DISEÑO Y ELABORACIÓN DE UN TEXTO PARA LA PROGRAMACIÓN EN HARDWARE BASADA EN LA PLATAFORMA DE DESARROLLO FPGA SPARTAN 3A

MARGARITA ALEJANDRA REBOLLEDO COY

UNIVERSIDAD PONTIFICIA BOLIVARIANA
FACULTAD DE INGENIERÍA ELECTRÓNICA
BUCARAMANGA
2010

DISEÑO Y ELABORACIÓN DE UN TEXTO PARA LA PROGRAMACIÓN EN HARDWARE BASADA EN LA PLATAFORMA DE DESARROLLO FPGA SPARTAN 3A

MARGARITA ALEJANDRA REBOLLEDO COY

Trabajo de tesis para optar al título de
Ingeniero Electrónico

Director
CLAUDIA LEONOR RUEDA GUZMAN
Ingeniera Electrónica

UNIVERSIDAD PONTIFICIA BOLIVARIANA
FACULTAD INGENIERÍA ELECTRÓNICA
BUCARAMANGA
2010

Nota de aceptación

____

____

____

____
Presidente del Jurado

____
Jurado

____
Jurado

Bucaramanga, 23 marzo 2010

AGRADECIMIENTOS

Claudia Leonor Rueda Guzmán, ingeniera electrónica y directora de la investigación por sus valiosas orientaciones y su constante motivación por este trabajo.

CONTENIDO

1 RESUMEN .... ¡ERROR! MARCADOR NO DEFINIDO.
2 INTRODUCCIÓN **3**
3 OBJETIVOS **5**
3.1 GENERALES 5
3.2 ESPECÍFICOS 5
4 METODOLOGÍA **6**
5 MARCO TEÓRICO **7**
5.1 CONCEPTO DE FPGA 7
5.2 HISTORIA 7
5.3 CARACTERÍSTICAS 8
5.4 ARQUITECTURA 8
5.4.1 Caja de interruptores *10*
5.5 SPARTAN 3A STARTER KIT 11
5.6 VHDL 13
5.7 VERILOG 14
5.7.1 Niveles de abstracción *15*
6 ESTADO DEL ARTE **16**
7 INTRODUCCIÓN A LAS FPGA **17**
8 TECNOLOGÍAS DE PROGRAMACIÓN EN LAS FPGA **19**
8.1 TECNOLOGÍA ENLACE DE FUSIBLE (FUSIBLE LINK) 19
8.2 TECNOLOGÍA DE ENLACE ANTI-FUSIBLE (ANTIFUSE) 20
8.3 TECNOLOGÍA EPROM 20
8.4 TECNOLOGÍA EEPROM 21
8.5 TECNOLOGÍA SRAM 21
9 TARJETA DE DESARROLLO SPARTAN3A **22**
9.1 CARACTERÍSTICAS 22
9.1.1 Componentes y características disponibles *22*
9.2 DESCRIPCIÓN DE LOS COMPONENTES DE LA TARJETA DE DESARROLLO SPARTAN3A 23
9.2.1 Switch de suspensión *24*
9.2.2 Pulsadores *25*
9.2.3 Perilla de rotación *26*
9.2.4 LEDs discretos *28*
9.2.5 Fuentes de reloj *29*
9.2.6 Opciones de configuración de la FPGA *29*
9.2.7 Display LCD *30*
9.2.8 Puerto VGA *31*
9.2.9 Puerto serial RS-232 .... *33*

9.2.10 Puerto PS2 para mouse y teclado ... 34
9.2.11 Circuito de captura análoga (ADC) ... 34
9.2.12 Conversor digital análogo (DAC) ... 37

10 LENGUAJE DE PROGRAMACIÓN HDL ... 40

10.1 INTRODUCCIÓN AL HDL ... 40
10.2 INTRODUCCIÓN AL VERILOG ... 40

11 DISEÑO EN VERILOG ... 42

11.1 CONCEPTOS BASICOS ... 43
11.1.1 Módulos ... 43
11.1.2 Módulo estructural ... 43
11.1.3 Sintetización ... 44
11.1.4 Lógica combinacional ... 44
11.1.5 Lógica secuencial ... 44
11.1.6 Glitch ... 44
11.1.7 Metaestabilidad ... 44

12 ESCRITURA DEL LENGUAJE VERILOG HDL ... 46

12.1 ELEMENTOS DEL LENGUAJE ... 46
12.1.1 Comentarios ... 46
12.1.2 Identificadores ... 46
12.1.3 Palabras clave ... 47
12.1.4 Representación numérica ... 48
12.1.5 Tipos de variables ... 49
12.1.6 Operadores ... 50
12.1.7 Construcciones en Verilog ... 56
12.1.8 Estructura básica de un programa en Verilog ... 65
12.2 MÁQUINAS DE ESTADO ... 66
12.3 PICOBLAZE ... 73
12.3.1 Instrucciones lógicas ... 76
12.3.2 Instrucciones de corrimiento y rotación ... 77
12.3.3 Instrucciones aritméticas ... 78
12.3.4 Instrucciones de control de flujo del programa ... 79
12.3.5 Instrucciones de movimiento de datos ... 81
12.3.6 Instrucciones de interrupción ... 81
12.3.7 Declaraciones en Picoblaze ... 82

13 PRÁCTICAS ... 83

13.1 PRÁCTICA 1: CONCEPTOS BASICOS ... 83
13.2 PRÁCTICA 2: MAQUINAS DE ESTADO ... 84
13.3 PRÁCTICA 3: PUERTO PS2/ PICOBLAZE/ LCD ... 84
13.4 PRÁCTICA 4: PUERTO VGA ... 84
13.5 PRÁCTICA 5: CORE GENERATOR ... 85

14 PRÁCTICA 1: CONCEPTOS BASICOS ... 86

14.1 OBJETIVOS ... 86

14.2 RECURSOS UTILIZADOS 86
14.3 PREREQUISITOS 86
14.4 EXPLICACIÓN DE LA PRÁCTICA. 86
14.5 DESARROLLO DE LA PRÁCTICA. 88
14.5.1 Creación de un nuevo proyecto. 92
14.5.2 Funcionamiento del módulo contador 99
14.5.3 Módulo variador_frecuencia 102
14.5.4 Módulo LFSR_7bits 104
14.5.5 Módulo rotación_leds 105
14.5.6 Módulo control_rot_knob 107
14.5.7 Módulo principal 109
14.5.8 Ejercicios propuestos 115

15 PRÁCTICA 2: MÁQUINAS DE ESTADO 116

15.1 OBJETIVOS 116
15.2 RECURSOS A UTILIZAR 116
15.3 PREREQUISITOS 116
15.4 EXPLICACIÓN DE LA PRÁCTICA 116
15.5 DESARROLLO DE LA PRÁCTICA 117
15.5.1 Módulo control_inout 117
15.5.2 Módulo Anti_Rebote_Mec 121
15.5.3 Módulo principal 126
15.5.4 Ejercicios propuestos: 127

16 PRÁCTICA 3: PUERTO PS2/ PICOBLAZE/ LCD 128

16.1 OBJETIVOS 128
16.2 RECURSOS A UTILIZAR 128
16.3 PREREQUISITOS 128
16.4 EXPLICACIÓN DE LA PRÁCTICA 128
16.5 DESARROLLO DE LA PRÁCTICA 130
16.5.1 Módulo receptor_ps2_teclado 130
16.5.2 Módulo hex_ascii 135
16.5.3 Módulo Picoblaze_control 137
16.5.4 Módulo estructural picoblaze_practica 148
16.5.5 Ejercicios propuestos 149

17 PRÁCTICA 4: PUERTO VGA 151

17.1 OBJETIVOS 151
17.2 RECURSOS A UTILIZAR 151
17.3 PREREQUISITOS 151
17.4 EXPLICACIÓN DE LA PRÁCTICA 151
17.5 DESARROLLO DE LA PRÁCTICA 152
17.5.1 Módulo VGA_module 155
17.5.2 Módulo estructural top_vga 156
17.5.3 Módulo mov_obj 159
17.5.4 Módulo bola 161
17.5.5 Ejercicios propuestos 161

18 PRÁCTICA 5: CORE GENERATOR 163

18.1 OBJETIVOS...163
18.2 RECURSOS UTILIZADOS 163
18.3 PREREQUISITOS 163
18.4 EXPLICACIÓN DE LA PRÁCTICA 163
18.5 DESARROLLO DE LA PRÁCTICA 165
18.5.1 Módulo paralelo_serial 165
18.5.2 Módulo variador_frecuencia 172
18.5.3 Módulo estructural ctrl_FIFO 172
18.5.4 Ejercicios propuestos 174

