suele usarse Matlab y *Simulink* por la gran librería y facilidades de simulación de que disponen. Con esta etapa del diseño se persigue comprobar la funcionalidad total del sistema; por ejemplo, si se tratase de un filtro, comprobar la respuesta en frecuencia. Posteriormente puede diseñarse el sistema en un lenguaje de alto nivel usando aritmética de punto fijo; con esto se consigue comprobar el efecto del número de bits en la funcionalidad del sistema. Una vez fijada la arquitectura en punto fijo se diseña el circuito con alguna de las técnicas anteriores. El modelado de punto fijo con un lenguaje en alto nivel no es obligatorio, porque se dispone de herramientas de diseño de circuitos que permiten evaluar el efecto del número de bits [System Generator, 2014; DSP Builder, 2014]. Estas últimas simulaciones son más lentas que las simulaciones en lenguajes de alto nivel. Esto es debido a que las herramientas simulan el comportamiento del circuito y necesita compilarlo cada vez que se modifica.

A veces en los entornos académicos un diseñador desarrolla el modelo en punto flotante, con un lenguaje de alto nivel; y posteriormente, otro diseña el circuito en punto fijo, por ejemplo en usando un HDL. Esto tiene serios inconvenientes:

- Obliga a una traducción manual, en inglés llamada *hand-coded*, lo que es una tarea tediosa si el algoritmo es complicado.
- El segundo diseñador cuando empieza a diseñar no conoce el algoritmo, lo debe estudiar previamente, luego sería aconsejable que las dos etapas las realizase el mismo equipo de diseño.
- El segundo diseñador debe fijar la longitud de los datos y operadores en punto fijo, lo que con un HDL es igualmente tedioso; además las compilaciones y simulaciones son muy lentas. De hecho hay herramientas más apropiadas para esta fase del diseño [System Generator, 2014; DSP Builder, 2014].
- Es muy difícil comprobar la funcionalidad total del sistema usando un HDL. Estas simulaciones son lentas porque incluyen los detalles del circuito. Si el sistema es

complicado, o hay gran cantidad de datos en la entrada, las simulaciones son muy lentas y puede resultar imposible comprobar la total funcionalidad.

- Por el motivo anterior pueden producirse errores en la codificación manual que sean difíciles de detectar y aparezcan cuando el circuito está funcionando.
- Finalmente, pueden haber especificaciones en el algoritmo de punto flotante que faciliten el diseño en punto fijo y mejore sus prestaciones; esto lo puede desconocer el diseñador del modelo en punto flotante.

En conclusión, debe recurrirse a una metodología de diseño que permita la total comprobación de la funcionalidad del sistema, aun cuando el sistema sea complicado y las señales de entrada tengan gran cantidad de datos. Por otro lado, se debe facilitar el estudio del efecto del número de bits en punto fijo sobre la funcionalidad del sistema. Por último, debe ser posible tomar todas las decisiones en la etapa de punto flotante que mejoren el diseño en punto fijo.

Todo lo anterior se muestra en la figura 2.12. Por los motivos anteriores en esta tesis doctoral se usa *System Generator*, que es la herramienta de Xilinx que funciona sobre *Simulink*. Evita los inconvenientes enumerados; a la vez que permite verificar la funcionalidad total del sistema y comprobar el efecto del número de bits.

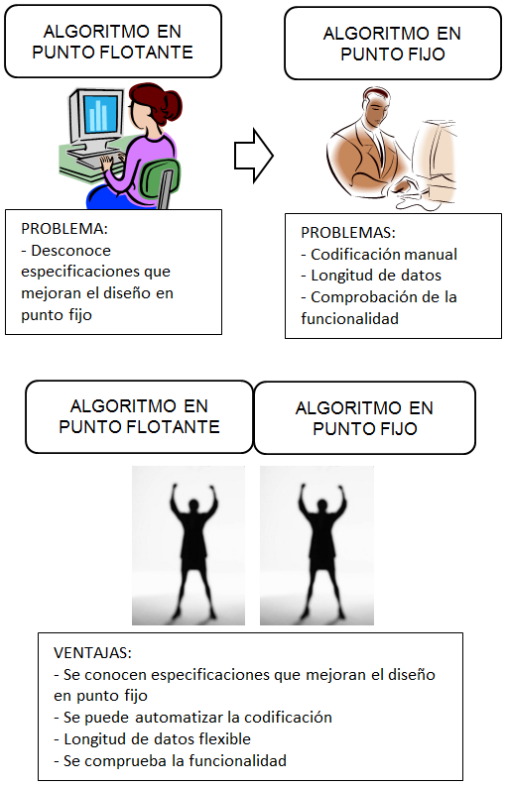


Figura 2.12. Las herejías metodológicas y el proceso apropiado.

CAPÍTULO 3

METODOLOGÍA EXPERIMENTAL

"Después de escalar una gran colina, uno encuentra solo que hay muchas más colinas que escalar".

Nelson Mandela

En este capítulo se plantea una metodología para demostrar la hipótesis indicada en el primer capítulo. El proceso a describir parte de las bases de datos usada y llega, tras diferentes etapas a la descripción física y prestaciones en la FPGA.

3.1 Esquema general de la metodología experimental

Se desarrollaron experimentos sobre distintos escenarios. En cada uno de ellos se dispone de una base de datos, cada una con señales de distinta naturaleza que determinan las entradas de la NN. La arquitectura de las NN será distinta en cada caso, incluso variará el tipo de función de transferencia. En unas situaciones, la NN actuará como clasificador; en otras ocasiones, operará como predictor.

En la figura 3.1 se muestra el esquema típico con las etapas de los experimentos para la metodología planteada. A continuación se describen las etapas mostradas en esa figura.

- ***Base de datos***. Estos son los conjuntos de datos necesarios para diseñar la NN. Pueden ser naturales o sintéticas, y pueden incluir cierto nivel de ruido.
- ***Preprocesado***. Es el bloque que se usa para disminuir el nivel de ruido o distorsión de las señales de entrada, básicamente mediante técnicas de filtrado. Con esto se consigue adecuar las señales de la base de datos como entrada de la NN.
- ***Parametrización***. En esta fase se realiza la extracción de los parámetros, también llamados características. A veces la propia señal de la base de datos no es usada directamente como entrada para la NN. En su lugar se transforma con operaciones matemáticas, y estos nuevos valores son los usados como entrada.
- ***Modelado en punto flotante de la NN***. El primer proceso en el modelado en punto flotante de la NN es la normalización de los datos de entrada. Mediante la normalización los datos se reducen a los rangos [0,+1] o [-1,+1]. Con esto se consigue que todas las señales de entrada tengan el mismo orden de magnitud y se facilita el entrenamiento. Si a priori las entradas tuvieran el mismo orden de magnitud esta operación no es necesaria. Tras la normalización, que es opcional, ya se tienen definidas las entradas. Posteriormente, con un lenguaje o entorno de alto nivel se modela la NN. En este proceso se realiza el entrenamiento y testeo de la NN. Las decisiones a tomar tienen que ver con el número de capas de la NN, el número de neuronas en cada capa y las funciones de transferencia usadas. Obviamente, al final, se tienen los

coeficientes de la NN en formato de punto flotante. La NN queda especificada según la funcionalidad requerida.

- ***Modelado en punto fijo de la NN***. La arquitectura y los coeficientes obtenidos anteriormente constituyen la llamada "regla de oro", junto a la funcionalidad de la NN. A partir de este momento se precisa pasar la aritmética de punto flotante a punto fijo. Esto es así por la idoneidad del formato de punto fijo para ser implementado en dispositivos digitales. La cuestión es buscar el mínimo número de bits que mantenga la funcionalidad de la NN. La disminución del número de bits, aparte de minimizar el área ocupada, disminuye la potencia consumida y aumenta la velocidad de respuesta. En este punto es importante la elección de la herramienta y del método de diseño. El método debe permitir comprobar la total funcionalidad de la NN mediante su simulación.
- ***Implementación en la FPGA***. En la última fase se implementa el diseño en el dispositivo digital programable elegido. Las herramientas de diseño permiten obtener las prestaciones físicas de área, potencia y velocidad. Finalmente se puede configurar la FPGA elegida y comprobar su funcionamiento.

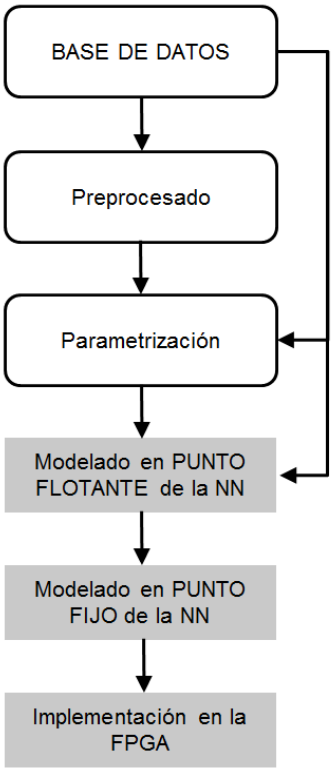


Figura 3.1. Esquema general de la metodología experimental.

La tesis se centra sobre los últimos tres bloques, que aparecen sombreados en la figura 3.1. Esto se debe a que estas etapas son las que tratan expresamente la implementación de las NN sobre una FPGA. Dicho de otra forma, una vez determinadas las entradas de la NN y la funcionalidad deseada, estas etapas son las que consiguen trasladar el diseño a un circuito digital. El preprocesado y la parametrización son opcionales, su uso depende del escenario y de las características de la base de datos usada. Como resumen, puede observarse la figura 3.2, donde confluyen los diferentes conceptos sobre la metodología propuesta.

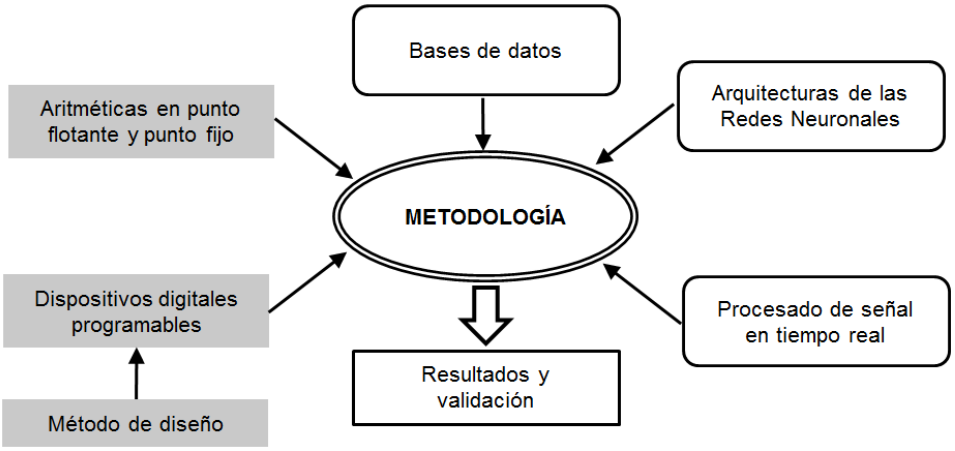


Figura 3.2. Confluencia de conceptos en la metodología experimental.

3.2 Las bases de datos

Para el modelado de una NN se precisa de una base de datos que cumpla una serie de condiciones; entre ellas cabe destacar que debe componerse de un conjunto representativo de elementos, debe estar convenientemente etiquetada y ser accesible mediante programas informáticos. Por otro lado, pueden ser gratuitas o libre difusión, o de pago. En esta tesis se usan bases de datos gratuitas, bien por ser de libre difusión, fruto de colaboraciones con otras instituciones o ser generadas de forma sintética.

En los apartados posteriores se describen las bases de datos usadas, una para cada escenario. Las hay naturales y sintéticas, en cualquier caso etiquetadas por el especialista correspondiente. Algunas proceden de señales biológicas; otra de parámetros físicos de la atmósfera terrestre; y una de ellas de señales usadas por el hombre. Los cuatro escenarios usados son:

- clasificación de la palmera pejibaye atendiendo a sus marcadores moleculares,

- clasificación de pulsos de electrocardiograma,
- predicción de temperatura,
- y ecualizador para señal binaria con ruido.

3.2.1 La palmera pejibaye

La palmera pejibaye (*Bactris gasipaes Kunth*) es una especia nativa de la América tropical. Sus frutos y brotes tiernos se usan como alimento humano y para cebar al ganado; además, el tronco de la palmera adulta se usa como madera. La base de datos usada procede del Banco de Germoplasma de la Universidad de Costa Rica, que está constituida por los marcadores moleculares de amplificación aleatoria del ácido desoxirribonucleico polimórfico (RAPD, *Random Amplification of Polymorphic Deoxyribonucleic acid*). Las muestras están clasificadas en seis razas primitivas: *utilitis*, *tuira*, *pará*, *yurimagua*, *putumayo* y *tembé*. Se desarrolló un sistema previo de reconocimiento automático para esta base de datos en [Travieso et al., 2008]. Los objetivos de este sistema fueron dos; por un lado, comprobar la viabilidad de la clasificación usando marcadores moleculares; y por otro, determinar el patrón con el mínimo número de atributos para poder clasificar las razas de pejibaye.

La estructura de la base de datos se detalla en la tabla 3.1. Esta consta de las seis razas primitivas, con 13 elementos de cada una de ellas; por tanto hay 78 elementos convenientemente etiquetados con un código. Cada elemento dispone de 10 parámetros, que son los marcadores empleados para la clasificación. Durante la fase de testeo de la NN se obtuvo un 100% de acierto para cada clase.

Tabla 3.1. Estructura de la base de datos para la palmera pejibaye.

Clase	Denominación	Código	Número de elemento
Clase 1	*Costa rica - utilitis*	c	01-13
Clase 2	*Tuira*	t	14-26
Clase 3	*Pará*	p	27-39
Clase 4	*Yurimagua*	y	40-52
Clase 5	*Putumayo*	u	53-65
Clase 6	*Bolivia – tembé*	b	66-78

3.2.2 Pulsos electrocardiográficos

Tras el proceso de búsqueda de una base de datos electrocardiográfica se optó por la MIT-BIH (*Massachusetts Institute of Technology-Beth Israel Hospital*) *Arrhythmia Database*. Esta base de datos de arritmias está compuesta por 48 registros de electrocardiograma (ECG, *Electrocardiogram*) de 30 minutos cada uno, y contiene una amplia representación de tipos de cardiopatías [MIT-BIH Arrhythmia Database, 2014]. Esta base de datos se encuentra dentro de un conjunto más amplio disponible en el MIT-BIH *Database Distribution* [MIT-BIH Database Distribution, 2014].

La MIT-BIH *Arrhythmia Database* dispone de 19 tipos de cardiopatías diferentes, aparte de los pulsos normales; además, incluye 15 tipos de ritmos cardiacos convenientemente etiquetados. Estas anotaciones permiten la correcta identificación de pulsos y ritmos cardiacos. Los registros de esta base de datos fueron obtenidos entre los años 1975 y 1979; corresponden a 47 personas, 25 varones entre 32 y 89 años, y 22 mujeres entre 23 y 89 años.

Los registros fueron tomados a 360 muestras por segundo. Se realizó una conversión analógica a digital con 11 bits, el rango de la señal estaba entre $\pm$5 mV. Los valores de las muestras, por tanto, se codificaron con los valores de 0 a 2047 inclusive, donde el valor 1024 corresponde a 0 voltios.

Se determinó la detección de los 7 tipos de pulsos más frecuentes [Chadnani, 2011], que se muestran en la figura 3.3. Los pulsos usados son: Latido Normal (N), Bloqueo de Rama Izquierda (L), Bloqueo de Rama Derecha (R), Latido Auricular Prematuro (A), Contracción Ventricular Prematuro (V), Fusión de Latido Ventricular y Normal (F) y Latido Lento (/).

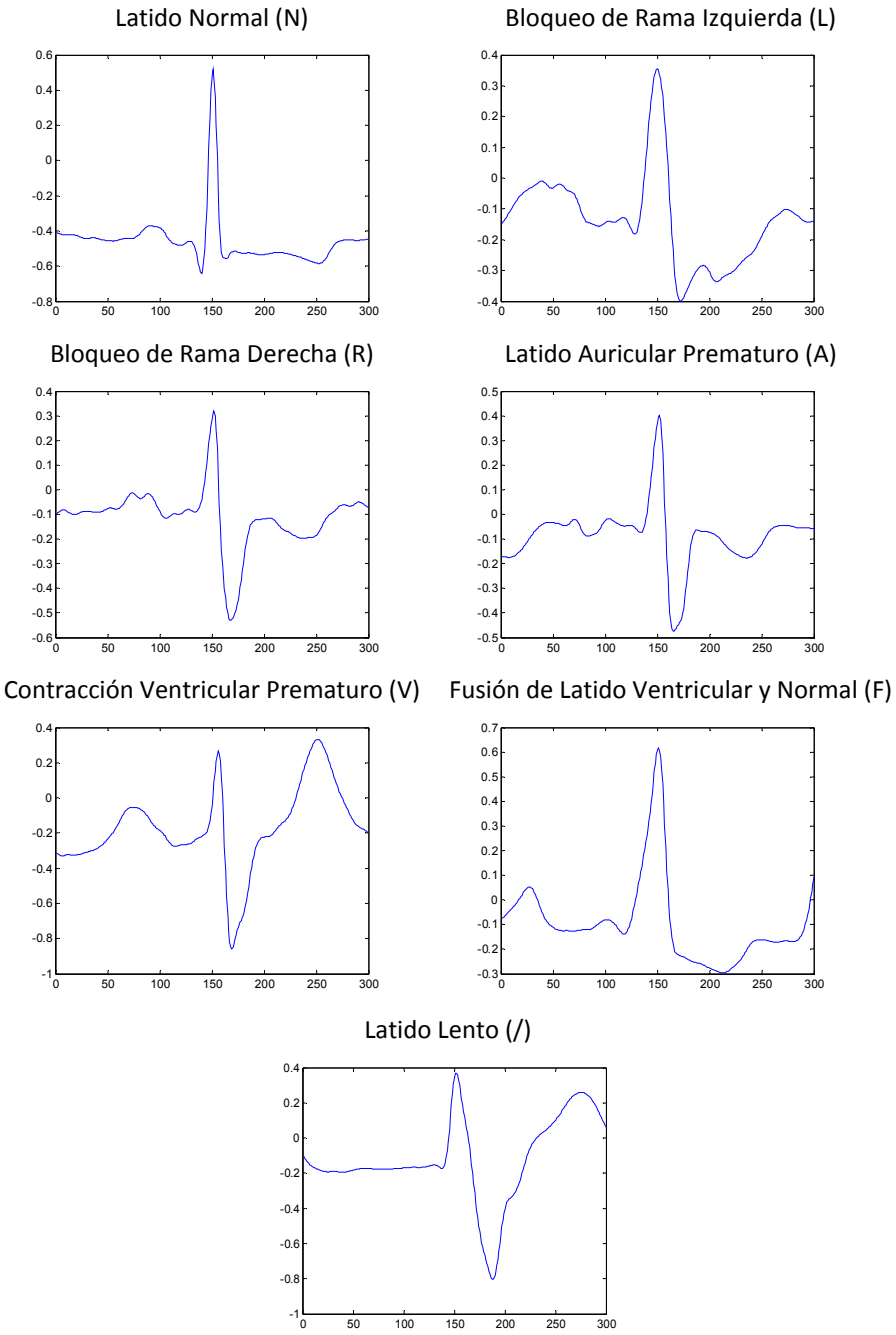


Figura 3.3. Tipos de pulso elegidos para la detección.

Inicialmente se realiza un preprocesado. Esta fase tiene como objetivo la reducción del ruido, de la interferencia de la red eléctrica y de las variaciones de la línea de base. Para ello se realiza un filtrado usando la transformada Wavelet discreta y un filtrado clásico. La transformada Wavelet es útil principalmente en la reducción del ruido y la interferencia de la red; por otro lado, el filtrado clásico es usado para eliminar las variaciones de la línea de base. La familia Wavelet utilizada en esta fase es la *daubechies3*, con un nivel 3.

De cada uno de los 7 tipos de pulso se tomaron 800 de la base de datos, teniendo en total 5.600 pulsos. Esto se realizó con herramientas de detección del complejo QRS, que permitieron localizar los pulsos cardiacos. Una vez localizados, se extrajeron y catalogaron, aprovechando el etiquetado de la base de datos.

Tras el preprocesado se tienen 5.600 pulsos caracterizados por 77 parámetros. Se dispuso la mitad de los pulsos para entrenamiento de la NN y la mitad para el testeo; es decir, 400 pulsos para cada fase.

Seguidamente se realiza un procesado con la intención de reducir el número de parámetros por pulso, ya que una NN de 77 entradas es relativamente grande. La reducción del número de parámetros se realizó con el análisis de componentes independientes (ICA*, Independent Components Analisys*). De esta forma se consiguió reducir el número de parámetros hasta 15. Finalmente se realizó la normalización de estos parámetros y las entradas quedaron restringidas al intervalo [-1,+1].

En el testeo en punto flotante se alcanzó una tasa de acierto del 97,21% con 30 neuronas en la capa oculta. O sea, la NN tiene 15 entradas, 30 neuronas en la capa oculta y 7 salidas; se podría denotar como 15-30-7. Las funciones usadas fueron tipo *tansig* en la capa intermedia y *purelin* en la salida. Conviene resaltar que se tienen 400 pulsos de cada tipo para entrenar y 400 para testear, con 15 parámetros para cada pulso. La matriz de confusión obtenida en este testeo es la mostrada en la tabla 3.2. De esta forma ya se tiene la regla de oro para este caso, los valores de los coeficientes de la NN obtenida en el entrenamiento se guardan para la implementación en la FPGA.

Tabla 3.2. Matriz de confusión obtenida en el testeo de punto flotante para ECG.

	N	**L**	**R**	**A**	**V**	**F**	**/**	**Éxito(%)**
N	400	0	0	0	0	0	0	100
L	0	400	0	0	0	0	0	100
R	1	0	394	3	2	0	0	98,50
A	4	0	6	375	3	12	0	93,75
V	0	1	0	10	380	7	2	95
F	0	0	0	9	15	374	2	93,50
/	0	0	0	0	0	1	399	99,75
Total								**97,21**

Conviene resaltar que el sistema usado es de tipo balanceado; o sea, tiene el mismo número de pulsos para cada caso. Esto quiere decir que en el entrenamiento se usaron 400 pulsos normales, y 400 para cada tipo de los seis patológicos. Estas cantidades se van a repetir en el testeo.

3.2.3 Temperatura

La tercera base de datos usada corresponde con la temperatura de una estación meteorológica. Esta se encuentra en la ciudad de Turrialba y pertenece a la Universidad de Costa Rica. Se dispuso de la temperatura desde mediados del año 2007 hasta mediados de 2010, tomada a intervalos de media hora. En ese intervalo de tiempo la temperatura fluctuó entre 11,9 y 31,8 ^{0}C, con un valor medio de 21,6 ^{0}C. En la figura 3.4 se observan las muestras de la primera semana de la base de datos; además se marca la temperatura media del periodo de tiempo de la base de datos.

El objetivo fue implementar un predictor de temperatura mediante una NN. Para entrenar la NN se usó el año 2008 y se testeó con el año 2009. Se consiguió predecir la señal con un valor medio del error en valor absoluto de 0,317 ^{0}C.

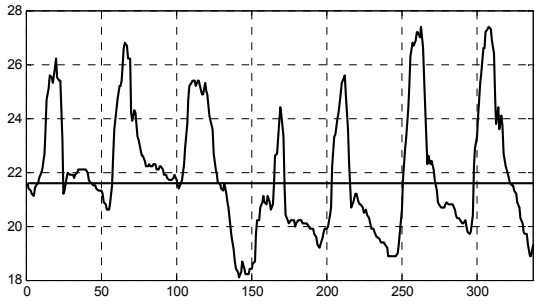


Figura 3.4. Temperatura de la primera semana de la base de datos y su valor medio.

3.2.4 Señal binaria con ruido

La última base de datos es sintética [Pérez et al., 2013]. Para ello se usó una señal binaria unipolar tipo NRZ, inicialmente de 1 kilobit por segundo. A esta señal se le sumó ruido blanco gaussiano aditivo. Para procesar la señal se consideraba muestreada con 10 kHz, lo que daba 10 muestras por bit. La relación señal a ruido

inicial era de +10 dB. La señal usada era de 2.000 bits generada de forma aleatoria. Finalmente, para comprobar las prestaciones de la NN, se testeó con una relación señal a ruido entre -5 y +20 dB. En la figura 3.5 se representan 20 bits de la base de datos citada, para una relación señal a ruido de +10 dB. En la señal superior se representa la señal binaria original, y en la inferior la señal afectada por el ruido.

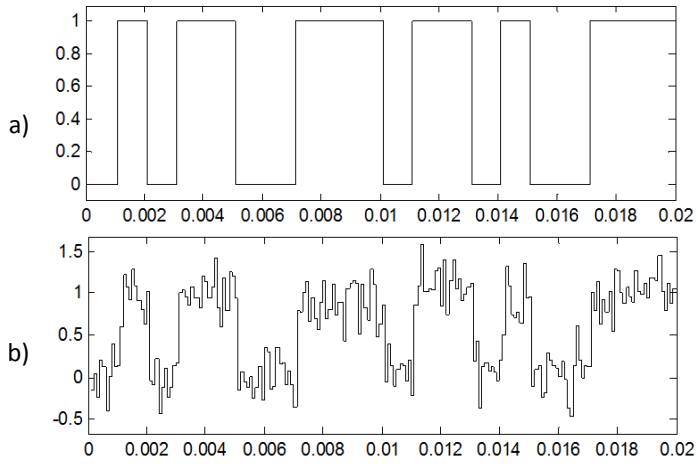


Figura 3.5. a) Señal binaria original, b) señal con una relación señal a ruido de +10 dB.

El objetivo en este escenario fue diseñar un ecualizador mediante una NN que disminuyera el ruido en la salida. Dado que el proceso de entrenamiento fue realizado en su totalidad por el autor de esta tesis, y dada su relativa complejidad, se describe en el apéndice de este documento.

3.2.5 Características de las bases de datos

En la tabla 3.3 se muestran las principales características de las bases de datos descritas anteriormente.

Tabla 3.3. Características de las bases de datos.

	Pejibaye	**ECG**	**Temperatura**	**Señal binaria con ruido**
Tipo numérico	Real	Real	Real	Real
Dimensionalidad	Multidimensional	Unidimensional	Unidimensional	Unidimensional
Origen	Natural	Natural	Natural	Sintética
Aleatoriedad	Aletoria	Pseudoperiódica	Pseudoperiódica	Aletoria
Nivel de ruido	Fijo	Variable	Fijo	Variable
Etiquetado	Especialista externo	Especialista externo	No etiquetada	Etiquetado propio
Coste	Gratuito	Gratuito	Gratuito	Gratuito

3.3 Preprocesado y parametrización

El preprocesado consiste en la reducción de ruido y distorsión de la señal, normalmente se realiza por filtrado. En resumen, se trata de extraer la componente de señal útil y rechazar las aportaciones indeseadas de que disponga la señal. La parte indeseada de señal puede generarse en el momento de su captación o generación, al almacenarse o realizar un tratamiento posterior. La etapa de preprocesado será de uso obligado u opcional, según el caso.

De la parametrización de la señal, una vez preprocesada la base de datos, se puede decir que es una etapa igualmente opcional. A veces las propias muestras de la base de datos son apropiadas para ser la entrada de la NN; en otras ocasiones, la señal no es apropiada y debe sufrir una transformación. Dicho de otra forma, es necesaria cuando los resultados iniciales son muy pobres o se consiguen sustanciales mejoras en la NN.

En general, la existencia de preprocesado y parametrización, dependerá del escenario, de la base de datos, y de la función que se espera que realice la NN. En la tabla 3.4 se indica la existencia de estos procesos en las cuatro bases de datos usadas.

Tabla 3.4. Uso del preprocesado y parametrización en los distintos escenarios.

	Pejibaye	**ECG**	**Temperatura**	**Señal binaria con ruido**
Preprocesado	No	Sí	No	No
Parametrización	No	Sí	No	No

3.4 Modelado en punto flotante

Debe destacarse, que llegados a este punto, ya no importa la naturaleza de la base de datos, ni el preprocesado y parametrización previas; dicho de otra forma, en adelante se tratarán números y el diseñador puede abstraerse de su significado físico. Una vez determinada la señal de entrada en la NN y la funcionalidad deseada, el siguiente paso es determinar la arquitectura de la NN. Para esto hay que fijar una serie de parámetros de la arquitectura, que se describen a continuación.

- ***Número de entradas***. Vendrá dado por el número de parámetros si se trata de una clasificación, o del número de muestras de entrada si opera como un predictor.

- ***Número de salidas***. Viene especificado por la naturaleza del problema.
- ***Número de capas***. Al menos es necesaria una sola capa intermedia, también llamada oculta. Se puede comprobar las prestaciones de la NN en función del número de capas. En la mayoría de las aplicaciones es suficiente con una sola capa intermedia; es decir, una configuración de tres capas.
- ***Número de neuronas en cada capa***. En la capa de salida el número de neuronas es igual al número de salidas. El número de entradas y salidas viene dada por el problema a resolver. Donde es posible variar el número de neuronas es en la capa (o capas) intermedias, lo que puede afectar a las prestaciones de la NN.
- ***Función de transferencia usada***. Normalmente se usa una de las indicadas en la tabla 2.1. Puede usarse el mismo tipo de función en todas las neuronas; es decir, para todas las capas; aunque a veces, la naturaleza del problema sugiere formas distintas para la capa de salida. Lo normal es usar el mismo tipo de función para todas las neuronas de una misma capa.

El diseñador debe probar diferentes configuraciones. Mediante el proceso de entrenamiento y testeo se determina el valor de los coeficientes de la NN. En esta tesis se usa el *Neural Network Toolbox* [Neural Network Toolbox, 2014] y el *Neural Network Time Series Tool* [ntstool, 2015], ambos de Matlab. El motivo es que Matlab se ha convertido en un estándar, y todo el diseño de la NN para la FPGA opera sobre Matlab, en gran parte sobre su entorno de diseño gráfico *Simulink*. La etapa de modelado de punto flotante, de forma detallada, se muestra en la figura 3.6.

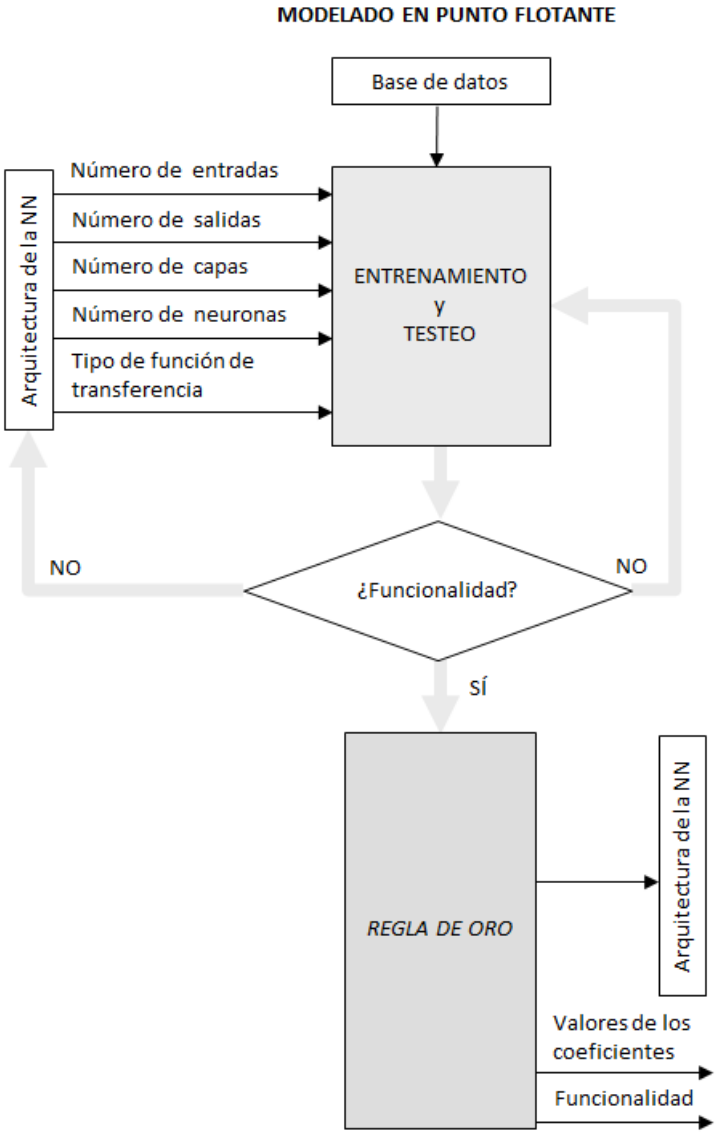


Figura 3.6. Modelado detallado de la NN en punto flotante.

En la figura 3.6 el diseñador fija la arquitectura deseada, o la que espera que cumpla la funcionalidad. Con el proceso de entrenamiento y testeo se comprueba si se alcanza la funcionalidad. Si no se alcanzase se puede repetir el entrenamiento cambiando alguno de sus parámetros, o bien se puede cambiar la arquitectura de la NN. Cuando se alcanza la funcionalidad deseada se dice que se tiene la regla de oro del diseño. La regla de oro, como muestra la figura 3.6, es la arquitectura definida para la NN, los coeficientes obtenidos y la funcionalidad alcanzada.

Algunos de los parámetros que se pueden cambiar en el entrenamiento son: el propio algoritmo de entrenamiento, la función de error usada, el número de iteraciones, y la parte de la base de datos usada. Debe destacarse que en la regla de

oro se tienen los valores de los coeficientes en formato de punto flotante. La funcionalidad puede medirse de diferentes formas y dependerá del escenario; por ejemplo puede ser: tasa de acierto en la clasificación, error en una predicción o relación señal a ruido en una ecualización. En la tabla 3.5 se indica la forma de operar la NN y cómo se evalúa la funcionalidad en los cuatro escenarios de estudio.

Tabla 3.5. Forma de operar la NN y de evaluar la funcionalidad.

	Pejibaye	**ECG**	**Temperatura**	**Señal binaria con ruido**
Operación de la NN	Clasificador	Clasificador	Predictor	Predictor
Funcionalidad	Tasa de acierto	Tasa de acierto	Valor medio del error absoluto	Relación señal a ruido

3.5 Modelado en punto fijo

Como ya se indicó, al final del modelado en punto flotante se tiene la regla de oro. O sea, la propia NN con su arquitectura y coeficientes, que alcanza una cierta funcionalidad. Esta NN puede ponerse a funcionar en cualquier entorno de punto flotante, normalmente sobre sistemas con microprocesadores. De forma inmediata puede funcionar usando la propia *Neural Network Toolbox* de Matlab en ese formato. Como se dijo anteriormente, es objetivo pasar el cálculo a aritmética de punto fijo para que sea soportable sobre dispositivos digitales programables. El proceso de modelado en punto fijo se muestra en detalle en la figura 3.7.

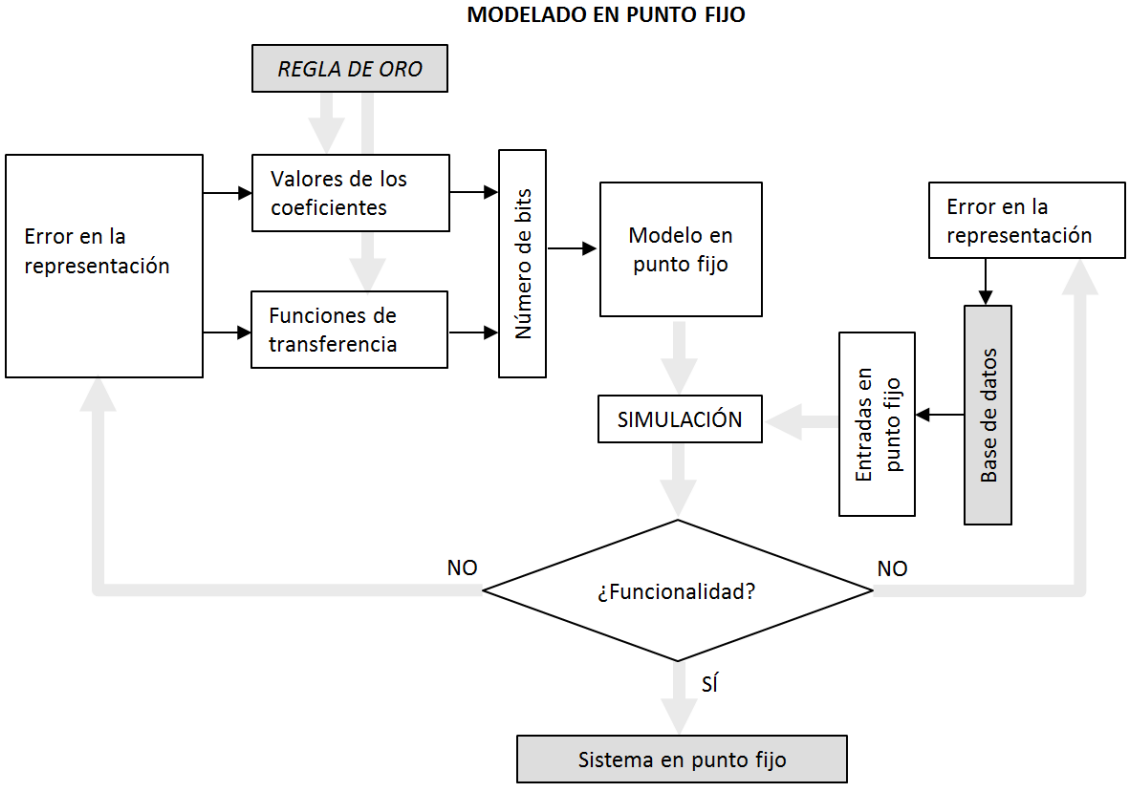


Figura 3.7. Modelado detallado de la NN en punto fijo.

En el modelo de punto fijo la clave es determinar el número de bits idóneo en los diferentes puntos del sistema, esto hace que se alcance la funcionalidad, a la vez que se minimiza el área y la potencia consumida, y se maximiza la velocidad. Un excesivo número de bits no mejora la funcionalidad a la vez que empeora el área, la potencia y la velocidad de la NN. Si se disminuye el número de bits por debajo de ciertos valores, el error de cuantificación hace que la NN pierda la funcionalidad, y las salidas no tomen los valores esperados.

Inicialmente se fija un error pequeño en la representación de los valores en punto fijo, lo que da un número de bits relativamente grande. La simulación del modelo en punto fijo, con la base de datos usada, hace posible comprobar la funcionalidad. También, se representan las entradas en punto fijo según el error permitido en la representación. Si no se alcanzase la funcionalidad, claramente debe disminuirse el error y aumentar el número de bits del sistema. Normalmente en un primer intento, con un error suficientemente pequeño, se alcanza la funcionalidad de forma inmediata, a consta de gastar mucha área, potencia y perjudicar el retardo.

Lo que debe hacerse entonces es ir aumentando el error de la representación, lo que disminuye el número de bits y mejora las prestaciones físicas del sistema. Llegará el momento en el que para un determinado error se perderá la funcionalidad.

En resumen, debe buscarse el máximo error permitido en la representación que mantiene la funcionalidad, este error minimiza el número de bits y mejora las prestaciones físicas de área, potencia y velocidad. El cálculo del número de bits en función del error se realiza con un programa, en esta tesis es un fichero tipo “m” de Matlab. Ese mismo error también se usa para la representación de las funciones de transferencia.

3.6 Flujo de diseño para la FPGA

De la etapa anterior se tiene el modelo en punto fijo con la funcionalidad comprobada. En esta última fase debe trasladarse el modelo en punto fijo a un dispositivo digital programable, en particular una FPGA. Esto puede hacerse con algún método de diseño de los descritos en el capítulo 2, atendiendo siempre a las características deseadas para el método de diseño elegido. El flujo de diseño para la FPGA se detalla en la figura 3.8.

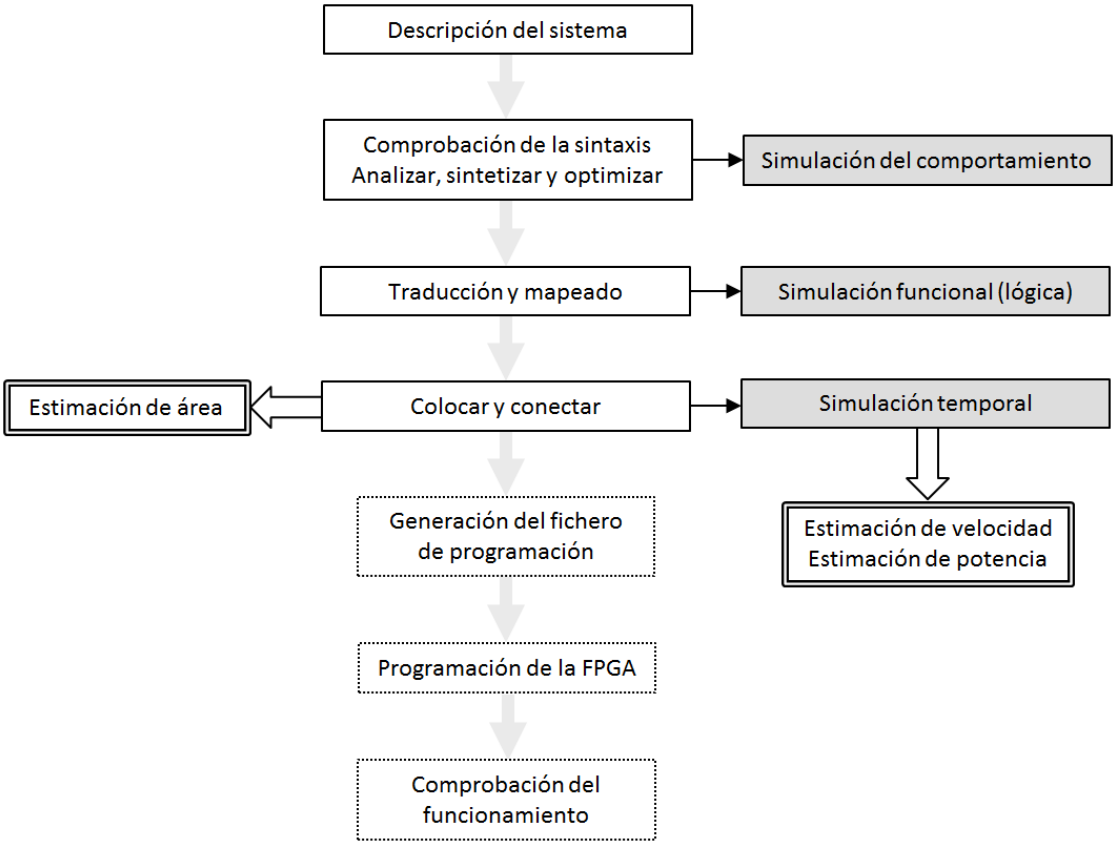


Figura 3.8. Flujo de diseño para la FPGA.

El flujo de la figura 3.8 tiene carácter general; los diferentes desarrolladores de herramientas, o suministradores de FPGA, usan nombres distintos para estas etapas. Además, en un diagrama más detallado, se pueden agrupar o generar subprocesos. A continuación se detallan las etapas del flujo de diseño mostrado.

- ***Descripción del sistema***. En esta etapa se tiene el diseño descrito con alguno de los métodos vistos en el capítulo 2. Es muy común, por ejemplo, describir el sistema usando un lenguaje de descripción hardware.
- ***Comprobación de la sintaxis. Analizar, sintetizar y optimizar***. La "comprobación de la sintaxis" realiza una comprobación de errores a priori de la descripción; por ejemplo: pines no asignados, errores en el lenguaje, errores de esquemático, etc. Después de "analizar, sintetizar y optimizar" se tienen las ecuaciones de la lógica digital en un álgebra de Boole simplificado, se puede decir que es una descripción genérica de la lógica digital. Por este motivo en esta fase es posible una simulación del comportamiento del circuito. Esta simulación, y las posteriores, son muy costosas computacionalmente; esto se

debe a que se simula el circuito con cierto nivel de detalle, y no solo la aritmética en punto fijo como en el apartado anterior. Este puede ser un motivo, por el que normalmente, en esta fase no es posible la comprobación de la total funcionalidad del sistema. Dicho de otra forma, no es posible simular la NN para toda la base de datos, por dos razones: la complejidad del sistema y la gran cantidad de datos de entrada.

- ***Traducción y mapeado***. Con esta etapa se consigue el circuito digital para una FPGA determinada; es decir, se usan las puertas lógicas y bloques disponibles en esa FPGA. Es decir, se tiene la descripción que usa los tipos de puertas, registros, etc., disponibles en la FPGA elegida. En este punto es posible una simulación funcional, también llamada lógica, con las mismas restricciones que la simulación de comportamiento de la etapa anterior.
- ***Colocar y conectar***. En "traducción y mapeado" se usó el tipo de recursos lógicos disponibles, pero sin seleccionar recursos particulares de la FPGA. En la etapa de "colocar y conectar" se traslada el diseño a los circuitos de la FPGA y se procede al conexionado final entre ellos, y con los pines de entrada-salida. Por este motivo ahora es posible una estimación precisa del área requerida, que en las fases anteriores solo era aproximada. Además, ahora es posible realizar simulaciones temporales, que sirven para comprobar la velocidad máxima del sistema; bien estimando la frecuencia máxima, bien el máximo retardo combinacional entre entradas y salidas. En las simulaciones de las fases anteriores la estimación de velocidad era nula o muy deficiente. Hecha la estimación de velocidad es posible disponer de la estimación de potencia consumida. Esto se debe en primer lugar a que se tiene el circuito final, con toda la lógica y el cableado interno; y en segundo lugar, a que se tiene la velocidad máxima. La potencia estática depende, entre otras cosas, de la temperatura; y la potencia dinámica de la frecuencia. En resumen, en esta fase ya se tienen las tres prestaciones físicas del sistema: área, velocidad y potencia.
- ***Generación del fichero de programación***. Una vez que se ha comprobado las prestaciones físicas del diseño es posible generar el fichero para programar la FPGA elegida. A este fichero normalmente se le llama *bitstream* o fichero de

programación. Este fichero es el que describe el conexionado interno en la FPGA, para que realice la función del diseño.

- ***Programación de la FPGA***. La configuración o programación de la FPGA se hace conectado el ordenador a la FPGA, y ejecutando el proceso de programación. La FPGA está en una placa de circuito impreso donde existe un conector para la programación, este está conectado a los pines específicos de programación. Actualmente es habitual el uso del Bus Universal Serie (USB, *Universal Serial Bus*).
- ***Comprobación del funcionamiento***. La FPGA está en condiciones de realizar su operación. Antes de ponerla a funcionar es recomendable comprobar su funcionamiento en un laboratorio mediante el uso de la correcta alimentación e instrumentos electrónicos apropiados: generadores de señal, osciloscopios, analizadores lógicos, analizadores de espectro, etc.

Debe ponerse de relieve que a las entradas en las simulaciones en el flujo de diseño de la FPGA se les llama *testbench* (banco de pruebas). Para el de la NN, el *testbench* debe generarse con los datos obtenidos de la base de datos, o después del preprocesado y parametrización si se hicieron. Por otro lado, durante todo el flujo, las herramientas generan ficheros de tipo *report*, donde se informa de las características y estado del diseño.

3.7 Herramienta y flujo de diseño

Al llegar a este punto, se han especificados los escenarios sobre los que se va a probar la metodología y sus bases de datos asociadas. Igualmente, se ha planteado la metodología de forma global, en lo que respeta a la aritmética de punto flotante y punto fijo. Por último, se ha fijado el uso de las FPGA como tecnología. Queda por determinar los programas de diseño y su flujo de diseño; es decir, el método de diseño sobre la FPGA.

En general, el método de diseño elegido, debe garantizar algunos de los parámetros descritos en el capítulo segundo. A continuación se vuelven a enumerar, comentándolos convenientemente.

El coste económico. Dado que la tesis se centra en un entorno académico las herramientas deben ser gratuitas o poder conseguirse mediante donación. Esto se justifica por la escasez de fondos y abrir la posibilidad a que cualquier miembro de la institución, sobre todo alumnos, lo puedan usar de forma gratuita.

El periodo de aprendizaje del diseñador. Si se consiguen ventajas sustanciales por usar un método de diseño, se justifica un periodo de aprendizaje por parte del diseñador. Dicho de otra forma, el tiempo invertido en el aprendizaje se ahorra posteriormente en la fase de diseño; más aún, pueden conseguirse grandes mejoras. Estas mejoras pueden consitir en la comparación de diferentes arquitecturas, la comprobación total de la funcionalidad, el tiempo total del diseño, etc.

El soporte de las herramientas. Los programas usados, dado que pueden ser novedosos, deben estar documentados con ficheros de referencia rápida, manual de usuario y tutoriales de aprendizaje.

La actualización del sistema. El entorno de diseño debe tener constante actualización.

Los sistemas operativos sobre los que funciona. Los programas deben funcionar sobre Windows o Linux, al ser los más habituales.

La ayuda ante errores. Cuando aparezcan errores debe haber mecanismos de consulta a los desarrolladores de las herramientas.

La portabilidad entre fabricantes y dispositivos. Es deseable, aunque no imprescindible, el poder cambiar de suministrador de FPGA durante el diseño. Lo que sí será posible es el cambio de dispositivos dentro del mismo suministrador, siempre que la FPGA se ajuste a las restricciones del diseño.

La flexibilidad del diseño. Este parámetro es especialmente interesante y deseable. Implica poder rediseñarlo fácil y rápidamente, si se cambia alguno de los parámetros del diseño. En este sentido, se permitiría la comprobación de diferentes arquitecturas.

El tiempo de diseño y compilación. Deben ser de duración razonable, y que no aumenten excesivamente el tiempo de diseño. Obviamente, debe obtenerse el área ocupada finalmente.

El tiempo de simulación. Este tiempo debe ser igualmente razonable. Las simulaciones deben permitir comprobar la total funcionalidad del sistema, este punto es especialmente importante. De forma especial, debe generar automáticamente los *testbench* necesarios para ser usados como señales de entrada. Por otro lado, debe extraerse la velocidad del diseño y la potencia consumida.

El tipo de reconfiguración del sistema. Este punto no es crítico en esta tesis, no importa si la FPGA se reconfigura total o parcialmente, dado que no es un objetivo reprogramarla durante su funcionamiento.

La seguridad y privacidad del diseño. Este punto no es crítico, aunque las herramientas y los dispositivos disponen de mecanismos para evitar la copia de los diseños.

Interactuación con otras herramientas. Es deseable un entorno de diseño compacto, que interactúe con fidelidad con simuladores y los entornos de punto flotante donde se tiene la regla de oro.

Tras un periodo de búsqueda y estudio, de diferentes métodos de diseño, se optó por los entornos que los principales suministradores tienen disponibles sobre *Simulink* de Matlab. Estos programas cumplen con los requisitos enunciados anteriormente y sus características serán expuestas en apartados posteriores. Ambos entornos, de Altera y Xilinx, son muy similares e igualmente recomendables; pero se optó finalmente por usar el de Xilinx. Esto solo se debió a una serie de errores que aparecieron en la fase de estudio de las herramientas de Altera; y a que la documentación y tutoriales de Xilinx resultó ser más clara. De todas formas, debe resaltarse, que ya ambos fabricantes disponen de la documentación convenientemente estructurada en su página web.

Ambos entornos aprovechan las características de *Simulink*, que deben resaltarse. Por un lado, está totalmente integrado en Matlab; esto hace que acceda a su espacio de variables, tanto para la toma de señales de entrada, como para analizar y estudiar las salidas. En *Simulink* se diseña usando diagrama de bloques de forma gráfica, cada bloque se configura mediante su ventana de diálogo. En *Simulink* existen multitud de conjuntos de bloques, llamados *Blocksets*; entre las que cabe destacar, las fuentes de

señal (*Sources*), las utilidades para interconectar (*Signal Routing*) y los elementos para evaluar las salidas (*Sinks*).

Obviamente, para usar este tipo de herramientas, se precisa conocer el uso de Matlab y *Simulink*. Por otro lado, es aconsejable pero no obligatorio, tener conocimientos de VHDL y Verilog. Esto se debe a que desde *Simulink* se generará el diseño completo descrito en uno de esos HDL, y normalmente no es necesaria su modificación.

Conviene resaltar que la búsqueda, puesta a punto, y uso de estas herramientas puede llevar meses si se parte de un total desconocimiento. Pero, si existe asesoramiento previo suficiente, el tiempo de estudio y uso puede reducirse a semanas; en cualquier caso, depende de la experiencia y habilidad del individuo. Es decir, con suficiente asesoramiento, este periodo de tiempo puede aproximarse al del uso de un HDL estándar. Si se involuciona; por ejemplo, se diseñase usando esquemáticos, el inicio es casi instantáneo; pero como se indicó en temas anteriores, se dificulta la edición, modificación y portabilidad de los diseños.

3.8 El entorno de diseño de Xilinx

Ya se introdujo el entorno de Xilinx en el segundo capítulo. Como se observa en la figura 3.9, está conformado por dos grandes herramientas. Por un lado *System Generator*, que opera sobre *Simulink* de Matlab; y por otro, la herramienta estándar ISE (*Integrated System Environment*). Como alternativa a ISE, a partir del año 2012 aparece la nueva versión llamada Vivado, a la que se pueden migrar los diseños.

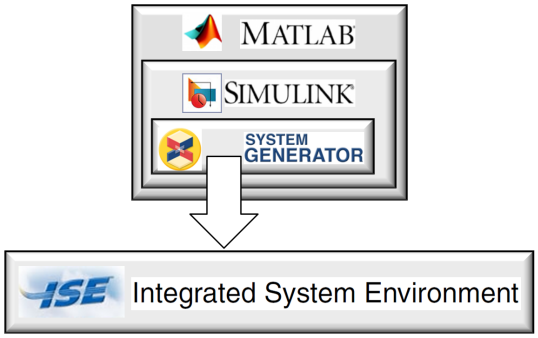


Figura 3.9. El escenario de diseño de Xilinx para System Generator.

3.8.1 System Generator

Como se acaba de indicar, el entorno de diseño de Xilinx sobre *Simulink* se llama *System Generator*. Este software requiere de la correcta versión de Matlab, y por tanto de *Simulink*; a la vez que de la correcta versión de ISE, o Vivado si fuera el caso. Cuando se instala *System Generator* en *Simulink* aparecen los *Blocksets* propios de Xilinx. A partir de ese momento se puede diseñar un sistema usando diagrama de bloques, que se configuran mediante sus ventanas de diálogo. El diseño, normalmente en punto fijo, queda especificado entre los buses de entrada y salida. Fuera de los límites del diseño pueden colocarse el resto de bloques de *Simulink*, normalmente sistemas de punto flotante. En el interior del diseño pueden usarse bloques de *Signal Routing* de *Simulink*, dado que son bloques de conexionado.

Los bloques de *System Generator* tienen acceso al espacio de variables de Matlab, lo que es una gran ventaja. Por otro lado, el sistema se diseña con las misma filosofía que en el propio *Simulink*; es decir, diagramas de bloque configurables mediante sus ventanas de diálogo. Esto hace que el diseño sea rápido. El cambio de los parámetros en las ventanas de diálogo; por ejemplo el número de bits, permite un diseño flexible, porque se pueden probar diferentes arquitecturas de forma rápida.

El acceso al espacio de variables de Matlab hace posible tomar las entradas de la base de datos; o después de la parametrización, si fuera el caso. Este mismo acceso hace posible guardar las salidas de la simulación, para ser analizadas o representadas. Las simulaciones de los diseños de *System Generator* son muy rápidas, porque se usa un bajo nivel de detalle de los circuitos. Todo lo anterior hace posible comprobar la total funcionalidad del sistema; es decir, simular el sistema para todas las entradas posibles. En este caso, la estimación de área es aproximada; y no se tiene estimación de velocidad ni de potencia.

Una vez comprobada la funcionalidad del sistema, se procede a su compilación; donde se obtiene la descripción estructural en un lenguaje HDL estándar; Verilog o VHDL según se elija. En esta compilación, entre otras cosas, puede generarse el *testbench*; que es la señal de entrada que será usada en las simulaciones de la herramienta posterior. La generación automática del *testbench* es importante; porque generarla de forma manual es muy costoso. Esto puede deberse a la cantidad de datos

que se quiere simular; y también, al posible cambio de formato de la señal de entrada. La interacción descrita entre *System Generator*, *Simulink*, Matlab y el ISE de Xilinx queda reflejada en la figura 3.10.

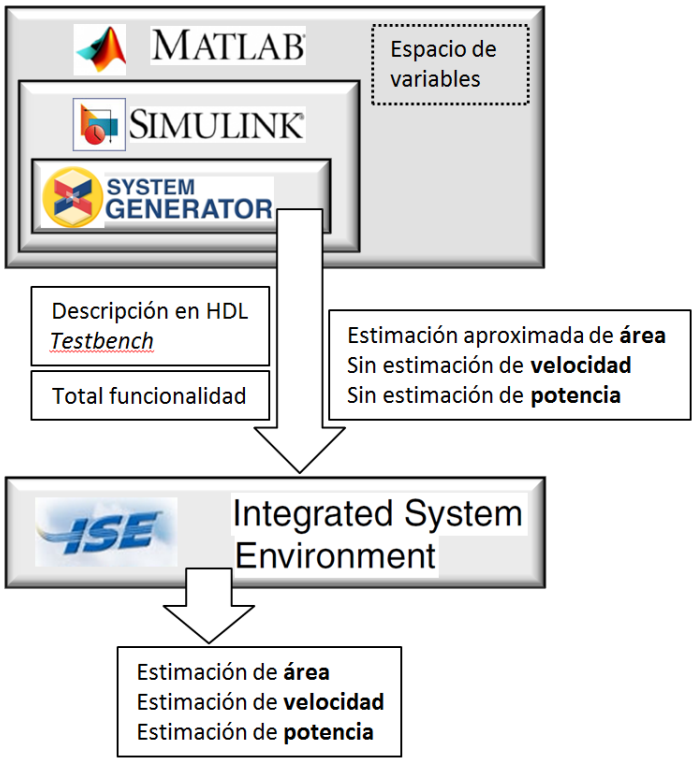


Figura 3.10. Interacción entre System Generator, Simulink, Matlab y el ISE de Xilinx.

Llegados a este punto, conviene resaltar, que con las herramientas elegidas, el preprocesado y parametrización se realiza con Matlab en formato de punto flotante. El modelado de la NN en punto flotante, igualmente se realiza con Matlab. El paso del modelo de punto flotante a punto fijo se hace mediante *System Generator*. El flujo final del diseño para la FPGA, que se corresponde con el apartado 3.6 y figura 3.8, se realiza con el *Integrated System Environment* de Xilinx.

3.8.2 Integrated System Environment

Como se acaba de exponer, el flujo final del diseño para la FPGA, se realiza con el *Integrated System Environment* de Xilinx, como muestra la figura 3.10. Su diagrama de flujo se corresponde con la figura 3.8. Como ya se ha indicado, las simulaciones son

ahora muy lentas por el alto nivel de detalle de los circuitos usados; esto permite una precisa estimación de área, velocidad y potencia. Normalmente, con ISE, no es posible comprobar la total funcionalidad; lo que no tiene importancia, porque con las herramientas propuestas puede comprobarse con *System Generator* con anterioridad.

Conviene destacar que del paquete que incluye *System Generator* e ISE se usaron las versiones 10.1 y 13.1, con los sistemas operativos y versiones de Matlab que se muestran a continuación.

- *System Generator for DSP and AccelDSP 10.1*
    - *Integrated System Environment 10.1*
    - Windows XP (32 bits)
    - Matlab R2007a
- *System Generator for DSP 13.1*
    - *Integrated System Environment 13.1*
    - Windows XP (32 bits) y Windows 7 (64 bits)
    - Matlab R2010a y R2010b

3.9 El entorno de diseño de Altera

De Altera, segundo suministrador de FGPA a nivel mundial, conviene destacar que dispone de *DSP Builder*, que es la herramienta sobre *Simulink* para el diseño de FPGA. De ella se puede decir que tiene las mismas características que su equivalente de Xilinx, su escenario se muestra en la figura 3.11. También dispone de *Quartus II*, que es el entorno estándar, igualmente equivalente a su homólogo de Xilinx.

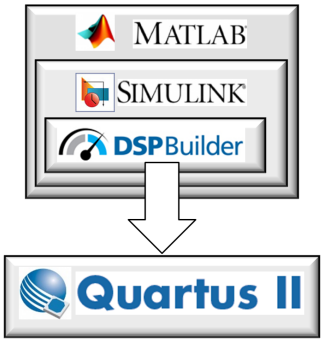


Figura 3.11. El escenario de diseño de Altera para DSP Builder.

CAPÍTULO 4

EXPERIMENTOS Y RESULTADOS

"Esto no es el final, ni siquiera es el principio del final, pero quizás sea el final del principio".

Winston Churchill

Este capítulo es, con pocas dudas, el núcleo de esta tesis. Después de todas las descripciones previas, hasta llegar a elegir el entorno de diseño para las FPGA, se procede a realizar los diseños de las NN para los diferentes escenarios; esto se hace con el fin de demostrar la hipótesis de esta tesis doctoral.

4.1 Introducción

Para cada base de datos se realiza la elección de la arquitectura de la NN y se entrena en punto flotante usando Matlab. Con un entrenamiento apropiado se obtiene la regla de oro del diseño; es decir, la arquitectura final de la NN, los coeficientes y los tipos de función de transferencia usadas. Durante el entrenamiento y el testeo debe comprobarse la funcionalidad requerida de la NN. Con la regla de oro obtenida se diseña el sistema con *System Generator* en aritmética de punto fijo, donde se comprueba la funcionalidad mediante simulaciones. Finalmente, con el *Integrated System Environment* se finaliza el flujo de diseño para la FPGA elegida y se extraen las prestaciones físicas.

4.2 La clasificación de la palmera pejibaye

Como se indicó en el capítulo anterior, donde se describió esta base de datos, finalmente es posible la clasificación con solo diez parámetros en la entrada. Se tomaron los diez marcadores moleculares que permiten la clasificación, sin realizar ningún preprocesado. Entonces, la NN elegida, debe tener diez entradas y seis salidas, pues son seis las razas de pejibaye a clasificar.

4.2.1 Modelado en punto flotante

Como se ha indicado repetidamente, el entrenamiento se realizó con el *Neural Network Toolbox* de Matlab. Igualmente, como se explicó previamente, se usará una NN artificial tipo perceptrón multicapa. Debe recordarse que de cada raza existen 13 individuos en la base de datos, de cada clase se usaron para el entrenamiento 4 individuos (30% aproximadamente), y los 9 restantes para el testeo (70% aproximadamente). Para el entrenamiento se escogieron las muestras de forma aleatoria. Las clases están etiquetadas, luego el entrenamiento es supervisado. En la fase de entrenamiento se varió el número de capas intermedia, y el número de neuronas en esas capas. Finalmente se determinó que se consigue una clasificación del 100% con una sola capa intermedia de 8 neuronas. Podría decirse entonces, que la NN

es del tipo 10-8-6; es decir, con 10 entradas, 8 neuronas en la capa oculta y 6 neuronas en la capa de salida.

Como funciones de transferencia se usaron las tipo *tansig* y *logsig*; para todas las neuronas de la arquitectura, alcanzado en ambos casos el 100% de aciertos. Finalmente, se optó por la función tipo *logsig* por ser una función unipolar positiva, y esto ahorra el bit de signo. En la figura 4.1 se observa la ventana obtenida al finalizar el entrenamiento de la NN con Matlab. Cabe destacar el uso del algoritmo de entrenamiento llamado *traingdx*, que es un algoritmo de entrenamiento adaptativo con retroalimentación del error, que minimiza dicho error con respecto a los coeficientes de la NN usando la minimización del momento descendiente del gradiente. Por otro lado, como función de error, se usó el tipo *sse* (*Sum squared error*), que evalúa la suma de los errores al cuadrado.

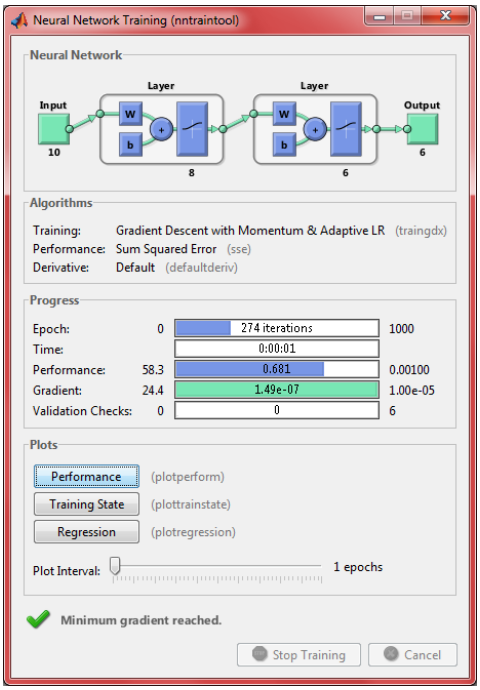
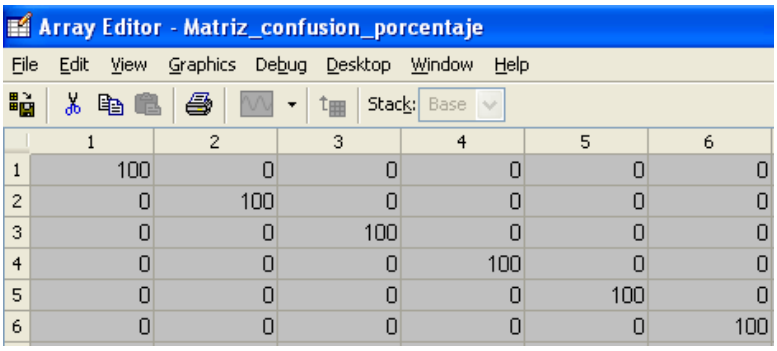


Figura 4.1. Ventana obtenida al finalizar el entrenamiento de la NN para pejibaye.

Al finalizar el entrenamiento se testeó el comportamiento de la NN tomando como entradas la parte de la base de datos no usada en el entrenamiento; se consiguió una clasificación del 100%, como indica la matriz de confusión de la figura 4.2.



	1	2	3	4	5	6
1	100	0	0	0	0	0
2	0	100	0	0	0	0
3	0	0	100	0	0	0
4	0	0	0	100	0	0
5	0	0	0	0	100	0
6	0	0	0	0	0	100

Figura 4.2. Matriz de confusión obtenida en la etapa de testeo de la NN para pejibaye.

Tal y como se ha repetido, después del testeo, se tiene la regla de oro. Que consiste en los ítems que se describen a continuación.

- ***La arquitectura de la NN***. Es decir, el número y formato de entradas y salidas, el número de capas, la cantidad de neuronas en cada capa y el tipo de función de transferencia usada en cada neurona.
- ***El valor de los coeficientes***. De cada neurona se tienen los pesos y el valor del desplazamiento de polarización. Estos valores están ordenados convenientemente en matrices para cada capa.
- ***La funcionalidad***. En este caso se consigue el 100% de acierto para la base de datos usada.

4.2.2 Diseño en punto fijo

Como se ha indicado el diseño se realizará en una primera etapa con *System Generator*, y posteriormente usando el *Integrated System Environment*. Esas dos fases se describen en los dos apartados que se exponen a continuación.

4.2.2.1 Diseño con System Generator

Una vez obtenida la regla de oro en formato de punto flotante, el siguiente paso es diseñar el sistema en aritmética de punto fijo, para ello se usará *System Generator* de Xilinx sobre *Simulink* de Matlab. La etapa de entrada de la NN se muestra en la figura 4.3.

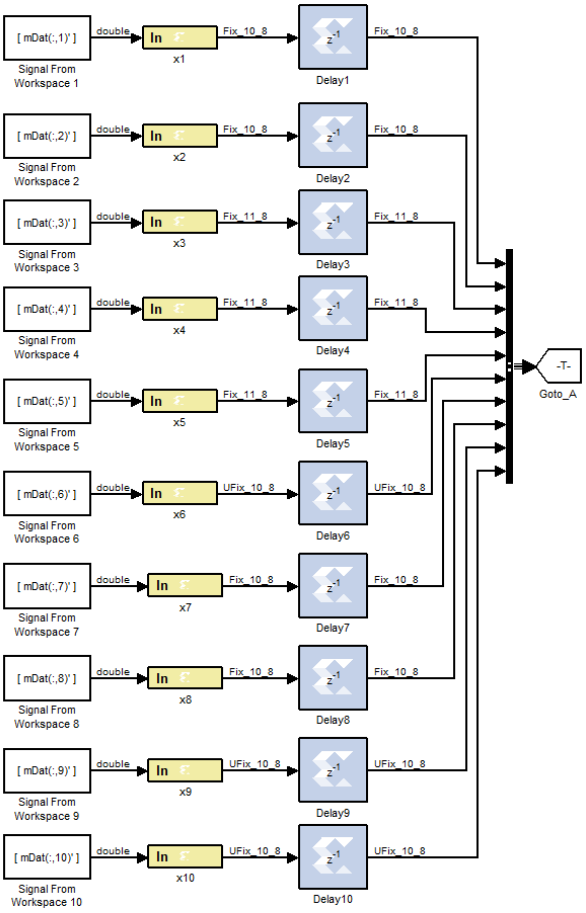


Figura 4.3. Etapa de entrada de la NN para pejibaye.

En la parte izquierda de la figura 4.3 se tienen las señales de entrada en formato de punto flotante (tipo *double*), que procede del espacio de trabajo de Matlab. Se dispone de 10 entradas, por ser 10 los parámetros para la clasificación. A continuación se observan los puertos de entrada a la FPGA (*Gateway In*), donde se realiza una conversión a punto fijo. Conviene resaltar que es *Simulink* y Matlab los que hacen la conversión a punto fijo de forma numérica; estos puertos de entrada no son convertidores analógico-digitales, sino meramente buses de entrada. Estos bloques pueden ser manipulados accediendo a su ventana de configuración, el de la primera entrada se muestra en la figura 4.4. En esta ventana, lo primero que se configura es el tipo de dato: booleano, o complemento a dos con signo o sin signo. También se configura el número de bits total usado en la representación, y el número de bits de la parte fraccionaria. Además, puede fijarse el tipo de cuantificación y desbordamiento. Finalmente, se fija un periodo de reloj, para marcar la velocidad con la que son tomadas las muestras de la entrada; este valor en este caso es arbitrario, y se fijó para una frecuencia de 5 MHz.

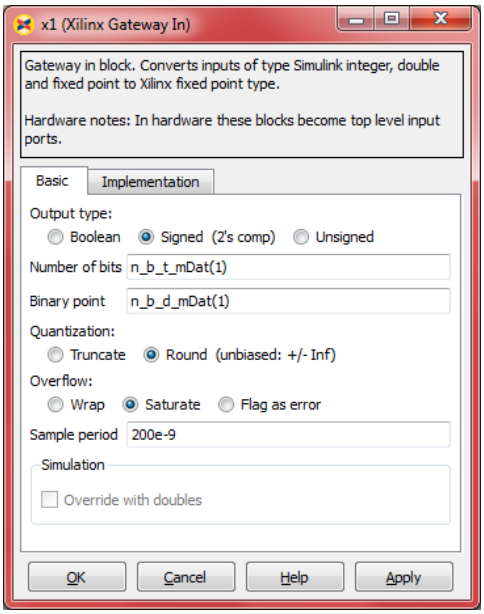


Figura 4.4. Ventana de configuración de un Gateway In.

Lo primero que hay que hacer es estimar el formato de representación necesario para cada entrada. Esto se hace según lo explicado en el apartado 3.5 del capítulo anterior, y mostrado en la figura 3.7. Inicialmente se fijó un error máximo del 1%. Con este objetivo, se desarrolló un programa en Matlab. Este fue el encargado de determinar el número de bits para la representación; que consta de bit de signo, si fuera necesario; y del número de bits de la parte entera y fraccionaria. A la derecha de los puertos de entrada se tiene el tipo de formato obtenido. Si es sin signo se etiqueta como *UFix*, de *unsigned fixed*; si es con signo se etiqueta como *Fix*; el número de la izquierda indica el total de bits, y el de la derecha el número de bits de la parte fraccionaria. Estos buses de entrada son registrados con elementos retardadores (bloques *Delay*), que son registros gobernados por el reloj que fija la tasa de entrada. Aunque inicialmente la NN está pensada para operar de forma combinacional, al entrar los datos sincronizados con un reloj, es preciso registrar la lógica a la entrada y salida, para que la compilación desde *System Generator* mantenga la señal de reloj, y sea aprovechable el *testbench* generado para la siguiente herramienta. Conviene resaltar que los bloques *Delay* usados en este diseño tienen latencia 1; es decir, retrasan la señal un ciclo de reloj. Estos 10 buses deben conectarse a las 8 neuronas de la capa intermedia; para facilitar la edición de este cableado, los buses se agrupan en un bloque *Goto*.

En la figura 4.5 se tiene la capa intermedia de la NN, que consta de 8 neuronas. Las señales que provienen de la capa de entrada se conectan a esta capa mediante un bloque *From*. Cada neurona consta de dos etapas; una primera, donde se multiplican las entradas por los coeficientes y se suma la polarización; y una segunda, que conforma la función de transferencia.

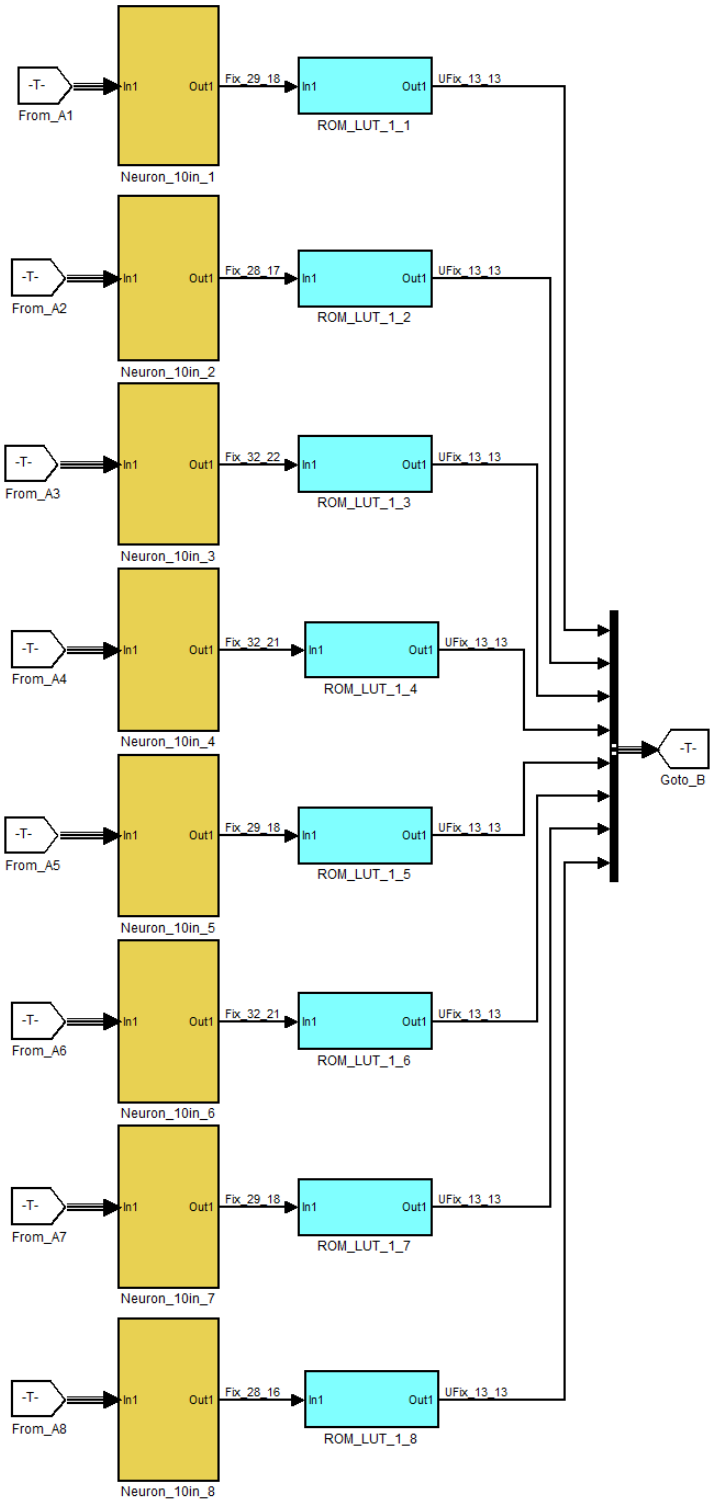


Figura 4.5. Capa intermedia de la NN para pejibaye.

En la figura 4.6 se tiene la primera fase de la primera neurona de la capa oculta. De forma análoga a las entradas, es preciso determinar el número de bits necesarios para la representación de los coeficientes en formato de punto fijo. Igualmente, se hace según lo explicado en el apartado 3.5 y la figura 3.7 del capítulo anterior. Inicialmente se fijó un error máximo del 1% en la representación. El mismo programa de Matlab es el encargado de determinar el formato de representación: bit de signo, si fuera necesario; y número de bits de la parte entera y fraccional. Por ejemplo, el coeficiente de valor -2.060084996074594 precisa de 9 bits para su representación; de los que 1 es bit de signo, 2 la parte entera, y 6 forman la parte fraccionaria; con este formato el valor representado en punto fijo es -2,063. Este valor corresponde con el coeficiente del multiplicador *CMult3* de la figura 4.6.

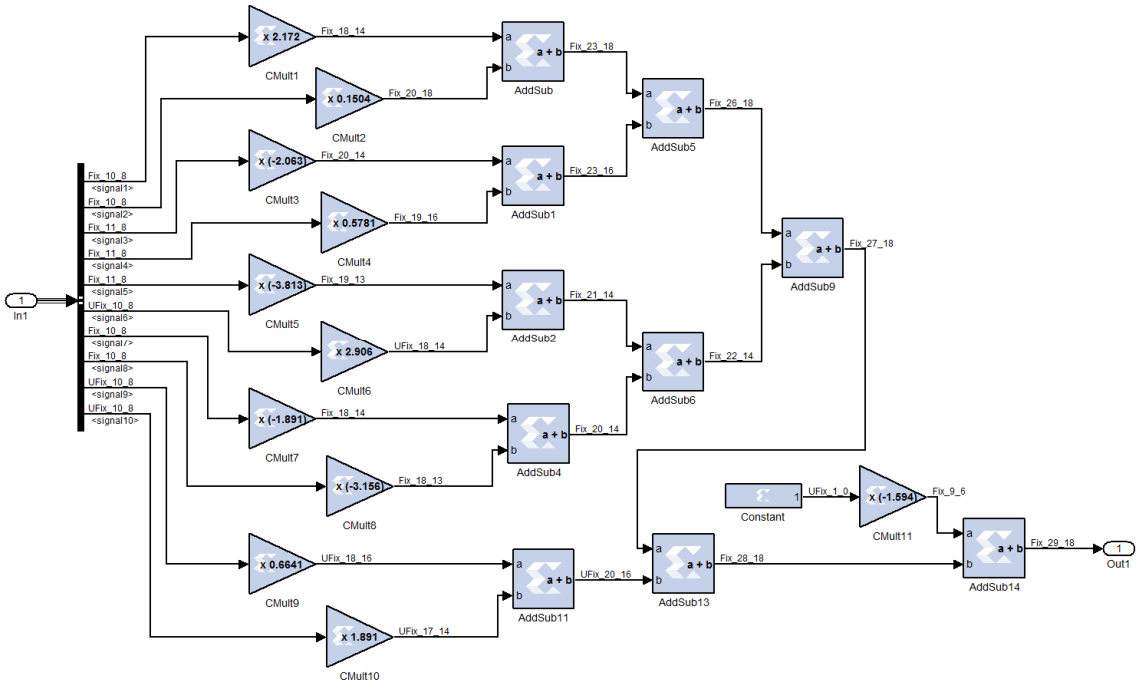


Figura 4.6. Primera fase de la primera neurona de la capa oculta para pejibaye.

La segunda etapa de las neuronas consiste en la implementación de la función de transferencia, en este caso se optó por la función *logsig* para todas las neuronas de la NN. Su diseño se realizó atendiendo al diagrama de bloques de la figura 4.7. La constante *Constant1* establece el límite de la abscisa, por debajo de la cual se

representa con un valor mínimo (próximo a cero) y no se supera el error máximo especificado. Este error inicialmente fue del 1%. De la misma forma, la constante *Constant2* establece el límite de la abscisa, por encima de la que se representa con un valor máximo (próximo a uno) y no se supera el error máximo especificado. Los comparadores de la figura 4.7, *Relational1* y *Relational2*, determinan si el valor de la entrada es menor o mayor que esas constantes. El concatenador *Concat*, a cuyas entradas llegan las salidas de los comparadores, determina si la entrada es menor que *Constant1*, si es mayor que *Constant2*, o si se encuentra entre ellos; esta información se tiene codificada con sus dos bits de salida. Los dos bits de salida del concatenador controlan un multiplexor, que deja pasar *Constant1,* el valor de la entrada o *Constant2;* dependiendo de si la entrada era menor que *Constant1*, si se encuentra entre *Constant1* y *Constant2*, o es mayor que *Constant2*, respectivamente. Con los bloques entre la salida del multiplexor y la entrada de la memoria de solo lectura (ROM, *Read Only Memory*), se produce una conversión entre el valor de la abscisa y la posición de memoria donde se encuentra almacenado el valor de la función en la ROM. En esta memoria se almacena un número de muestras de la función de transferencia, que es potencia de dos, para aprovechar la capacidad del bus de direcciones de la entrada. Existen por tanto dos parámetros a configurar en la ROM: el número de muestras almacenadas y el error de la representación en esas muestras. En el sistema mostrado se almacenaron 16 palabras con un error del 1%.

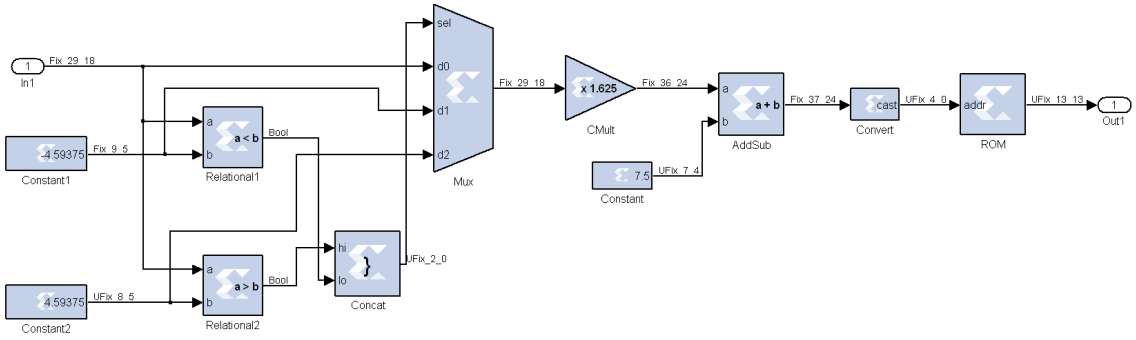


Figura 4.7. Implementación de la función de transferencia logsig mediante una ROM.