19 CONCLUSIONES 175

20 RECOMENDACIONES 176

21 BIBLIOGRAFÍA 177

LISTA DE FIGURAS

FIGURA 1 BLOQUES LÓGICOS PROGRAMABLES (CLB) 9
FIGURA 2 MATRIZ DE SWITCHES PROGRAMABLES 9
FIGURA 3 BLOQUES DE ENTADA Y SALIDA 10
FIGURA 4 ARQUITECTURA INTERNA DE UN FPGA 11
FIGURA 5 TARJETA DE DESARROLLO SPARTAN3A 12
FIGURA 6 OPCIONES DE CONFIGURACIÓN DE LA TARJETA SPARTAN3A. 13
FIGURA 7 ESTADO INICIAL DEL ENLACE POR FUSIBLE 19
FIGURA 8 ENLACE POR FUSIBLE PROGRAMADO 19
FIGURA 9 TRANSISTOR EPROM 20
FIGURA 10TRANSISTOR EEPROM 21
FIGURA 11 ELEMENTOS A UTILIZAR EN LA TARJETA 24
FIGURA 12 SWITCH DE SUSPENSIÓN 25
FIGURA 13 PULSADORES 26
FIGURA 14 PERILLA DE ROTACIÓN 27
FIGURA 15 CODIFICADOR DE EJE 27
FIGURA 16LEDS DISCRETOS 28
FIGURA 17DISPLAY LCD 30
FIGURA 18PUERTO VGA 32
FIGURA 19CONEXIONES ENTRE LA FPGA Y EL PUERTO VGA 32
FIGURA 20CONECTORES PUERTO RS-232 33
FIGURA 21VISTA FRONTAL PUERTO PS2 34
FIGURA 22PALABRA DE SALIDA DEL ADC 35
FIGURA 23SEÑALES DE CONTROL DEL AMPLIFICADOR 35
FIGURA 24SEÑALES DE CONTROL DEL AMPLIFICADOR 36
FIGURA 25SEÑALES DE CONTROL DEL ADC 36
FIGURA 26FUNCIONAMIENTO DEL ADC 37
FIGURA 27VOLTAJE DE SALIDA DEL DAC 38
FIGURA 28SEÑALES DE CONTROL DEL DAC 38
FIGURA 29SEÑALES DE CONTROL DEL DAC 39
FIGURA 30SEÑAL SPI_MOSI 39

FIGURA 31DISEÑO BOTTOM-UP 42
FIGURA 32DISEÑO TOP-DOWN 43
FIGURA 33TIEMPO DE SETUP Y HOLD EN LAS ENTRADAS DE UN FLIPFLOP 45
FIGURA 34TABLA PALABRA CLAVE EN VERILOG 47
FIGURA 35EJEMPLOS DE REPRESENTACIONES NUMÉRICAS 49
FIGURA 36ELEMENTOS DEL PROGRAMA 50
FIGURA 37OPERADORES DE TIPO ARITMÉTICO. 50
FIGURA 38OPERADORES DE DESPLAZAMIENTO 51
FIGURA 39OPERADORES DE RELACIÓN 51
FIGURA 40OPERADORES DE IGUALDAD 52
FIGURA 41OPERADORES BIT A BIT 53
FIGURA 42OPERADORES DE REDUCCIÓN. 53
FIGURA 43OPERADORES LÓGICOS. 54
FIGURA 44OPERADORES DE CONCATENACIÓN Y REPLICACIÓN. 54
FIGURA 45ELEMENTOS DEL PROGRAMA 55
FIGURA 46ESTRUCTURA IF-ELSE 56
FIGURA 47ESTRUCTURA IF-ELSE EN HARDWARE 57
FIGURA 48EJEMPLO DE CONSTRUCCIÓN IF-ELSE EN CASCADA 57
FIGURA 49 ....EJEMPLO DE CONSTRUCCIÓN IF-ELSE TRADUCIDO A HARDWARE 57
FIGURA 50ENCODER DE PRIORIDAD CONSTRUCCIÓN IF-ELSE 58
FIGURA 51DIAGRAMA ESQUEMÁTICO DEL ENCODER DE PRIORIDAD .. 58
FIGURA 52DIAGRAMA ESQUEMÁTICO EJERCICIO 12.1.7.1 59
FIGURA 53ESTRUCTURA CASE 60
FIGURA 54EJEMPLO DEL USO DE LA ESTRUCTURA CASE 60
FIGURA 55CONSTRUCCIÓN CASE EN HARDWARE 61
FIGURA 56ENCODER DE PRIORIDAD CONSTRCCIÓN CASE 61
FIGURA 57ESTRUCTURA BÁSICA DEL BLOQUE ALWAYS 62
FIGURA 58EJEMPLO DE LÓGICA COMBINACIONAL UTILIZANDO EL BLOQUE ALWAYS 63
FIGURA 59DEFINICIÓN DE UN FLIPFLOP CON EL BLOQUE ALWAYS 63
FIGURA 60FLIPFLOP TIPO D 64

FIGURA 61FLIPFLOP TIPO D CON RESET ASÍNCRONO. 64
FIGURA 62FLIPFLOP TIPO D CON ENABLE Y RESET SÍNCRONO 65
FIGURA 63CODIGO VERILOG EJERCICIO 4... 65
FIGURA 64EJEMPLO MÓDULO CONTADOR 66
FIGURA 65DIAGRAMA DE TRANSICIÓN DE ESTADOS PARA UNA MÁQUINA TIPO MEALY Y MOORE 68
FIGURA 66MÓDULO FSM, SALIDAS MOORE Y MEALY 68
FIGURA 67DIAGRAMA ESQUEMÁTICO MÓDULO FSM............................. 70
FIGURA 68DIAGRAMA DE ESTADOS DEL MÓDULO DE DETECCIÓN SE SECUENCIA. 71
FIGURA 69CÓDIGO VERILOG DETECTOR DE SECUENCIA. 71
FIGURA 70DIAGRAMA SIMPLIFICADO DE UN MICROCONTROLADOR 74
FIGURA 71DIAGRAMA DE BLOQUES DE PICOBLAZE. 75
FIGURA 72INSTRUCCIONES LÓGICAS EN PICOBLAZE 77
FIGURA 73INSTRUCCIONES DE CORRIMIENTO Y ROTACIÓN PARA PICOBLAZE. 77
FIGURA 74INSTRUCCIONES ARITMÉTICAS PARA PICOBLAZE. 79
FIGURA 75 ..INSTRUCCIONES DE CONTROL DE FLUJO PARA PICOBLAZE. 80
FIGURA 76INSTRUCCIONES DE MOVIMIENTO DE DATOS PARA PICOBLAZE. 81
FIGURA 77INSTRUCCIONES DE INTERRUPCIONES PARA PICOBLAZE. . 82
FIGURA 78DIAGRAMA DE BLOQUES DE LA PRÁCTICA 1......................... 88
FIGURA 79WEBSITE DE XILINX 89
FIGURA 80DESCARGA ISE WEBPACK. 89
FIGURA 81CREACIÓN DE UN NUEVO REGISTRO..................................... 90
FIGURA 82FORMULARIO PARA DESCARGAS 90
FIGURA 83DESCARGA DEL WEB INSTALL 91
FIGURA 84ISE PROJECT NAVIGATOR 91
FIGURA 85ESPECIFICACIONES DE LA FPG 92
FIGURA 86ESPECIFICACIONES SPARTAN3A.. 92
FIGURA 87VENTANA PARA LA CREACIÓN DE UN NUEVO PROYECTO... 93
FIGURA 88INGRESO DE LAS PROPIEDADES DE LA FPGA 94
FIGURA 89VENTANA SOURCE.. 95