En la figura 4.8 se tiene la representación de la función *logsig* (a) y las muestras que la aproximan con la función de transferencia implementada (b). Además se

representa el error en (c) y el error relativo en (d). Debe tenerse en cuenta que el error en (c), se representa como la salida de la función en punto fijo implementada menos el valor en punto flotante que toma la función *logsig*, sin tomar el valor absoluto. De la misma forma, para el error relativo, no se toma el valor absoluto.

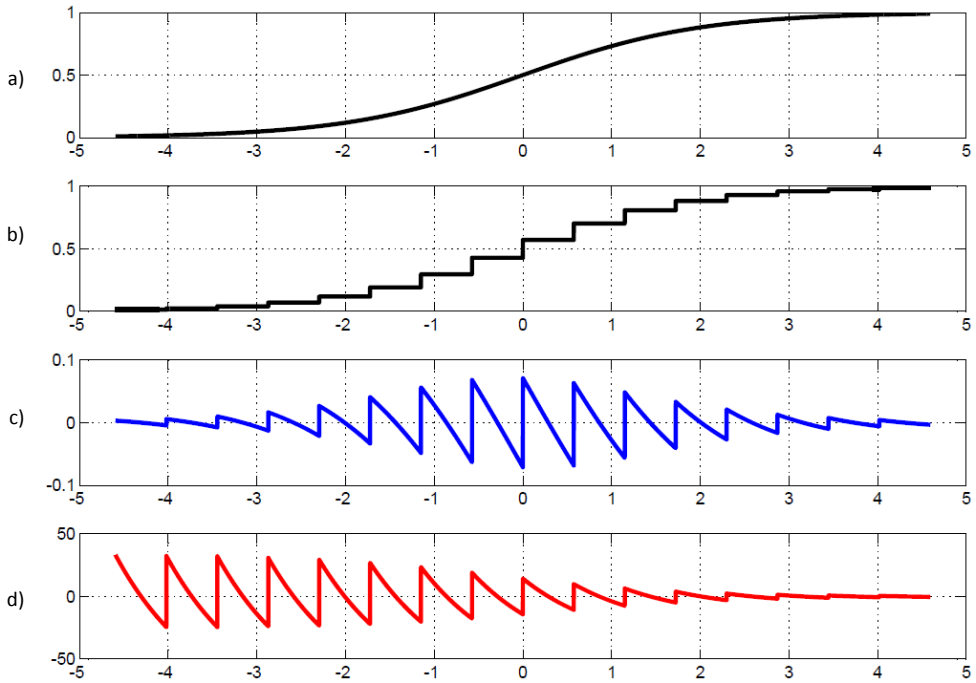


Figura 4.8. a) Función logsig, b) salida de la función implementada, c) error y d) error relativo.

Las ocho neuronas diseñadas con las arquitecturas de las dos etapas anteriores (figura 4.6 y figura 4.7), y conectadas como indica la figura 4.5, forman la capa oculta. Las ocho salidas de estas neuronas se agrupan con un bloque *Goto* para conectarla a la capa de salida. La capa de salida se muestra en la figura 4.9.

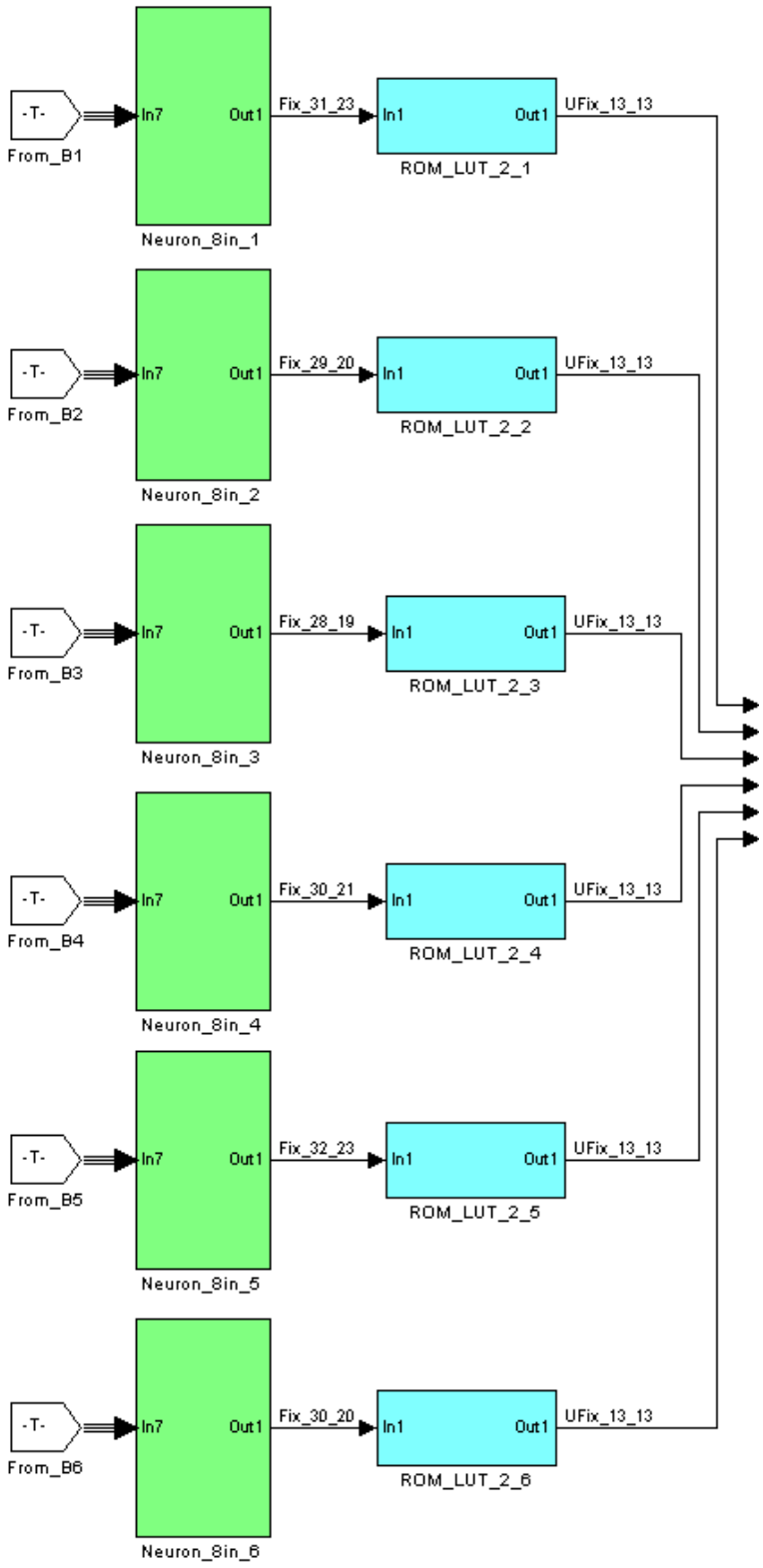


Figura 4.9. Capa de salida de la NN para pejibaye.

Las neuronas de la capa de salida se diseñaron con dos etapas como en la capa oculta. La primera etapa de la primera neurona se muestra en la figura 4.10, que dispone de 8 entradas. La función de transferencia es igual que las de la capa oculta.

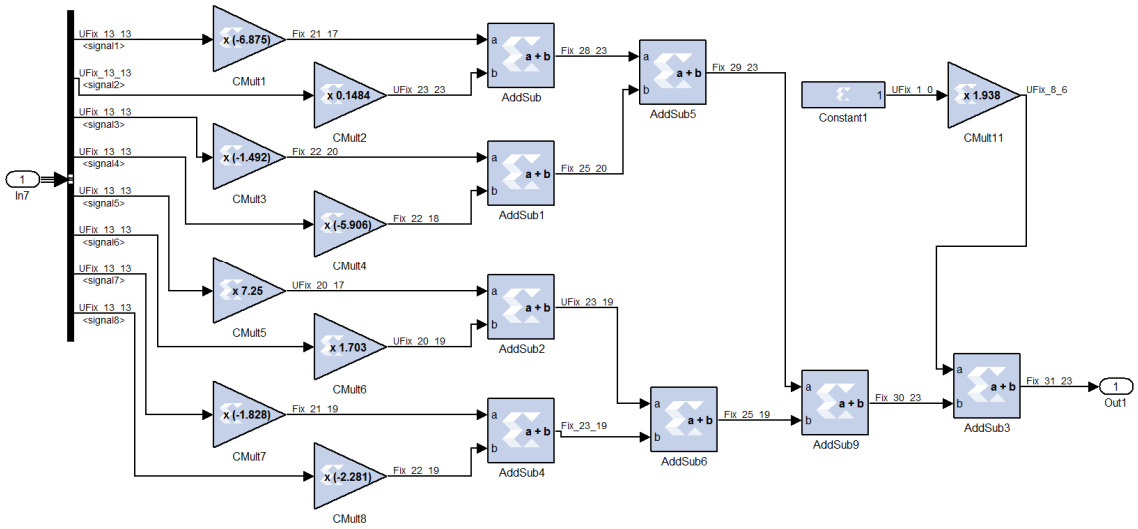


Figura 4.10. Primera fase de la primera neurona de la capa de salida para pejibaye.

La capa de salida se conectó a un bloque decodificador, llamado *DECODER*, que se muestra en la figura 4.11. Este bloque es el encargado de detectar cuál de las neuronas de salida tiene mayor amplitud, poniendo un "1" en la salida correspondiente y un "0" en las 5 restantes. Así se indica la clase a la que pertenece las muestras de la entrada. Este bloque está formado por comparadores, multiplexores y funciones lógicas. Obviamente sus entradas son sin signo y sus seis salidas booleanas.

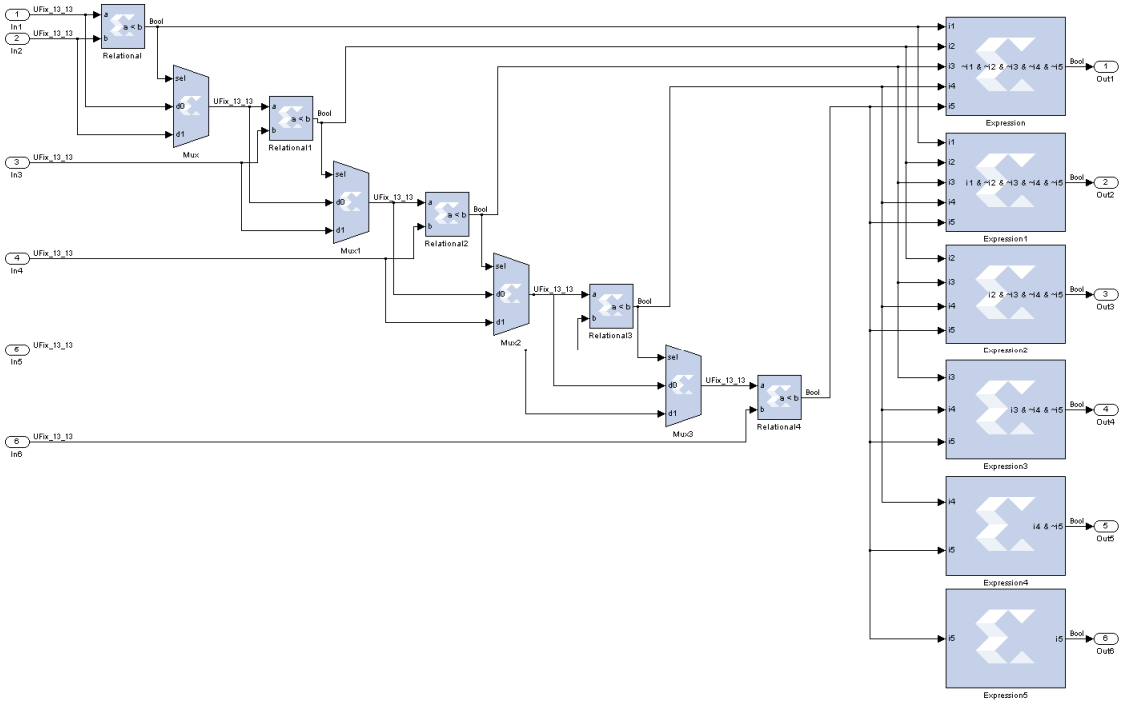


Figura 4.11. Bloque decodificador de las neuronas de salida para pejibaye.

El bloque *DECODER* se agrupa en un subsistema de *Simulink* como muestra la figura 4.12. Los seis bits de salida del decodificador se registrar con bloques *Delay* por los mismos motivos que se registró las entradas. Finalmente, las salidas de los bloques *Delay* se conectan a los pines de la salida de la FPGA mediante bloques *Gateway Out*, como indica la figura 4.12.

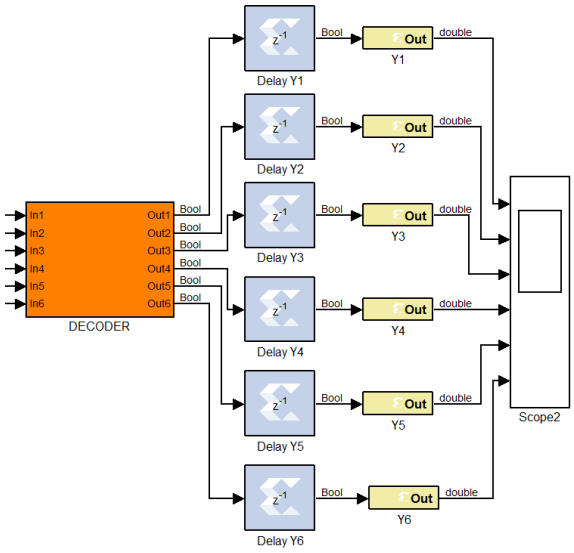


Figura 4.12. Conexión de las salidas del decodificador a los registros de salida y a los pines de salida de la FPGA para pejibaye.

El sistema final diseñado tiene el aspecto de la figura 4.13. Entre los buses de entrada *Gateway In*, y los bits de salida *Gateway Out*, se localiza el sistema de punto fijo diseñado para la FPGA. Además, en el modelo de *Simulink*, se observan algunos bloques auxiliares. Unos son los bloques *Scope*, que permiten la visualización de las formas de ondas en *Simulink*. Por otro lado, el bloque *System Generator*, es de obligado uso, permite elegir el tipo de FPGA y compilar el diseño.

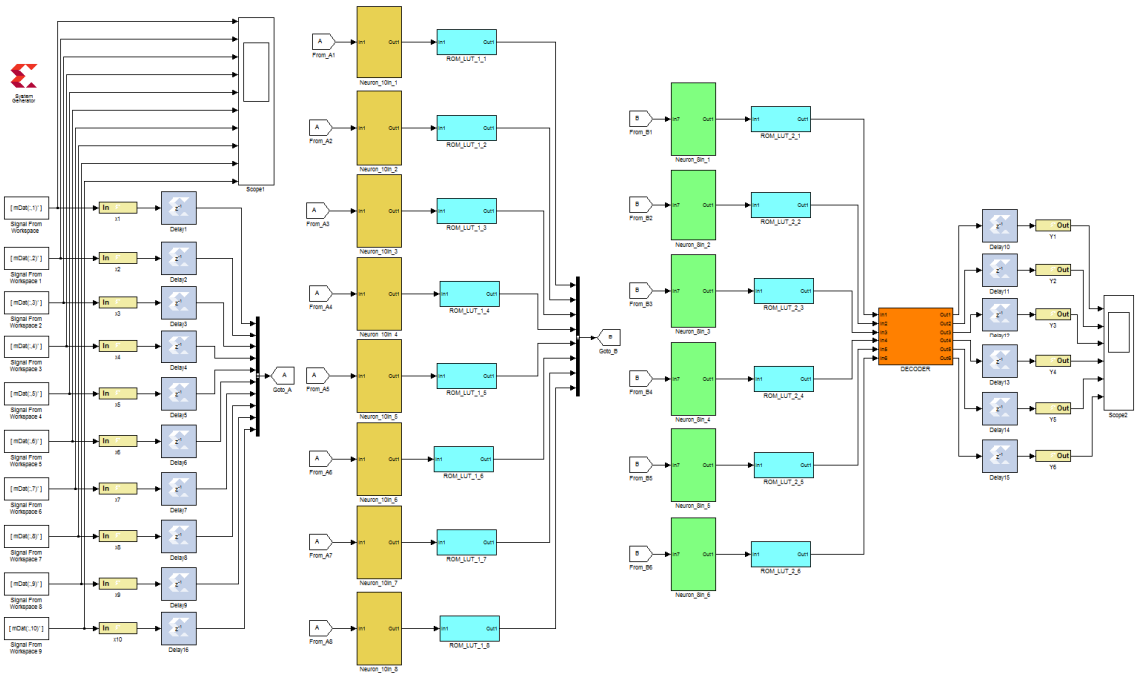


Figura 4.13. Diseño en Simulink del sistema final para pejibaye.

Descrito el modelo de Simulink, se puede realizar la simulación funcional. Las diez entradas se muestran en la figura 4.14 y las salidas en la figura 4.15. Las salidas se retrasan dos periodos de reloj respecto a las entradas por la existencia de los elementos de retardo en las entradas y salidas. Se introdujeron los 9 elementos de cada clase usados en el testeo, esto se hizo de forma secuencial. La tasa de acierto fue del 100% y se obtuvo la matriz de confusión de la figura 4.2. Es posible enviar los datos de salida al espacio de variables de Matlab, con el bloque *To Workspace*, lo que permitió elaborar la matriz de confusión de forma automática. Debe ponerse de relieve que esta simulación solo tarda unos minutos y permite comprobar la funcionalidad del diseño en punto fijo. Debe destacarse que se puede visualizar el reloj de entrada, que no se muestra por simplificación, y será mostrado en las simulaciones de la herramienta posterior. Este reloj de entrada se puede mostrar como señal de salida con el bloque *Clock Probe* que envía el reloj a un pin de salida.

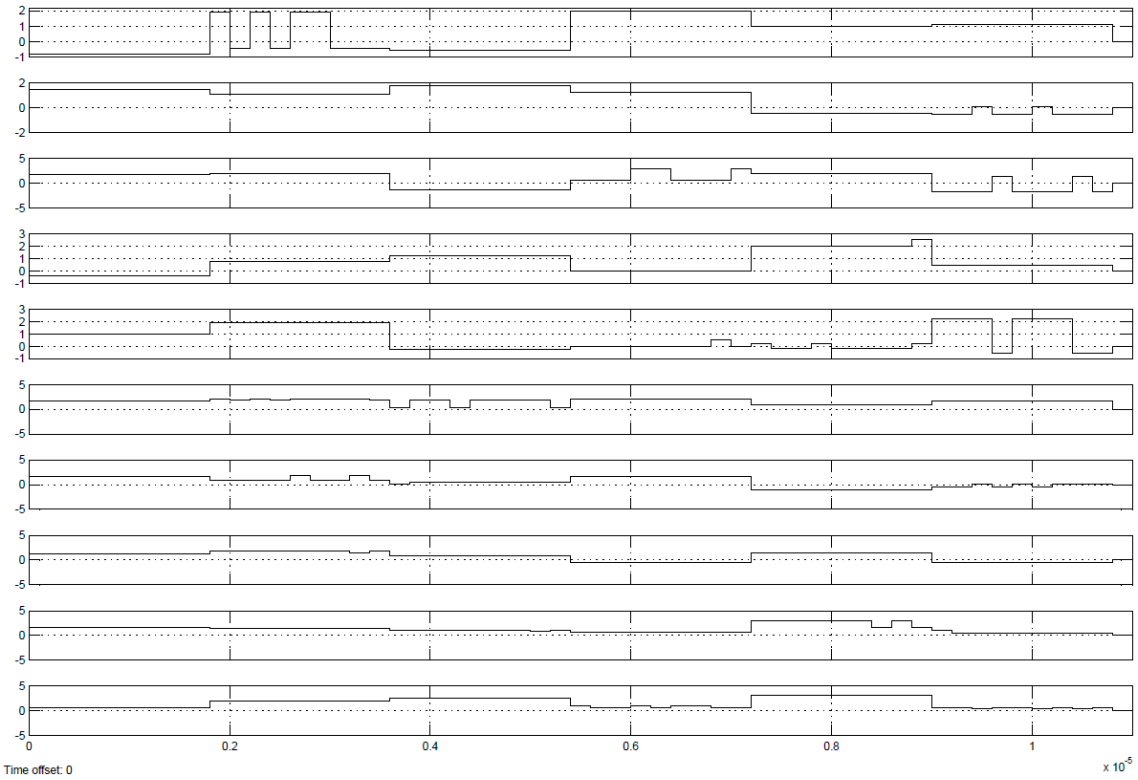


Figura 4.14. Señales de entrada en la NN para pejibaye.

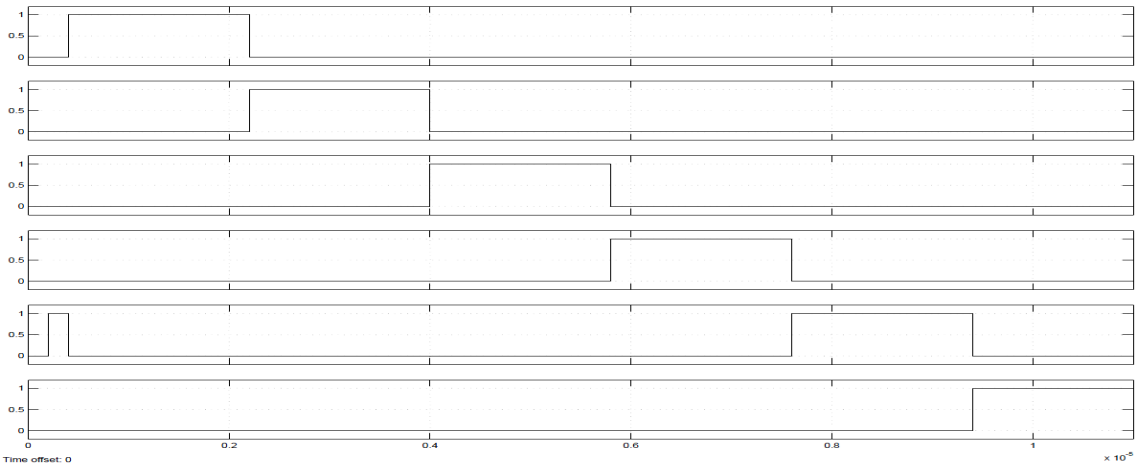


Figura 4.15. Señales de salida en la NN para pejibaye.

El bloque *System Generator* y su ventana de configuración se muestran en la figura 4.16, con este bloque se realiza la compilación del diseño. En el bloque *System Generator* primeramente se puede elegir el tipo de compilación. En segundo lugar se puede elegir el dispositivo FPGA. En la opción herramienta de síntesis se especifica la que se elige para sintetizar el circuito: *Synplify*, *Synplify Pro* o *Xilinx Synthesis Tool*

(XST). En la opción del lenguaje de descripción hardware el diseñador puede elegir entre VHDL o Verilog.

Al activar la opción *Create testbench* se genera el fichero en el HDL con las señales de entrada para el *Integrated System Environment*, que de otra forma tendrían que ser generadas de forma manual. Con el directorio de destino se define donde se van a escribir los ficheros en HDL para el proyecto en *Integrated System Environment*. Al activar la opción *Create interface document* se genera un fichero HTM (*Hypertext Markup Language*) con las principales características del diseño.

En las opciones de reloj (*Clocking*) se define su periodo, su pin de entrada, el modo de implementación en el caso de multitasa; y el periodo en el sistema de *Simulink*, que es el máximo común divisor de los periodos de las tasas que aparecen en el sistema. Finalmente, en *General* se especifica el tipo de información que se muestra en el icono de bloque.

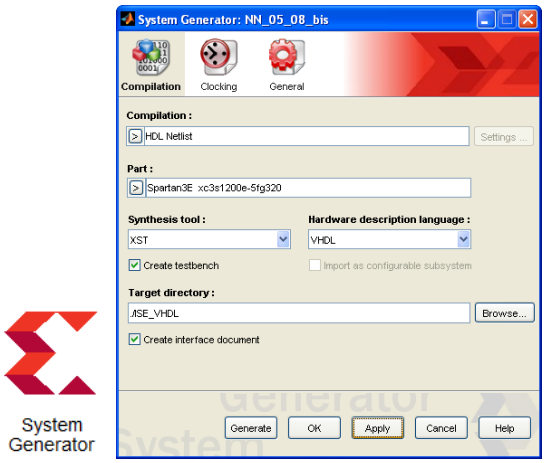


Figura 4.16. Bloque System Generator y su ventana de configuración.

Una vez configurado el bloque *System Generator* se puede proceder a la compilación del diseño, que se inicia al iniciar *Generate*. Al finalizar la compilación se obtiene una ventana como la de la figura 4.17.

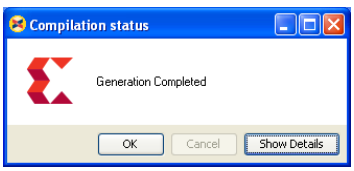


Figura 4.17. Ventana que se muestra al finalizar la compilación con System Generator.

Inicialmente, se usó para la representación de los coeficientes un error del 1%. Igualmente se usaron 16 posiciones de memoria, con un error del 1%, para la representación de las muestras de las funciones de transferencia almacenadas en la ROM. Con estos valores se alcanzó la funcionalidad deseada. Cabe preguntarse si se puede aumentar el error, lo que disminuye el número de bits, a la vez que se mantiene la funcionalidad. Igualmente, cabe preguntarse si se puede disminuir el número de palabras en la ROM y mantener la funcionalidad. Obviamente, la variación de estos parámetros a partir de ciertos valores hace que se pierda la funcionalidad. En la tabla 4.1 se muestra la funcionalidad del sistema en función de estos parámetros. En las simulaciones se comprobó que con un error del 14% no se conseguía la funcionalidad por mucho que se aumentara el número de palabras en la ROM. En el mismo sentido, con memorias ROM de 2 palabras, no se alzanza la funcionalidad por mucho que se disminuya el error.

Tabla 4.1. Funcionalidad del sistema frente al error en la representación y el número de palabras en las ROM para pejibaye.

		Error en la representación (%)													
		1	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***
Número de palabras en las ROM	***1024***	SI	SI	SI	SI	SI	SI	SI	SI	SI	SI	SI	SI	SI	NO
	512	SI	SI	SI	SI	SI	SI	SI	SI	SI	SI	SI	SI	SI	NO
	256	SI	SI	SI	SI	SI	SI	SI	SI	SI	SI	SI	SI	SI	NO
	128	SI	SI	SI	SI	SI	SI	SI	SI	SI	SI	SI	SI	SI	NO
	64	SI	SI	SI	SI	SI	SI	SI	SI	SI	SI	SI	SI	NO	NO
	32	SI	SI	SI	SI	SI	SI	SI	SI	SI	SI	SI	SI	NO	NO
	16	SI	SI	SI	SI	SI	SI	SI	SI	SI	SI	SI	SI	NO	NO
	8	SI	SI	SI	SI	SI	SI	SI	SI	SI	SI	SI	SI	NO	NO
	4	SI	SI	SI	SI	SI	SI	SI	SI	NO	NO	NO	NO	NO	NO
	2	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO

4.2.2.2 Implementación con Integrated System Environment

Claramente interesa comparar los casos de 8% de error y 4 palabras en la ROM, 12% de error y 8 palabras en la ROM; y 13% de error y 128 palabras en la ROM. Entre estos casos, se produce para un parámetro un aumento de recursos hardware, y para el otro una disminución. Los tres casos se compilaron con la intención de comparar el área ocupada, la potencia y la velocidad.

Resultado para 8% de error y 4 palabras en la ROM. Lenguaje VHDL.

La compilación se realizó con *System Generator* para obtener la descripción en VHDL. En el proyecto obtenido para *Integrated System Environment* se realizó la simulación para la fase final de la implementación en la FPGA (*Post Place and Route Simulation*). Como simulador se usó ISim (*ISE Simulator*), integrado en la herramienta. Se obtuvieron las formas de onda de la figura 4.18; donde cabe destacar la existencia del reloj de entrada de 5 MHz. La compilación se realizó para un dispositivo Spartan3E, tipo xc3s1200e, grado de velocidad -5, tipo de encapsulado ft y 256 pines; de forma compacta se denota como Spartan3E xc3s1200e-5ft256.

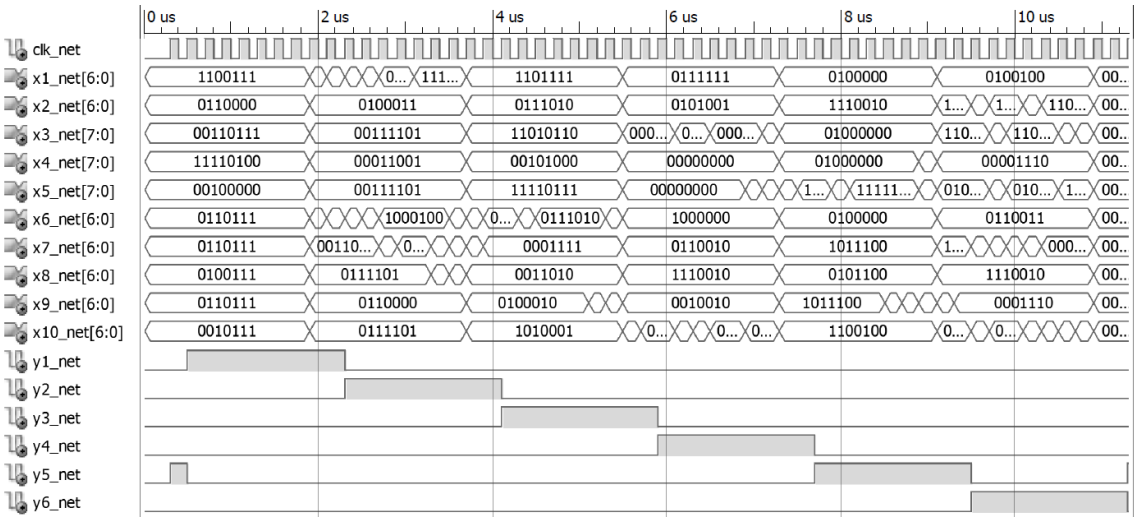


Figura 4.18. Simulación de la implementación para pejibaye en la FPGA, 8% de error y 4 palabras en las ROM.

Los recursos hardware necesarios se resumen en la tabla 4.2; estos datos se tienen en *Design Summary*. Los SLICES son los bloques de lógica internos en los que se divide la FPGA; los LUT son tablas de búsqueda (*Lookup-Table*). Debe resaltarse el número de SLICES necesarios, porque los LUT están incluidos dentro de los SLICES. El segundo parámetro importante es el número de pines de entrada-salida necesarios.

Tabla 4.2. Recursos hardware para 8% de error y 4 palabras en la ROM, aplicado a pejibaye y compilado en VHDL.

Entidad	Usados	Disponibles	Tasa de uso
SLICES	2.750	8.672	31%
LUT de 4 entradas	5.149	17.344	29%
Bloques de E/S	80	190	42%

Para la frecuencia de reloj de 5 MHz se obtienen las potencias de la tabla 4.3; para esto se usó la utilidad *XPower Analyzer* del ISE. La frecuencia máxima de funcionamiento es de 11,034 MHz; este dato está disponible en el informe posterior a la elección y conexión de los circuitos de la FPGA (*Place and Route Report*). En la tabla 4.3 se muestran las potencias para 11,0 MHz, que es el valor redondeado que permite introducir la herramienta.

Tabla 4.3. Potencias para 8% de error y 4 palabras en la ROM, aplicado a pejibaye y compilado en VHDL.

		Potencia (mW)		
		Estática	**Dinámica**	**Total**
Frecuencia (MHz)	**5,0**	159,16	7,86	167,02
	11,0	159,41	17,30	176,71

Resultado para 8% de error y 4 palabras en la ROM. Lenguaje Verilog.

En este apartado se muestra el caso de 8% de error y 4 palabras en la ROM, se compila con *System Generator* para generar el proyecto en Verilog para el mismo dispositivo. La simulación en este caso presenta el mismo aspecto que la de la figura 4.18. Los requisitos de área se muestran en la tabla 4.4.

Tabla 4.4. Recursos hardware para 8% de error y 4 palabras en la ROM, aplicado a pejibaye y compilado en Verilog.

Entidad	**Usados**	**Disponibles**	**Tasa de uso**
SLICES	2.780	8.672	32%
LUT de 4 entradas	5.161	17.344	29%
Bloques de E/S	80	190	42%

Los resultados de la potencia para la frecuencia de 5 MHz se tienen en la tabla 4.5. La frecuencia máxima de funcionamiento es de 10,200 MHz, en la tabla 4.5 se muestran las potencias para esta frecuencia.

Tabla 4.5. Potencias para 8% de error y 4 palabras en la ROM, aplicado a pejibaye y compilado en Verilog.

		Potencia (mW)		
		Estática	**Dinámica**	**Total**
Frecuencia (MHz)	**5,0**	159,15	7,70	166,86
	10,2	159,37	15,71	175,08

Resultado para 12% de error y 8 palabras en la ROM. Lenguaje VHDL.

A continuación se analiza el caso de 12% de error y 8 palabras en la ROM, compilado en *System Generator* para generar el proyecto en VHDL, para el mismo dispositivo. La simulación en este caso presenta el mismo aspecto que la de la figura 4.18. Los requisitos de área se muestran en la tabla 4.6.

Tabla 4.6. Recursos hardware necesarios para 12% de error y 8 palabras en la ROM, aplicado a pejibaye y compilado en VHDL.

Entidad	**Usados**	**Disponibles**	**Tasa de uso**
SLICES	2.574	8.672	29%
LUT de 4 entradas	4.784	17.344	27%
Bloques de E/S	80	190	42%

Los resultados de la potencia para la frecuencia de 5 MHz se tienen en la tabla 4.7. La frecuencia máxima de funcionamiento es de 11,561 MHz. En la tabla 4.7 se muestran las potencias para 11,6 MHz, que es el valor redondeado que permite introducir la herramienta.

Tabla 4.7. Potencias para 12% de error y 8 palabras en la ROM, aplicado a pejibaye y compilado en VHDL.

		Potencia (mW)		
		Estática	**Dinámica**	**Total**
Frecuencia (MHz)	**5,0**	159,15	7,51	166,66
	11,6	159,41	17,36	176,77

Resultado para 12% de error y 8 palabras en la ROM. Lenguaje Verilog.

Seguidamente se muestra el caso de 12% de error y 8 palabras en la ROM, compilado en *System Generator* para generar el proyecto en Verilog, para el mismo

dispositivo. La simulación en este caso presenta el mismo aspecto que la de la figura 4.18. Los requisitos de área se muestran en la tabla 4.8.

Tabla 4.8. Recursos hardware necesarios para 12% de error y 8 palabras en la ROM, aplicado a pejibaye y compilado en Verilog.

Entidad	Usados	Disponibles	Tasa de uso
SLICES	2.640	8.672	30%
LUT de 4 entradas	4.793	17.344	27%
Bloques de E/S	80	190	42%

Los resultados de la potencia para la frecuencia de 5 MHz se tienen en la tabla 4.9. La frecuencia máxima de funcionamiento es de 10,800 MHz. En la tabla 4.9 se muestran las potencias para la máxima frecuencia de funcionamiento.

Tabla 4.9. Potencias para 12% de error y 8 palabras en la ROM, aplicado a pejibaye y compilado en Verilog.

		Potencia (mW)		
		Estática	Dinámica	Total
Frecuencia (MHz)	**5,0**	159,15	7,45	166,60
	10,8	159,38	16,10	175,48

Resultado para 13% de error y 128 palabras en la ROM. Lenguaje VHDL.

A continuación se analiza el caso de 13% de error y 128 palabras en la ROM, compilado en *System Generator* para generar el proyecto en VHDL, para el mismo dispositivo. La simulación en este caso presenta el mismo aspecto que la de la figura 4.18. Los requisitos de área se muestran en la tabla 4.10.

Tabla 4.10. Recursos hardware necesarios para 13% de error y 128 palabras en la ROM, aplicado a pejibaye y compilado en VHDL.

Entidad	Usados	Disponibles	Tasa de uso
SLICES	2.802	8.672	32%
LUT de 4 entradas	5.121	17.344	29%
Bloques de E/S	80	190	42%

Los resultados de la potencia para la frecuencia de 5 MHz se tienen en la tabla 4.11. La frecuencia máxima de funcionamiento es de 11,014 MHz. En la tabla 4.11 se muestran las potencias para 11,0 MHz, que es el valor redondeado que permite introducir la herramienta.

Tabla 4.11. Potencias para 12% de error y 8 palabras en la ROM, aplicado a pejibaye y compilado en VHDL.

		Potencia (mW)		
		Estática	**Dinámica**	**Total**
Frecuencia (MHz)	**5,0**	159,16	7,97	167,13
	11,0	159,41	17,53	176,94

Resultado para 13% de error y 128 palabras en la ROM. Lenguaje Verilog.

Seguidamente se muestra el caso de 13% de error y 128 palabras en la ROM, compilado en *System Generator* para generar el proyecto en *Verilog*, para el mismo dispositivo. La simulación en este caso presenta el mismo aspecto que la de la figura 4.18. Los requisitos de área se muestran en la tabla 4.12.

Tabla 4.12. Recursos hardware necesarios para 13% de error y 128 palabras en la ROM, aplicado a pejibaye y compilado en Verilog.

Entidad	**Usados**	**Disponibles**	**Tasa de uso**
SLICES	2.800	8.672	32%
LUT de 4 entradas	5.093	17.344	29%
Bloques de E/S	80	190	42%

Los resultados de la potencia para la frecuencia de 5 MHz se tienen en la tabla 4.13. La frecuencia máxima de funcionamiento es de 10,686 MHz. En la tabla 4.13 se muestran las potencias para 10,7 MHz, que es el valor redondeado que permite introducir la herramienta.

Tabla 4.13. Potencias para 13% de error y 128 palabras en la ROM, aplicado a pejibaye y compilado en Verilog.

		Potencia (mW)		
		Estática	**Dinámica**	**Total**
Frecuencia (MHz)	**5,0**	159,16	7,83	166,99
	10,7	159,39	16,75	176,15

Resumen de los casos estudiados.

Como resumen debe observarse la tabla 4.14, donde se muestran las prestaciones en área, velocidad y potencia. En cuanto al área, no se muestra el número de pines, que casualmente coincide en los tres casos. Las mejores características de área y velocidad se consiguen en VHDL con 12% de error y 8 palabras en la ROM. La mejor prestación en potencia se consigue para el mismo caso, pero usando Verilog. Se tomará como óptimo el caso de VHDL, porque es el mejor en dos de los tres parámetros. En cualquier caso el empeoramiento en potencia es del 0,04%, cuando la mejora en área es del 2,5% y en frecuencia del 7%.

Tabla 4.14. Resumen de los casos estudiados para pejibaye.

	VHDL		**Verilog**	
8% de error 4 palabras en la ROM	*Área*	2.750 SLICES	*Área*	2.780 SLICES
	Frecuencia máxima	11,034 MHz	*Frecuencia máxima*	10,2 MHz
	Potencia (5 MHz)	167,02 mW	*Potencia (5 MHz)*	166,86 mW
12% de error 8 palabras en la ROM	*Área*	2.574 SLICES	*Área*	2.640 SLICES
	Frecuencia máxima	11,561 MHz	*Frecuencia máxima*	10,8 MHz
	Potencia (5 MHz)	166,66 mW	*Potencia (5 MHz)*	166,60 mW
13% de error 128 palabras en la ROM	*Área*	2.802 SLICES	*Área*	2.800 SLICES
	Frecuencia máxima	11,014 MHz	*Frecuencia máxima*	10,686 MHz
	Potencia (5 MHz)	167,13 mW	*Potencia (5 MHz)*	166,99 mW

Llegados a este punto se podría comparar el tiempo de respuesta de la NN en punto flotante, ejecutada con el *Neural Network Toolbox* de Matlab, con el sistema en punto fijo sobre la FPGA elegida.

En el ordenador personal en punto flotante una clasificación tarda 1,7 ms y en la FPGA elegida unos 86,5 ns; del orden de 20.000 veces más rápida. Estos resultados son meramente orientativos, claramente depende del ordenador personal y de la FPGA que se usen; sus características se muestran en la tabla 4.15.

Tabla 4.15. Comparación en velocidad entre un ordenador personal y la FPGA para pejibaye.

	Tipo	***Tiempo para una clasificación***
Ordenador personal	Windows 7 Home Premium 64 bits Intel(R) Core(TM) i7-3770 CPU @ 3.40 GHz RAM: 8 Gbytes Matlab R2010b 64 bits	1,7 ms
FPGA	Xilinx Spartan3E xc3s1200e-5ft256	86,5 ns

4.3 La clasificación de los pulsos electrocardiográficos

En este apartado se describe la implementación de una regla de oro obtenida en [Chadnani, 2011]; en particular, la descrita en el capítulo anterior. Tras la descripción del modelo en punto flotante, se realiza la implementación en punto fijo, con la misma metodología que la descrita para pejibaye.

4.3.1 Modelado en punto flotante

Igual que el caso anterior el entrenamiento se realizó con el *Neural Network Toolbox* de Matlab, y se usó una NN artificial tipo perceptrón multicapa. Debe recordarse que de cada pulso existen 400 casos para el entrenamiento y 400 para el testeo. Los pulsos estaban etiquetados, luego el entrenamiento fue supervisado. En la fase de entrenamiento se varió el número de capas intermedias, y se obtuvo un resultado óptimo con una sola capa oculta de 30 neuronas. Como se tiene 15 parámetros por pulso y existen 7 tipos se pulso, puede decirse entonces, que la NN es del tipo 15-30-7.

Como funciones de transferencia se usó la tipo *tansig* para la capa oculta y *purelin* para la capa de salida. Al finalizar el entrenamiento, se testeó el comportamiento de la NN tomando como entradas la parte de la base de datos no usada en el entrenamiento; se alcanzó los resultados que muestran la matriz de confusión de tabla 3.2 del capítulo anterior.

4.3.2 Diseño en punto fijo

Igual que en primer escenario se realizará en una primera etapa con *System Generator*, y posteriormente usando el *Integrated System Environment*. Estas dos fases del diseño se describen convenientemente en los siguientes apartados.

4.3.2.1 Diseño con System Generator

Una vez obtenida la regla de oro en formato de punto flotante, el siguiente paso es diseñar el sistema en aritmética de punto fijo, para ello se usó *System Generator* de Xilinx sobre *Simulink* de Matlab. La etapa de entrada de la NN se muestra en la figura 4.19. Con ciertas diferencias, el diseño sigue el mismo método que el escenario anterior. La NN mostrada obedece al máximo error del 4,2% y 64 posiciones en las ROM.

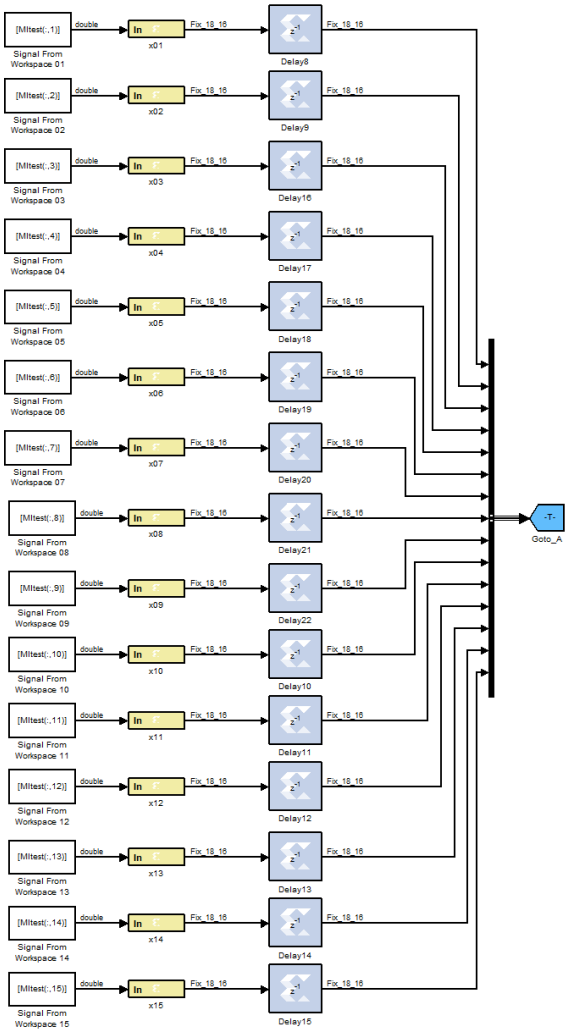


Figura 4.19. Etapa de entrada de la NN para ECG.

En la parte izquierda de la figura 4.19 se tienen las señales de entrada en formato de punto flotante (tipo *double*), que proceden del espacio de trabajo de Matlab. Se dispone de 15 entradas, por ser 15 los parámetros para la clasificación. A continuación se observan los puertos de entrada a la FPGA (*Gateway In*), donde se realiza una conversión a punto fijo. Estos puerto se configuran de la misma forma que en el escenario anterior. Por último, se fija un periodo de reloj, para marcar la velocidad con la que son tomados los parámetros en la entrada; este valor en este caso es arbitrario, y se fijó para una frecuencia de 5 MHz. Obviamente, este tasa de entrada es mucho más rápida que si proveniese de pulsos cardiacos, se elevó para comprobar cuál es el alcance en frecuencia de la NN. Por otro lado, el sistema para ECG no está pensado para operar en tiempo real; en ese caso el sistema de [Chadnani, 2011] debe rediseñarse usando memorias intermedias de almacenamiento que permitan conectarlo a una señal de ECG.

Los puertos de entrada se configuran con el mismo método del apartado anterior. Estos 15 buses deben conectarse a las 30 neuronas de la capa intermedia, para facilitar la edición de este cableado los buses se agrupan en un bloque *Goto*.

En la figura 4.20 se tiene la capa intermedia de la NN, que consta de 30 neuronas, agrupadas en grupos de 10. En la figura 4.21 se muestra la primera agrupación de 10 neuronas. Las señales que provienen de la capa de entrada se conectan a esta capa mediante un bloque *From*. Cada neurona consta de dos etapas; una primera, donde se multiplican las entradas por los coeficientes y se suma la polarización; y una segunda, que conforma la función de transferencia.

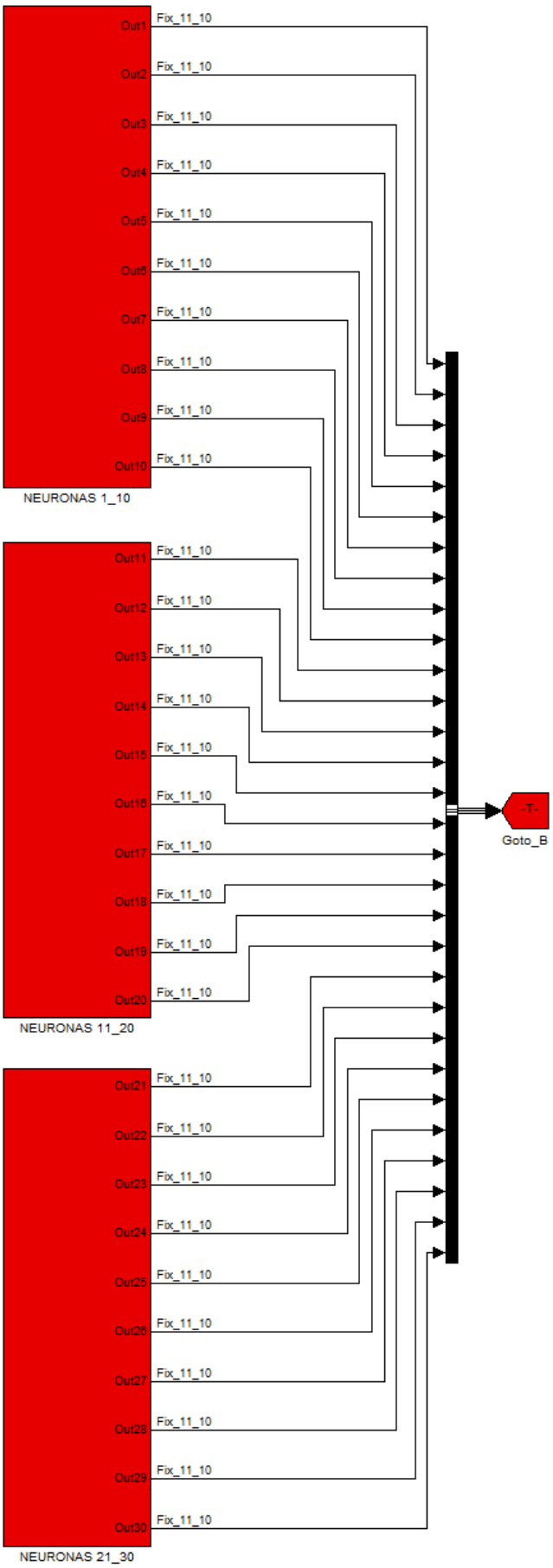


Figura 4.20. Capa intermedia de la NN para ECG.

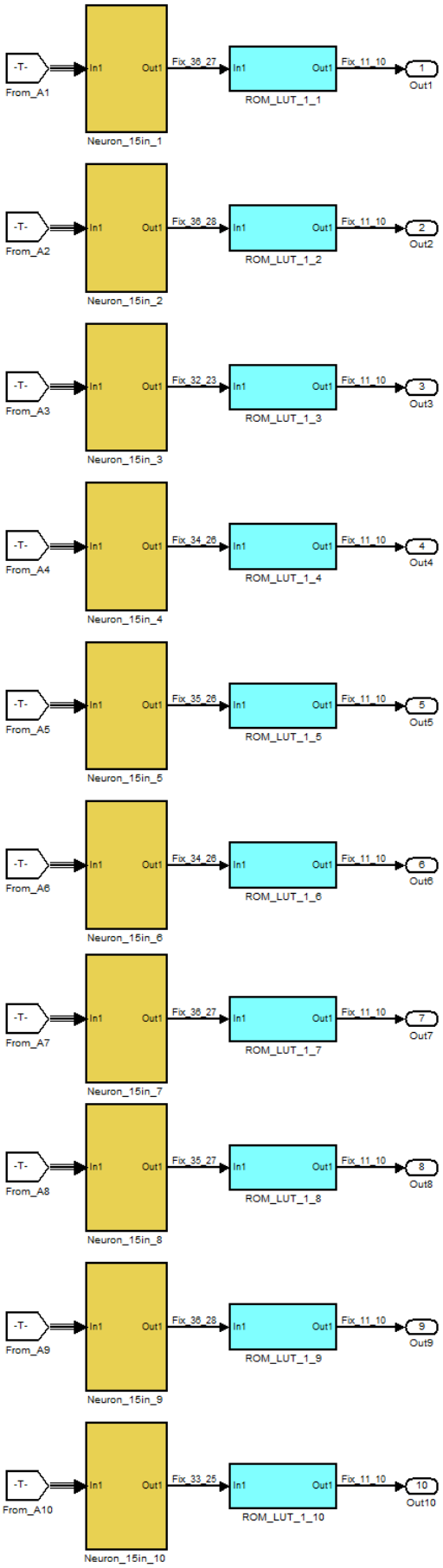


Figura 4.21. Agrupación de diez neuronas en la capa oculta de la NN para ECG.

En la figura 4.22 se tiene la primera fase de la primera neurona de la capa oculta. La configuración de esta etapa se realizó con el mismo método que el escenario anterior.

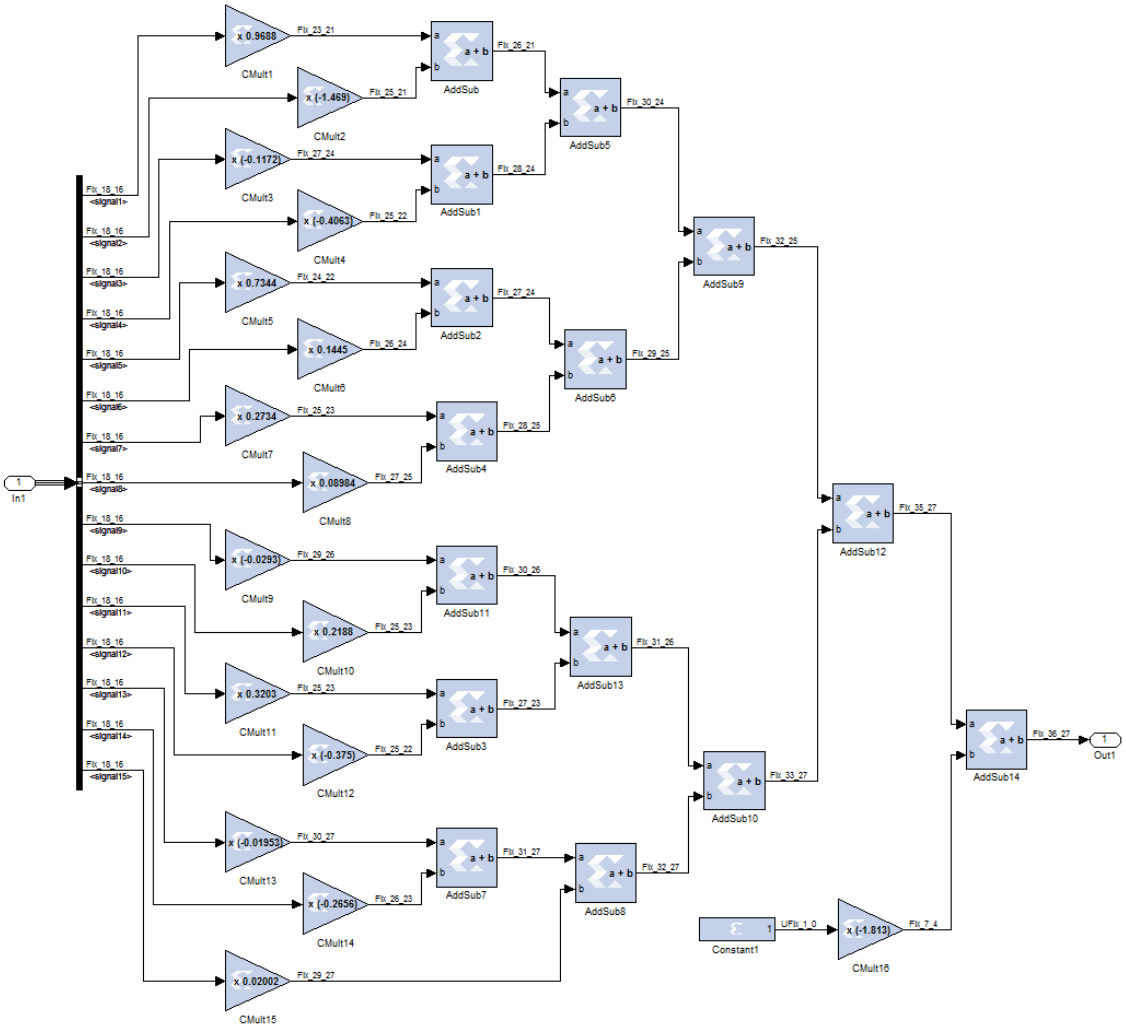


Figura 4.22. Primera fase de la primera neurona de la capa oculta para ECG.

La segunda etapa de las neuronas de la capa oculta se muestra en la figura 4.23. Esta consiste en la implementación de la función de transferencia, en este caso se optó por la función *tansig* para todas las neuronas de la capa oculta. Su diseño es análogo al caso de la función *logsig* del escenario anterior; la única diferencia es que los valores en la ROM se almacenan como *Fix*; es decir, complemento a dos con signo, dado que la función *tansig* es bipolar.

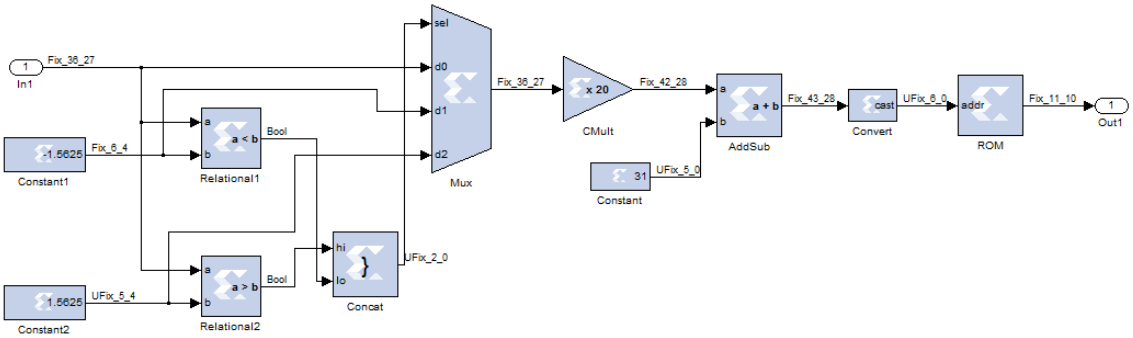


Figura 4.23. Implementación de la función de transferencia tansig mediante una ROM.

En la figura 4.24 se tiene la representación de la función *tansig* (a) y el valor de las muestras almacenadas en la ROM (b), la figura se muestra para 64 posiciones de memoria. Además se representa el error en (c) y el error relativo en (d). Debe tenerse en cuenta que el error en (c), se representa como la salida de la función en punto fijo implementada menos el valor en punto flotante que toma la función *tansig*, sin tomar el valor absoluto. De la misma forma, para el error relativo, no se toma el valor absoluto.

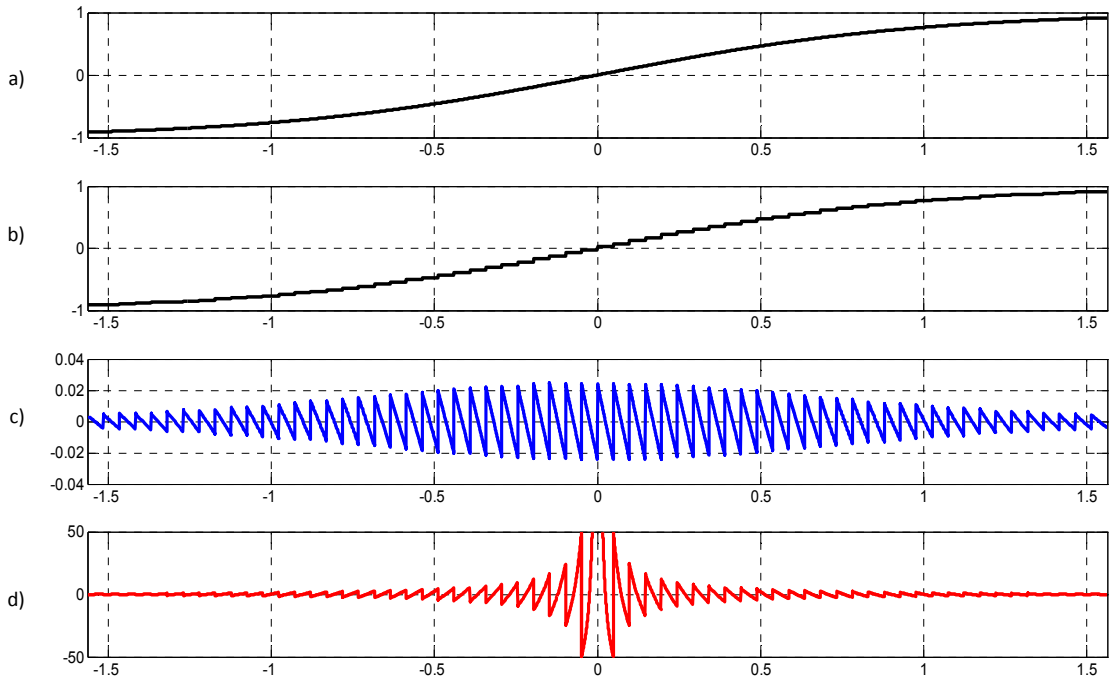


Figura 4.24. a) Función tansig, b) salida de la función implementada, c) error y d) error relativo.

Las treinta neuronas diseñadas con las dos etapas anteriores (figuras 4.22 y 4.23), y conectadas como indican las figuras 4.20 y 4.21, forman la capa oculta. Las treinta

salidas de estas neuronas se agrupan con un bloque *Goto* para conectarla a la capa de salida. La capa de salida se muestra en la figura 4.25. Las funciones de transferencia de estas neuronas son *purelin*; es decir, la función identidad, donde la salida es igual a la entrada. El diseño de estas funciones es trivial, mediante un cortocircuito, se deja un bloque que guarda esta conexión para recordar el tipo de función usada.

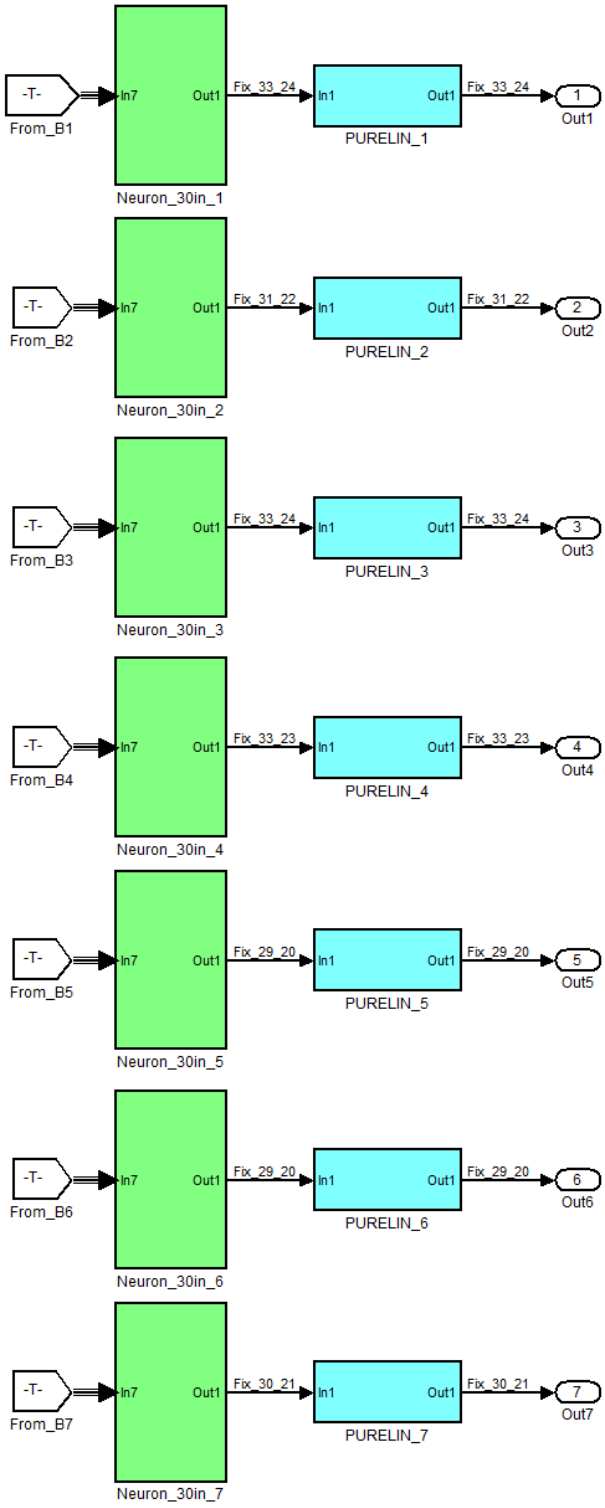


Figura 4.25. Capa de salida de la NN para ECG.

Las neuronas de la capa de salida se diseñaron con dos etapas como en la capa oculta. La primera etapa de la primera neurona se muestra en la figura 4.26, que dispone de 30 entradas. Abajo y a la derecha de esta figura se muestra el multiplicador que introduce el valor de polarización. Los multiplicadores por los pesos se agruparon en grupos de 10, el primer agrupamiento de la primera neurona se muestra en la figura 4.27.

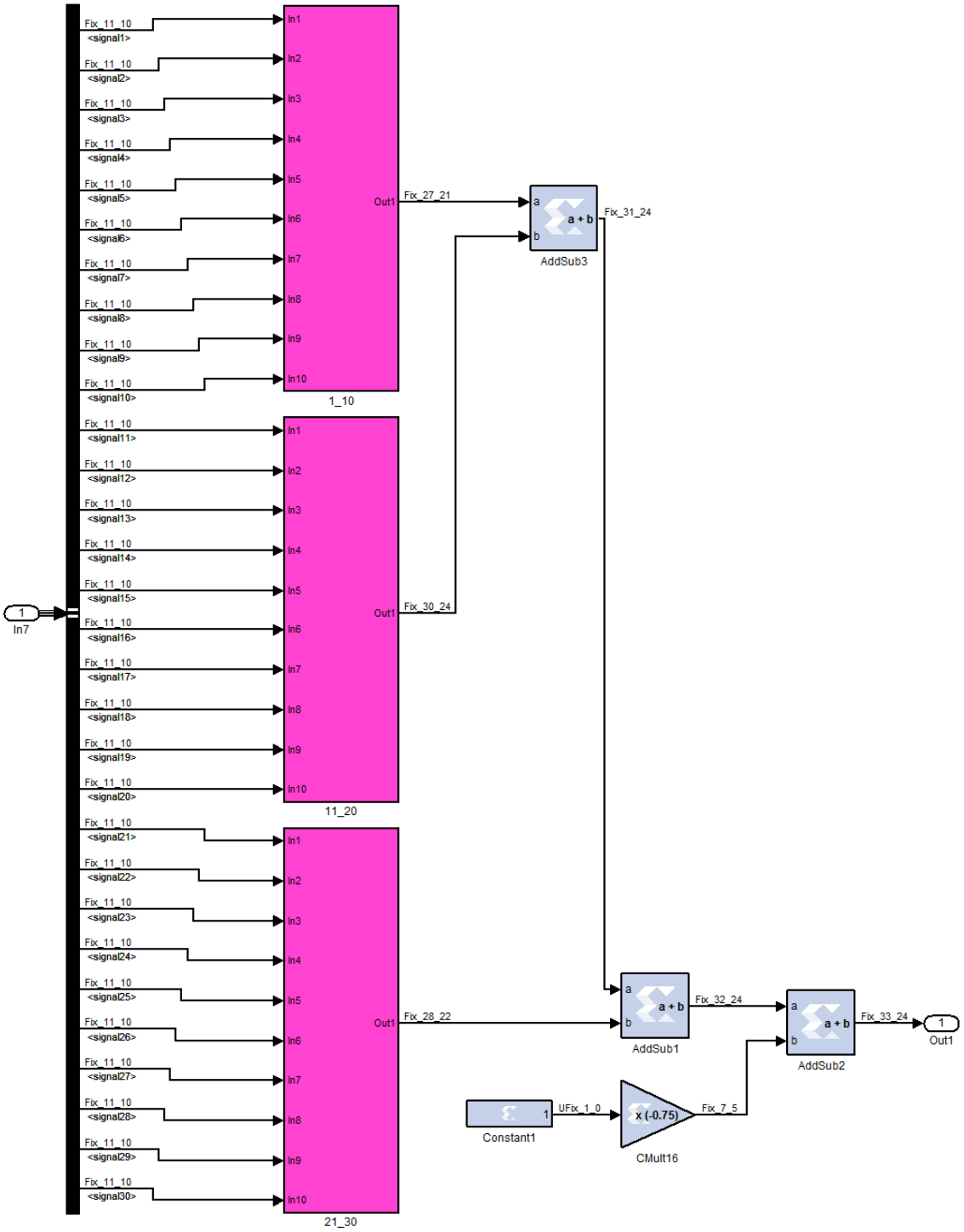


Figura 4.26. Primera fase de la primera neurona de la capa de salida para ECG.

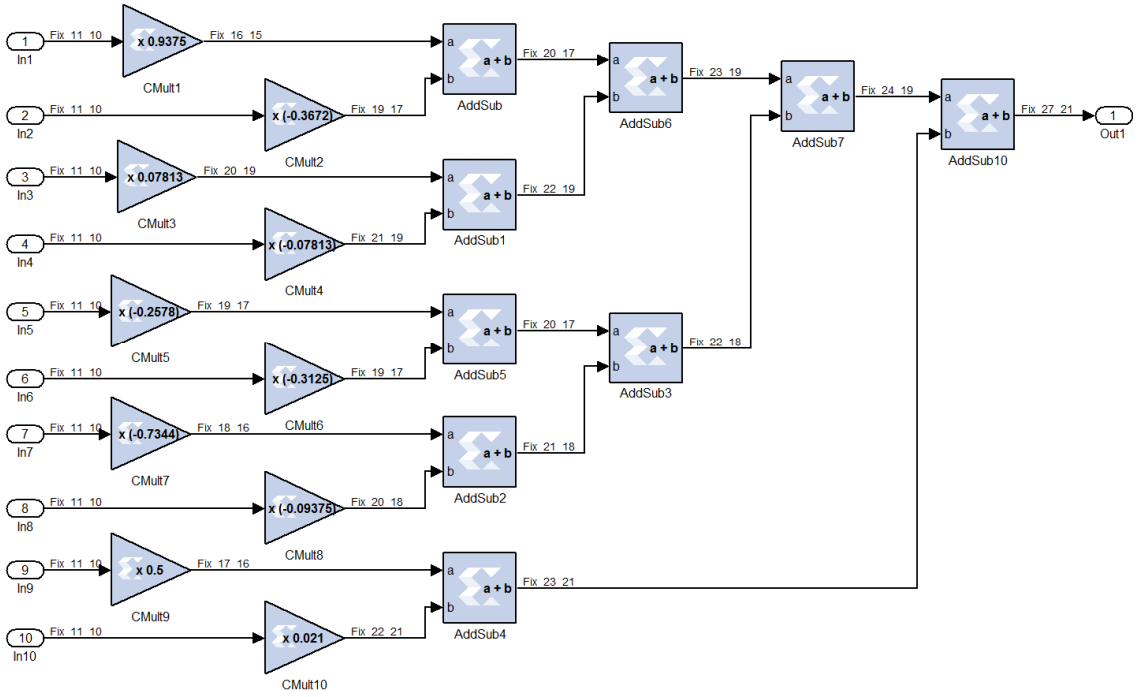


Figura 4.27. Primer agrupamiento de diez multiplicadores de la primera neurona de la capa de salida para ECG.

La capa de salida se conectó a un bloque decodificador, llamado *DECODER*, que se muestra en la figura 4.28. Este bloque es el encargado de detectar cuál de las neuronas de salida tiene mayor amplitud, poniendo un “1” en la salida correspondiente y un “0” en las 6 restantes. Así se indica la clase a la que pertenece las muestras de la entrada. Este bloque está formado por comparadores, multiplexores y funciones lógicas. Obviamente sus entradas son con signo y sus siete salidas booleanas.

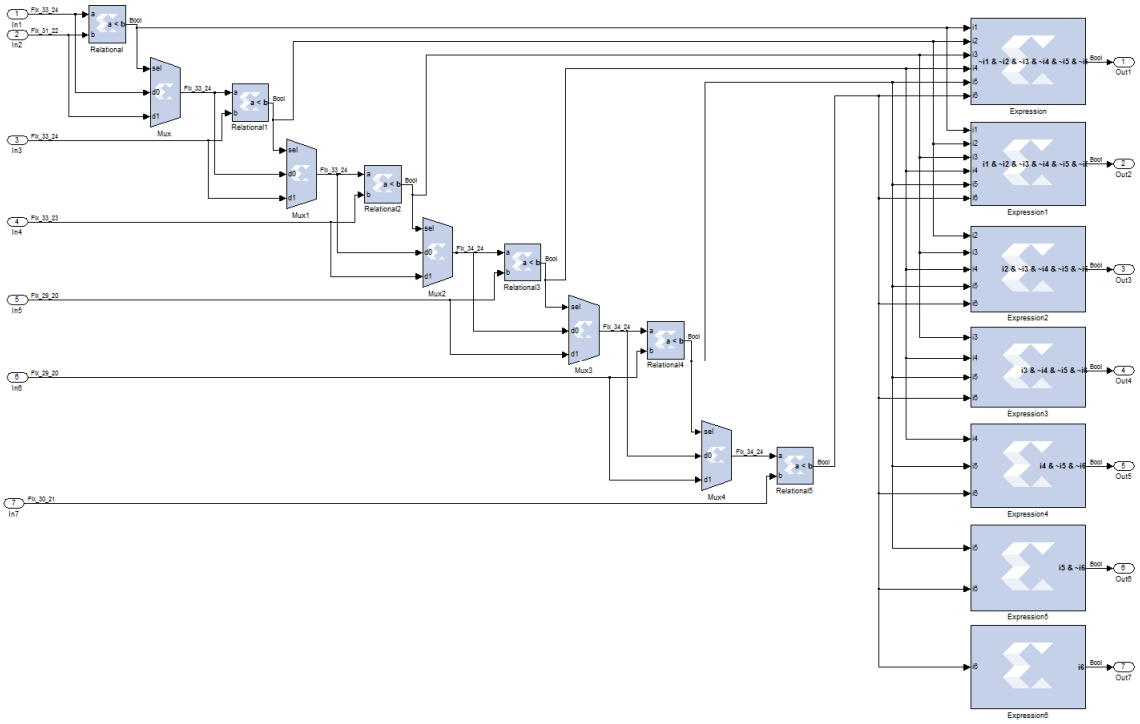


Figura 4.28. Bloque decodificador de las neuronas de salida para ECG.

El bloque *DECODER* se agrupa en un subsistema de *Simulink* como muestra la figura 4.29. Los siete bits de salida del decodificador se registran con bloques *Delay* por los mismos motivos que se registró las entradas. Finalmente, las salidas de los bloques *Delay* se conectan a los pines de la salida de la FPGA mediante bloques *Gateway Out*, como indica la figura 4.29. El sistema final diseñado tiene el aspecto de la figura 4.30.

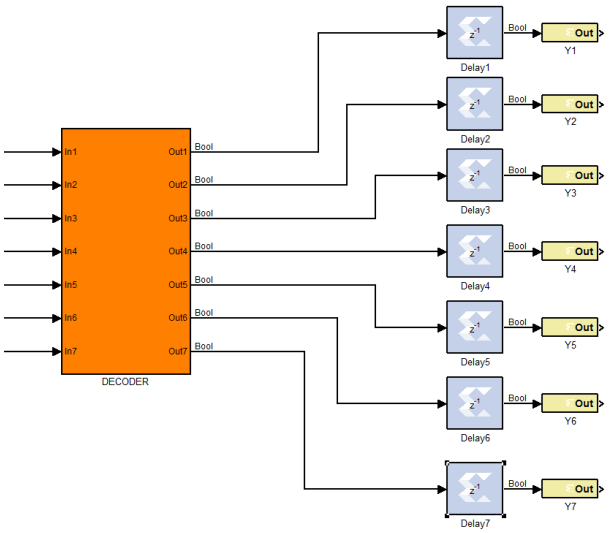


Figura 4.29. Conexión de las salidas del decodificador a los registros de salida y a los pines de salida de la FPGA para ECG.

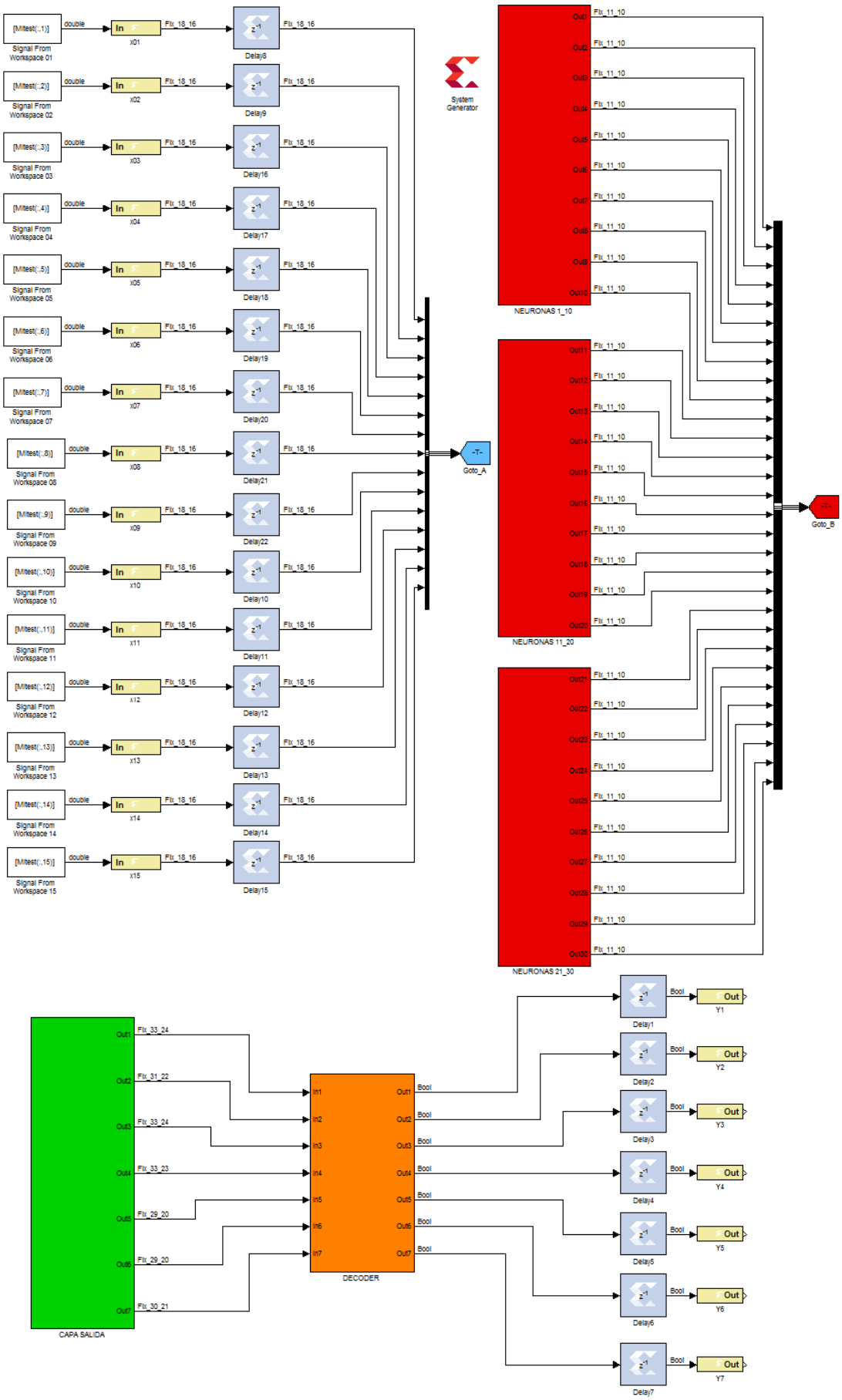


Figura 4.30. Diseño en Simulink del sistema final para ECG.

Una vez descrito el modelo de *Simulink*, se puede realizar la simulación funcional. Las quince entradas se muestran en la figura 4.31 y las salidas en la figura 4.32. Las salidas se retrasan dos periodos de reloj respecto a las entradas por la existencia de los elementos de retardo en las entradas y salidas. Se introdujeron los 400 elementos de cada clase usados en el testeo.

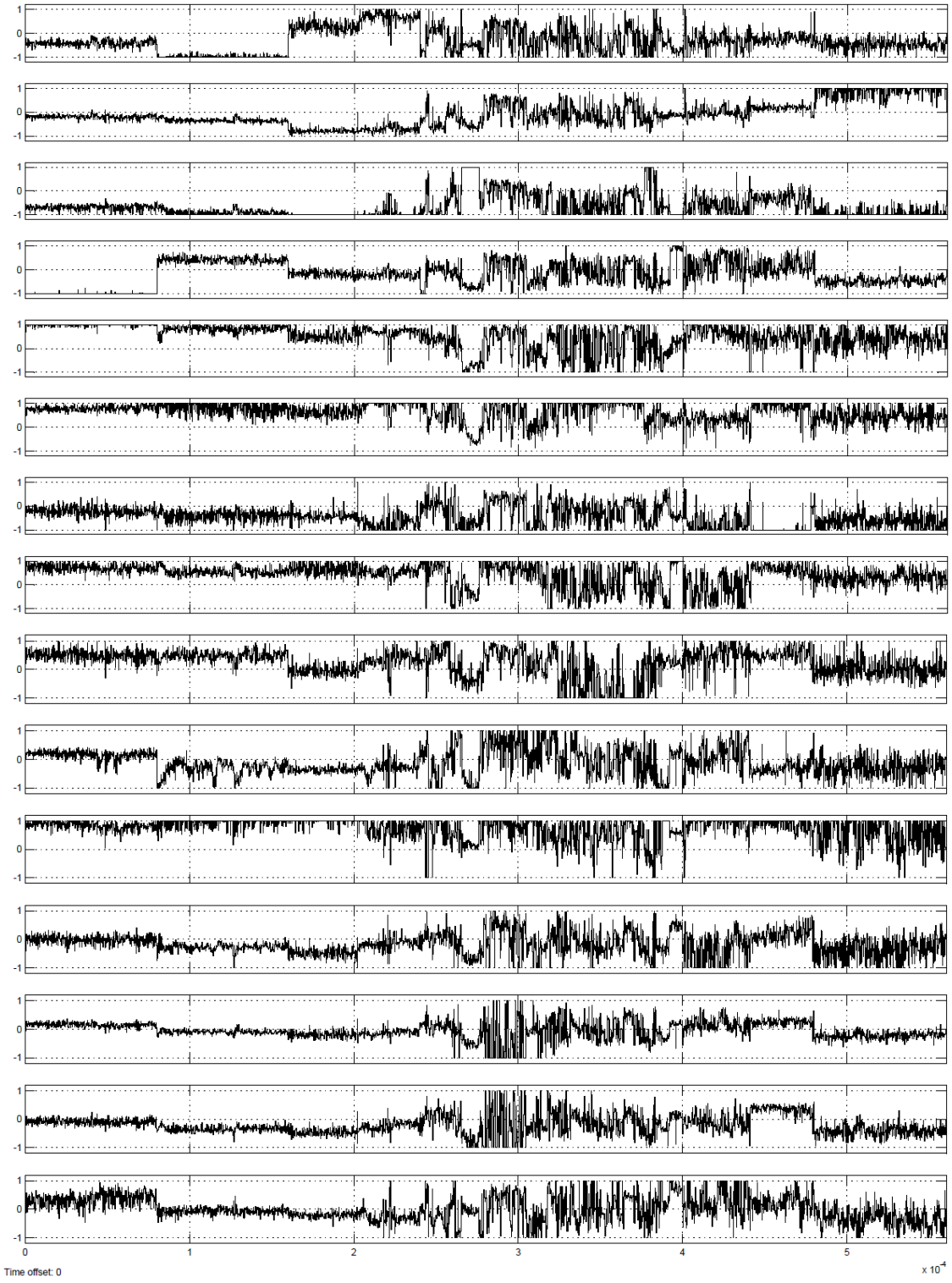


Figura 4.31. Señales de entrada en la NN para ECG.