FIGURA 90VENTANA DE CREACIÓN DE NUEVO MÓDULO VERILOG....... 95
FIGURA 91NUEVO MÓDULO 96
FIGURA 92VENTANA PARA LA CREACIÓN DEL UCF 97
FIGURA 93EDICIÓN DEL ARCHIVO UCF 98
FIGURA 94FRAGMENTO DEL ARCHIVO UCF... 99
FIGURA 95INSERCIÓN DE NUEVO MÓDULO AL PROYECTO................... 99
FIGURA 96NUEVO MÓDULO CONTADOR 100
FIGURA 97MÓDULO CONTADOR EN TÉRMINOS DE HARDWARE...........101
FIGURA 98VERIFICACIÓN DE LA SINTAXIS DEL PROGRAMA..................102
FIGURA 99BARRA DE ERRORES DE ISE 102
FIGURA 100 MÓDULO VARIADOR_FRECUENCIA EN CIRCUITO HARDWARE 104
FIGURA 101 PESTAÑA DE LANGUAGES TEMPLATES 105
FIGURA 102 ASIGNACIÓN DEL MÓDULO ESTRUCTURAL.......................112
FIGURA 103 PROCESOS DE SÍNTESIS, IMPLEMENTACIÓN Y GENERACIÓN DEL ARCHIVO DE PROGRAMA 113
FIGURA 104 EJECUCIÓN DEL IMPACT..113
FIGURA 105 SELECCIÓN DEL TIPO DE PROGRAMACIÓN.......................114
FIGURA 106 CONEXIÓN ESTABLECIDA ENTRE COMPUTADOR Y FPGA. 114
FIGURA 107 SELECCIÓN DEL ARCHIVO .BIT..114
FIGURA 108 PROGRAMACIÓN EN LA FPGA 115
FIGURA 109 DIAGRAMA DE BLOQUES PARA LA PRÁCTICA 2. 117
FIGURA 110 SENSORES DEL MÓDULO CONTROL_INOUT 118
FIGURA 111 ENTRADA DE UNA PERSONA. 118
FIGURA 112 DIAGRAMA DE ESTADOS DEL MÓDULO CONTROL_INOUT 119
FIGURA 113 ENTRADA Y SALIDA DEL MODULO ANTI_REBOTE_MEC 122
FIGURA 114 DIAGRAMA DE ESTADOS PARA EL MÓDULO ANTI_REBOTE_MEC 123
FIGURA 115 DIAGRAMA DE BLOQUES PARA LA PRÁCTICA 3. 129
FIGURA 116 CÓDIGOS DEL TECLADO 131
FIGURA 117 PROTOCOLO DE COMUNICACIÓN DEL TECLADO 131

FIGURA 118 DIAGRAMA DE TRANSICIÓN DE ESTADOS PARA EL MÓDULO PS2_RECEPTOR_TECLADO 132
FIGURA 119 TIEMPOS DE FUNCIONAMIENTO EN LA PANTALLA LCD 138
FIGURA 120 DESCARGA DEL ARCHIVO KCPSM3 143
FIGURA 121 FORMULARIO DE DESCARGA 143
FIGURA 122 FIN DEL REGISTRO PARA DESCARGA 144
FIGURA 123 PANTALLA DE DESCARGA DEL ARCHIVO KCPSM3 144
FIGURA 124 DIRECCIONAMIENTO PARA INGRESAR A KCPSM3............145
FIGURA 125 INSTRUCCIONES DE EJECUCIÓN PARA KCPSM3...............145
FIGURA 126 CONJUNTO DEL DIAGRAMA DE PICOBLAZE. 146
FIGURA 127 DIAGRAMA DE BLOQUES PARA LA PRACTICA 4. 152
FIGURA 128 DIAGRAMA DE UN MONITOR DE TUBO DE RAYOS CATÓDICOS..153
FIGURA 129 BARRIDO EN UNA PANTALLA VGA 154
FIGURA 130 SEÑALES VGA_HSYNC Y VGA_VSYNC 155
FIGURA 131 DIAGRAMA DE BLOQUES PARA LA PRÁCTICA 5.................164
FIGURA 132 VENTANA PARA LA SELECCIÓN DE UNA FUENTE CORE...166
FIGURA 133 SELECCIÓN DE LA HERRAMIENTA CORE A UTILIZAR........167
FIGURA 134 IMPLEMENTACIÓN DEL FIFO: TIPO DE MEMORIA...............168
FIGURA 135 IMPLEMENTACIÓN DEL FIFO: PARÁMETROS DE PUERTOS Y TIPO DE LECTURA 169
FIGURA 136 IMPLEMENTACIÓN DEL FIFO: SELECCIÓN DE BANDERAS 169
FIGURA 137 IMPLEMENTACIÓN DEL FIFO: OPCIONES DE INICIALIZACIÓN. 170
FIGURA 138 IMPLEMENTACÓN DEL FIFO: OPCIONES DE CONTEO DE DATOS 171
FIGURA 139 IMPLEMENTACIÓN DEL FIFO: RESUMEN 171

RESUMEN GENERAL DE TRABAJO DE GRADO

TITULO: Diseño y elaboración de un texto para la programación en hardware basada en la plataforma de desarrollo Spartan3A

AUTOR(ES): Margarita Alejandra Rebolledo Coy

FACULTAD: Facultad de Ingeniería Electrónica

DIRECTOR(A): Claudia Rueda

RESUMEN

La creación del presente trabajo se debió a la falta de un texto que integrara en una misma fuente los conocimientos básicos sobre las Field Programmable Gate Array y sobre el lenguaje de programación en hardware Verilog. El texto presenta una explicación sobre los conceptos y la historia de las Field Programmable Gate Arrays, así como de los dos lenguajes más populares de descripción en hardware, profundizando en el Verilog. Además cuenta con un tema extra como el Picoblaze, todo enfocado al propósito de generar conocimientos sobre esta área que va en ascenso y para servir de incentivo al uso de las tarjetas Spartan3A que posee la Universidad Pontificia Bolivariana para este fin. El texto está enfocado a convertirse en el libro guía para un curso de Diseño en hardware implementando la programación de la tarjeta Spartan3A, el cual estaría dividido en dos secciones, una teórica y una práctica. La sección teórica tendrá una duración total de 10 horas y cubrirá ocho grandes temas que van desde la historia y creación de las Field Programmable Gate Array hasta el Verilog, centrándose en temas específicos y necesarios para dar inicio a la programación con dicho lenguaje La sección práctica tendrá una duración total de 22 horas y constara de cinco grandes prácticas que contienen cada una un conjunto de ejercicios propuestos para generar confianza en el programador y alentarlo a continuar con el desarrollo de diseños en hardware. En conclusión el texto ofrece una fuente de todos los conocimientos básicos para iniciar la programación en hardware, además al cubrir el uso del Picoblaze se provee de una gran alternativa para realizar diseños flexibles y sencillos. Como última idea, la realización de las prácticas y la explicación de las mismas buscan convertirse en un método autodidáctico para quienes así lo deseen o lo requieran.

PALABRAS CLAVES: FPGA, Verilog, Curso, Prácticas, Teoría, Programación orientada a hardware

RESUMEN GENERAL DE TRABAJO DE GRADO

TITLE: Design and elaboration of a text for hardware programming based in the development platform Spartan3A

AUTHOR(S): Margarita Alejandra Rebolledo Coy

FACULTY: Faculty of Electronic Engineering

DIRECTOR: Claudia Rueda

ABSTRACT

The primary motivation for the present work is the lack of a unified source of information regarding basic knowledge of Field Programmable Gate Arrays and corresponding hardware description languages, and more specifically, in Verilog. This work describes the concepts and history of Field Programmable Gate Arrays, and two of the most popular Hardware Description Languages, Very High Speed Integrated Circuit, and Verilog. In addition, to these topics, this work also provides details regarding topics such as Picoblaze, which may be of practical use to students, by helping them become more knowledgeable in these fields. This work awill serve as the basis to a guide-book for a course in hardware design where programming is emphasized on the Spartan3A board; this course will consist of two sections: i) a theoretical section, and ii) a practical section. The theoretical section will have a total duration of 10 hours and will cover eight major topics from the history and creation of Field Programmable Gates Array to Verilog, focusing on specific subjects required for the beginning of programming on the corresponding Language. The practical section will have a total duration of 22 hours and will consist of five major practices, each of them with a set of exercises that aim to build student programmer confidence and encourage them to continue with hardware design. In conclusión, this work provides a basic knowledge source to those interetsed in beginning hardware programming. In addition, it provides an overview of the use of Picoblaze as a mean of obtaining more flexibility in resulting designs. Finally, for the interested reader, the presented practices and its explanations can serve as an autodidactic method of learning hardware programming.

KEYWORDS:

FPGA, Verilog, Course, Practices, Theory, Hardware oriented programming

2 INTRODUCCIÓN

El enfoque de este proyecto es funcionar como una introducción a los estudiantes de ingeniería que estén comenzando a utilizar la programación en hardware en el sistema de desarrollo FPGA Spartan3A, mediante el uso de un texto y ayudas interactivas.