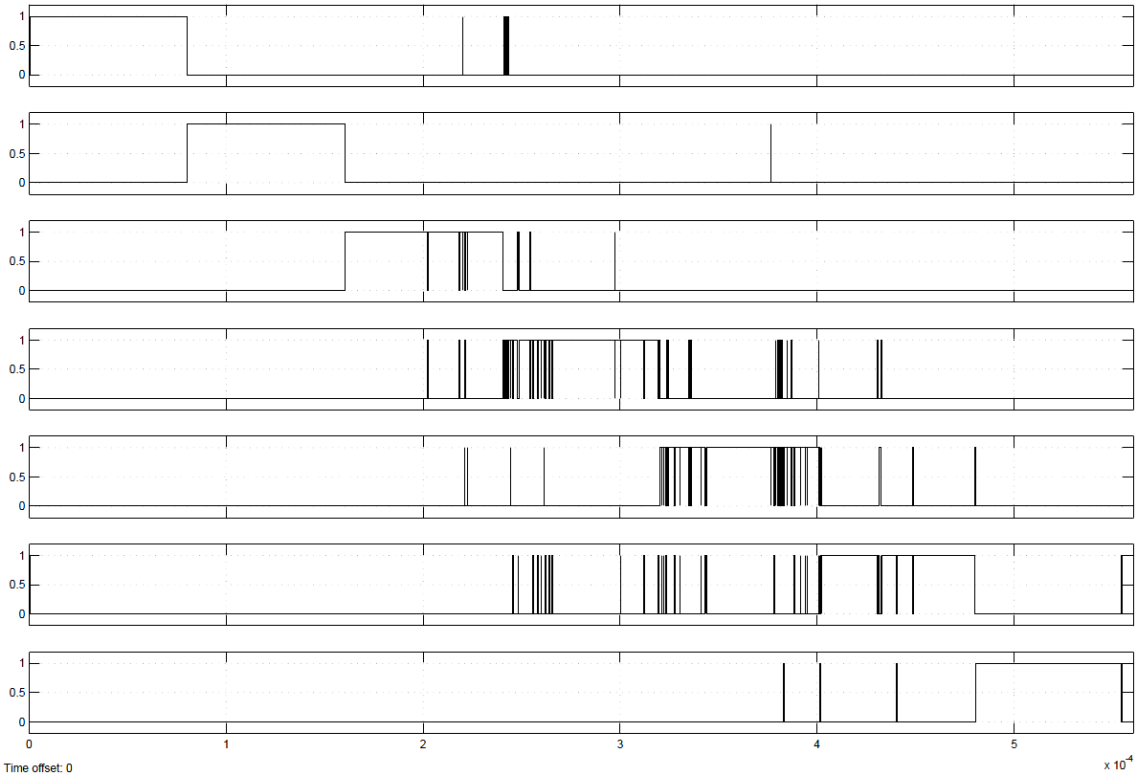


Figura 4.32. Señales de salida en la NN para ECG.

Para el caso que nos ocupa, 4,2% de error en la representación de los coeficientes y 64 palabras en las ROM, se obtuvo la matriz de confusión de la tabla 4.16. Esta se puede comparar con la tabla 3.2, obtenida en el modelo en punto flotante, casualmente mejora la tasa del algoritmo de punto flotante.

Tabla 4.16. Matriz de confusión obtenida en el modelo en punto fijo para 4,2% de error y 64 palabras en las ROM para ECG.

	N	**L**	**R**	**A**	**V**	**F**	**/**	**Éxito(%)**
N	400	0	0	0	0	0	0	100
L	0	400	0	0	0	0	0	100
R	1	0	394	3	2	0	0	98,50
A	6	0	6	372	2	14	0	93
V	0	1	0	10	374	13	2	93,50
F	0	0	0	3	11	384	2	96
/	0	0	0	0	0	1	399	99,75
Total								**97,25**

Posteriormente se tanteó la funcionalidad en función del error máximo en la representación y el número de muestras almacenadas en las memorias ROM. Se

obtuvo la funcionalidad indicada en la tabla 4.17. Se observan dos soluciones que deben ser comparadas: 4,2% de error y 64 palabras en las ROM, y 4,7% de error y 512 palabras en las ROM. Se observó en las simulaciones que con un 4,8% de error, no se alcanza la funcionalidad por mucho que se aumente el número de palabras en las ROM. En el mismo sentido, con 32 palabras en las ROM, tampoco se consigue la funcionalidad por mucho que se disminuya el error.

Tabla 4.17. Funcionalidad del sistema frente al error en la representación y el número de palabras en la ROM para ECG.

		Error en la representación (%)								
		4,00	***4,10***	***4,20***	***4,30***	***4,40***	***4,50***	***4,60***	***4,70***	***4,80***
Número de palabras en las ROM	***65.536***	SI	SI	SI	SI	SI	SI	SI	SI	NO
	32.768	SI	SI	SI	SI	SI	SI	SI	SI	NO
	16.384	SI	SI	SI	SI	SI	SI	SI	SI	NO
	8.192	SI	SI	SI	SI	SI	SI	SI	SI	NO
	4.096	SI	SI	SI	SI	SI	SI	SI	SI	NO
	2.048	SI	SI	SI	SI	SI	SI	SI	SI	NO
	1.024	SI	SI	SI	SI	SI	SI	SI	SI	NO
	512	SI	SI	SI	SI	SI	SI	SI	SI	NO
	256	SI	SI	SI	NO	NO	NO	NO	NO	NO
	128	SI	SI	SI	NO	NO	NO	NO	NO	NO
	64	SI	SI	SI	NO	NO	NO	NO	NO	NO
	32	NO	NO	NO	NO	NO	NO	NO	NO	NO

4.3.2.2 Implementación con Integrated System Environment

Claramente interesa comparar los casos de 4,2% de error y 64 palabras en la ROM y 4,7% de error y 512 palabras en la ROM. Los dos casos se compilaron con la intención de comparar el área ocupada, la potencia y la velocidad.

Resultado para 4,2% de error y 64 palabras en la ROM. Lenguaje VHDL.

La compilación se realizó con *System Generator* para obtener la descripción en VHDL. En el proyecto obtenido para *Integrated System Environment* se realizó la simulación para la fase final de la implementación en la FPGA (*Post Place and Route Simulation*). Como simulador se usó ISim (ISE Simulator), integrado en la herramienta. Se obtuvieron las formas de onda de la figura 4.33. En esta figura las señales no se ven con claridad dado que hay 2.802 ciclos de reloj, pero puede comprobarse las formas de

onda de las salidas y su parecido con las de la figura 4.32. En la figura 4.34 se observa el detalle de los primeros 2.500 ns de la simulación. La compilación se realizó para un dispositivo Virtex-4, tipo xc4vlx160, grado de velocidad -12, tipo de encapsulado ff y 1148 pines; de forma compacta se denota como Virtex-4 xc4vlx160-12ff1148.

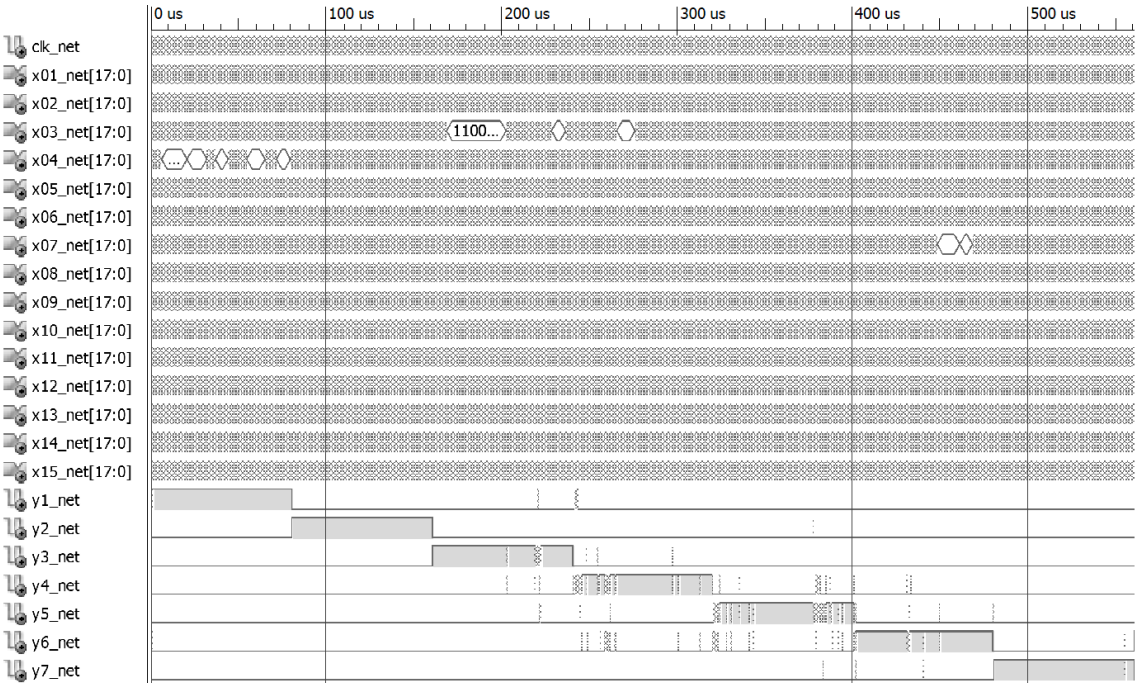


Figura 4.33. Simulación para la fase final de la implementación en la FPGA para 4,2% de error y 64 palabras en la ROM para ECG.

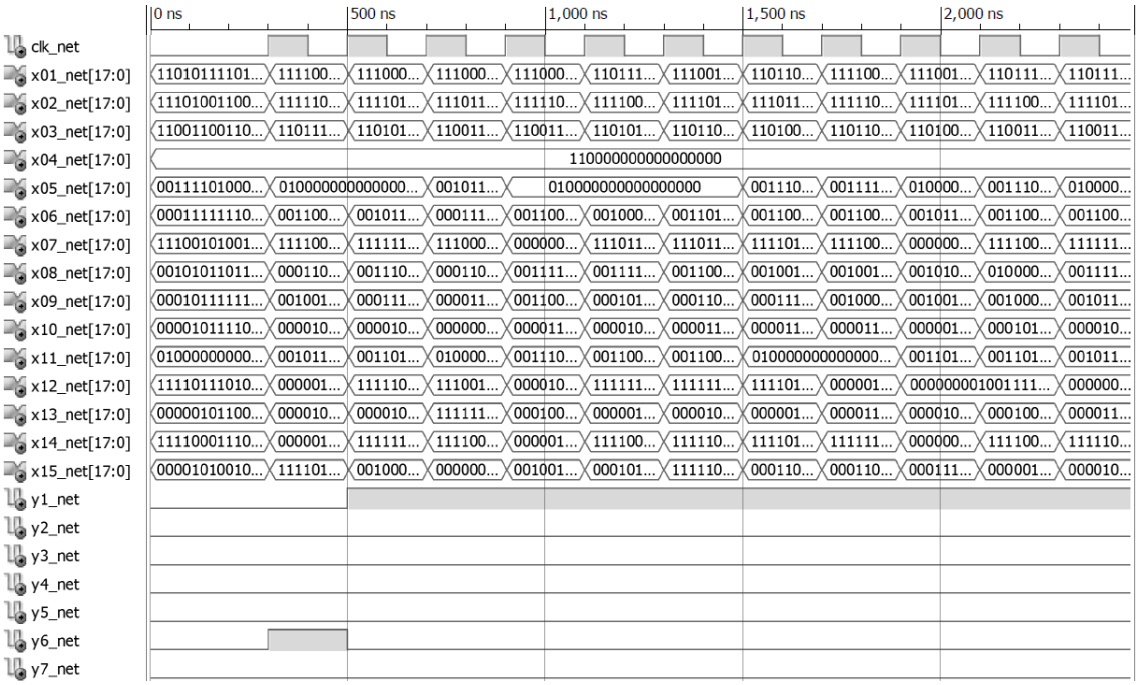


Figura 4.34. Detalle de los primeros 2.500 ns de la simulación en la FPGA para 4,2% de error y 64 palabras en la ROM para ECG.

Los recursos hardware necesarios se resumen en la tabla 4.18. Debe resaltarse el número de SLICES necesarios, porque los LUT están incluidos dentro de los SLICES. El segundo parámetro importante es el número de pines de entrada-salida necesarios.

Tabla 4.18. Recursos hardware necesarios para 4,2% de error y 64 palabras en la ROM, aplicado a ECG y compilado en VHDL.

Entidad	Usados	Disponibles	Tasa de uso
SLICES	29.401	67.584	43%
LUT de 4 entradas	55.013	135.168	40%
Bloques de E/S	278	768	36%

Para la frecuencia de reloj de 5 MHz se obtienen las potencias de la tabla 4.19; para esto se usó la utilidad *XPower Analyzer* del ISE. La frecuencia máxima de funcionamiento es de 12,086 MHz; este dato está disponible en el informe posterior a la elección y conexión de los circuitos de la FPGA (*Place and Route Report*). En la tabla 4.19 se muestran las potencias para 12,1 MHz, que es el valor redondeado que permite introducir la herramienta.

Tabla 4.19. Potencias para 4,2% de error y 64 palabras en la ROM, aplicado a ECG y compilado en VHDL.

		Potencia (mW)		
		Estática	**Dinámica**	**Total**
Frecuencia (MHz)	**5,0**	2.692,49	194,16	2.886,64
	12,1	2.707,53	368,22	3.075,76

Resultado para 4,2% de error y 64 palabras en la ROM. Lenguaje Verilog.

En este apartado se muestra el caso de 4,2% de error y 64 palabras en la ROM, se compila con *System Generator* para generar el proyecto en Verilog para el mismo dispositivo. La simulación presenta el mismo aspecto que las figuras 4.33 y 4.34. Los requisitos de área se muestran en la tabla 4.20.

Tabla 4.20. Recursos hardware necesarios para 4,2% de error y 64 palabras en la ROM, aplicado a ECG y compilado en Verilog.

Entidad	**Usados**	**Disponibles**	**Tasa de uso**
SLICES	29.429	67.584	43%
LUT de 4 entradas	54.985	135.168	40%
Bloques de E/S	278	768	36%

Los resultados de la potencia para la frecuencia de 5 MHz se tienen en la tabla 4.21. La frecuencia máxima de funcionamiento es de 12,057 MHz. En la tabla 4.21 se muestran las potencias para 12,1 MHz, que es el valor redondeado que permite introducir la herramienta

Tabla 4.21. Potencias para 4,2% de error y 64 palabras en la ROM, aplicado a ECG y compilado en Verilog.

		Potencia (mW)		
		Estática	**Dinámica**	**Total**
Frecuencia (MHz)	**5,0**	2.690,44	170,22	2.860,67
	12,1	2.705,41	343,81	3.049,22

Resultado para 4,7% de error y 512 palabras en la ROM. Lenguaje VHDL.

A continuación se analiza el caso de 4,7% de error y 512 palabras en la ROM, compilado en *System Generator* para generar el proyecto en VHDL para el mismo

dispositivo. La simulación en este caso presenta el mismo aspecto que las de las figuras 4.33 y 4.34. Los requisitos de área se muestran en la tabla 4.22.

Tabla 4.22. Recursos hardware necesarios para 4,7% de error y 512 palabras en la ROM ,aplicado a ECG y compilado en VHDL.

Entidad	Usados	Disponibles	Tasa de uso
SLICES	37.840	67.584	55%
LUT de 4 entradas	70.088	135.168	51%
Bloques de E/S	278	768	36%

Los resultados de la potencia para la frecuencia de 5 MHz se tienen en la tabla 4.23. La frecuencia máxima de funcionamiento es de 10,852 MHz. En la tabla 4.23 se muestran las potencias para 10,9 MHz, que es el valor redondeado que permite introducir la herramienta.

Tabla 4.23. Potencias para 4,7% de error y 512 palabras en la ROM, aplicado a ECG y compilado en VHDL.

		Potencia (mW)		
		Estática	Dinámica	Total
Frecuencia (MHz)	**5,0**	2.696,47	240,56	2.937,03
	10,9	2.713,82	439,97	3.153,78

Resultado para 4,7% de error y 512 palabras en la ROM. Lenguaje Verilog.

Seguidamente se muestra el caso de 4,7% de error y 512 palabras en la ROM, compilado en *System Generator* para generar el proyecto en Verilog, para el mismo dispositivo. La simulación en este caso presenta el mismo aspecto que las de las figuras 4.33 y 4.34. Los requisitos de área se muestran en la tabla 4.24.

Tabla 4.24. Recursos hardware necesarios para 4,7% de error y 512 palabras en la ROM, aplicado a ECG y compilado en Verilog.

Entidad	Usados	Disponibles	Tasa de uso
SLICES	37.393	67.584	55%
LUT de 4 entradas	69.865	135.168	51%
Bloques de E/S	278	768	36%

Los resultados de la potencia para la frecuencia de 5 MHz se tienen en la tabla 4.25. La frecuencia máxima de funcionamiento es de 11,560 MHz. En la tabla 4.25 se muestran las potencias para 11,6 MHz, que es el valor redondeado que permite introducir la herramienta.

Tabla 4.25. Potencias para 4,7% de error y 512 palabras en la ROM, aplicado a ECG y compilado en Verilog.

		Potencia (mW)		
		Estática	**Dinámica**	**Total**
Frecuencia (MHz)	**5,0**	2.695,34	227,36	2.922,70
	11,6	2.714,23	444,65	3158,88

Resumen de los casos estudiados.

Como resumen debe observarse la tabla 4.26, donde se muestran las prestaciones en área, velocidad y potencia. En cuanto al área, no se muestra el número de pines, que casualmente coincide en los dos casos. Las mejores características de área y velocidad se consiguen en VHDL con 4,2% de error y 64 palabras en la ROM. La mejor prestación en potencia se consigue para el mismo caso, pero usando Verilog. Se puede tomar como óptimo el caso de VHDL, porque es el mejor en dos de los tres parámetros. Se podría tomar como óptimo el caso de Verilog si hubieran restricciones en el consumo de potencia.

Tabla 4.26. Resumen de los casos estudiados para ECG.

	VHDL		**Verilog**	
4,2% de error 64 palabras en la ROM	*Área*	29.401 SLICES	*Área*	29.429 SLICES
	Frecuencia máxima	12,086 MHz	*Frecuencia máxima*	12,057 MHz
	Potencia (5 MHz)	2.886,64 mW	*Potencia (5 MHz)*	2.860,67 mW
4,7% de error 512 palabras en la ROM	*Área*	37.840 SLICES	*Área*	37.393 SLICES
	Frecuencia máxima	10,852 MHz	*Frecuencia máxima*	11,560 MHz
	Potencia (5 MHz)	2.937,03 mW	*Potencia (5 MHz)*	2.922,70 mW

Llegados a este punto se podría comparar el tiempo de respuesta de la NN en punto flotante ejecutada con el *Neural Network Toolbox* de Matlab y el sistema en punto fijo sobre la FPGA elegida.

En el ordenador personal en punto flotante una clasificación tarda 22.285 ns y en la FPGA elegida unos 82,7 ns; del orden de 270 veces más rápida. Estos resultados son meramente orientativos, claramente depende del ordenador personal y de la FPGA que se use; sus características se muestran en la tabla 4.27.

Tabla 4.27. Comparación en velocidad entre un ordenador personal y la FPGA usada para ECG.

	Tipo	***Tiempo para una clasificación***
Ordenador personal	Windows 7 Home Premium 64 bits Intel(R) Core(TM) i7-3770 CPU @ 3.40 GHz RAM: 8 Gbytes Matlab R2010b 64 bits	22.285 ns
FPGA	Xilinx Virtex-4 xc4vlx160-12ff1148	82,7 ns

Conviene resaltar que la simulación en *Simulink* del sistema diseñado con *System Generator* tarda unos 45 segundos; mientras la simulación en *Integrated System Environment* tarda una hora y veinte minutos aproximadamente. Estos tiempos son para el ordenador de la tabla 4.27.

4.4 La predicción de temperatura

En el capítulo anterior se describió la base de datos de temperatura que se ha usado. El objetivo es realizar un predictor de temperatura mediante una NN, se hará uso de la metodología y de las herramienta de diseño explicadas. En la base de datos solo se disponía de dos años completos, 2008 y 2009; por esto se usó el año 2008 para el entrenamiento y 2009 para el testeo.

4.4.1 Modelado en punto flotante

El sistema predictor consta de una NN y una etapa previa de líneas de retardo, que almacena las últimas muestras recibidas en la entrada del predictor, como indica la figura 4.35. En dicha figura *x(t)* es la señal de entrada e *y(t)* la señal de salida. Debe observarse que la salida *y(t)* es función de *n* muestras de la señal de entrada, como indica la expresión 4.1. El objetivo es entrenar la NN para que *y(t)* sea una estimación

de *x(t+1)*. En adelante al sistema de la figura 4.35 se le denotará como red neuronal con línea de retardo (TDNN, *Time Delay Neural Network*).

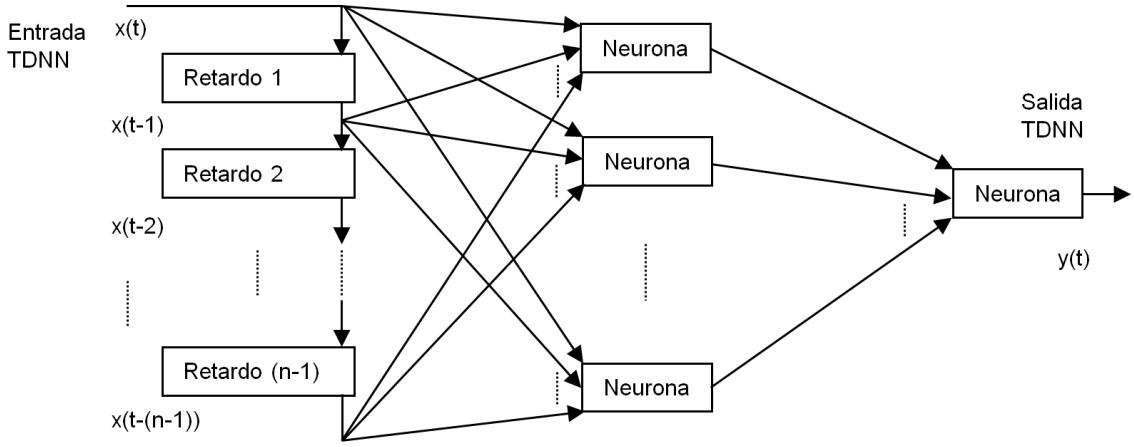


Figura 4.35. Red neuronal con línea de retardo.

$$y(t) = f(x(t), x(t-1), x(t-2), \ldots, x(t-(n-1))) \tag{4.1}$$

En la estructura de una TDNN debe resaltarse el transitorio en el inicio; es decir, la salida no es válida hasta que en la entrada llegan las *n* primeras muestras de *x(t)*. El modelado de la TDNN se puede realizar usando el *Neural Network Toolbox* de Matlab en la forma habitual para clasificación y reconocimiento de patrones; creando una matriz para las *n* entradas de la NN interna, y usando como objetivo en el entrenamiento la señal *x(t)*. La segunda dimensión de la matriz de las entradas dependerá del número de muestras en la entrada, y de valor *n*. La creación de esta matriz puede resultar tediosa; además el número de muestras válidas de *y(t)* no coincide con la longitud de *x(t)*. En las primeras versiones de la predicción de la temperatura se operó de esta forma [Pérez et al., 2011d; Vásquez et al., 2013], en realidad se entrenaba la NN interna de la TDNN; es decir, la figura 4.35 sin líneas de retardo.

Por los motivos anteriores, en los últimos entrenamientos se recurrió al *Neural Network Time Series Tool*, esta utilidad de Matlab maneja series temporales para NN [ntstool, 2015]. Las ventajas de esta utilidad es que añade la serie de retardos en la

entrada, maneja el inicio del transitorio de entrada y prepara las matrices internas para la NN. Esto es posible tanto para el entrenamiento como para el testeo. El manejo de esta utilidad se realiza mediante ventanas de diálogo, finalmente permite generar el código. Este código; por un lado, evita volver a ejecutar las ventanas de diálogo; y por otro, permite cambiar algunos parámetros del proceso, que no se pueden ajustar en la ventanas. Entre estos parámetros cabe destacar, entre otros, el algoritmo de entrenamiento, y las funciones de transferencias usadas. En la figura 4.36 se observa una ventana típica obtenida en el entrenamiento, donde cabe resaltar la inserción de líneas de retardo en la entrada.

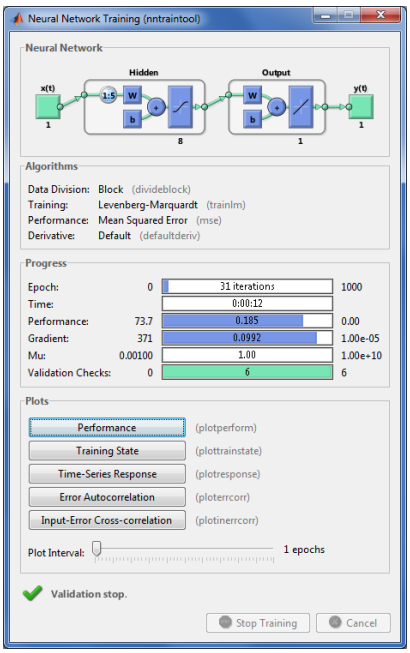


Figura 4.36. Ventana obtenida en el entrenamiento con el Neural Network Time Series Tool.

La TDNN se entrenó con los datos del año 2008. Tras el periodo de entrenamiento se obtuvo una TDNN con 4 elementos de retardo, 8 neuronas en la capa intermedia con funciones *tansig* y una neurona en la capa de salida con la función identidad (*purelin*). Los datos no sufrieron ningún tipo de procesado ni normalización.

El testeo de la TDNN se realizó con la temperatura obtenida en el año 2009. Las simulaciones dieron un error medio de -0,032 ^{0}C, un valor medio del valor absoluto del error de 0,317 ^{0}C y un valor medio del error al cuadrado de 0,210 ^{0}C. Para comprobar

la funcionalidad del sistema se tendrá en cuenta el valor medio del error en valor absoluto.

4.4.2 Diseño en punto fijo

Obviamente, la implementación física obedece a la expresión 4.2. En esta fórmula *Tm* es el intervalo de muestro de la señal de entrada, que además fijará el reloj del sistema. La TDNN se entrena con el objetivo de que *y(t)* sea *x(t+Tm)*; es decir la salida estimará el valor de la próxima muestra en la entrada.

$$y(t) = f(x(t), x(t - Tm), x(t - 2Tm), \dots, x(t - (n-1)Tm)) \qquad (4.2)$$

4.4.2.1 Diseño con System Generator

El sistema diseñado con *System Generator* obedece a la figura 4.37, donde cabe destacar la línea de retardos en la entrada. El sistema es diseñado en la misma forma que los dos anteriores; con funciones *tansig* en la capa intermedia y función identidad en la neurona de salida. En el estudio se tendrá en cuenta que cumple la funcionalidad si el valor medio del error absoluto es menor que 0,32 ^{0}C; si este error es mayor que ese valor se considerará que no cumple la funcionalidad. En la figura 4.37 se muestran los tipos de datos para un error de 0,52% y 1024 palabras en la ROM.

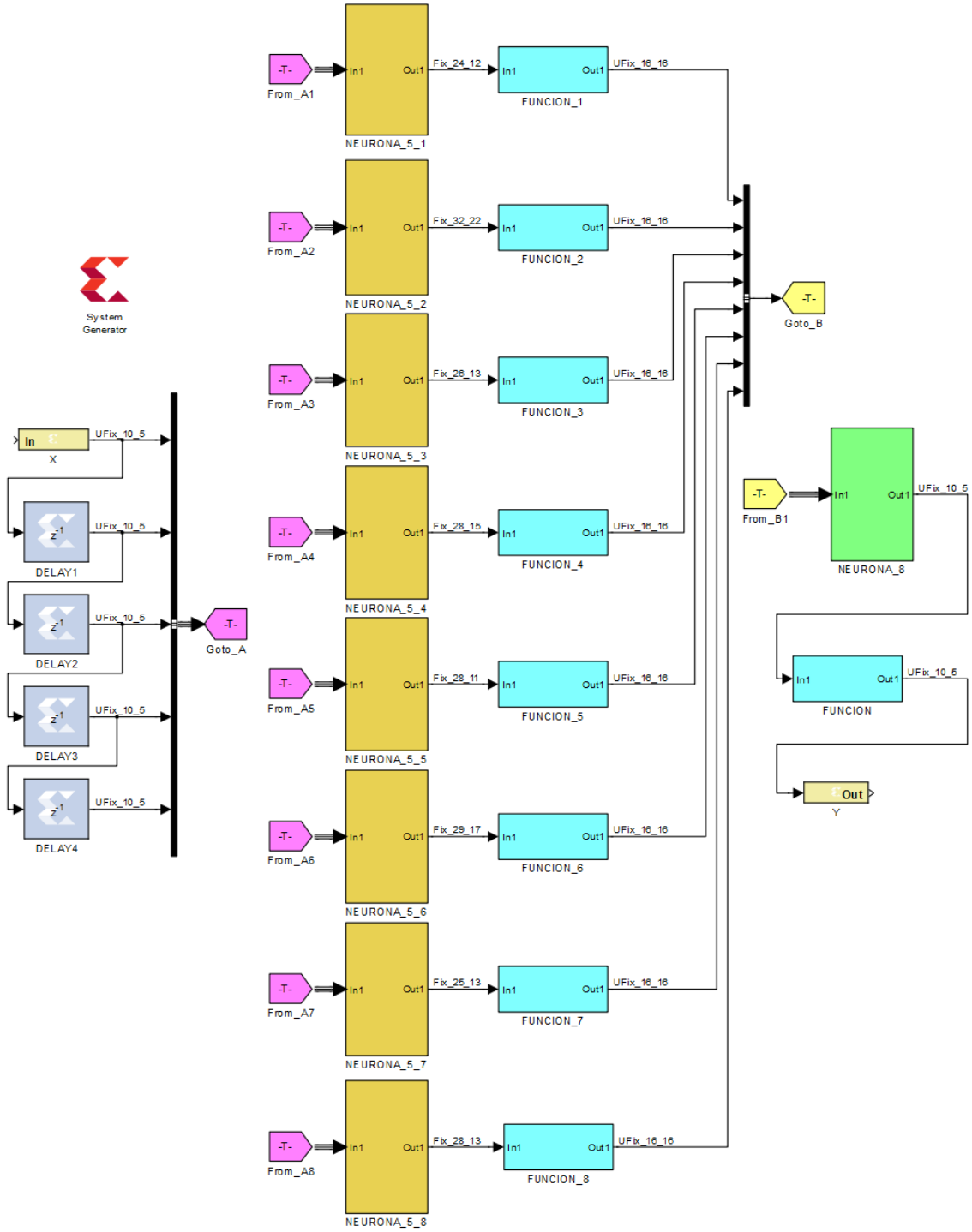


Figura 4.37. Implementación de la TDNN para predicción de la temperatura en la FPGA.

Cabe destacar el ajuste realizado en la neurona de la capa de salida, figura 4.38. Inicialmente la resolución de los multiplicadores y sumadores se mantuvo completa, obteniéndose el número de bits que se muestra en la figura 4.38. Para el caso que nos ocupa, se obtiene en la salida una señal *Fix_16_5*; es decir, con signo, cinco bits para la parte fraccionaria y 16 bits en total. Se puede realizar un ajuste en la señal de salida

que es definida por el diseñador. El primer ajuste es definir la salida sin signo, esto es posible porque la temperatura en la base de datos siempre es positiva. Por otro lado se fija el formato de la salida como en la entrada. Si el formato de la salida tuviese más bits que en la entrada, la cuantificación sería mejor en teoría, pero esto no mejora el error medido. En resumen, el formato de la salida es igual que el formato de la entrada. A partir de esta decisión es posible ajustar de forma manual la resolución de los sumadores de esta etapa; desde la salida hacia las entradas, hasta llegar a los multiplicadores por contantes, cuya resolución viene fijada por el error fijado para los coeficientes. Este ajuste se muestra en la figura 4.39, esta disminución del número de bits provoca mejora en las prestaciones físicas del sistema.

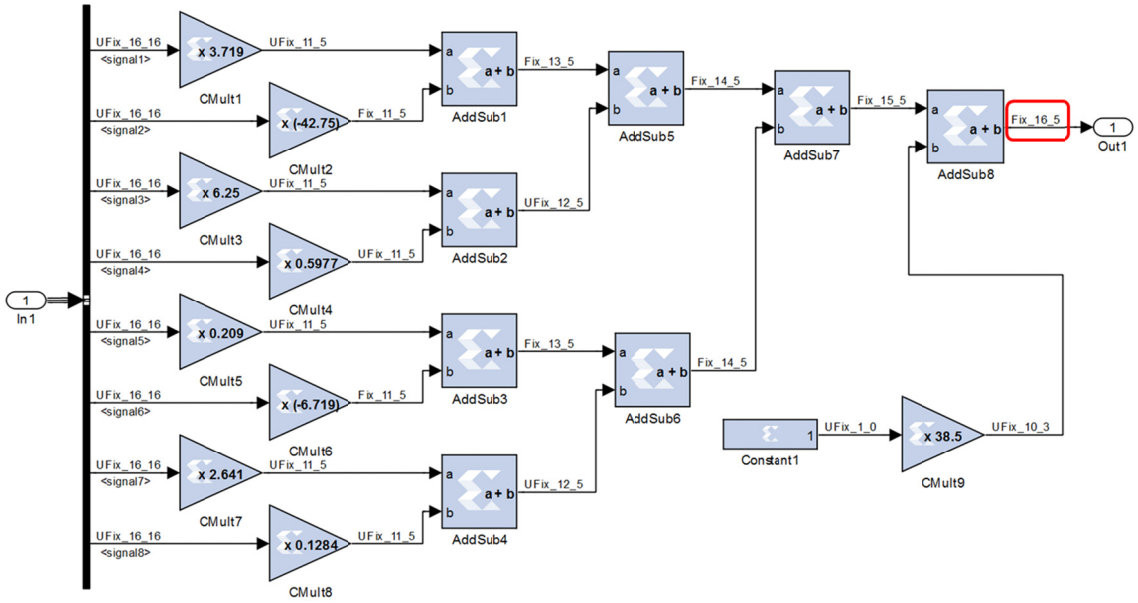


Figura 4.38. Neurona de salida en el predictor de temperatura con resolución completa.

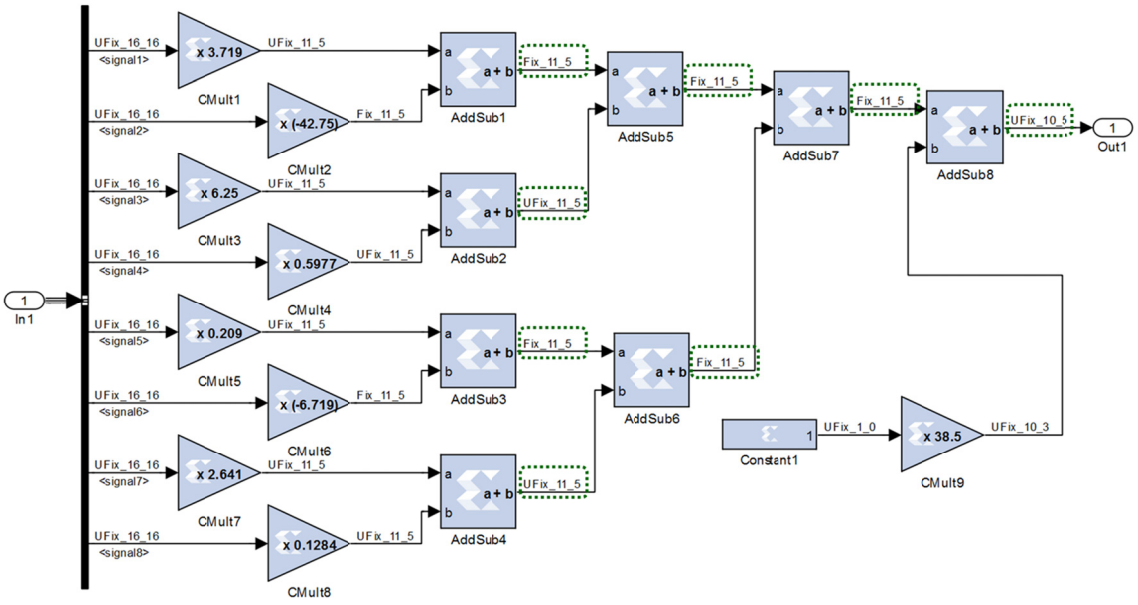


Figura 4.39. Neurona de salida en el predictor de temperatura con resolución ajustada de forma manual.

En la figura 4.40 se observan las formas de ondas obtenidas en *Simulink*, la unidad del eje vertical son grados centígrados y en el eje horizontal se indica el número de muestras. Esta simulación se corresponde con los siete primeros días de la base de datos del año 2009. En color rojo se tiene la señal de entrada, en color azul la señal de salida y en negro la señal de error. Las dos señales de temperatura están convenientemente retardas en *Simulink*, para evaluar el error de manera apropiada. En la figura 4.40 se observa el transitorio en el inicio de la simulación, tanto en la señal de salida como en la señal de error.

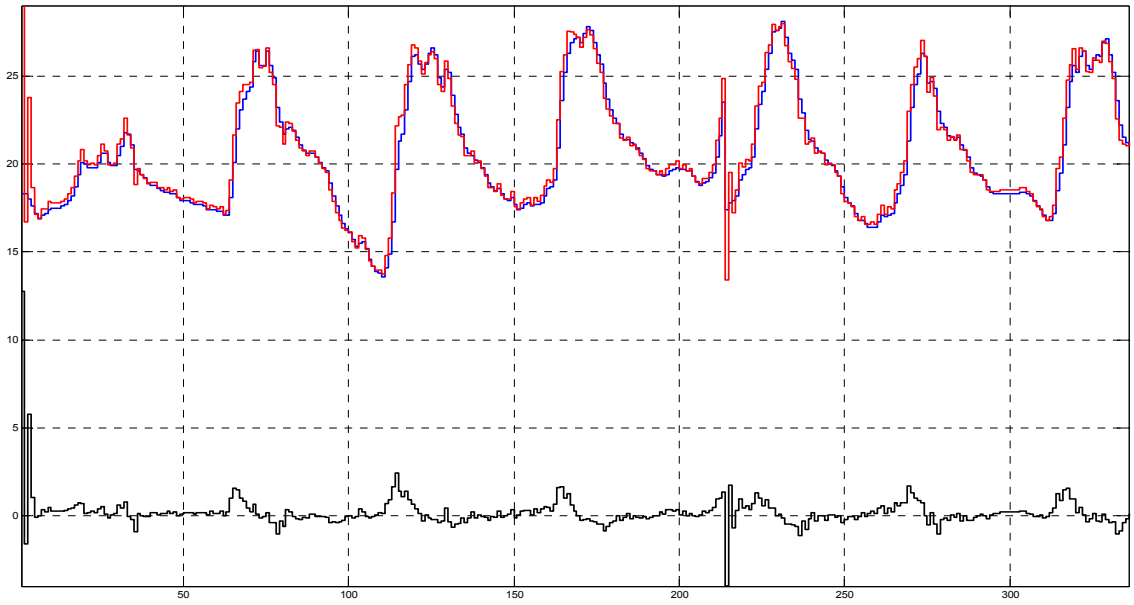


Figura 4.40. Simulación de los siete primeros días de 2009 con System Generator en el predictor de temperatura: muestras en la entrada (azul), estimación en la salida (rojo) y señal de error (negro).

Tras un periodo de tanteo se llega a la tabla 4.28, donde se indica si se cumple la funcionalidad en función del error permitido en la representación; y el número de palabras en las memorias ROM, que implementan las funciones de transferencia. Interesa estudiar los casos de 0,52% de error y 1024 palabras en la ROM; y 0,56% de error y 2048 palabras en la ROM.

Tabla 4.28. Funcionalidad del predictor de temperatura en función del error permitido en la representación y el número de palabras en las memorias ROM.

		Error en la representación (%)												
		0,001	**⋯**	***0,40***	***0,42***	***0,44***	***0,46***	***0,48***	***0,50***	***0,52***	***0,54***	***0,56***	***0,58***	***0,60***
Número de palabras en las ROM	2^{20}	SI	⋯	SI	SI	SI	SI	SI	SI	SI	SI	SI	NO	NO
	⋯	⋯	⋯	⋯	⋯	⋯	⋯	⋯	⋯	⋯	⋯	⋯	⋯	⋯
	2^{14}	SI	⋯	SI	SI	SI	SI	SI	SI	SI	SI	SI	NO	NO
	2^{13}	SI	⋯	SI	SI	SI	SI	SI	SI	SI	SI	SI	NO	NO
	2^{12}	SI	⋯	SI	SI	SI	SI	SI	SI	SI	SI	SI	NO	NO
	2^{11}	SI	⋯	SI	SI	SI	SI	SI	SI	SI	SI	SI	NO	NO
	2^{10}	SI	⋯	SI	SI	SI	SI	SI	SI	SI	NO	NO	NO	NO
	2^{9}	NO	⋯	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO

4.4.2.2 Implementación con Integrated System Environment

Los dos casos anteriores se compilaron en *System Generator*, para VHDL y Verilog, y se obtuvieron cuatro proyectos para el *Integrated System Environment* de Xilinx. El dispositivo elegido fue el Spartan3E xc3s1200e-5ft256, el mismo que el usado para pejibaye.