Mediante la lectura y comprensión de este trabajo, el estudiante interesado en aprender programación en hardware se familiarizará con temas como la historia de las FPGA, su arquitectura, su configuración, el procedimiento necesario para la programación de las mismas, los diferentes lenguajes HDL con algunas de sus más importantes características pero enfocándose de manera primaria en el lenguaje Verilog, la solución de problemas prácticos propuestos, y a seguir una forma de programación que brinde alta fiabilidad e igualmente un alto nivel de optimización de recursos dentro de la FPGA para asegurar una alta velocidad de desempeño en caso de ser esta necesaria.

Todo esto inspirado en la actual importancia que han adquirido las FPGA en las aplicaciones industriales y académicas debido a todas sus ventajas sobre los sistemas utilizados anteriormente, razón que conlleva a un uso más amplio de los lenguajes de programación en hardware necesarios para poder utilizarlas que por lo tanto se convierten en temas de estudio necesarios.

Al finalizar este texto el estudiante de ingeniería que lo haya utilizado contará con los conocimientos necesarios para resolver problemas de distintas dificultades mediante el uso de Verilog y podrá utilizar todos los recursos disponibles en la tarjeta de desarrollo Spartan3A.

El contenido total del texto se divide en los diferentes capítulos de la siguiente manera, el cuarto capítulo buscan introducir a grandes rasgos al lector en el tema de las FPGA y los lenguajes de programación en hardware Verilog y VHDL.

El quinto capítulo se refiere a otros textos que podrían servir como complemento al propósito del presente.

En el sexto y séptimo capítulo se aborda de manera específica el tema de las FPGA buscando lograr una mayor comprensión acerca de dichos dispositivos, algo que es indispensable al momento de trabajar con ellas.

A lo largo del octavo capítulo se realizará una introducción a la tarjeta de desarrollo Spartan3A a la cual irán enfocadas todas las prácticas propuestas en el texto, para reconocer sus diferentes características, posibilidades y componentes.

Los capítulos 9, 10, 11 estarán todos enfocados al lenguaje de programación en hardware Verilog presentando una introducción al mismo, recomendaciones y parámetros de diseño, elementos del lenguaje y diferentes construcciones.

De los capítulos 12 a 17 se tratarán las prácticas propuestas enfocadas a generar una comprensión y manejo tanto del lenguaje de programación en hardware Verilog, como de la tarjeta de desarrollo Spartan3A y el programa de sintetización utilizado ISE Project Navigator.

3 OBJETIVOS

3.1 GENERALES

- Brindar un texto junto con ayudas interactivas que permitan al estudiante de ingeniería electrónica realizar un estudio del sistema de desarrollo FPGA Spartan 3A para diseñar aplicaciones sobre el mismo en las diferentes áreas de la electrónica.

3.2 ESPECÍFICOS

- Brindar una página web interactiva que permita un uso del texto de desarrollo más explicito y productivo.
- Dar un diseño de prácticas que posean un nivel de dificultad acorde con las expectativas de enseñanza del proyecto y que además dichas prácticas cuenten con soluciones que eviten errores en su resolución por parte del usuario del texto.
- Brindar un recurso explicito y sencillo mediante una buena organización de los temas del libro que permita que cualquier persona interesada en el tema de las FPGA pueda encontrar en el libro una forma de empezar a construir su conocimiento de manera ordenada y progresiva.

4 METODOLOGÍA

Teoría							
introducción a las FPGA	tecnologías de programación de las FPGA	Tarjeta de desarrollo Spartan3A	Lenguaje de programación HDL	Diseño en Verilog	Escritura del lenguaje Verilog HDL	Maquinas de estado	Picoblaze
30 Minutos	30 Minutos	1 Hora	30 Minutos	30 Minutos	2 Horas	3 Horas	3 Horas

Práctica				
Conceptos básicos	Maquinas de estado	Picoblaze/ Puerto PS2/ Display LCD	Puerto VGA	CORE Generator
3 Horas	3 Horas	6 Horas	6 Horas	2 Horas

5 MARCO TEÓRICO

5.1 CONCEPTO DE FPGA

Una FPGA (del inglés Field Programmable Gate Array) es un dispositivo semiconductor que contiene bloques de lógica cuya interconexión y funcionalidad se puede programar. La lógica programable puede reproducir desde funciones tan sencillas como las llevadas a cabo por una puerta lógica o un sistema combinacional hasta complejos sistemas en un chip.

Las FPGAs se utilizan en aplicaciones similares a los ASICs (circuito integrado para aplicaciones especificas en sus siglas en ingles) sin embargo son más lentas, tienen un mayor consumo de potencia y no pueden abarcar sistemas tan complejos como ellos. A pesar de esto, las FPGAs tienen las ventajas de ser reprogramables (lo que añade una enorme flexibilidad al flujo de diseño), sus costes de desarrollo y adquisición son mucho menores para pequeñas cantidades de dispositivos y el tiempo de desarrollo es también menor.[1]

5.2 HISTORIA

Las FPGAs fueron inventadas en el año 1984 por Ross Freeman y Bernard Vonderschmitt, co-fundadores de Xilinx, y surgen como una evolución de los CPLDs, son el resultado de la convergencia de dos tecnologías diferentes, los dispositivos lógicos programables (PLDs -Programmable Logic Devices-) y los circuitos integrados de aplicación específica (ASIC -application-specific integrated circuit-). La historia de los PLDs comenzó con los primeros dispositivos PROM (Programmable Read-Only Memory) y se les añadió versatilidad con los PAL (Programmable Array Logic) que permitieron un mayor número de entradas y la inclusión de registros.

Los ASIC siempre han sido potentes dispositivos, pero su uso ha requerido tradicionalmente una considerable inversión tanto de tiempo como de dinero. Intentos de reducir esta carga han provenido de la modularización de los elementos de los circuitos, como en los ASIC basados en celdas, y de la estandarización de las máscaras.

El paso final era combinar las dos estrategias con un mecanismo de interconexión que pudiese programarse utilizando fusibles, anti fusibles o celdas RAM, como los innovadores dispositivos Xilinx de mediados de los 80. Los circuitos resultantes son similares en capacidad y aplicaciones a los PLDs más grandes, aunque hay diferencias puntuales que delatan antepasados diferentes. Además de en computación reconfigurable, las FPGAs se utilizan en

---

[1] FPGA. Internet: es.wikipedia.org/wiki/FPGA [consulta: 17 septiembre 2009]

controladores, codificadores/decodificadores y en la realización de prototipos de circuitos VLSI y microprocesadores a medida.

El primer fabricante de estos dispositivos fue Xilinx, cuyos dispositivos se mantienen como uno de los más populares en compañías y grupos de investigación. Otros vendedores en este mercado son Atmel, Altera, AMD y Motorola.[2]

5.3 CARACTERÍSTICAS

Los bloques lógicos programables de una FPGA pueden ser interconectados según las necesidades de la aplicación con la cual se desea trabajar, de esta manera el programador puede trabajar con una FPGA como un sistema que se adecua a las necesidades de la aplicación.

Una tendencia reciente ha sido combinar los bloques lógicos e interconexiones de las FPGA con microprocesadores y periféricos relacionados para formar un “Sistema programable en un chip”. Ejemplo de tales tecnologías híbridas pueden ser encontradas en los dispositivos Virtex-II PRO y Virtex-4 de Xilinx, los cuales incluyen uno o más procesadores PowerPC embebidos junto con la lógica del FPGA. El FPSLIC de Atmel es otro dispositivo similar, el cual utiliza un procesador AVR en combinación con la arquitectura lógica programable de Atmel. Otra alternativa es el empleo de núcleos de procesadores implementados haciendo uso de la lógica del FPGA. Esos núcleos incluyen los procesadores MicroBlaze y Picoblaze de Xilinx, Nios y Nios II de Altera, y los procesadores de código abierto LatticeMicro32 y LatticeMicro8.