Resultado para 0,52% de error y 1024 palabras en la ROM. Lenguaje VHDL.

Al intentar simular con un intervalo de muestreo de media hora (1800 segundos), por ser una frecuencia extremadamente baja, el simulador da error. La herramienta indica que la frecuencia mínima es de unos 47 Hz. Esto no es problema, pues el sistema es capaz de funcionar a esa frecuencia, 555 µHz. Para solventar este problema el sistema se simuló con un reloj de 5 MHz. En la figura 4.41 se tienen algunos tramos de la simulación, después del colocado y conexionado de los componentes en la FPGA. En este caso el formato de la entrada y la salida es *UFix_10_5*; es decir, sin signo, diez bits en total y cinco para la parte fraccionaria.

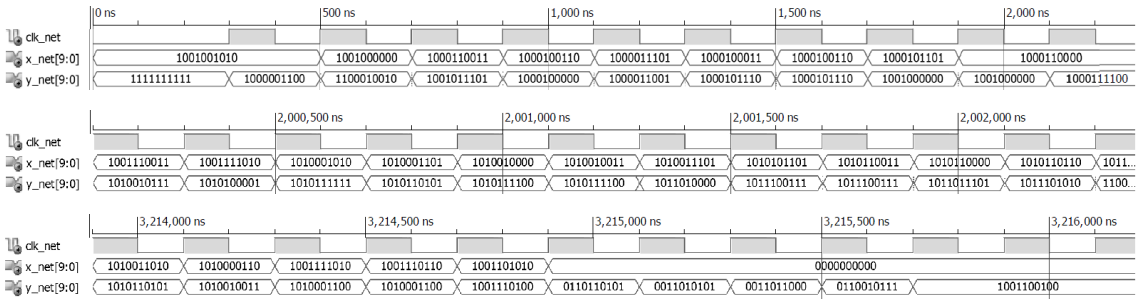


Figura 4.41. Simulación del predictor de temperatura después del colocado y conexionado de los componentes en la FPGA para 0,52% de error y 1024 palabras en la ROM.

La ocupación de área se resume en la tabla 4.29. La frecuencia máxima del diseño es de 275,482 MHz. Las potencias se muestran en la tabla 4.30.

Tabla 4.29. Recursos hardware necesarios para 0,52% de error y 1024 palabras en la ROM, predicción de temperatura y compilado en VHDL.

Entidad	Usados	Disponibles	Tasa de uso
SLICES	6.228	8.672	71%
LUT de 4 entradas	10.674	17.344	61%
Bloques de E/S	21	190	11%

Tabla 4.30. Potencias para 0,52% de error y 1024 palabras en la ROM, , predicción de temperatura y compilado en VHDL.

		Potencia (mW)		
		Estática	Dinámica	Total
Frecuencia (MHz)	**5,0**	159,35	15,10	174,45
	275,48	185,27	831,87	1017,14

Resultado para 0,52% de error y 1024 palabras en la ROM. Lenguaje Verilog.

Las simulaciones en este caso tienen el mismo aspecto que la figura 4.41. En la tabla 4.31 se tiene los requisitos de área, la frecuencia máxima de funcionamiento es de 301,023 MHz; finalmente, en la tabla 4.32 se muestra la estimación de potencia.

Tabla 4.31. Recursos hardware necesarios para 0,52% de error y 1024 palabras en la ROM, predicción de temperatura y compilado en Verilog.

Entidad	Usados	Disponibles	Tasa de uso
SLICES	6.239	8.672	71%
LUT de 4 entradas	11.728	17.344	67%
Bloques de E/S	21	190	11%

Tabla 4.32. Potencias para 0,52% de error y 1024 palabras en la ROM, predicción de temperatura y compilado en Verilog.

		Potencia (mW)		
		Estática	Dinámica	Total
Frecuencia (MHz)	**5,0**	159,35	15,06	174,41
	301,02	188,14	906,81	1094,95

Resultado para 5,6% de error y 2048 palabras en la ROM. Lenguaje VHDL.

En la figura 4.42 se tienen algunos tramos de la simulación para una frecuencia de reloj de 5 MHz, después del colocado y conexionado de los componentes. En este caso el formato de la entrada y la salida es *UFix_9_4*; es decir, sin signo, nueve bits en total y cuatro para la parte fraccionaria.

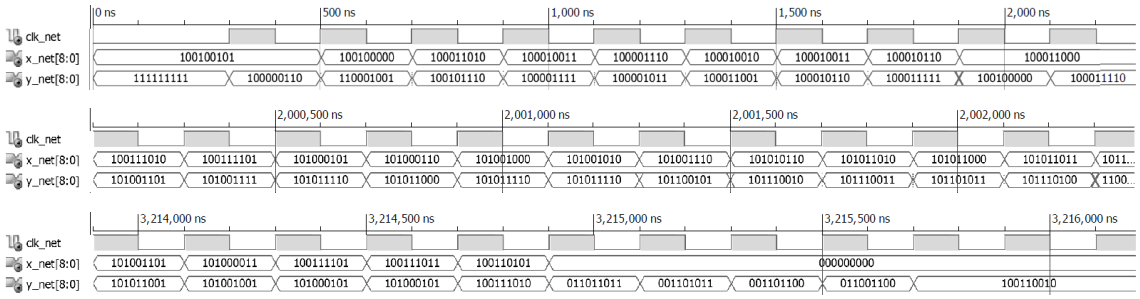


Figura 4.42. Simulación del predictor de temperatura después del colocado y conexionado de los componentes en la FPGA para 0,56% de error y 2048 palabras en la ROM.

En la tabla 4.33 se tiene los requisitos de área. La frecuencia máxima de funcionamiento vale 382,409 MHz. En la tabla 4.34 se muestra la estimación de potencia.

Tabla 4.33. Recursos hardware necesarios para 0,56% de error y 2048 palabras en la ROM, predicción de temperatura y compilado en VHDL.

Entidad	Usados	Disponibles	Tasa de uso
SLICES	7.672	8.672	88%
LUT de 4 entradas	14.700	17.344	84%
Bloques de E/S	19	190	10%

Tabla 4.34. Potencias para 0,56% de error y 2048 palabras en la ROM, predicción de temperatura y compilado en VHDL.

		Potencia (mW)		
		Estática	Dinámica	Total
Frecuencia (MHz)	**5,0**	159,45	18,92	178,37
	382,41	211,81	1447,03	1658,84

Resultado para 5,6% de error y 2048 palabras en la ROM. Lenguaje Verilog.

Las simulaciones en este caso tienen el mismo aspecto que la figura 4.42. En la tabla 4.35 se tiene los requisitos de área, la frecuencia máxima estimada es de 358,038 MHz; por último, en la tabla 4.36 se muestra la estimación de potencia.

Tabla 4.35. Recursos hardware necesarios para 0,56% de error y 2048 palabras en la ROM, predicción de temperatura y compilado en Verilog.

Entidad	Usados	Disponibles	Tasa de uso
SLICES	7.621	8.672	87%
LUT de 4 entradas	14.698	17.344	84%
Bloques de E/S	19	190	10%

Tabla 4.36. Potencias para 0,56% de error y 2048 palabras en la ROM, predicción de temperatura y compilado en Verilog.

		Potencia (mW)		
		Estática	**Dinámica**	**Total**
Frecuencia (MHz)	**5,0**	159,44	18,42	177,86
	358,04	205,68	1.319,05	1.524,74

Resumen de los casos estudiados.

Como resumen debe observarse la tabla 4.37, donde se muestran las prestaciones en área, velocidad y potencia. En la tabla 4.37 están sombreados los mejores casos de SLICES necesarios, potencia y velocidad. Si existe restricción en el número de pines, este parámetro debe tenerse en cuenta en la elección; esto dependerá del tipo de diseño, por ejemplo convertidor analógico-digital de entrada y formato del dispositivo de destino en la salida.

Tabla 4.37. Resumen de los casos estudiados para el predictor de temperatura.

	VHDL		**Verilog**	
0,52% de error 1024 palabras en la ROM	*Área*	6.228 SLICES	*Área*	6.239 SLICES
	Número de pines	21	*Número de pines*	21
	Frecuencia máxima	275,482 MHz	*Frecuencia máxima*	301,023 MHz
	Potencia (5 MHz)	174,45 mW	*Potencia (5 MHz)*	174,41 mW
0,56% de error 2048 palabras en la ROM	*Área*	7.672 SLICES	*Área*	7.621 SLICES
	Número de pines	19	*Número de pines*	19
	Frecuencia máxima	382,409 MHz	*Frecuencia máxima*	358,038 MHz
	Potencia (5 MHz)	178,37 mW	*Potencia (5 MHz)*	177,86 mW

Llegados a este punto, se podría comparar el tiempo de respuesta de la NN en punto flotante ejecutada con el *Neural Network Toolbox* de Matlab y el sistema en punto fijo sobre la FPGA elegida. En el ordenador personal en punto flotante una clasificación tarda 83,29 μs y en la FPGA elegida unos 2,62 ns; del orden de 32.000 veces más rápida. Estos resultados son meramente orientativos, claramente depende del ordenador personal y de la FPGA que se use; sus características se muestran en la tabla 4.38.

Tabla 4.38. Comparación en velocidad entre un ordenador personal y la FPGA usada en la predicción de temperatura.

	Tipo	***Tiempo para una clasificación***
Ordenador personal	Windows 7 Home Premium 64 bits Intel(R) Core(TM) i7-3770 CPU @ 3.40 GHz RAM: 8 Gbytes Matlab R2010b 64 bits	83,29 µs
FPGA	Spartan3E xc3s1200e-5ft256	2,62 ns

Conviene resaltar que la simulación en *Simulink* del sistema diseñado con *System Generator* tarda unos 3 segundos; mientras la simulación en *Integrated System Environment* tarda unos 10 minutos. Estos tiempos son para el ordenador de la tabla 4.38.

4.5 El ecualizador para señal binaria

En el último escenario se pretende examinar las prestaciones de una red neuronal para ser usada como ecualizador de señal binaria [Pérez et al., 2013]. La señal es unipolar sin retorno a cero y es sometida a AWGN.

4.5.1 Modelado en punto flotante

Para este escenario, dado que el proceso de entrenamiento fue realizado en su totalidad por el autor de esta tesis, y dada su relativa complejidad, se describe en el anexo de este documento.

4.5.2 Diseño en punto fijo

Seguidamente se diseñó la TDNN, descrita en el anexo, en formato de punto fijo en complemento a dos. El diseño se afrontó con la misma metodología y con las mismas herramientas usadas en los escenarios anteriores: *System Generator* e *Integrated System Environment*.

4.5.2.1 Diseño con System Generator

El diseño del sistema usando *System Generator* de Xilinx queda como muestra la figura 4.43. En ella se observan los nueve elementos de retardo de la entrada, la

neurona de la capa intermedia con función de transferencia identidad y la neurona de salida con función *satlin*. Se diseñó de forma análoga a los escenarios anteriores. Cabe destacar el diseño de la función *satlin*, con el uso de un multiplexor, como se muestra en la figura 4.44.

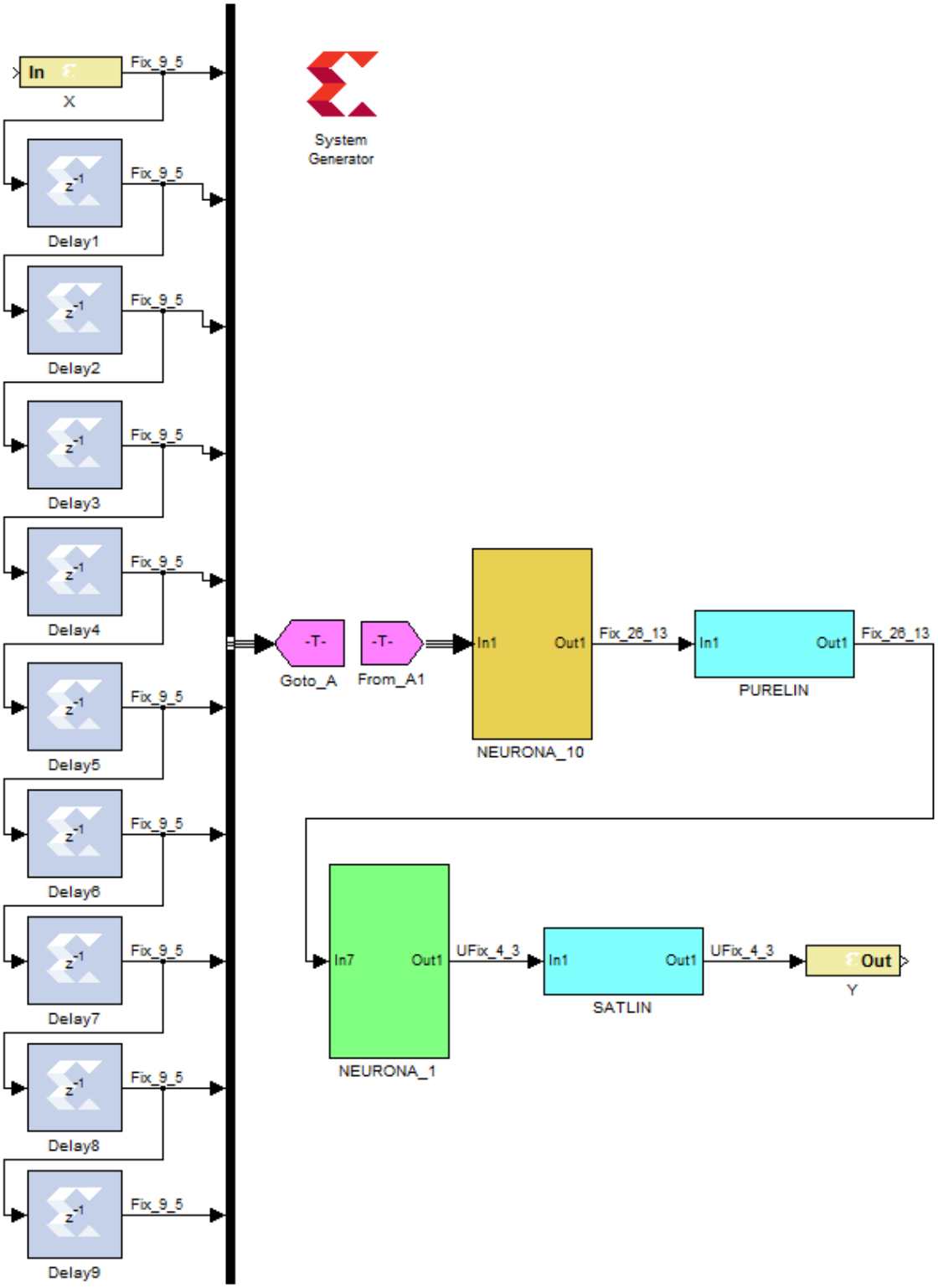


Figura 4.43. Ecualizador diseñado con System Generator.

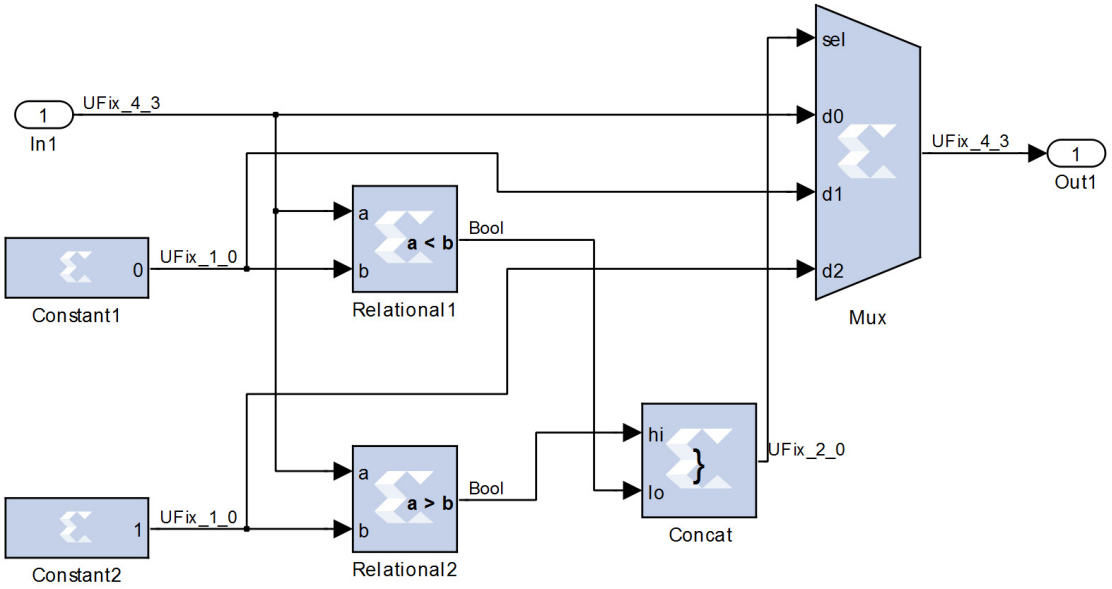


Figura 4.44. Diseño de la función satlin con el uso de un multiplexor.

Si el ruido fuese nulo la señal de entrada valdría 0 o +1, y sería suficiente una representación de un bit; esta situación sería trivial y no haría falta ecualizador. En cuanto aparece ruido la señal es bipolar, es necesario un bit de signo. En el caso peor, la SNR es de -5 dB; la varianza y potencia de ruido es 1,58; y la desviación típica del ruido es 1,26. Los valores en una función de densidad de probabilidad gaussiana de la variable *n*, se encuentran en torno a la media, según indica la expresión 4.3, con una probabilidad de 0,997.

$$Pr\ (\ \mu-3\sigma \le n \le \mu+3\sigma\ ) = 0{,}997 \qquad (4.3)$$

Las muestras se encuentran con una probabilidad mayor que 0,997 entre los límites dados por la ecuación 4.4; y la señal de entrada será representable en el intervalo [-3,78, +4,78].

$$\begin{aligned} \text{Límite inferior} &= 0 - 3\sigma = -3{,}78 \\ \text{Límite superior} &= 1 + 3\sigma = +4{,}78 \end{aligned} \qquad (4.4)$$

Para establecer el número de bits fraccionarios de la señal de entrada (*nbf*), el intervalo de cuantificación (Δ) se fijó como una fracción (*p*) del valor de pico a pico de

la señal de entrada, según la expresión 4.5. En la fórmula 4.5 la variable *A* es igual a 1, y el error máximo de cuantificación es $nq=\Delta/2$. Por ejemplo, si *p* vale 0,01 entonces el número de bits fraccionarios es 7.

$$\Delta = 2^{-nbf} \leq p(A-0) \tag{4.5}$$

No se puede fijar el número de bits fraccionarios para la señal de entrada usando un error porcentual para la representación del valor menor en valor absoluto; si una muestra en la entrada tiende a cero entonces el número de bits crece enormemente. En vez de eso, se fija en función de una fracción del valor de pico a pico de la señal sin ruido, según la expresión 4.6.

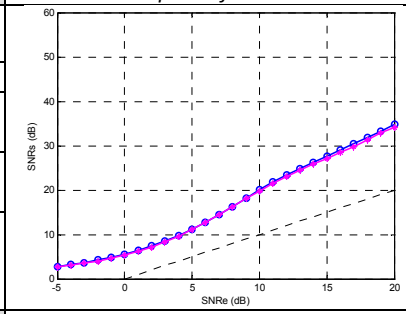
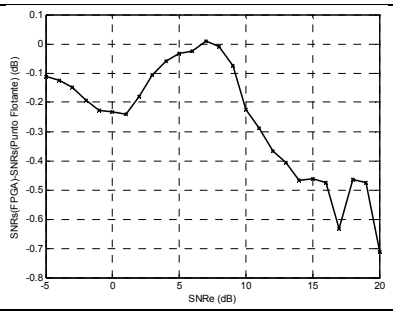
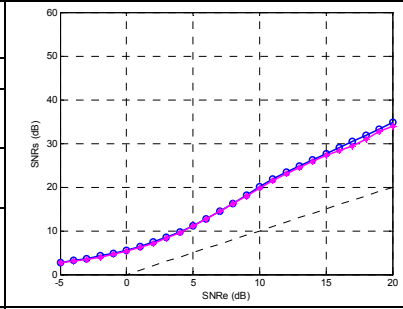
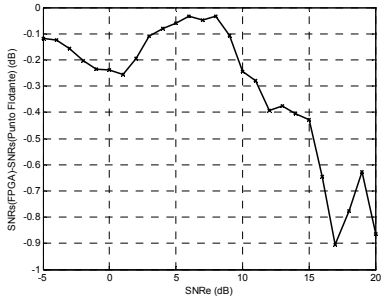
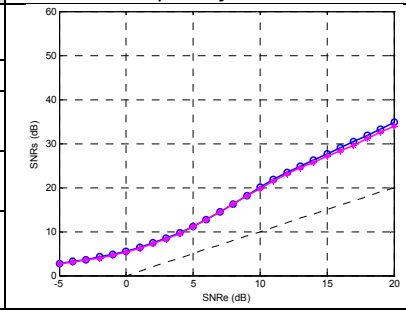
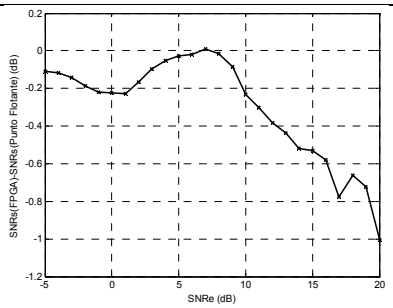
$$nbf \geq -\log_2[p(A-0)] \tag{4.6}$$

En el sistema anterior se estudió la SNR en la salida frente a la SNR en la entrada. El error máximo debe fijarse con algún criterio; por ejemplo, que la curva obtenida en punto fijo se desvíe menos de 1 dB respecto a la de punto flotante de la figura A.11, mostrada en el anexo. En la tabla 4.39 se comprobó la funcionalidad del sistema en función del error en la representación de los coeficientes, se observa que se consigue la funcionalidad con un 3,9% de error. Este valor se alcanzó tras un proceso de tanteo. Dada la simplicidad de las funciones de transferencia usadas, no existe ningún parámetro en su diseño que afecte a la funcionalidad.

Inicialmente se fijó la resolución en la salida de todos los operadores de forma completa (*Full*). El último operador es el multiplexor de la figura 4.44, con resolución completa la salida tiene el formato *Fix_35_20*; es decir, con bit de signo, 20 bits para la parte fraccionaria y 35 bits en total. Claramente este formato es excesivo para representar la salida de una función *satlin*, cuya valor está en el entorno [0,+1]. Por este motivo se procede a hacer un ajuste manual de la resolución en la salida: se elimina el bit de signo; por tanteo se ajusta el número de bits fracionarios a 3 y se mantiene un bit para la parte entera. El bit para la parte entera hace que +1 sea representable sin error. De esta forma el formato de la salida es *UFix_4_3*; es decir,

este es el formato que se define en la salida del multiplexor de la figura 4.44. Posteriormente se fijó la resolución en las salidas de los circuitos de forma manual, en dirección a la entrada, hasta llegar a los operadores que tienen los coeficientes en la neurona de salida.

Tabla 4.39. Efecto del error de representación en el ecualizador.

Error	**3,9%**	*SNRs punto fijo: ** *SNRs punto flotante: o*	*SNRs punto fijo - SNRs punto flotante*
SNRs (SNRe=+7 dB)	14,46		
Caso peor	-0,71 dB		
Formato de entrada	Fix_9_5		
Formato de salida	Fix_35_20		
Resolución en la salida	Completa		
Error	**3,9%**	*SNRs punto fijo: ** *SNRs punto flotante: o*	*SNRs punto fijo - SNRs punto flotante*
SNRs (SNRe=+7 dB)	14,47		
Caso peor	-0,91		
Formato de entrada	Fix_9_5		
Formato de salida	UFix_4_3		
Resolución en la salida	Ajustada		
Error	**4,0%**	*SNRs punto fijo: ** *SNRs punto flotante: o*	*SNRs punto fijo - SNRs punto flotante*
SNRs (SNRe=+7 dB)	14,47		
Caso peor	-1,01		
Formato de entrada	Fix_9_5		
Formato de salida	Fix_34_19		
Resolución en la salida	Completa		

En la figura 4.45 se muestran las formas de ondas obtenidas en la simulación con *Simulink* del sistema de la figura 4.43. Esta simulación se corresponde con un error del 3,9%, resolución ajustada en la salida, y SNR en la entrada de +7 dB.

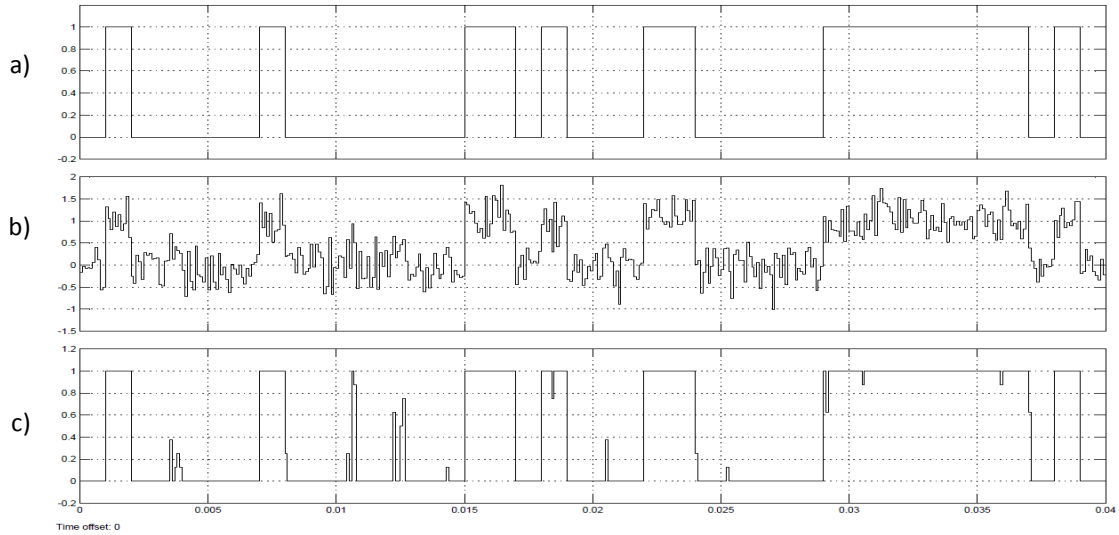


Figura 4.45. Formas de onda de Simulink en el ecualizador para los primeros 40 bits: a) señal de datos original, b) entrada en la FPGA, c) salida de la FPGA.

4.5.2.2 Implementación con Integrated System Environment

Una vez acabado el diseño en *System Generator* se procede a su compilación. El sistema final se corresponde con un error del 3,9% y resolución ajustada en la salida. Se compiló para obtener el sistema en VHDL y Verilog. El dispositivo elegido fue el Spartan3E, tipo xc3s100e, encapsulado tipo vq, 100 pines, velocidad -5; dicho de forma compacta el Spartan3E xc3s100e-5vq100.

Resultado para 3,9% de error. Lenguaje VHDL.

La compilación se realizó con *System Generator* para obtener la descripción en VHDL. En el proyecto obtenido para *Integrated System Environment* se realizó la simulación para la fase final de la implementación en la FPGA (*Post Place and Route Simulation*). Como simulador se usó *ISE Simulator*, integrado en la herramienta. Se obtuvieron las formas de onda de la figura 4.46.

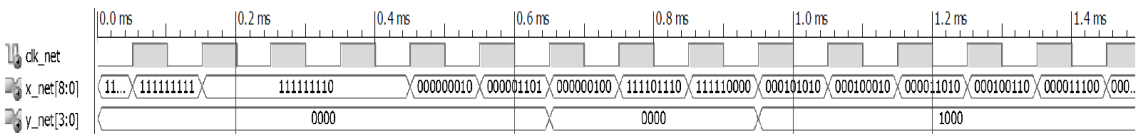


Figura 4.46. Formas de onda de las primeras quince muestras en el ecualizador después del colocado y conexionado de los componentes en la FPGA.

La ocupación de área se resume en la tabla 4.40. La frecuencia máxima del diseño es de 312,695 MHz. Las potencias se muestran en la tabla 4.41, a efectos de comparación se muestran las potencias para un reloj de 5 MHz, dado que para la frecuencia de 10 kHz la potencia dinámica es pequeña y no se observa variación frente al caso de Verilog, que se muestra en la tabla 4.43.

Tabla 4.40. Recursos hardware necesarios en el ecualizador compilado en VHDL.

Entidad	Usados	Disponibles	Tasa de uso
SLICES	401	960	41%
LUT de 4 entradas	690	1.920	35%
Bloques de E/S	14	66	21%

Tabla 4.41. Potencias en el ecualizador compilado en VHDL.

		Potencia (mW)		
		Estática	Dinámica	Total
Frecuencia	10,0 kHz	33,59	0,01	33,60
	5,0 MHz	33,60	1,09	34,69
	312,69 MHz	34,11	68,25	102,36

Resultado para 3,9% de error. Lenguaje Verilog.

La compilación se realizó con *System Generator* para obtener la descripción en Verilog. En el proyecto obtenido para *Integrated System Environment* se realizó la simulación para la fase final de la implementación en la FPGA (*Post Place and Route Simulation*), como simulador se usó *ISE Simulator*. Se obtuvieron formas de onda similares a la figura 4.46. La ocupación de área se resume en la tabla 4.42. La frecuencia máxima del diseño es de 357,654 MHz. Las potencias se muestran en la tabla 4.43.

Tabla 4.42. Recursos hardware necesarios en el ecualizador compilado en Verilog.

Entidad	Usados	Disponibles	Tasa de uso
SLICES	387	960	40%
LUT de 4 entradas	690	1.920	35%
Bloques de E/S	14	66	21%

Tabla 4.43. Potencias en el ecualizador compilado en Verilog.

		Potencia (mW)		
		Estática	**Dinámica**	**Total**
Frecuencia	**10,0 kHz**	33,59	0,01	33,60
	5,0 MHz	33,60	1,05	34,65
	357,65 MHz	34,17	74,88	109,04

Resumen de los casos estudiados.

Como resumen debe observarse la tabla 4.44, donde se muestran las prestaciones en área, velocidad y potencia. Las mejores prestaciones físicas se obtuvieron para la compilación en Verilog. El hecho de que el diseño alcance una frecuencia de reloj de 357,654 MHz quiere decir que esa es la máxima frecuencia de muestras en la entrada; como se tienen 10 muestras por bit, la tasa máxima de datos es de 35,765 Mbits por segundo. Si se usase un dispositivo más rápido puede aumentarse la velocidad del sistema.

Tabla 4.44. Resumen de los casos estudiados para el ecualizador.

	VHDL		**Verilog**	
3,9% de error	*Área*	401 SLICES	*Área*	387 SLICES
	Frecuencia máxima	312,695 MHz	*Frecuencia máxima*	357,654 MHz
	Potencia (5 MHz)	34,69 mW	*Potencia (5 MHz)*	34,65 mW

Llegado este punto en el diseño se podría comparar el tiempo de respuesta de la NN en punto flotante, ejecutada con el *Neural Network Toolbox* de Matlab, con el sistema en punto fijo sobre la FPGA elegida.

En el ordenador personal en punto flotante la clasificación de 10.000 muestras de entrada tarda 0,3 segundos, en la FPGA elegida la misma clasificación tarda unos 27,96 µs; del orden de 11.000 veces más rápida. Estos resultados son meramente orientativos, claramente depende del ordenador personal y de la FPGA que se use; sus características se muestran en la tabla 4.45.

Tabla 4.45. Comparación en velocidad entre un ordenador personal y la FPGA usada para el ecualizador.

	Tipo	***Tiempo clasificación 10000 muestras***
Ordenador personal	Windows 7 Home Premium 64 bits Intel(R) Core(TM) i7-3770 CPU @ 3.40 GHz RAM: 8 Gbytes Matlab R2010b 64 bits	0,3 s
FPGA	Spartan3E xc3s100e-5vq100	27,96 μs

Conviene resaltar que la simulación en *Simulink* del sistema diseñado con *System Generator* tarda unos 2 segundos; mientras la simulación en *Integrated System Environment* tarda un minuto aproximadamente. Estos tiempos son para el ordenador de la tabla 4.45.

4.6 Comparación con el estado del arte

El estado del arte se describió extensamente en el capítulo 1. Es difícil, cuando no imposible, hacer una comparación cuantitativa con los trabajos presentados. La dificultad viene de que los diferentes trabajos tienen enfoques distintos, luego las comparaciones tendrán carácter cualitativo. La gran mayoría de los trabajos descritos en el estado del arte, son de gran calidad. Lo único que se pretende en este apartado es establecer las diferencias con la tesis expuesta, y mostrar que esta constituye un trabajo original.

En el presente párrafo podrían nombrarse largas listas de la bibliografía, solo se nombran los trabajos más siginificativos; esto es por que ya se explicaron en el primer capítulo y sería una larga redacción de difícil lectura. Lo primero que debe ponerse de manifiesto es el tipo de NN estudiada, esta es del tipo perceptrón multicapa con conexión hacia adelante (*Feedforward Multilayer Perceptron*), algunos de los elementos de la bibliografía se centran en otro tipo de NN [Nambiar et al., 2014]. La mayoría de los trabajos están implementados en aritmética de punto fijo, aunque algunos se han desarrollado para aritmética de punto flotante [Cavuslu et al., 2011]. En esta tesis, salvo casos triviales, las funciones de transferencia se implementan mediante LUT, otros trabajos usan otras formas de aproximación [Nascimento et al.,

2013]. El método de diseño planteado es rápido y flexible; algunos autores usan métodos menos versátiles, como puede ser la descripción usando un HDL [Ogrenci, 2008]. Esta tesis insiste en la extracción de las prestaciones de área, velocidad y potencia; la mayoría de los autores no extraen el consumo de potencia [Orlowska y Kaminski, 2011]. Muchos trabajos implementa NN capaces de realizar un entrenamiento *online* [Gomperts et al., 2011]; fue objetivo de esta tesis realizar el entrenamiento *offline*. En la mayoría de las referencias se presenta el cálculo secuencializado en las neuronas, mediante uso de elementos MAC [Oniga et al., 2009]; en esta tesis, para priorizar la velocidad, se paralelizó el cálculo en las neuronas, igual que en [Bahoura y Park, 2011]. Pero ningún autor que paraleliza el cálculo en las neuronas asigna un número distinto de bits para cada coeficiente; más aún, asigna el mismo número de bits a los coeficientes de una capa [Bahoura y Park, 2011]. En general, cuando el cálculo está secuencializado o paralelizado los autores usan un número de bits fijado de forma arbitraria [Mishra et al., 2007]; como mucho las posibles arquitecturas son probadas de forma discreta, sin barrer todas las posibles soluciones [Gomperts et al., 2011]. Finalmente, en esta tesis se define la calidad de una implementación atendiendo a las prestaciones físicas (área, velocida y potencia), para una cierta funcionalidad; algunos autores definen la calidad de un diseño mezclando la funcionalidad con las prestaciones físicas, pero excluyendo la potencia [Tommiska, 2003].

La principal aportación de esta tesis es que el método planteado permite comprobar rápidamente diferentes arquitecturas, hasta llegar a su implementación física. En particular, en la NN totalmente paralelizada, se asigna una representación binaria diferente para cada coeficiente, y una resolución diferente para las funciones de cada capa. Este formato binario se asigna en función de un error para la representación. De esta forma solo se emplean los bits estrictamente necesarios en cada punto del sistema, mejorando las prestaciones físicas. La cantidad de bits usados no se varían directamente por el diseñador, lo que resultaría tedioso; sino a través del valor del error. De esta forma se consiguen diversas soluciones con la funcionalidad esperada, comparando las prestaciones físicas se puede elegir la que se considere apropiada.

CAPÍTULO 5

CONCLUSIONES Y LÍNEAS FUTURAS

"Se puede engañar a mucha gente durante poco tiempo, se puede engañar a poca gente durante mucho tiempo, pero no se puede engañar a mucha gente durante mucho tiempo".

Abraham Lilcoln

Para finalizar, en este capítulo se exponen las conclusiones observadas y posibles líneas de trabajo futuras.

5.1 Conclusiones

En esta tesis se ha tratado de validar la hipótesis que se expuso en el capítulo primero.

> *"Es posible encontrar métodos de diseño sobre dispositivos digitales programables para implementar redes neuronales que operen en tiempo real en procesado digital de la señal. Los métodos deben ser rápidos y flexibles; y permitir evaluar el efecto del número de bits sobre la arquitectura. Además, deben posibilitar la comprobación de la total funcionalidad del sistema y las prestaciones físicas de área, potencia y velocidad".*

Tras el desarrollo mostrado en los capítulos anteriores; especialmente en el cuarto, donde se muestran los resultados obtenidos, se comprueba que se ha conseguido verificar dicha hipótesis.

A continuación se enumeran una serie de conclusiones observadas. La primera conclusión es referente a como un profano puede introducirse en el estudio de las NN. Si bien existen multitud de libros, y es aconsejable la consulta de alguno de ellos, su estudio puede resultar tedioso por sus limitaciones gráficas y la falta de interacción con el usuario. A esto hay que añadir la variación de la nomenclatura entre los autores. Existen diferentes plataformas para el uso y apredizaje de las NN, pero cabe destacar Matlab, que se ha convertido en un entorno estándar para el modelado de las NN en punto flotante. La ventaja de Matlab es que permite un apredizaje interactivo y visual mediante sus tutoriales y ejemplos; además, de que su nomenclatura se está convirtiendo en estándar en lo referido a NN. A esto hay que añadir la ventaja de que los métodos avanzados de diseño para FPGA se basan en *Simulink* y en Matlab. En los dos últimos escenarios de esta tesis, se usó una TDNN, como sistema predictor de series temporales; es conveniente el uso del *Neural Network Time Series Tool* de Matlab por las ventajas comentadas en el capítulo cuarto. Las herramientas que han aparecido en años recientes para el diseño en FPGA, y se apoyan en Matlab, serán la vanguardia de los métodos y diseños en el futuro.

Por otro lado, los diseños presentados con *System Generator* no son portables a otros fabricantes; aunque podrían ser traducidos para otros entornos, lo que puede resultar costoso. Si se quisiese un diseño portable, una alternativa sería recurrir a los métodos que se apoyan en Matlab, que se expusieron en el capítulo segundo.

Un factor importante de los métodos de diseño es el soporte ante errores. Estos errores aparecen normalmente en las compilaciones, y se deben a la comunicación entre los diferentes programas, y a la interacción con el sistema operativo. Si bien, el aprendizaje y uso de un paquete de herramientas puede ser una importante tarea, la aparición de errores inesperados puede desalentar al diseñador.

La compilación con el *Integrated System Environment* puede optimizarse por área, velocidad o ser balanceada, la optimización en área o velocidad dio para las NN peores resultados. Sí se observó dependencia en la compilación desde *System Generator*, frente al HDL elegido, VHDL o Verilog.

En cuanto a la extracción de las prestaciones físicas caben destacar las idéas que se exponen a continuación.

- El área ocupada es la expresada con mayor frecuencia en el estado del arte, esto se debe a que son tasas de ocupación de recursos, y son mostrados con claridad y exactitud por las herramientas.
- La máxima frecuencia de funcionamiento ocupa una posición intermedia; su extracción, aunque fácil, no es inmediata; además, puede haber ciertas diferencias con el funcionamiento real del dispositivo.
- En último lugar se encuentra la potencia, esto se debe a que es la característica más difícil de extraer. Se divide en estática y dinámica; la estática depende de la temperatura, y la dinámica de la frecuencia. Para su estimación las herramientas suponen una temperatura ambiente estándar, y unos valores de disipación térmica para la FPGA; esto puede ser variado por el diseñador. Además la verificación de la potencia en funcionamiento obliga a medir las corrientes de la alimentación, pero una misma alimentación puede alimentar a otros dispositivos de la placa donde se encuentre la FPGA, lo que dificulta o imposibilita la medida.

En el capítulo segundo se presentaron métodos para comparar las prestaciones físicas de diferentes soluciones. Estas comparaciones, y la elección de un diseño óptimo, puede hacerse fácilmente usando un pequeño programa. En esta tesis, por ser relativamente pocas el número de soluciones, se presentaron los resultados en forma de tabla.

Después del desarrollo de los experimentos y del análisis del estado del arte, cabe preguntarse por qué se usa habitualmente el mismo número de bits en los coeficientes de una capa, aún cuando las neuronas están totalmente paralelizadas. Esto puede deberse a que los diseños provienen de códigos en punto flotante, donde el cálculo está secuencializado en un bucle, con tamaño de datos fijo. O visto de otra forma, los diseños provienen de arquitecturas secuenciales mediantes el uso de MAC.

En pruebas finales, realizadas como curiosidad al compilador de *System Generator*, en diseños con un solo multiplicador, se observó que no responde como fuera de esperar antes casos triviales. Al multilplicar una señal por uno o cero mantiene el circuito multiplicador, cuando es eliminable. También se mantiene el sumador, aunque una de sus entradas sea nula.

5.2 Líneas futuras

A partir del trabajo desarrollado y las conclusiones indicadas se exponen posibles líneas de trabajo.

Como mejora del método propuesto se puede operar en cada capa con un error distinto para los coeficientes, diferente error para las funciones de trasferencia en cada capa; e incluso, un error diferente para las entradas. Esto complica el proceso de modelado en punto fijo, pero puede llevar a implementaciones más eficientes.

En los dos últimos escenarios, en la salida de la TDNN, se realiza un ajuste en el número de bits. Esto se debe a que inicialmente la resolución de los operadores se configura con resolución completa. Este proceso podría automatizarse mediante el uso de parámetros y el conveniente código ejecutable.

Ha quedado de manifiesto, en el diseño del ecualizador, que a veces es posible usar las funciones de Matlab que son lineal por tramos. Esto supone grandes ventajas en la implementación de punto fijo.

De todas formas, pueden usarse funciones no incluidas en Matlab, como las formada por dos tramos parabólicos, lo que simplifica la implementación.

Finalmente, aún usando funciones sigmoideas, pueden implementarse otras aproximaciones; y estudiar sus prestaciones físicas. De hecho, se puede afirmar, que en el caso de NN la mitad del esfuerzo de los investigadores se centra en la implementación de la funciones.

Otra línea de trabajo puede ser no implementar la NN totalmente paralelizada; es decir, cambiar el secuenciamento del cálculo de diferentes formas. Se puede secuencializar cada neurona, cada capa o realizar el cálculo con un solo elemento MAC. El último caso estaría totalmente secuencializado. Las diferentes soluciones ahorran área, aunque aumenta el tiempo requerido para el cálculo. Para este tipo de arquitectura, donde se introducen máquinas de estados, es conveniente el uso de otro tipo de herramientas, donde la descripción sea de forma algorítmica.

En el futuro pueden implementarse otro tipo de clasificadores o NN. Cabe destacar la NN realimentadadas porque tienen mejor comportamiento en la predicción de series temporales. Debe insistirse en la conveniencia del *Neural Network Time Series Tool* de Matlab, porque en estos casos a las líneas de retardo en la entrada de la NN se conecta la entrada y la salida realimentada. Esta utilidad facilitaría su estudio y modelado en punto flotante.

Mediante el uso de las TDNN, realimentadas o no, puede procederse a la demodulación y detección de señales, tanto analógicas como digitales. Esto ofrece una interesante línea de estudio. Se podría comprobar sus prestaciones bajo los efectos del canal: distorsión, ruido e interferencia; frente a las configuraciones convencionales. Esto puede hacerse para señales sintéticas, o capturadas de un sistema de comunicaciones real.

La implementación realizada del ecualizador podría compararse con otros tipos de ecualización. Debe destacarse que en el ecualizador no es posible normalizar la señal,

dada la SNR en la entrada, esto afectaría a la potencia de la señal de entrada. La etapa que procedería añadir, previa al ecualizador, sería un control automático de ganancia, para garantizar la potencia de la señal de entrada. Esto no sería difícil evaluando el valor medio de la señal recibida.

Con el método planteado con *System Generator*, cada vez que se implementa una NN, es preciso su rediseño. Esto afecta al número de neuronas y al número de entradas en cada neurona. Se prodría crear una NN suficientemente grande, donde quepan los casos esperados, y al valer cero los correpondientes coeficientes, por no existir, desaparecieran los circuitos tras la complilación. Es decir, se ocupe el número de entradas necesario; no se activen los multiplicadores innecesarios y no se usen las salidas que sobran. Esto debe revisarse; más cuando el compilador mantiene circuitos innecesarios en casos triviales; explicados al final del apartado anterior.

ANEXO

Modelado en punto flotante del ecualizador

"… no se le olvide lo que de la ínsula me tiene prometido, que yo la sabré gobernar por grande que sea."

El Ingenioso Hidalgo Don Quijote de la Mancha
Miguel de Cervantes

A.1 Introducción

En este anexo se expone el modelado en punto flotante de la TDNN usada como ecualizador. Al final se obtiene la regla de oro que se implementará en punto fijo en la FPGA; es decir, la arquitectura de la NN y su funcionalidad.

A.2 Modelado en punto flotante de la TDNN ecualizadora

El sistema propuesto obedece a la figura A.1; donde la señal de entrada es del tipo NRZ unipolar y binaria. Es decir, el *"1"* y el *"0"* se representan respectivamente por *+A* y *0* voltios (o amperios), durante un tiempo de bit, que vale *Tb* segundos. Por otro lado, se asume que los ceros y los unos son equiprobables. La tasa binaria vale *Rb=1/Tb* bits por segundo. Puede suponerse que *A* es igual a 1 por simplificación y sin pérdida de generalidad.

Se supone que la señal está afectada por AWGN. La señal NRZ recibida tiene ancho de banda infinito y no sufre distorsión; sus componentes espectrales más significativas están cerca del origen de frecuencia. El AWGN tiene también ancho de banda infinito y su densidad espectral es uniforme con la frecuencia, su potencia es infinita, y por tanto la SNR es nula. El AWGN no está acotado en amplitud; aunque valores muy grandes en valor absoluto son poco probables, como indica su función de densidad de probabilidad gaussiana. Hasta aquí la señal es analógica y continua.

En resumen, se supone que la señal se ha transmitido por un canal con ancho de banda infinito que solo añade AWGN. La señal con ruido se muestrea cada *Tm* segundos; la frecuencia de muestreo vale *fm=1/Tm* Hz. Se tomará un número entero de muestras en cada intervalo de bit. En cada muestra de la señal se tiene la componente de señal recibida más la componente del ruido. La componente del ruido muestreado es gaussiano aditivo de valor medio nulo. La potencia del ruido en la salida del muestreador es finita y viene dada por la varianza, que es lo mismo que el cuadrado de la desviación típica. La nueva SNR deja de ser nula. La nueva señal es discreta en el tiempo, y puede ser introducida a un sistema digital con la conveniente cuantificación. Se puede crear un sistema digital en tiempo discreto.

El objetivo principal es comprobar si una TDNN puede ser usada como preamplificador o ecualizador de la señal; mejorando en su salida la SNR. Igual que la TDNN del predictor de temperatura para su modelado se usó el *Neural Network Time Series Tool* de Matlab [ntstool, 2015].

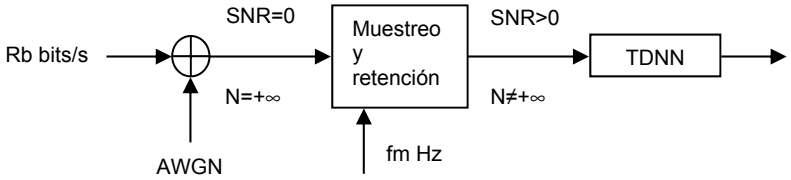


Figura A.1. Modelo propuesto para el ecualizador.

Inicialmente la tasa binaria (*Rb*) se establece en 1 kbit/s. Se tomaron 10 muestras por bit (*n=10*); es decir, la frecuencia de muestreo (*fm*) es de 10 kHz. El valor de la tasa binaria no tiene transcendencia para la predicción, lo que importa es el número de muestras por bit; en el sistema final la tasa binaria se puede aumentar tanto como lo permita la tecnología; es decir, la frecuencia máxima de reloj. En el bloque generador de ruido de *Simulink*, que genera *fm* muestras por segundo, se puede variar su potencia y su semilla generadora. En la figura A.2 se observa la señal de datos original y la señal con ruido muestreada, en esta figura la SNR es de +10 dB.

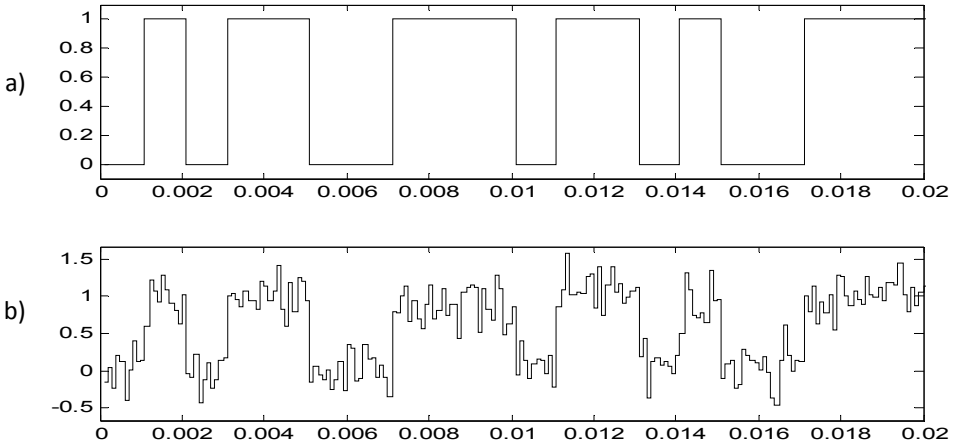


Figura A.2. Señal de datos original binaria (a) y señal con ruido muestreada (b).

La TDNN usada es similar a la de la figura 4.35, que se explicó en el predictor de temperatura. La cuestión es si esta TDNN serviría para mejorar la SNR de la señal muestreada. Para ello se entrenó con la señal ruidosa en su entrada y como objetivo se fija la señal de datos original. Se llamará *r(t)* a la señal recibida en la entrada del muestreador, que es igual a la señal de datos *d(t)* más el ruido *n(t)*, como indica la expresión A.1.

$$r(t) = d(t) + n(t) \qquad (A.1)$$

En un instante dado, llamado t_o, en los elementos de retardo de la red neuronal se tienen las muestras de valor $r(t_o\text{-}kTm)$, donde $k=0, 1, 2, 3, \ldots, 9$. Para este valor de la entrada el objetivo de la red neuronal en la salida es el valor del dato original en t_o; es decir, $d(t_o)$. La ventana de observación vale *Tb* segundos. Para entrenar, validar y testear la TDNN se usó:

- SNR entrenamiento y testeo de +10 dB;
- una secuencia de 2.000 bits aleatoria;
- se usó una capa intermedia, con cinco neuronas (tipo 10-5-1);
- como función de transferencia se usó la función *logsig* en todas las neuronas;
- de las muestras se usó el 40% para entrenar, el 5% para validar y el 55% para testear;
- el algoritmo de entrenamiento fue el de *Levenberg-Marquardt*;
- como función de error se usó el error cuadrático medio (*Mean squared error*).

En la figura A.3 se observan 20 bits de la señal de datos original, la señal con ruido muestreada y la salida de la TDNN. Se comprueba que la señal ha mejorado su SNR, que se medirá más adelante.

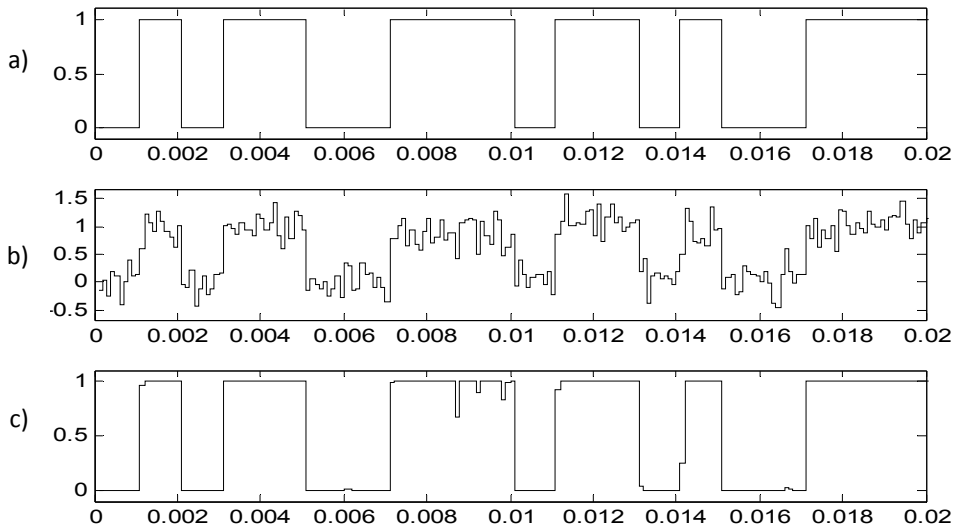


Figura A.3. Señal de datos original (a), la señal con ruido muestreada (b) y la salida de la TDNN (c).

La TDNN restringe la señal de salida al intervalo [0,1]. Esto se debe a que tiene una función *logsig* en la salida; además, el error máximo en la salida está acotado a 1. En la entrada de la TDNN el ruido no está acotado. Debe destacarse que la potencia de la señal en la entrada y salida de la TDNN coinciden; esto se debe a la forma de onda obtenida en la salida de la TDNN; es decir, a los valores objetivos (*target*) especificados en el entrenamiento.

Posteriormente se varió la SNR de testeo en la entrada de la TDNN. Es decir, la TDNN entrenada con una SNR de +10 dB, se testea de la siguiente forma:

- se varió la SNR de testeo desde -5 dB hasta +25 dB, en incrementos de 0,5 dB;
- se simularon 1000 bits para cada valor de SNR.

En la figura A.4 se observa la SNR en la salida (SNRs) de la TDNN frente a la SNR en su entrada (SNRe). Está marcada, de forma discontinua, la recta donde la SNRs es igual a la SNRe. Se observa que la SNR en la salida siempre es mayor que en la entrada. Hay que resaltar que el resultado depende del entrenamiento y esta es una curva típica obtenida.

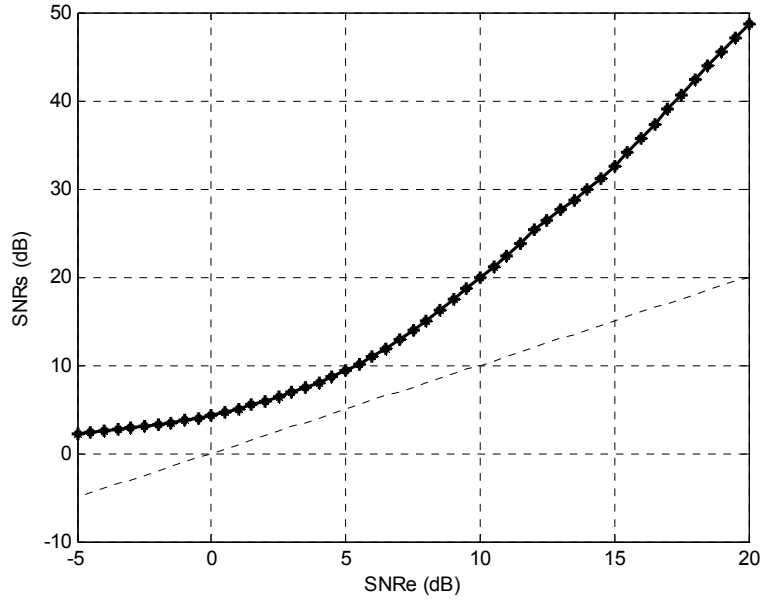


Figura A.4. Curva de SNR en la salida de la TDNN frente a la SNR en la entrada para el entrenamiento con +10 dB.

Conviene resaltar el elevado número de parámetros que intervienen en este estudio. En el sistema puede ajustarse:

- el número de capas intermedias;
- el número de neuronas en cada capa;
- la función de transferencia usada en cada capa;
- el algoritmo de entrenamiento;
- la función de error usada en el entrenamiento;
- la ventana de observación;
- la potencia del ruido usada en el entrenamiento;
- las potencias de ruido usadas en el testeo;
- la duración de la señal usada para entrenamiento, validación y testeo;
- el método de división de la señal usada para el entrenamiento, validación y testeo.

Para observar el efecto del aumento de las capas intermedias se probaron configuraciones con 2 y 3 capas ocultas. Se mantuvieron las demás condiciones anteriores: funciones *logsig*, etc. Ninguna de ellas entrenó bien. Luego hay que recurrir a una sola capa intermedia. Ahora se variará el número de neuronas en la capa intermedia, entre 1 y 20. En la figura A.5 se muestra la curva obtenida para 1, 10 y 20 neuronas.

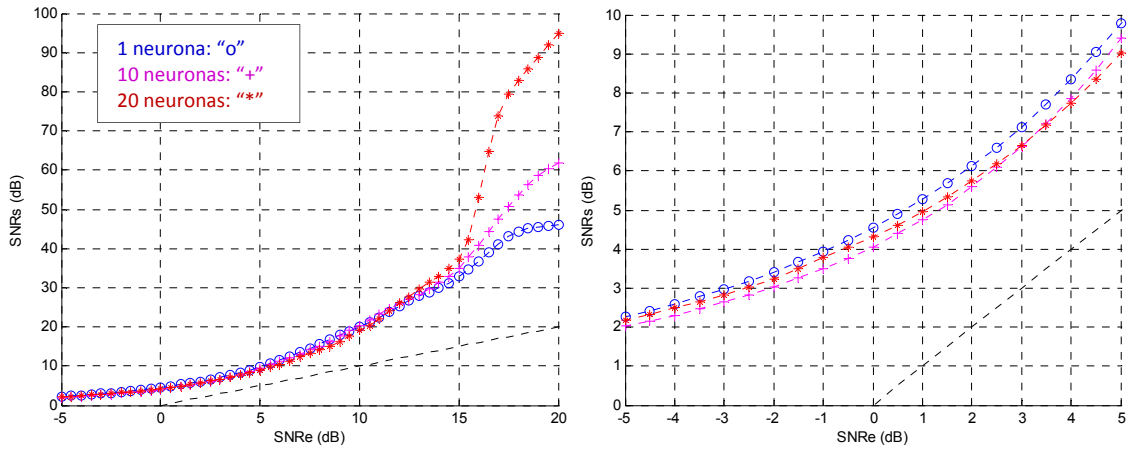
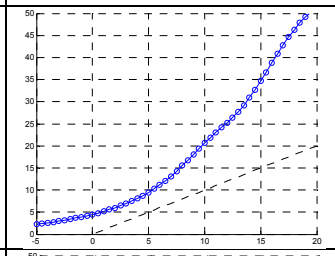
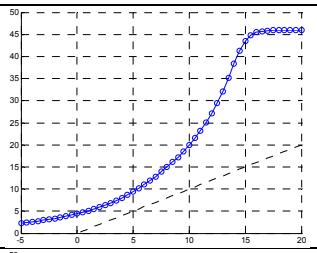
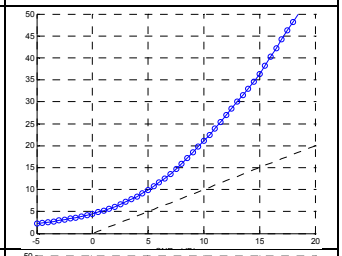
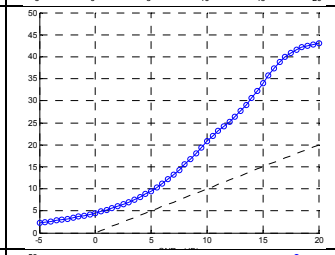
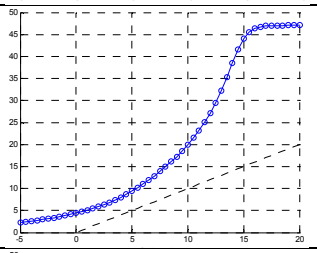
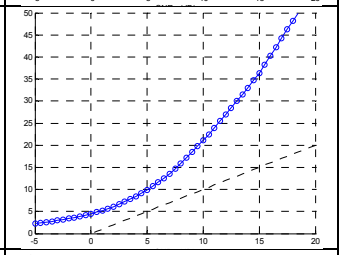
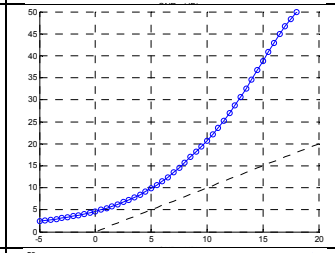
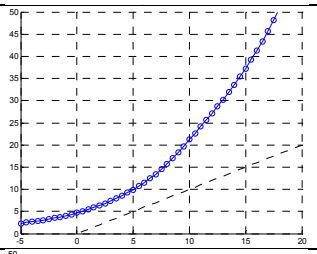
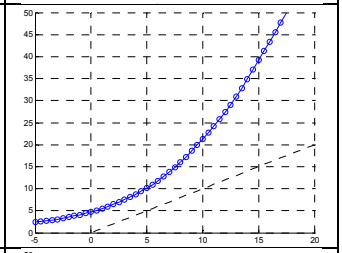
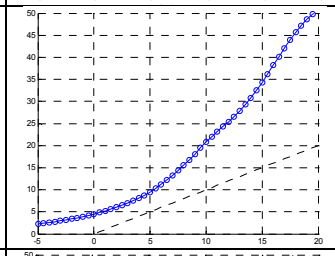
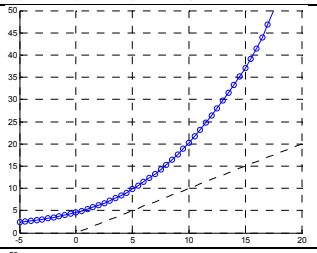
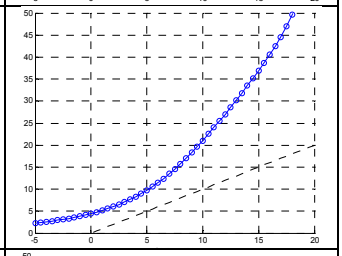
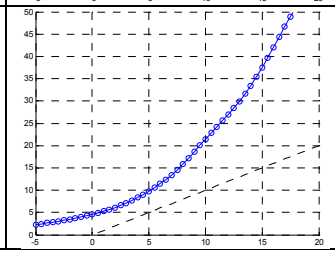
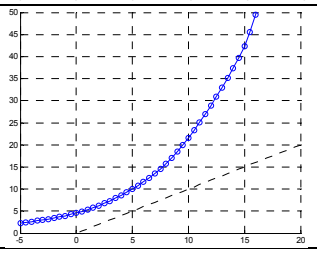
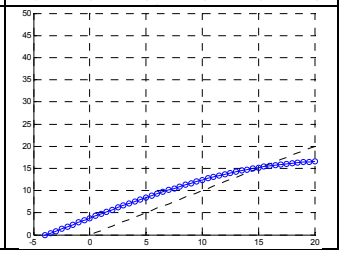


Figura A.5. Efecto del número de neuronas en la capa intermedia para el ecualizador.

Para una SNR en la entrada mayor que 15 dB, con una neurona en la capa intermedia, se obtienen altos valores de SNR en la salida, mayores que 30 dB. Para una baja SNR en la entrada las curvas convergen. Es suficiente, en general, una sola neurona en la capa intermedia. Esto es un criterio de diseño, se puede aumentar el número de neuronas si fuera necesario. Una sola neurona es la arquitectura más pequeña en la capa oculta: minimiza el área, el tiempo de respuesta y el consumo de potencia. En adelante se usará una sola capa intermedia con una neurona. Hay que resaltar que el resultado depende del entrenamiento, las curvas mostradas son curvas típicas.

A continuación se tratará de comprobar el efecto de las diferentes funciones de transferencia en la TDNN. Esto en principio no parece decisivo, siempre que sean funciones derivables y crecientes, pero pueden tener impacto en la implementación hardware. Dado que la señal de salida es unipolar, y dada la consideración de derivabilidad, se probaron tres tipos de función en la neurona de salida: *logsig, satlin* y *purelin*. En la capa intermedia se probaron las funciones: *logsig, tansig, satlin, satlins* y *purelin*. Las formas de estas funciones se tienen en la tabla 2.1. Los demás parámetros se mantuvieron, solo se cambió el tipo de función. Se probaron las combinaciones que muestra la tabla A.1, en ella se observan las curvas típicas obtenidas en el testeo.

Tabla A.1. Dependencia de la curva SNRs-SNRe frente a las funciones de transferencia usadas en el ecualizador.

Funciones en la capa intermedia	Funciones en la neurona de salida: *logsig*	*satlin*	*purelin*
logsig			
tansig			
satlin			
satlins			
purelin			

Con el objetivo de facilitar el diseño de la función de transferencia en un dispositivo digital programable, capaz de operar a alta velocidad, se desprecian los casos en los que la capa intermedia tienen función *logsig* o *tansig* (primeras dos filas de la tabla A.1. En ambos casos se obtiene una SNR en la salida parecida.

Cuando en la capa intermedia se tiene una función *satlin* o *satlins* se tiene una SNR en la salida parecida a la anterior, esto sucede en las filas tercera y cuarta de la tabla

A.1. El diseño de estas funciones es más fácil que en el caso anterior; básicamente se pueden implementar usando comparadores y multiplexores. Además, estas funciones no introducirán error de aproximación. Estos bloques tienen menos área, consumo y retardo, respecto a las dos primeras filas.

En la última fila, se usa para la capa oculta la función *purelin*, que es la función identidad. El diseño de esta función es trivial, la salida es igual a la entrada; un cortocircuito, sin error de aproximación. Por tanto, no conlleva gasto de área, consumo o retardo, que sean significativos. Se desecha cuando la función en la neurona de salida es *purelin* por la mala SNR en la salida. Quedan dos casos, para función *logsig* y *satlin* en la salida. Se elige en la salida la función *satlin*, por los motivos anteriores. En adelante para la TDNN se tendrán 10 muestras de la señal de entrada, una capa intermedia con una neurona y función *purelin*; y una neurona en la salida con función *satlin*, tal y como se muestra en la figura A.6. Debe ponerse de relieve que Matlab tiene una errata en la neurona de salida, muestra la función *satlin* como bipolar, cuando en realidad es unipolar.

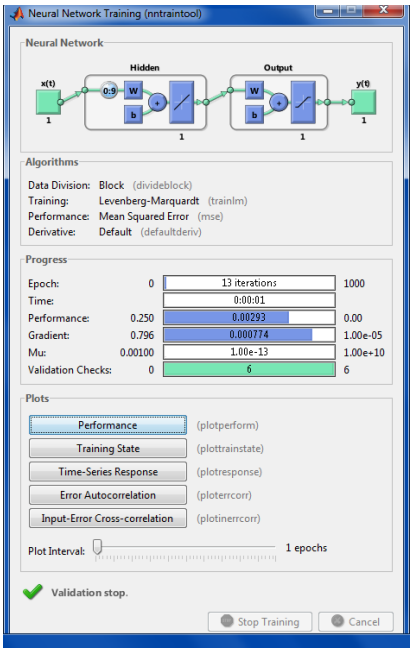
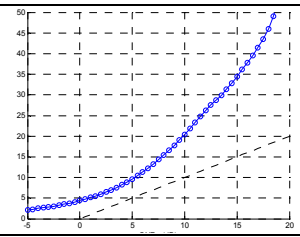
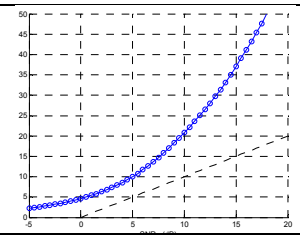
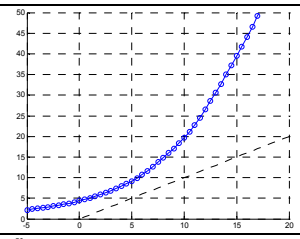
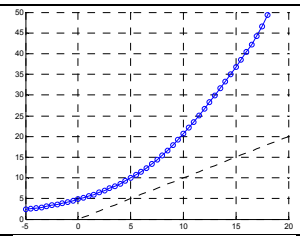
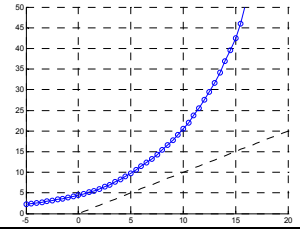
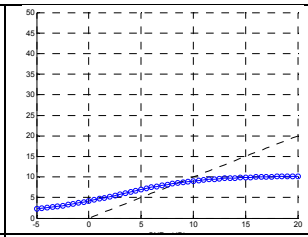
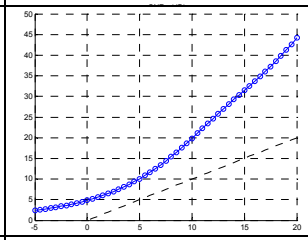
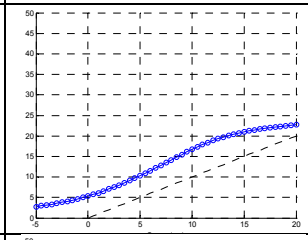
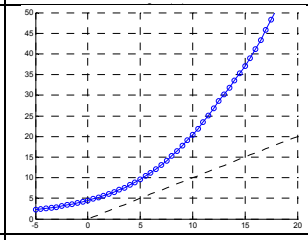
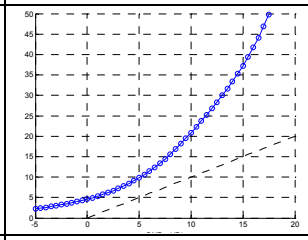
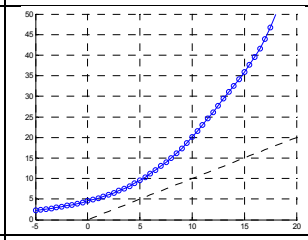
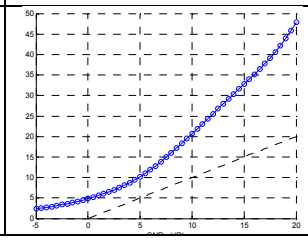


Figura A.6. Ventana de entrenamiento en Matlab para el ecualizador.

Para las condiciones anteriores, se probaron los algoritmos de entrenamiento de la tabla A.2, donde se observan curvas típicas obtenidas. En el futuro se fijará el algoritmo *traincgp* (*Conjugate gradient backpropagation with Polak-Ribiére updates*) para el entrenamiento. El motivo de esta elección es la SNR obtenida en la salida, el limitado número de iteraciones, y la rapidez de la convergencia.

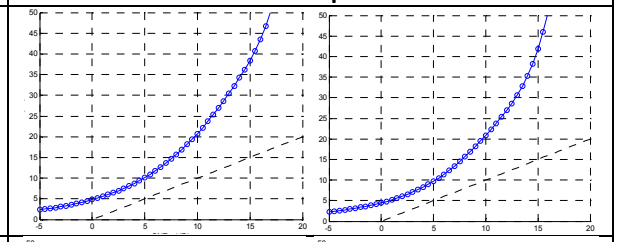
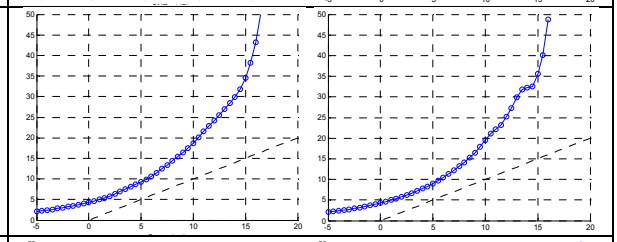
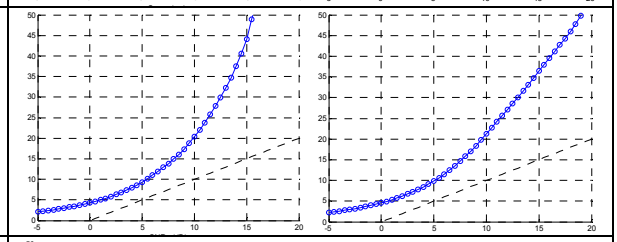
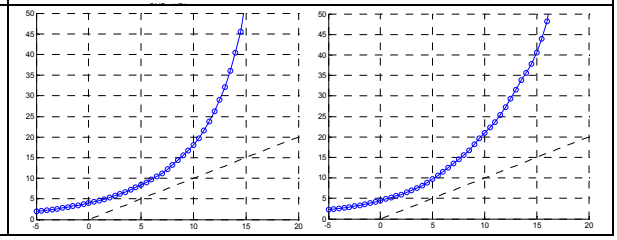
Tabla A.2. Curvas obtenidas en función del algoritmo de entrenamiento para el ecualizador.

ALGORITMO	Resultado	Comentarios
trainb	---	Entrena, no converge
trainbfg		Número de iteraciones: 25
trainbfgc	---	No entrena, produce error
trainbr		Número de iteraciones: 1000
trainbu	---	No entrena, produce error
trainc	---	No entrena, produce error
traincgb		Número de iteraciones: 22
traincgf		Tarda 1 segundo Número de iteraciones: 17
traincgp		Tarda 1 segundo Número de iteraciones: 23

traingd		Número de iteraciones: 1000
traingda		Número de iteraciones: 167
traingdm	---	Entrena, no converge
traingdx		Número de iteraciones: 110
trainlm		Número de iteraciones: 14
trainoss		Número de iteraciones: 26
trainr	---	No entrena, produce error
trainrp		Número de iteraciones: 162
trainru	---	Entrena, no converge
trains	---	Entrena, no converge
trainscg		Número de iteraciones: 28

Hasta el momento se ha usado como función de error el valor cuadrático medio (*mse*, *mean squared error*). Fijadas todas las condiciones anteriores, incluido el algoritmo *traincgp*, se comprobará el efecto de la función de error empleada en el algoritmo de entrenamiento. Hay cuatro opciones disponibles: error cuadrático medio (mse, *mean squared error*), error absoluto medio (mae, *mean absolute error*), suma del error al cuadrado (sse, *sum squared error*) y la suma del error absoluto (sae, *sum absolute error*). Los resultados se muestran en la tabla A.3.

Tabla A.3. Curvas obtenidas dependiendo de la función de error usada para el ecualizador.

Función de error	Curvas típicas
mse *mean squared error*	
mae *mean absolute error*	
sse *sum squared error*	
sae *sum absolute error*	

En la tabla A.3 se observa que la función de error no es decisiva; se mantiene *mse* (*mean squared error*), dada la pequeña mejora para baja SNR. La TDNN queda como en la figura A.7.

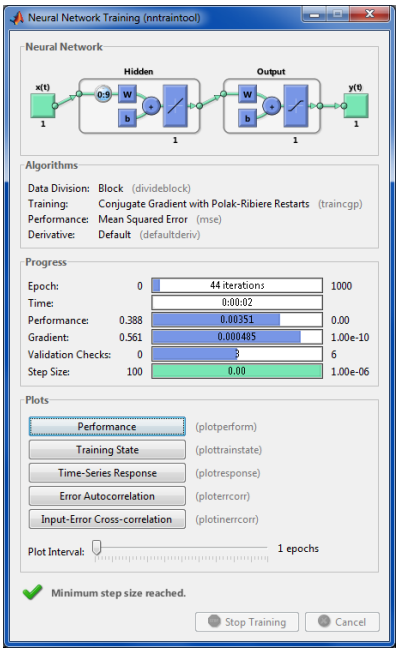


Figura A.7. Ventana de entrenamiento en Matlab, con el algoritmo de entrenamiento finalmente usado, para el ecualizador.

Posteriormente se estudia el efecto del tamaño de la ventana de observación; o sea, el número de muestras que entran en la TDNN. Es decir, se toman 10 muestras por tiempo de bit (*fmuestreo = 10 kHz*, *n=10*), la SNR de entrenamiento es de +10 dB, y se varía el número de entradas en la TDNN entre 1 y 20. En la figura A.8 se muestran las curvas para 8, 10 y 12 muestras en la entrada. Se observa que para una ventana de 10 muestras se consigue la mayor SNR; este intervalo coincide con la duración de un bit, *Tb* segundos.

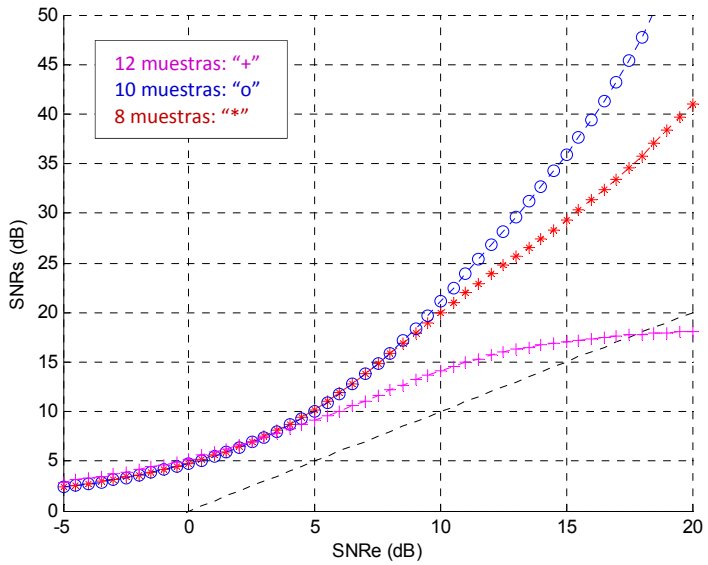


Figura A.8. Efecto del tamaño de la ventana de observación en el ecualizador.

En este apartado se justifica el uso de un intervalo de observación de *Tb* segundos. La decisión es tanto mejor cuantas más muestras se tiene, si la ventana es más pequeña que *Tb* segundos entonces hay que aumentar la frecuencia de muestreo, que puede llegar a ser una restricción del diseño. Si dura más de *Tb* segundos, cuando empieza un símbolo, que debe forzar la decisión en la TDNN, se está procesando no solo el símbolo anterior, sino hasta dos símbolos anteriores al de interés. El símbolo más antiguo recarga el sistema innecesariamente.

Posteriormente se estudió el efecto de la potencia de ruido en el entrenamiento. El entrenamiento se realizó variando la SNR desde -5 hasta +20 dB, en pasos de 1 dB. En la figura A.9 se muestran las curvas para diferentes SNR de entrenamiento. El testeo se realizó barriendo la SNR desde -5 hasta +20 dB, en pasos de 0,5 dB y 1000 bits para cada valor de SNR de testeo.

SNR entrenamiento	Leyenda
+20 dB	+
+15 dB	o
+10 dB	*
+5 dB	□
0 dB	x
-5 dB	Δ

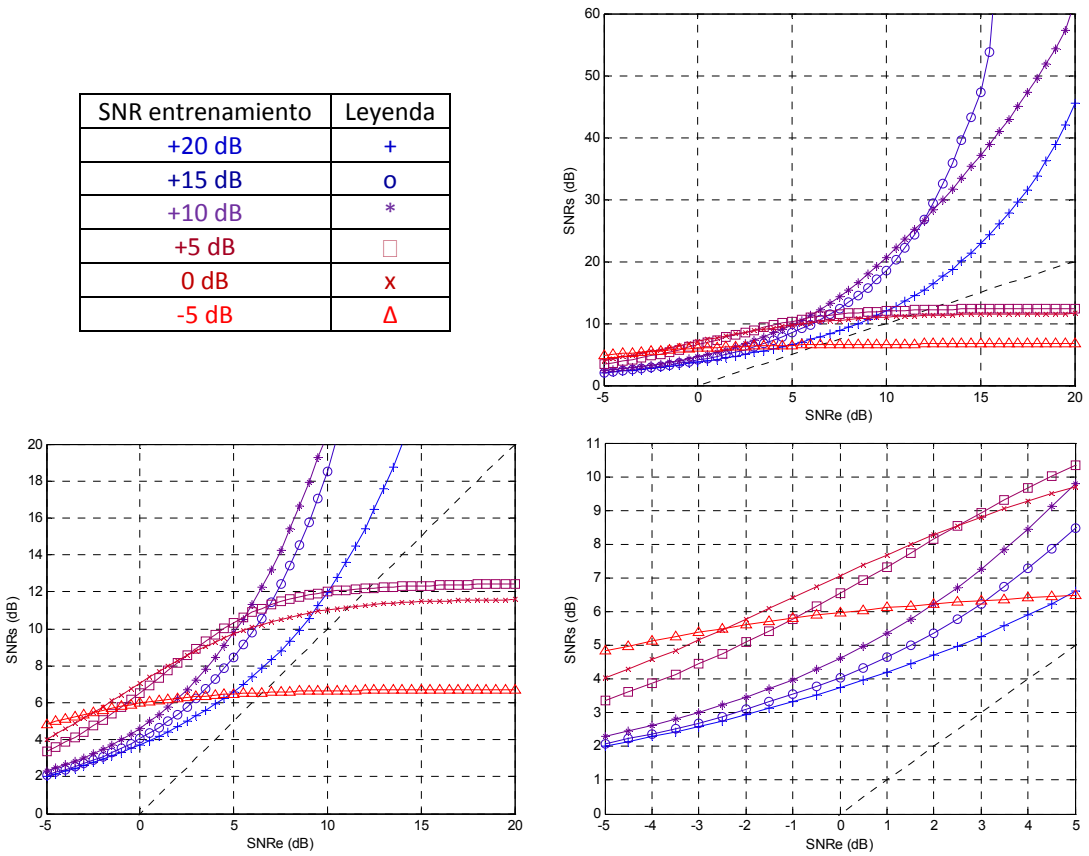


Figura A.9. Efecto de la SNR en el entrenamiento del ecualizador (de -5 dB a +20 dB).

En la figura A.9 se observa que:

- entrenando con -5 dB, se obtiene la mejor SNR para -5 dB en el testeo, se consigue entrenar con una SNR muy baja;
- entrenando con 0 dB, se obtiene la mejor SNR para 0 dB en el testeo;
- entrenando con +5 dB, se obtiene la mejor SNR para +5 dB en el testeo;
- entrenando con +10 dB, se obtiene la mejor SNR para +10 dB en el testeo;
- entrenando con +15 dB, se obtiene la mejor SNR para +15 dB en el testeo;
- pero entrenando con +20 dB no se obtiene la mejor SNR para +20 dB en el testeo, si no con el entrenamiento realizado para +15 dB.

Se comprobó que si la entrada disminuye por debajo de -5 dB la TDNN no entrena bien, hay demasiado ruido. Por encima de +15 dB hay muy poco ruido, esto produce una relajación del entrenamiento.

Se podría establecer el criterio de la SNR de entrenamiento como aquella que produce la mayor SNR en la salida para una SNR de testeo en la entrada, o rango en la entrada. Dicho de otra forma, se podría elegir como criterio; por ejemplo, una SNR de entrenamiento que maximizara la SNR en la salida para una determinada SNR en la entrada.