Muchas FPGA modernas soportan la reconfiguración parcial del sistema, permitiendo que una parte del diseño sea reprogramada, mientras las demás partes siguen funcionando. Este es el principio de la idea de la "computación reconfigurable", o los "sistemas reconfigurables". Los fabricantes, además, pueden proporcionar versiones de los FPGA más baratas y menos flexibles, las cuales no pueden ser modificadas después de que se han programado con un diseño.[3]

5.4 ARQUITECTURA

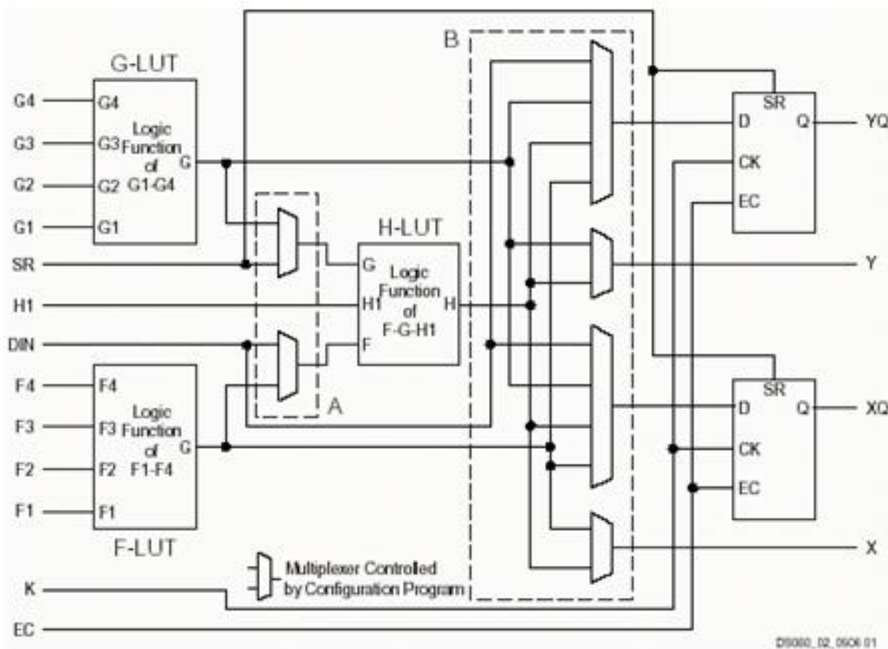
Las FPGA están compuestas principalmente por CLB o Bloques lógicos programables, los cuales a su vez poseen canales de comunicación con los demás CLB de los sistemas y las características adicionales que posee la FPGA. Las entradas de interconexión son manejadas en cada CLB en una tabla de funciones lógicas llamas “Look Up Table” que su vez se interconecta con un Flip Flop de almacenamiento que maneja la salida del CLB. En la siguiente figura puede observarse una arquitectura básica de un CLB.

---

[2] FPGA. internet: es.wikipedia.org/wiki/FPGA [consulta: 17 septiembre 2009]

[3] FPGA. internet: es.wikipedia.org/wiki/FPGA [consulta: 17 septiembre 2009]

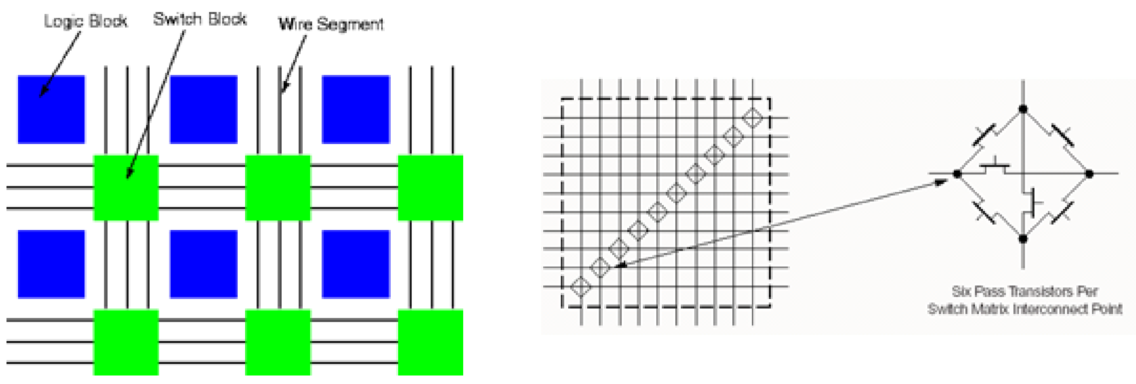
Figura 1 Bloques lógicos programables (CLB)[4]



Se pueden Implementar en cada CLB una función máxima de 5 entradas o dos independientes de 4 entradas, además se puede conectar cualquier función de Flip Flop D o Latch con 4 salidas secuenciales. Se pueden usar funciones de nueve variables, las HLUT se pueden usar como RAM 16x2.

Se cuenta además con un PSM (matriz de Switch programables), esta matriz se puede observar en la grafica siguiente, donde se visualizan las diferentes formas de conectar los CLB, se cuenta con osciladores de diferentes frecuencias según la FPGA utilizada.

Figura 2 Matriz de Switches programables[5]



---

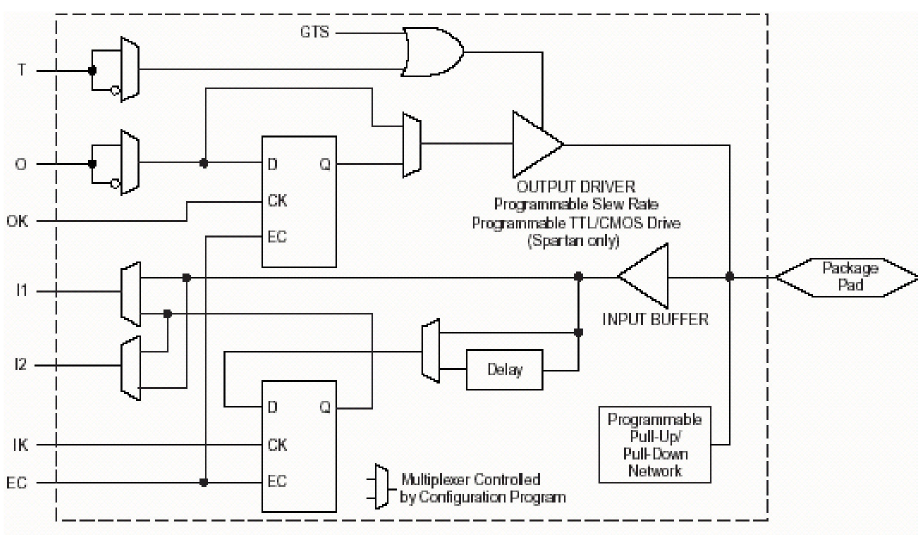
[4] XILINX. Spartan-II 2.5V FPGA Family: Functional Description, DS001-2 (v2.2) September 3, 2003

[5] XILINX. Spartan-II 2.5V FPGA Family: Functional Description, DS001-2 (v2.2) September 3, 2003

5.4.1 Caja de interruptores

Existen, además, bloques usados como interfaz entre el FPGA y otros dispositivos. Estos son llamados IOB (Bloques de Entrada/Salida) (observados en la figura 3). Los IOBs son los que definen si un pin de la FPGA será usado como entrada o como salida en el sistema implementado.

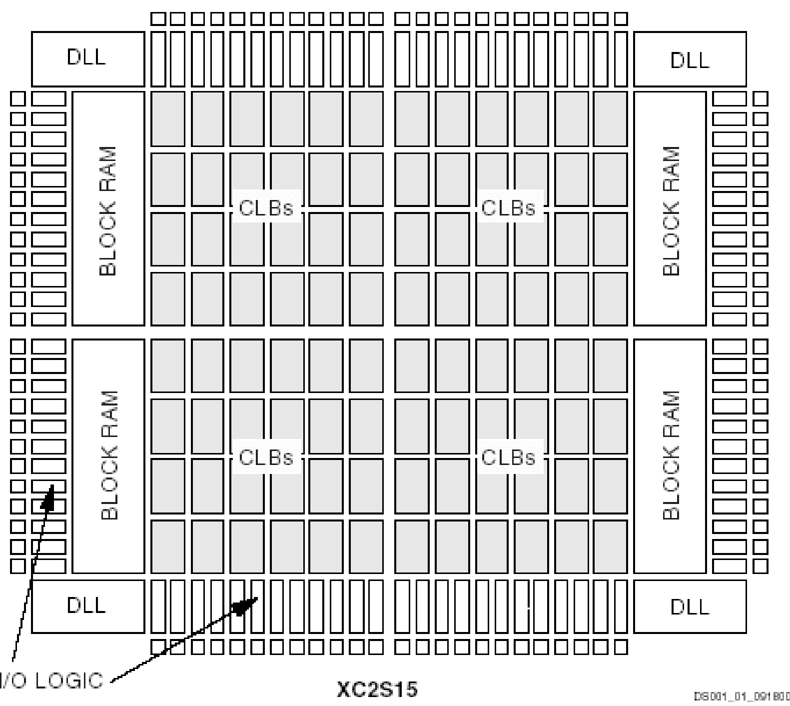
Figura 3 Bloques de entada y salida[6]



Una representación de la arquitectura interna de una FPGA en donde se pueden encontrar los CLB, los IOB y los demás componentes del sistema puede apreciarse en la siguiente figura, aquí se pueden observar el oscilador, pequeños bloques de START UP -que tienen la función de manejar la energía- y una pequeña memoria RAM.