En adelante se intentará buscar la SNR de entrenamiento óptima con el criterio de tener máxima SNR en la salida para baja SNR en la entrada. Por otro lado, se mantendrá el criterio de que la SNR de la salida sea mayor que la entrada en todo el rango. Para ello se observan las curvas indicadas en la figura A.10. El testeo se realizó variando la SNR desde -5 hasta +20 dB, en pasos de 0,5 dB y 1000 bits para cada valor de SNR de testeo. Observando la figura A.10 se tomó como SNR óptima de entrenamiento +7 dB.

SNR entrenamiento	Leyenda
+10 dB	+
+9 dB	o
+8 dB	*
+7 dB	·
+6 dB	x

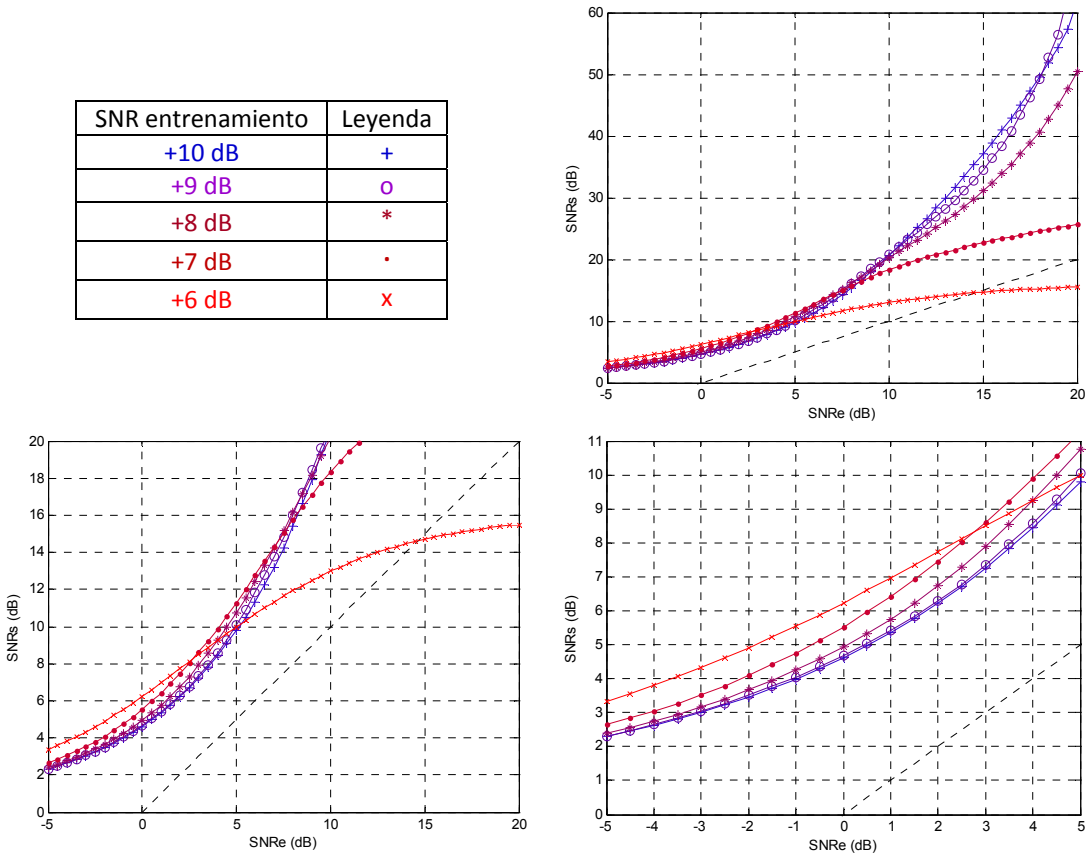


Figura A.10. Efecto de la SNR en el entrenamiento del ecualizador (de +6 dB a +10 dB).

Variando de +6 dB hasta +7 dB (en paso de 0,1 dB) no se consiguen resultados significativos, las curvas no siguen un comportamiento regular o claramente diferenciado. Para futuras referencias se muestra la curva para una SNR óptima de entrenamiento de +7 dB en la figura A.11. Testeando el sistema con una SNR de +7 dB en la entrada se consiguen 14,52 dB en la salida.

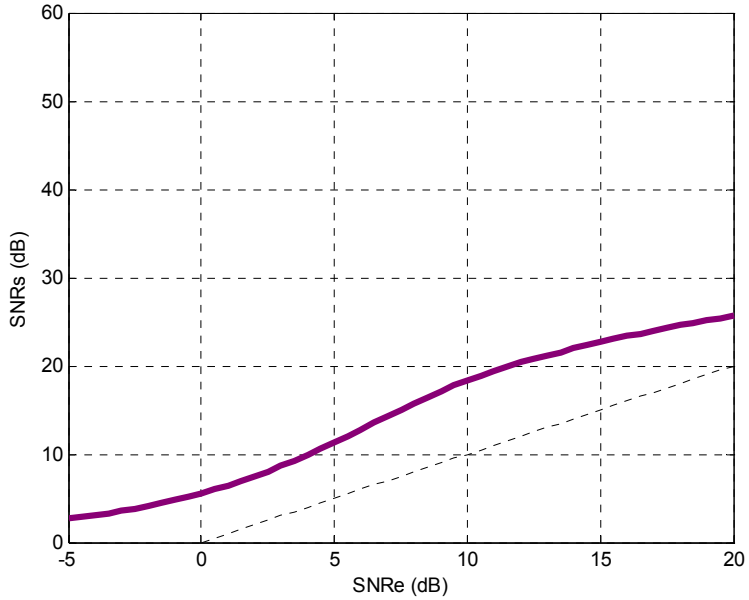


Figura A.11. Testeo de la SNR de salida frente a la SNR de entrada para una SNR de entrenamiento de +7 dB.

El dividir las muestras para entrenemiento, validación y testeo; en forma aleatoria o por bloques, no tuvo efecto en el comportamiento de la TDNN.

En resumen, el objetivo es diseñar una TDNN para una señal de entrada de 1 kbit/s aleatoria; la señal es NRZ unipolar, con amplitud igual a 1; y se toman 10 muestras por de bit, esto implica una frecuencia de muestreo de 10 kHz.

Los parámetros de la TDNN, según los estudios anteriores, son los que se enumeran a continuación.

- La SNR de entrenamiento es de +7 dB.
- Ventana de entrada de 10 muestras.
- Una neurona en la capa oculta.
- Funciones de transferencia: *purelin* en la capa oculta y *satlin* en la salida.
- Se usaron 2.000 bits (40% para entrenamiento, 5% para validación y 55% para testeo).
- Algoritmo de entrenamiento es *traincgp*.
- Función de error: mse (*mean squared error*)

En el testeo final se usó:

- SNR de testeo desde -5 dB hasta +20 dB, en pasos de 0,5 dB.
- Se simularon 1.000 bits para cada nivel de potencia.

En este punto se ha llegado a la arquitectura de la TDNN y a sus parámetros asociados. Pero cada vez que se entrena se obtiene una TDNN con distintos coeficientes, cuyas curvas de prestaciones son parecidas, pero no iguales. Es decir, puede haber variaciones entre las diferentes TDNN obtenidas en diferentes entrenamientos. Esto no solo se debe a la inicialización del algoritmo de entrenamiento, sino también a que en cada entrenamiento se usa una señal de datos y señal de ruido distinta; donde se mantienen el valor de sus parámetros: potencia, características estadísticas, etc. Por lo tanto se puede definir el concepto de "banda de estabilidad", esta es la zona donde se puede asegurar que encajan la mayoría de las curvas de las TDNN obtenidas.

Con el fin de obtener la banda de estabilidad se hicieron 100 entrenamientos. Se despreciaron aquellos en los que para SNR en la entrada de +20 dB daba menos de +20 dB en la salida. Quedaron 94 entrenamientos válidos; o sea, 94 TDNN con sus respectivas curvas. En la figura A.12 (a) se observan las 94 curvas de testeo.

Para cada valor de la SNR de entrada se calculó el valor medio de las curvas, que se muestra en la figura A.12 (b) en línea de puntos. Para cada valor de la SNR de entrada se calculó la desviación típica. Con esos valores se define una frontera inferior, restando a los valores medios la desviación típica. De la misma forma se define una frontera superior, sumando a los valores medios la desviación típica. Estas fronteras se muestran con líneas continuas en la figura A.12 (b). La banda de estabilidad es la zona que está entre las dos fronteras. Dentro de la banda de estabilidad cayeron 71 de los 94 entrenamiento válidos; es decir, el 76 % de las curvas. Se considera que una curva está dentro de la banda de estabilidad cuando para una SNR de +20 dB en la entrada el valor de la salida está dentro de la banda.

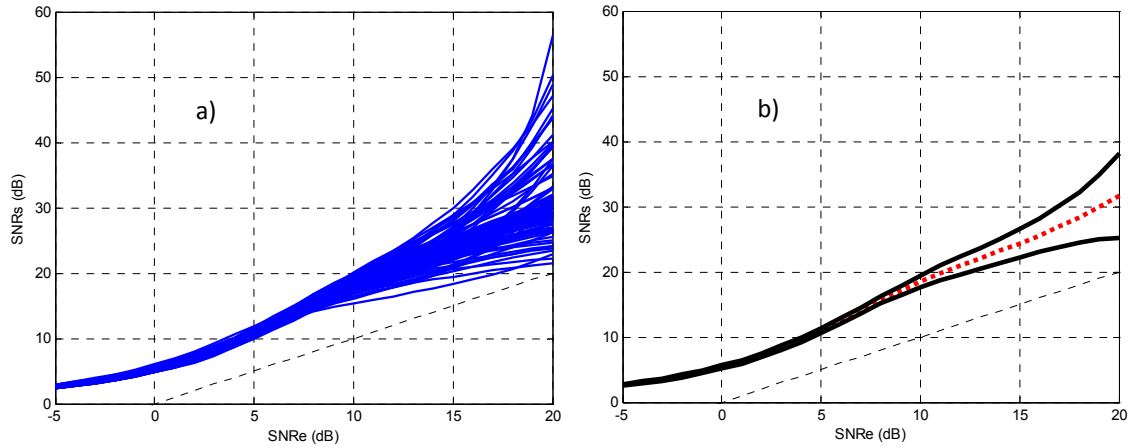


Figura A.12. Curvas de entrenamientos existosos (a), curva promedio y límites de la banda de estabilidad (b).

La solución obtenida se somete a una segunda vuelta de variación de los parámetros, con el objetivo de comprobar si la solución obtenida es local; es decir, si cambiando algún parámetro se puede mejorar. El comportamiento, al variar los parámetros en orden distinto, se mantiene.

Una de las ventajas de este método es que se puede elegir la SNR de entrenamiento; es decir, la curva de SNR obtenida no es rígida. Otra ventaja es que su arquitectura y parámetros pueden ser cambiados para que se adapte a otro tipo de señal o ruido.

BIBLIOGRAFÍA

"… se le pasaban las noches leyendo de claro en claro, y los días de turbio en turbio, y así, del poco dormir y del mucho leer, se le secó el cerebro, …"

El Ingenioso Hidalgo Don Quijote de la Mancha
Miguel de Cervantes

[AccelDSP, 2008] Xilinx User Guide; "AccelDSP Synthesis Tool", release 10.1, 2008

[AccelDSP, 2014] AccelDSP Synthesis Tool, *http://www.xilinx.com/tools/acceldsp.htm*, última visita el 11 de junio de 2014

[Active-HDL, 2014] Active-HDL de *Aldec*, *http://www.aldec.com/en/products/fpga_simulation/active-hdl*, última visita el 4 de junio de 2014

[AHDL, 2014] Altera Hardware Description Language, *http://www.alterawiki.com/uploads/c/c9/Altera_AHDL_Language_Reference.pdf* , última visita el 4 de junio de 2014

[Alonso et al., 2013] Alonso, J. B.; Cabrera, J.; de León, J.; Ferrer, M. A.; Travieso, C. M.; Sánchez, D.; Henríquez, P.; Morales, A.; Rivero, J. F.; Ayudarte, F.; Pérez, S. T.; Cabrera, F.; Caballero, J. M.; "Proyecto e-VOICE: Sistema de Evaluación Remota del Sistema Fonador", *I Jornadas Multidisciplinares de Usuarios de la Voz, el Habla y el Canto* (JVHC 2013), pp. 46-58, 2013

[Altera, 2014] Altera Corporation, *http://www.altera.com/*, última visita el 11 de junio de 2014

[Anadigm, 2015] Anadigm Inc., *http://www.anadigm.com/*, última visita el 1 de julio de 2015

[Armato et al., 2011] Armato, A.; Fanucci, L.; Scilingo, E. P.; De Rossi, D "Low-error digital hardware implementation of artificial neuron activation functions and their derivative", *Microprocessors and Microsystems*, vol. 35, nº 6, pp. 557-567, 2011

[Bahoura y Park, 2011] Bahoura, M.; Park, C. W.; "FPGA-Implementation of High-Speed MLP Neural Network", *18th IEEE International Conference on Electronics, Circuits and Systems* (ICECS 2011), pp. 426 - 429, 2011

[Basterretxea et al., 2004] Basterretxea, K.; Tarela, J. M.; del Campo, I.; "Approximation of sigmoid function and the derivative for hardware implementation of neural networks", *IEE Proceedings-Circuits Devices and Systems*, vol. 151, nº 1, pp. 18-24, 2004

[Basterretxea, 2012] Basterretxea, K.; "Recursive Sigmoidal Neurons for Adaptive Accuracy Neural Network Implementations", *NASA/ESA Conference on Adaptive Hardware and Systems* (AHS 2012), pp. 152 - 158, 2012

[Belanovic y Leeser, 2002] Belanovic, P; Leeser, M; "A library of parameterized floating-point modules and their use", *12th International Conference on Field-Programmable Logic and Applications*, Lecture Notes in Computer Science, vol. 2438, pp. 657–666, 2002

[Belanovic, 2002] Belanovic, P.; "Library of Parameterized Hardware Modules for Floating-Point Arithmetic with An Example Application", Trabajo Fin de Grado, Universidad de Northeastern, 2002

[Bishop, 2005] Bishop, C. M.; *Neural Networks for Pattern Recognition*, 13ª edición, Oxford University Press, 2005

[Cao, 2003] Cao, J. D.; "Global asymptotic stability of delayed bi-directional associative memory neural networks", *Applied Mathematics and Computation*, vol. 142, nº 2-3, pp. 333-339, 2003

[Cavuslu et al., 2011] Cavuslu, M. A.; Karakuzu, C.; Sahin, S.; Yakut, M.; "Neural network training based on FPGA with floating point number format and it's performance", *Neural Computing and Applications*, vol. 20, nº 2, pp. 195-202, 2011

[Chadnani, 2011] Chadnani, B. V.; "Análisis de reducción de características en la detección de arritmias cardíacas", Proyecto Fin de Carrera, Universidad de Las Palmas de Gran Canaria, 2011

[Chadnani, 2014] Chadnani, B. V.; "Implementación de un Kernel sobre la reducción de dimensionalidad en la identificación de cardiopatías", Trabajo Fin de Master, Universidad de Las Palmas de Gran Canaria, 2014

[Chandrasetty, 2011] Chandrasetty, V. A.; *VLSI Design A Practical Guide for FPGA and ASIC Implementations*, Springer, 2011

[Chen et al., 2006] Chen, X.; Wang, G. F.; Zhou, W.; Chang, S.; Sun, S. L.; "Efficient Sigmoid Function for Neural Networks Based FPGA Design", *International Conference on Intelligent Computing* (ICIC), Lecture Notes in Computer Science, vol. 4113 , nº 1, pp. 672-677, 2006

[Cofer y Harding, 2006] Cofer, R. C.; Harding, B. F.; *Rapid System Prototyping with FPGAs,* Elsevier, 2006

[Connor et al., 1994] Connor, J.T.; Martin, R. D.; Atlas, L. E.; "Recurrent neural networks and robust time series prediction", *IEEE Transactions on Neural Networks*, vol. 5, nº 2, pp. 240-254, 1994

[Costa et al., 1999] Costa, M.; Palmisano, D.; Pasero, E.; "A system design methodology for analog feed forward artificial neural networks", *Analog Integrated Circuit and Signal Processing*, vol. 21, nº 1, pp. 45-55, 1999

[C-to-Verilog, 2009] C-to-Verilog por Nadav Rotem, *http://www.c-to-verilog.com*, última visita el 6 de junio de 2014

[del Pozo et al., 2012] del Pozo, M.; Ticay, J. R.; Cabrera, J.; Arroyo, J.; Travieso, C. M.; Sánchez, L.; Pérez, S. T.; Alonso, J. B.; Ramírez, M.; "Image Processing for Pollen Classification", *Biodiversity Enrichment in a Diverse World*, InTech, páginas: 493-508, 2012

[Deng et al., 2011] Deng, L.; Sobti, K.; Zhang, Y.; Chakrabarti, C.; "Accurate Area, Time and Power Models for FPGA-Based Implementation", *Journal of Signal Processing Systems*, vol. 63, nº 1, pp. 39-50, 2011

[Dias et al., 2004] Dias, F. M.; Antunes, A.; Mota, A. M.; "Artificial neural networks: a review of commercial hardware", *Engineering Applications of Artificial Intelligence*, vol. 17, nº 8, pp. 945-952, 2004

[Dong et al., 2006] Dong, P.; Bilbro, G. L.; Chow, M. Y.; "Implementation of artificial neural network for real time applications using field programmable analog Arrays", *IEEE International Joint Conference on Neural Network* (IJCNN), pp. 1518-1524, 2006

[DSP Builder, 2014] DSP Builder, *http://www.altera.com/products/software/products/dsp/dsp-builder.html*, última visita el 11 de junio de 2014

[Duda et al., 2001] Duda, R. O.; Hart, P. E.; Stork, D.G.; Pattern Classification, 2ª edición, John Wiley & Sons, 2001

[EDIF, 2000] *Electronic design interchange format (EDIF), Part 2, Version 4.0.0*, International Electrotechnical Commission, 2000

[Ferreira et al., 2007] Ferreira, P.; Ribeiro, P.; Antunes, A.; Morgado, F.; "A high bit resolution FPGA implementation of a FNN with a new algorithm for the activation function", *Neurocomputing*, vol. 71, nº 1, pp. 71-77, 2007

[Filter Design HDL Coder, 2014] Matlab Filter Design HDL Coder, *http://www.mathworks.es/products/filterhdl*, última visita el 13 de junio de 2014

[Fixed-Point Designer, 2014] Fixed-Point Designer, *http://www.mathworks.es/products/fixed-point-designer,* última visita el 13 de junio de 2014

[Floyd, 2006] Floyd, T. L.; *Fundamentos de Sistemas Disgitales*, novena edición, Prentice Hall, 2006

[flpfxptoolbox, 2006] Floating-Point to Fixed-Point Transformation Toolbox, *http://users.ece.utexas.edu/~bevans/projects/wordlength/converter/*, última visita el 13 de junio de 2014

[Gokhale y Graham, 2005] Gokhale, M.; Graham, P.S.; *Reconfigurable Computing Accelerating Computation with Field-Programmable Gate Arrays*, Springer, 2005

[Gomperts et al., 2011] Gomperts, A.; Ukil, A.; Zurfluh, F.; "Development and Implementation of Parameterized FPGA-Based General Purpose Neural Networks for Online Applications", *IEEE Transactions on Industrial Informatics*, vol. 7, nº 1, pp. 78-89, 2011

[Graupe, 2013] Graupe, D.; *Principles of Artificial Neural Networks*, 3ª edición, World Scientific, 2013

[Hauck y Dehon, 2008] Hauck, S.; Dehon, A.; *Reconfigurable Computing*, Elsevier, 2008

[HDL Coder, 2014] Matlab HDL Coder, *http://www.mathworks.es/products/hdl-coder*, última visita el 13 de junio de 2014

[HDL Verifier, 2014] Matlab HDL Verifier, *http://www.mathworks.es/products/hdl-verifier*, última visita el 13 de junio de 2014

[Himavathi et al., 2007] Himavathi, S.; Anitha, D.; Muthuramalingam, A.; "Feedforward neural network implementation in FPGA using layer multiplexing for effective resource utilization", *IEEE Transactions on Neural Networks*, vol. 18, nº 3, pp. 880-888, 2007

[Ho et al., 2008] Ho, T.; Lam, P.; Leung, C.; "Parallelization of cellular neural networks on GPU", *Pattern Recognition*, vol. 41, nº 8, pp. 2684-2692, 2008

[IEEE Std 1076, 2009] *IEEE Standard VHDL Language Reference Manual*, enlace permanente: http://ieeexplore.ieee.org/servlet/opac?punumber=4772738

[IEEE Std 1364, 2006] *IEEE Standard for Verilog Hardware Description Language*, enlace permanente: http://ieeexplore.ieee.org/servlet/opac?punumber=10779

[IEEE Std 1666, 2011] IEEE Standard for Standard SystemC Language Reference Manual, enlace permanente: *http://ieeexplore.ieee.org/servlet/opac?punumber=6134617*

[IEEE Std 754, 2008] *IEEE Standard for Floating-Point Arithmetic*, enlace permanente: http://ieeexplore.ieee.org/servlet/opac?punumber=4610933

[ISE, 2013] Integrated System Environment, *http://www.xilinx.com/products/design-tools/ise-design-suite/index.htm*, última visita el 11 de junio de 2014

[JHDL, 2006] Java Hardware Description Language, *http://www.jhdl.org*, última visita el 6 de junio de 2014

[Krips, 2002] Krips, M.; Lammert, T.; Kummert, A.; "FPGA implementation of a neural network for a real-time hand tracking system", *1st IEEE International Workshop on Electronic Design, Test and Applications*, pp. 313-317, 2002

[Kwan, 1992] Kwan, H. K.; "Simple sigmoid-like activation function suitable for digital hardware implementation", *IET Electronics Letters*, vol. 28, nº 15, pp. 1379-1380, 1992

[LabView, 2015] Laboratory Virtual Instrumentation Engineering Workbench de National Instruments, *http://www.ni.com/labview/esa*, última visita el 29 de junio de 2015

[Lange, 2005] Lange, R.; "Design of a Generic Neural Network FPGA-Implementation", Trabajo Fin de Grado, *Universidad Tecnológica de Chemnitz*, 2005

[Larkin et .al, 2006] Larkin, D.; Kinane, A.; Muresan, V.; O'Connor, N.; "An Efficient Hardware Architecture for a Neural Network Activation Function Generator", *3rd International Symposium on Neural Networks* (ISNN 2006), Lecture Notes in Computer Science, vol. 3973, pp. 1319-1327, 2006

[Lee y Burgess, 2003] Lee, B.; Burgess, N.; "Some results on Taylor-series function approximation on FPGA", *37th Asilomar Conference on Signals, Systems and Computers*, pp. 2198-2202, 2003

[Lin y Wang, 2008] Lin, C; Wang, J; "A digital circuit design of hyperbolic tangent sigmoid function for neural networks", *IEEE International Symposium on Circuits and Systems*, pp. 856-859, 2008

[Maliuk et al., 2010] Maliuk, D.; Stratigopoulos, H. G.; Makris, Y.; "An Analog VLSI Multilayer Perceptron and its Application Towards Built-In Self-Test in Analog Circuits", *IEEE 16th International On-Line Testing Symposium* (IOLTS 2010), pp. 71 - 76, 2010

[Mandic y Chambers, 2001] Mandic, D. P.; Chambers, J. A.; *Recurrent Neural Networks for Prediction*, John Wiley & Sons, 2001

[Manjunath y Gurumurthy, 2003] Manjunath, R.; Gurumurthy, K.S.; "Artificial_neural networks_as_building_blocks_of_mixed_signal_FPGA", *2nd International Conference on Field-Programmable Technology* (ICFPT 2003), pp. 375-378, 2003

[Martínez, 2012] Martínez, J. J.; "Diseño e implementación sobre hardware reconfigurable de una arquitectura para la emulación en tiempo real de redes neuronales celulares", Tesis Doctoral, Universidad Politécnica de Cartagena, 2012

[Matlab, 2014] MATrix LABoratory de MathWorks, *http://www.mathworks.com*, última visita el 11 de junio de 2014

[Maxfield, 2004] Maxfield, C; *The Design Warrior's Guide to FPGAs*, Elsevier, 2004

[Meeus et al., 2012] Meeus, W.; Van, K.; Goedemé, T.; Meel, J.; Stroobandt, D.; "An overview of today's high-level synthesis tools", *Design Automation of Embedded Systems*, vol. 16, nº 3, pp. 31-51, 2012

[Mishra et al., 2007] Mishra, A.; Zaheeruddin; Raj, K.; "Implementation of a digital neuron with nonlinear activation function using Piecewise linear approximation technique", *International Conference on Microelectronics*, pp. 279-282, 2007

[Misra y Saha, 2010] Misra, J.; Saha, I.; "Artificial_neural_networks_in_hardware:_A survey_of_two_decades_of_progress", *Neurocomputing*, vol. 74, nº 3, pp. 239-255, 2010

[MIT-BIH Arrhythmia Database, 2014] Massachusetts Institute of Technology-Beth Israel Hospital Arrhythmia Database, *http://www.physionet.org/physiobank/database/mitdb/*, última visita el 18 de diciembre de 2014

[MIT-BIH Database Distribution, 2014] Massachusetts Institute of Technology-Beth Israel Hospital Database Distribution, *http://ecg.mit.edu/*, última visita el 18 de diciembre de 2014

[Myers y Hutchinson, 1989] Myers, D; Hutchinson, R; "Efficient implementation of piecewise linear activation function for digital VLSI neural networks", *Electronics Letters*, vol. 25, nº 24, pp. 1662-1663, 1989

[Nambiar et al., 2014] Nambiar, V. P.; Khalil, M.; Sahnoun, R.; Marsono, M. N.; "Hardware implementation of evolvable block-based neural networks utilizing a cost efficient sigmoid-like activation function", *Neurocomputing*, vol. 140, nº 1, pp. 228–241, 2014

[Namin et al., 2009] Namin, A. H.; Leboeuf, K.; Wu, H.; Ahmadi, M.; "Artificial Neural Networks Activation Function HDL Coder", *IEEE International Conference on Electro/Information Technology*, pp. 387-390, 2009

[Narendra y Parthasarathy, 1991] Narendra, K.S. ; Parthasarathy, K.; "Gradient methods for the optimization of dynamic-systems containing neural networks", *IEEE Transactions on Neural Networks*, vol. 2, nº 2, pp. 252-262, 1991

[Nascimento et al., 2013] Nascimento, I.; Jardim, R.; Morgado, F.; "A new solution to the hyperbolic tangent implementation in hardware: polynomial modeling of the fractional exponential part", *Neural Computing & Applications*, vol. 23, nº 2, pp. 363-369, 2013

[Neural Network Toolbox, 2014] Neural Network Toolbox de Matlab, *http://es.mathworks.com/products/neural-network* última visita el 22 de diciembre de 2014

[ntstool, 2015] Neural Network Time Series Tool de Matlab, *http://es.mathworks.com/help/nnet/ref/ntstool.html*, última visita el 10 de julio de 2015

[Ogrenci, 2008] Ogrenci, A. S.; "Abstraction in FPGA implementation of neural networks", *9th WSEAS International Conference on Neural Networks* (NN 08), pp. 221-224, 2008

[Oh y Jung, 2004] Oh, K.; Jung K.; "GPU implementation of neural networks", *Pattern Recognition*, vol. 37, nº 6, pp. 1311-1314, 2004

[Oldfield y Dorf, 1995] Oldfield, J.; Dorf, R.; *Field-Programmable Gate Array*, John Wiley and Sons, 1995

[Oniga et al., 2007] Oniga, S.; Tisan, A.; Mic, D.; Buchman, A.; Vida, A.; "Hand postures recognition system using artificial neural networks implemented in FPGA", *30th International Spring Seminar on Electronics Technology*, pp. 507-512, 2007

[Oniga et al., 2008] Oniga, S.; Tisan, A.; Mic, D.; Buchman, A.; Vida, A.; "Optimizing FPGA implementation of Feed-Forward Neural Networks", *11th International Conference on Optimization of Electrical and Electronic Equipment*, pp. 31-36, 2008

[Oniga et al., 2009] Oniga, S; Tisan, A; Mic, D.; Lung, C.; Orha, I.; Buchman, A.; "FPGA Implementation of Feed-Forward Neural Networks for Smart Devices Development", *International Symposium on Signals, Circuits and Systems* (ISSCS 2009), pp. 401-404 , 2009

[Oniga et al., 2010] Oniga, S.; Tisan, A.; Lung, C.; Buchman, A.; Orha, I.; "Adaptive Hardware-Software Co-Design Platform for Fast Prototyping of Embedded Systems", *12th International Conference on Optimization of Electrical and Electronic Equipment*, pp. 1004-1009, 2010

[Orlowska y Kaminski, 2011] Orlowska, T.; Kaminski, M.; "FPGA Implementation of the Multilayer Neural Network for the Speed Estimation of the Two-Mass Drive System", *IEEE Transactions on Industrial Informatics*, vol. 7, nº 3, pp. 436-445, 2011

[Osorio, 2014] Osorio, C.; "Diseño con System Generator de una Red Neuronal totalmente paralelizada sobre una FPGA", Trabajo Fin de Máster, Universidad de Las Palmas de Gran Canaria, 2014

[Palnitkar, 2003] Palnitkar, S.; *Verilog HDL: A Guide to Digital Design and Synthesis*, 2ª edición, Prentice Hall, 2003

[Pedroni, 2004] Pedroni, V. A.; *Circuit Design with VHDL*, MIT Press, 2004

[Pérez et al., 2009a] Pérez, S. T.; Alonso, J. B.; Travieso, C. M.; Ferrer, M. A.; Cruz, J. F.; "Design of a synchronous FFHSS modulator on a FPGA with System Generator", *WSEAS Transactions on Circuits and Systems*, vol. 8, nº 8, pp. 641-650, 2009

[Pérez et al., 2009b] Pérez, S. T.; Alonso, J. B.; Travieso, C. M.; Ferrer, M. A.; Cruz, J. F.; "Implementation of a Fast Frequency Hopping Spread Spectrum modulator with System Generator on a FPGA", *11th WSEAS International Conference on Mathematical Methods, Computational Techniques and Intelligent Systems* (MAMECTIS '09), pp. 213-217, 2009

[Pérez et al., 2009c] Pérez, S. T.; Alonso, J. B.; Travieso, C. M.; Ferrer, M.A.; "Diseño de un modulador FFHSS síncrono sobre FPGA con System Generator", *XXIV Simposium Nacional de la Unión Científica Internacional de Radio* (URSI-2009), Acta editada en CD-ROM sin numeración de páginas, 2009

[Pérez et al., 2011a] Pérez, S. T.; Alonso, J. B.; Travieso, C. M.; "Diseño de un transceptor FFHSS usando System Generator con evaluación ante Ruido Blanco Gaussiano Aditivo", *XI Jornadas de Computación Reconfigurable y Aplicaciones* (JCRA 2011), pp. 235-242, 2011

[Pérez et al., 2011b] Pérez, S. T.; Travieso, C. M.; Alonso, J. B.; "Design Methodology with System Generator in Simulink of a FHSS Transceiver on FPGA", *Applications of MATLAB in Science and Engineering*, InTech, pp. 293-316, 2011

[Pérez et al., 2011c] Pérez, S. T.; Vásquez, J. L.; Ticay, J. R.; Alonso, J. B.; Travieso, C. M.; "Artificial Neural Network in FPGA for Pejibaye Palm Classification Using Molecular Markers", *Thirteen International Conference on Computer Aided Systems Theory* (EUROCAST 2011), pp. 438-441, 2011

[Pérez et al., 2011d] Pérez, S. T.; Vásquez, J. L.; Travieso, C. M.; Alonso, J. B.; "Artificial Neural Network in FPGA for temperature prediction", *International Conference on NoLinear Speech Processing* (NoLISP 2011), vol. 7015, pp. 104-110, 2011

[Pérez et al., 2013] Pérez, S. T.; Travieso, C. M.; Alonso, J. B.; "Design Methodology of an Equalizer for Unipolar Non Return to Zero Binary Signals in the Presence of Additive White Gaussian Noise Using a Time Delay Neural Network on a Field Programmable Gate Array", *Sensors*, vol. 13, nº 12, pp. 16829-16850, 2013

[Pérez et al., 2014a] Pérez, S. T.; Osorio, C.; Vásquez, J. L.; Alonso, J. B.; Travieso, C. M.; "Design methodology of a fully parallelized Neural Network on a FPGA", *8th WSEAS International Conference on Circuits, Systems, Signal and Telecommunications* (CSST '14), vol. 29, pp. 115-119, 2014

[Pérez et al., 2014b] Pérez, S. T.; Travieso, C. M.; Alonso, J. B.; Vásquez, J. L.; "Metodologías de diseño para dispositivos digitales programables", *I Jornadas Iberoamericanas de Innovación Educativa en el ámbito de las TIC* (InnoEducaTIC 2014), pp. 1-11, 2014

[Quartus, 2014] Quartus, *http://www.altera.com/products/software/quartus-ii/web-edition/qts-we-index.html*, última visita el 11 de junio de 2014

[Raida, 2002] Raida, Z.K.; "Modeling EM structures in the neural network toolbox of MATLAB", *IEEE Antennas and Propagation Magazine*, vol. 44, nº 6, pp. 46-67, 2002

[Reis et al., 2011] Reis, L.; Aguiar, L.; Baptista, D.; Morgado, F.; "ANGE - Automatic Neural Generator", *Lecture Notes in Computer Science*, vol. 6792, nº xx, pp. 446-453, 2011

[Rivals y Personnaz, 2000] Rivals, I.; Personnaz, L.; "Nonlinear internal model control using neural networks: Application to processes with delay and design issues", *IEEE Transactions on Neural Networks*, vol. 11, nº 1, pp. 80-90, 2000

[Saichand et al., 2008] Saichand, V.; Nirmala, D. M.; Arumugam, S.; Mohankumar, N.; "FPGA Realization of Activation Function for Artificial Neural Networks", *8th International Conference on Intelligent Systems Design and Applications* (ISDA 2008), vol. 3, pp. 159-164, 2008

[Satyanarayana et al., 1992] Satyanarayana, S.; Tsividis, Y.; Graf, H.; "A reconfigurable VLSI Neural Network", *IEEE Journal of Solid-State Circuits*, vol. 27, nº 1, pp. 67-81, 1992

[Savich et al., 2007] Savich, A. W.; Moussa, M.; Areibi, S.; "The impact of arithmetic representation on implementing MLP-BP on FPGAs: A study", *IEEE Transactions on Neural Networks*, vol. 18, nº 1, pp. 240-252, 2007

[Schmit y Chandra, 2005] Schmit, H.; Chandra, V.; "Layout Techniques for FPGA Switch Blocks", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, nº 1, pp. 96-105, 2005

[Selow et al., 2009] Selow, R.; Lopes, H. S.; Erig, C. R; "A comparison of FPGA and FPAA technologies for a signal processing application", *International Conference on Field Programmable Logic and Applications*, pp. 230-235, 2009

[Simulink, 2014] Simulink de MathWorks, *http://www.mathworks.com/products/simulink*, última visita el 11 de junio de 2014

[Strik et al., 2005] Strik, D. P. B. T. B.; Domnanovich, A. M.; Zani, L.; Braun, R.; Holubar, P.; "Prediction of trace compounds in biogas from anaerobic digestion using the MATLAB Neural Network Toolbox", Environmental Modelling & Software, vol. 20, nº 6, pp. 803-810, 2005

[Synplify Pro, 2014] Synplify Pro de Synopsys, *http://www.synopsys.com/Tools/Implementation/FPGAImplementation/FPGASynthesis/Pages/SynplifyPro.aspx*, última visita el 4 de junio de 2014

[Synplify, 2014] Synplify, *http://www.synopsys.com/Tools/Implementation/FPGAImplementation/FPGASynthesis/Pages/SynplifyFeatureComparisonChart.aspx*, última visita el 11 de junio de 2014

[System Generator, 2014] System Generator for DSP, *http://www.xilinx.com/products/design-tools/vivado/integration/sysgen.html*, última visita el 11 de junio de 2014

[Tarapuez, 2015] Tarapuez, R. A.; "Diseño e implementación de un prototipo de entrenamiento basado en FPGA para las modulaciones analógicas", Proyecto Final de Carrera, Universidad de Las Palmas de Gran Canaria, 2015

[Tebbe, 1994] Tebbe, D.; Doner, J.; Billhartz, T.; *Neural Network Communications Signal Processing*, Harris Corporation, Government Communications Systems Division, 1994

[Ticay et al., 2011] Ticay, J. R.; del Pozo, M.; Travieso, C. M.; Arroyo, J.; Pérez, S. T.; Alonso, J. B.; Mora, F.; "Pollen Classification Based on Geometrical, Descriptors and Colour Features Using Decorrelation Stretching Method", *Artificial Intelligence Applications and Innovations 2011 Conference* (AIAI 2011), páginas: 342-349, 2011

[Tisan y Cirstea, 2013] Tisan, A.; Cirstea, M.; "SOM neural network design - A new Simulink library based approach targeting FPGA implementation", *Mathematics and Computers in Simulation*, vol. 91, nº 1, pp. 134-149, 2013

[Tommiska, 2003] Tommiska, M. T.; "Efficient digital implementation of the sigmoid function for reprogrammable logic", *IEE Proceedings-Computers and Digital Techniques*, vol. 150, nº 6, pp. 403-411, 2003

[Travieso et al., 2008] Travieso, C. M.; Briceño, J. C.; Vásquez, J. L.; Vásquez, J.; Castillo, E.; "Automatic Recognition System for Pejibaye palm DNA using SVM", *2nd European Computing Conference* (ECC'08), pp. 262-266, 2008

[Travieso et al., 2013a] Travieso, C. M.; Pérez, S. T.; Alonso, J. B.; "Using Fixed Point Arithmetic for Cardiac Pathologies Detection Based on Electrocardiogram", *Lecture Notes in Computer Science*, vol. 8112, nº 2, pp. 242–249, 2013

[Travieso et al., 2013b] Travieso, C. M.; Vázquez, A.; Pérez, S. T.; Alonso, J. B.; "Cardiovascular Disease Detection implemented on Fixed Point", *Fourteenth International Conference on Computer Aided Systems Theory* (EUROCAST 2013), pp. 264-267, 2013

[Vásquez et al., 2012] Vásquez, J. L.; Travieso, C. M.; Pérez, S. T.; Alonso, J. B.; Briceño, J. C.; "Temperature prediction based on different meteorological series", *Third Global Congress on Intelligent Systems* (GCIS 2012), pp. 104-107, 2012

[Vásquez et al., 2013] Vásquez, J. L.; Pérez, S. T.; Travieso, C. M.; Alonso, J. B.; "Meteorological Prediction Implemented on Field-Programmable Gate Array", *Cognitive Computation*, vol. 5, nº 4, pp. 551-557, 2013

[Vassiliadis et al., 2000] Vassiliadis, S.; Zhang, M.; Delgado, J. G.; "Elementary Function Generators for Neural-Network Emulators", *IEEE Transactions on Neural Networks*, vol. 11, nº 6, pp. 1438-1449, 2000

[Vázquez et al., 2012] Vázquez, A.; Pérez, S. T.; Travieso, C. M.; Alonso, J. B.; "Cardiac Pathologies Detection over FPGA using Electrocardiogram", *International Conference on Bio-inspired Systems and Signal Processing* (BIOSIGNALS 2012), pp. 360-364, 2012

[Vázquez, 2011] Vázquez, A.; "Diseño de un detector automático de cardiopatías a partir de señales electrocardiográficas en tiempo real usando dispositivos digitales programables", Proyecto Final de Carrera, Universidad de Las Palmas de Gran Canaria, 2011

[Vivado, 2014] Vivado Design Suite, http://www.xilinx.com/products/design-tools/vivado/index.htm, última visita el 11 de junio de 2014

[Wang y Ma, 2001] Wang, X.; Ma, Z.; "Discussion on the methodology of neural network hardware design and implementation", *6th International Conference on Solid-State and Integrated-Circuit Technology* (acrónimo), pp. 113-116, 2001

[Web of Science, 2014] Plataforma de base de datos Web of Science, *https://apps.webofknowledge.com*, última visita el 23 de julio de 2014

[Wu y Er, 2000] Wu, S. Q.; Er, M. J.; "Dynamic fuzzy neural networks - A novel approach to function approximation", *IEEE Transactions on Systems, Man, and Cybernetics*, *Part B: Cybernetics*, vol. 30, nº 2, pp. 358-364, 2000

[Xilinx, 2014] Xilinx Corporation, *http://www.xilinx.com*, última visita el 11 de junio de 2014

[Yen et al., 2004] Yen, C. T.; Weng, W. D.; Lin, Y. T.; "FPGA realization of a neural-network-based nonlinear channel equalizer", *IEEE Transactions on Industrial Electronics*, vol. 51, nº 2, pp. 472-479, 2004

[Younis, 2012] Younis, S.; "Synchronization Algorithms and Architectures for Wireless OFDM Systems", Tesis Doctoral, Universidad de Newcastle, 2012

[Zamanlooy y Mirhassani, 2014] Zamanlooy, B.; Mirhassani, M.; "Efficient VLSI Implementation of Neural Networks with Hyperbolic Tangent Activation Function", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, nº 1, pp. 39-48, 2014

[Zhu y Sutton, 2003] Zhu, J. H.; Sutton, P.; "FPGA implementations of neural networks - A survey of a decade of progress", *13th International Conference on Field-Programmable Logic and Applications* (FPL 2003), Lecture Notes in Computer Science, vol. 2778, pp. 1062-1066, 2003