[6] XILINX. Spartan-II 2.5V FPGA Family: Functional Description, DS001-2 (v2.2) September 3, 2003

Figura 4 Arquitectura interna de un FPGA[7]

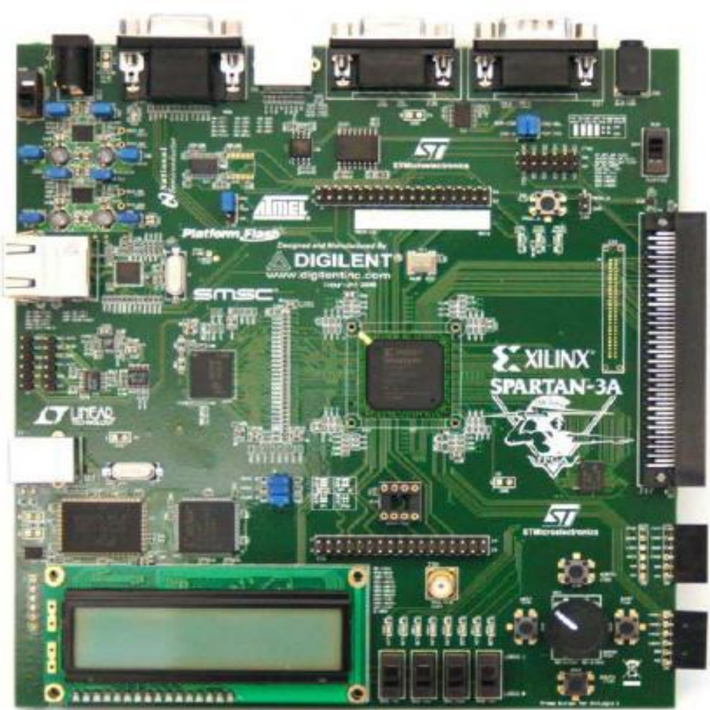


5.5 SPARTAN 3A STARTER KIT

La tarjeta de desarrollo Spartan3A puede observarse en la siguiente figura.

[7] XILINX. Spartan-II 2.5V FPGA Family: Functional Description, DS001-2 (v2.2) September 3, 2003

Figura 5 Tarjeta de desarrollo Spartan3A[8]



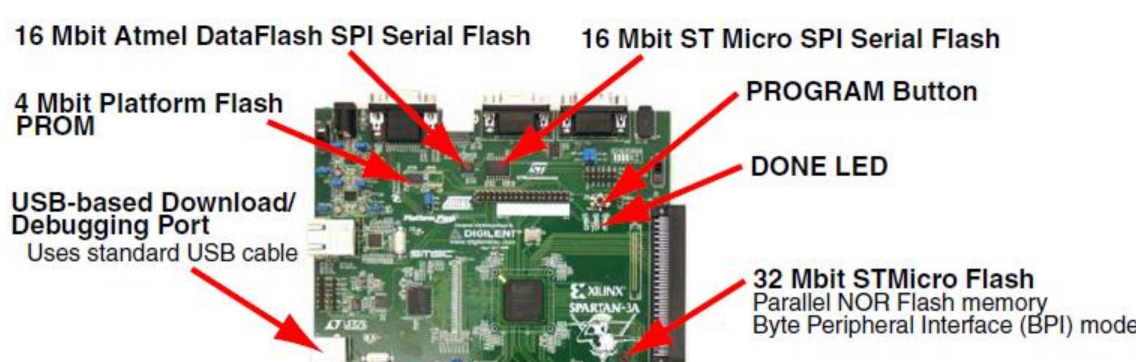
Este sistema de desarrollo puede ser usado en muchas aplicaciones, está basado en una FPGA Spartan 3A. Entre los principales componentes que presenta esta tarjeta se encuentran.

- Tres fuentes para relojes principales, la tarjeta incluye un reloj de 5OMHZ, otros relojes pueden ser incluidos mediante un conector tipo SMA (SubMiniature versión A) y opcionalmente la tarjeta incluye un espacio de 8 pines estilo DIP para acoplar un oscilador compatible.
- La tarjeta soporta una gran variedad de opciones de configuración. Los diseños pueden descargarse a la FPGA mediante el cable JTAG, utilizando el conector USB que posee la tarjeta. Se puede también programar la plataforma flash PROM que se encuentra en la tarjeta y seguidamente configurar la FPGA desde la imagen guardada en la plataforma Flash PROM usando el modo maestro serial. Mediante la programación del SPI (Serial Peripheral Interface) flash PROM de 16MBit, se puede configurar la FPGA utilizando la imagen almacenada en la memoria flash PROM utilizando el modo SPI. Una de las últimas maneras de programar la FPGA se obtiene de programar la NOR flash PROM paralela de 32 MBit y seguidamente configurar la FPGA usando el modo de configuración BPI Up.

En la figura a continuación se indica las locaciones de los elementos que permiten diferentes configuraciones

[8] XILINX, Spartan 3a fpga starter kit board user guide UG334 (v 1.1) June 19 de 2008

Figura 6 Opciones de configuración de la tarjeta Spartan3A[9]



- La tarjeta contiene una pantalla LCD de dos renglones y 16 caracteres que es controlada mediante una interfaz de 8 bits de datos, además también soporta una interfaz de 4 bits de datos para lograr ser compatible con otras tarjetas de desarrollo Xilinx.
- Cuenta con un display VGA con un conector estándar HD-DB15 permitiendo así una conexión con la mayoría de los monitores de computador.
- Dos puertos seriales RS-232, un conector tipo hembra DB9-DCE y un conector tipo macho DB9 DTE, lo que permite conectar la tarjeta a la mayoría de computadores personales o estaciones de trabajo utilizando un cable serial estándar y a otros RS-323 periféricos como una impresora.
- Incluye un puerto PS/2 para teclado y mouse y un conector mini DIN de seis pines J28.
- Un conversor análogo digital (DAC) serial LTC2624 con 12 bits de resolución, compatible con SPI y de cuatro canales.
- Una NOR flash PROM paralela que permite almacenar una configuración de la FPGA en la memoria flash, o bien varias configuraciones que pueden intercambiarse con la función Multiboot, almacena y ejecuta un código en Microblaze en la memoria flash.
- Se incluyen también una memoria DDR2 SDRAM de 16Bit, un sistema estándar LAN8700 10/100 capa física de Ethernet y un sistema de voltaje de alimentación de 5V que se consiguen con el adaptador incluido.[10]

5.6 VHDL

VHDL es un lenguaje utilizado para describir circuitos digitales que vio su origen en el programa VHSIC desarrollado por el Departamento de Defensa Americano a partir de 1980. Uno de los resultados de ese programa, fue la

[9] XILINX, Spartan 3a fpga starter kit board user guide UG330 (v 1.3) June 21 de 2007
[10] XILINX, Spartan 3a fpga starter kit board user guide UG330 (v 1.3) June 21 de 2007

necesidad de crear un lenguaje estándar capaz de describir la estructura y funcionamiento de los circuitos integrados, dando lugar al VHSIC HDL (Very High Speed Integrated Circuits Hardware Description Languaje) o VHDL. En 1987, VHDL es adoptado como estándar por la IEEE (Est. 1076-1987) arrastrando de esa forma un gran número de usuarios en todo el mundo y convirtiéndose en una referencia casi obligatoria para muchos diseñadores de circuitos digitales.

VHDL es un lenguaje de descripción y modelado de diseñado para describir la funcionalidad y la organización de los sistemas de hardware digitales, placas de circuitos y componentes. Fue desarrollado como un lenguaje para el modelado y simulación lógica dirigida por eventos de sistemas digitales y actualmente se lo utiliza también en la síntesis automática de circuitos.

VHDL es un lenguaje con sintaxis amplia y flexible que permite el modelado estructural, en flujo de datos y de comportamiento en hardware. VHDL permite el modelado preciso, en distintos estilos, del comportamiento de un sistema digital conocido y el desarrollo de modelos de simulación.

Otro de los usos de este lenguaje es la síntesis automática de circuitos. En el proceso de síntesis, se parte de una especificación de entrada con un determinado nivel de abstracción, y se llega a una implementación más detallada. La síntesis constituye hoy en día una de las principales aplicaciones del lenguaje con una gran demanda de su uso. Las herramientas de síntesis basadas en este lenguaje permiten en la actualidad ganancias importantes en la productividad y el diseño.[11]

5.7 VERILOG

Verilog fue inventado por Phil Moorby en1985 mientras trabajaba en Automated Integrated Design Systems, más tarde renombrada Gateway Design Automation. El objetivo de Verilog era ser un lenguaje de modelado de hardware. Gateway Design Automation fue comprada por Cadence Design Systems en 1990. Cadence ahora tiene todos los derechos sobre los simuladores lógicos de Verilog y Verilog-XL hechos por Gateway.

Con el incremento en el éxito de VHDL, Cadence decidió hacer el lenguaje abierto y disponible para estandarización. Se transfirió Verilog al dominio público a través de Open Verilog International, actualmente conocida como Accellera. Verilog fue después enviado a la IEEE que lo convirtió en el estándar IEEE 1364-1995, habitualmente referido como Verilog 95.

Extensiones a Verilog 95 fueron enviadas a la IEEE para cubrir las deficiencias que los usuarios habían encontrado en el estándar original de Verilog. Estas extensiones se volvieron el estándar IEEE 1364-2001 conocido como Verilog 2001.

[11] Carpio Prado. Fernando, VHDL lenguaje para descripción y modelado de circuitos, universidad de valencia

El advenimiento de los lenguajes de verificación de alto nivel como OpenVera y el lenguaje E de Verisity, impulsaron el desarrollo de Superlog, por Co-Design Automation Inc. Co-Design fue más tarde comprada por Synopsis. Las bases de Superlog y Vera han sido donadas a Accellera. Todo ello ha sido transformado y actualizado en forma de SystemVerilog, que probablemente se convierta en el próximo estándar de la IEEE.

Las últimas versiones del lenguaje incluyen soporte para modelado analógico y de señal mixta. Todos estos están descritos en Verilog-AMS[12].

Verilog, es un lenguaje de descripción en hardware (HDL), utilizado para describir sistemas digitales, tales como procesadores, memorias o un flipflop. El sistema permite la descripción el diseño en varios niveles de abstracción.

5.7.1 Niveles de abstracción

- Nivel de puerta: corresponde a una descripción de bajo nivel de diseño, también denominada modelo estructural. El diseñador describe el diseño mediante el uso de primitivas lógicas (AND, OR, etc.) y añadiendo las propiedades de tiempo de las diferentes primitivas.
- Nivel RTL: o nivel de transferencia de registro, los diseños en este nivel, especifican las características de un circuito mediante operaciones y la transferencia de datos entre registros. Mediante el uso de las especificaciones de tiempo las operaciones se realizan en instantes determinados. Este nivel le confiere al diseño la propiedad de diseño sintetizable. Todo código sintetizable se denomina código RTL
- Nivel de comportamiento: la característica principal de este nivel es su total independencia de la estructura del diseño. Se define más que la estructura el comportamiento del diseño. En este nivel, el diseño se define mediante algoritmos en paralelo. La descripción a este nivel puede hacer uso de sentencias o estructuras no sintetizables y su uso se justifica en la realización de los denominados testbench[13].

[12] Verilog. Internet:es.wikipedia.org/wiki/Verilog [consulta: 24 septiembre 2009]
[13] Tutorial Verilog. Internet: www.iuma.ulpgc.es/~nunez/clases-micros-para-com/verilog/Verilog%20Tutorial%20v1.pdf [consulta: 24 septiembre 2009]

6 ESTADO DEL ARTE

Se realizo una investigación de los trabajos relacionados con un texto para la programación de FPGA y se encontró un artículo de Stephen Brown y Jonatan Rose titulado **FPGA and CPLD Architectures: A Tutorial,** Este tutorial provee un estudio de las arquitecturas de los dispositivos de campo programables de alta capacidad FPDs (high capacity field programmable device), entre los que se encuentran las CPLD Y FPGA como tópico principal. Dando detalles de la arquitectura de los chips comerciales más importantes. Una segunda parte de este tutorial brinda resultados sobre estudios acerca de la arquitectura de los FPD durante los últimos seis años y da indicios de las posibilidades de futuras arquitecturas.

Relacionado al tema se encuentra una tesis de grado de Efrén Arenales titulada **Diseño de Guías Metodológicas en Lenguaje de Descripción de Hardware (VHDL) e Implementación en FPGA** que tiene como intención dar a conocer la versatilidad de las FPGA y abarca circuitos con lógica sencilla hasta lógica de gran complejidad en lenguaje VHDL.

La tesis titulada **Desarrollo de una herramienta web para la enseñanza de VHDL** de Edgar Eduardo Castellanos, enfocada a estudiantes con conocimientos básicos de ingeniería con el fin de aprender desde la historia del VHDL hasta el uso de este lenguaje para programar una FPGA de Xilinx, abarcando los conceptos básicos de las FPGA, su historia, su arquitectura, los elementos del lenguaje VHDL, el manejo del software y finalmente unos ejemplos para demostrar el correcto uso del lenguaje.

Otro texto titulado **The Design Warrior's Guide to FPGAs: Devices, Tools and Flows** de Max Clive enfocado a personas interesadas en el tema de las FPGA como estudiantes o personas que quieran trabajar en el campo de la FPGA con conocimientos básicos de ingeniería

Teniendo en cuenta que la Universidad Pontificia Bolivariana posee a la fecha varias tarjetas de desarrollo Spartan3A que no están siendo aprovechadas al máximo por falta de un texto que reúna dentro del mismo una explicación clara, concisa y sencilla de la tarjeta y del lenguaje de programación Verilog, se decidió dar avance al presente texto que tiene como objetivo fundar las bases de conocimiento necesarias para el manejo de la tarjeta de desarrollo Spartan3A mediante el lenguaje VerilogHDL.

7 INTRODUCCIÓN A LAS FPGA

Las FPGA (Field Programmable Gate Array) por sus siglas en ingles, vieron su aparición en el escenario de los productos electrónicos, con la empresa Xilinx alrededor de 1984, sin embargo su popularidad solo se evidencio unos años más tarde, cuando los diseñadores vieron su potencial e iniciaron a trabajar con ellas.

Cuando Xilinx introdujo la primera FPGA, se hallaban en el mercado dos tipos de componentes: PLD y ASIC.

Los PLD (Programmable Logic Device) fueron los primeros circuitos integrados disponibles que permitían modificaciones de manera rápida en caso de algún error en las funciones que ejecutaban. Estos dispositivos comenzaron con una capacidad muy limitada, como la primera PROM (Programmables Read Only Memory) en 1970, pero con el tiempo fueron evolucionando hasta que fueron divididos en dos subgrupos, los SPLD (Simple Programmable Logic Device) que fueron las primeras versiones del PLD y que soportaba arreglos simples. Después se comenzó a mencionar un nuevo término, el CPLD (Complex Programmable Logic Device) que permitió realizar funciones más complejas – de ahí su nombre- con mayor velocidad y con menor consumo de energía. Aunque los PLDs eran bastante atractivos por su capacidad de ser reprogramables, aun no podían competir contra el otro componente disponible en el mercado.

Los ASICs (Application Specific Integrated Circuit) que soportaban diseños extremadamente complejos y extensos –debido a la gran cantidad de transistores que podían contener en un espacio muy pequeño-, con un muy buen rendimiento y una alta velocidad, pero a pesar de estos puntos a su favor tenían varios problemas como su largo tiempo de fabricación, sus altos costos y el hecho de no poder ser modificados una vez terminados.

En sus primeros modelos, las FPGAs se encontraban en un estado intermedio entre los PLDs y ASICs ya que además de poseer la característica de ser programables también tenían la capacidad de soportar una gran cantidad de compuertas lógicas en un solo integrado que ocupaba poco espacio, esto permitía que desarrollase funciones largas y complejas que anteriormente solo podían realizarse con los ASICs pero con la ventaja de permitir modificar su función, otro punto favorable de las FPGA es que su precio de producción es mucho más económico que la de los ASIC en cantidades pequeñas y el tiempo de fabricación también es menor ya que su diseño es simple.

Las primeras aplicaciones que encontraron las FPGA fueron la de plataforma de prueba de diseños en software o hardware realizados por grupos pequeños de ingenieros que no podían costear los altos precios de los instrumentos

necesarios para diseñar ASICs. Las primeras funciones programadas en las FPGA, comenzaron como aplicaciones de un nivel medio que requería una capacidad baja de procesamiento, sin embargo al ir evolucionando, las FPGA comenzaron a abarcar mas campos de acción que antes estaban reservados exclusivamente para los ASICs y hoy en día muchos de los fabricantes de FPGAs señalan que sus productos poseen la misma capacidad que los ASICs.
Entre los usos que tiene las FPGA en estos días se pueden mencionar a modo de ejemplo:

- Plataforma de pruebas para nuevos diseños en hardware como base para la creación de ASICs, o que si se quiere, pueden llegar a convertirse en el producto final también utilizando FPGA tipo OTP (One-Time Programmable).
- Haciendo uso de la gran capacidad de memoria RAM que poseen, de su paralelismo y de multiplicadores embebidos, las FPGAs están siendo implementadas en el procesamiento digital de señales obteniéndose resultados de velocidad que superan a los DSPs (Digital Signal Processors) usados tradicionalmente.
- Acciones de control que antes realizaban los microprocesadores ahora pueden ser ejecutadas por las FPGA gracias a los procesadores embebidos con que cuentan, haciéndolas muy utilizadas para realizar aplicaciones de control embebido
- Las últimas FPGAs que pueden soportar transceptores de alta velocidad son usadas en funciones de comunicación y creación de redes, permitiendo integrar así estas dos funciones en un solo dispositivo.
- Aprovechando el paralelismo y reconfigurabilidad, las FPGA están siendo utilizadas para acelerar en términos de hardware los algoritmos de software

8 TECNOLOGÍAS DE PROGRAMACIÓN EN LAS FPGA

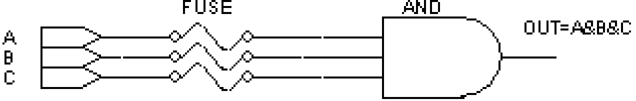
Una las razones por la cual la FPGA alcanzo una gran popularidad fue su capacidad de ser programada mediante un método rápido y ser borrada de la misma manera.

Existen varias tecnologías que pueden lograr esto, y las FPGA existentes por lo general poseen algunos de estos métodos.

8.1 TECNOLOGÍA ENLACE DE FUSIBLE (FUSIBLE LINK)

Este método ya no se utiliza en las FPGA pero es importante de mencionar ya que sentó un precedente para los dispositivos programables.
En la tecnología de enlace por fusible, cuando el dispositivo no está programado, todos los fusibles o uniones dentro del están activadas, es decir hay conexión, como se ve en la siguiente figura.

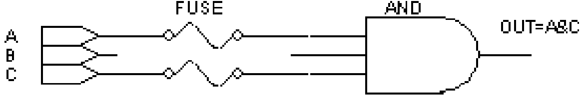
Figura 7 Estado inicial del enlace por fusible



Cuando se quiere programar el dispositivo, se eligen las conexiones que quieren ser desactivadas y mediante la aplicación de voltaje y corriente alta en el terminal que desea ser removido se puede remover esa conexión. Sin embargo este proceso es irreversible ya que una vez destruido el enlace no se puede reconstruir, esto da como resultado dispositivos OTP.

Una vez realizado el proceso de programación las conexiones quedan como lo demuestra la figura siguiente.

Figura 8 Enlace por fusible programado



8.2 TECNOLOGÍA DE ENLACE ANTI-FUSIBLE (ANTIFUSE)

Esta es una alternativa a la tecnología de enlace por fusible que se acaba de mencionar, pero en este caso en lugar de que todos los enlaces estén conectados, estos se encuentran abiertos, debido a una altísima resistencia en cada conexión.

Para programar se debe aplicar un alto voltaje y corriente en la entrada de la conexión que se desea activar, esto ocasiona que se cree un camino entre dos capas de silicio antes separadas por silicio no cristalino, permitiendo así un camino de conducción.

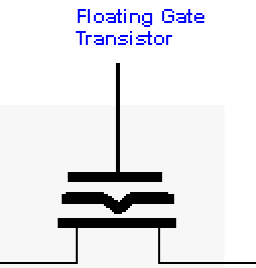
Al igual que en el enlace por fusible, este tipo de alternativa no permite una reprogramación del dispositivo ya que el camino creado no puede ser disuelto. Sin embargo esto puede ser muy útil.

8.3 TECNOLOGÍA EPROM

La tecnología EPROM (Erasable Programmable Read Only Memory), se hizo famosa ya que permitía ser programada y borrada cuantas veces se requiriera.

Su funcionamiento se logra gracias a un transistor tipo EPROM el cual, a diferentes de un CMOS normal, posee una segunda puerta como lo muestra la figura que en su estado inicial no afecta el funcionamiento normal del transistor.

Figura 9 Transistor EPROM[14]



Para programar el transistor, se requiere la aplicación de un voltaje alto entre puerta y drenador, esto obliga a los electrones de las capas de oxido que separan las dos puertas a dirigirse hacia la puerta flotante dejando una carga negativa en ella que interrumpe el funcionamiento del transistor. Esta carga puede llegar a permanecer hasta diez años.

Para restablecer el funcionamiento del transistor, es necesario aplicar radiación ultravioleta lo que descarga la puerta flotante, los integrados EPROM vienen con una ventana que permite que estos rayos entren, sin embargo aquí es donde se encuentra una de las mayores desventajas de esta tecnología, la energía necesaria para descargar la puerta es alta por lo tanto la exposición a

---

[14] EEPROM. Internet: dictionary.zdnet.com/definition/EEPROM.html [consulta: 26 septiembre 2009]

los rayos UV debe ser larga y además este tiempo se incrementa cuando la densidad de transistores lo hace.

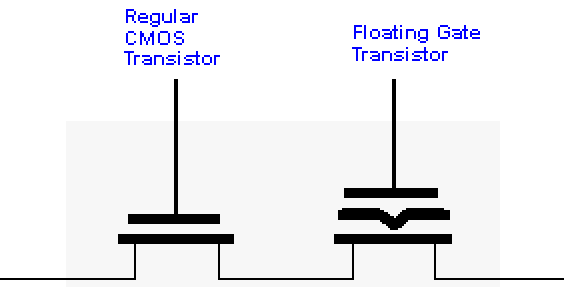
8.4 TECNOLOGÍA EEPROM

EEPROM de (Electrically Erasable Programmable Read Only Memory), es la tecnología que siguió a la EPROM.

Un transistor EEPROM es muy similar a su contraparte EPROM, pero las capas de oxido que separan a las dos puertas son mucho más delgadas, además cada celda cuenta con dos transistores, el primero es el que se utiliza para descargar la puerta del transistor EEPROM y el segundo es el que permite ser programado.

El conjunto de los dos transistores se aprecia en la figura.

Figura 10 Transistor EEPROM[15]



Este arreglo de transistores permitió tiempos de borrado mucho más cortos, aunque ocupara más espacio debido a los dos elementos utilizados y el espacio que debe existir entre ellos.

8.5 TECNOLOGÍA SRAM

SRAM se conoce como el arreglo de cuatro o seis transistores configurados como latch que permiten almacenar un valor lógico uno o cero según como sea el caso. Cada celda puede contener dicho valor hasta que sea reprogramada o el suministro de energía sea interrumpido, sin embargo esto no es un problema grande ya que este tipo de arreglo permite ser borrado o programado muy rápidamente.

Esta tecnología permite reemplazar a todas las anteriores mencionadas y es en la que está basada la FPGA que se trabajara a lo largo del texto.

---

[15] EEPROM. Internet: dictionary.zdnet.com/definition/EEPROM.html [consulta: 26 septiembre 2009]

9 TARJETA DE DESARROLLO SPARTAN3A

Esta sección sirve como una introducción a las características que ofrece la tarjeta de desarrollo con la cual se estará trabajando, más adelante se explicara la manera de utilizar dichas características mediante códigos Verilog.

9.1 CARACTERÍSTICAS

La tarjeta de desarrollo Spartan 3A contiene varias características y componentes que se describirán a lo largo de esta sección.

Como una introducción a las diferentes cualidades que posee la tarjeta, esta trae almacenado dentro de su memoria flash, una configuración que ejemplifica el funcionamiento de muchos de los dispositivos que están disponibles en este modelo, como ejemplo se pueden nombrar los puertos VGA y seriales además de características como el Multiboot y el modo ahorro de energía.

9.1.1 Componentes y características disponibles[16]

Dentro de los componentes de la tarjeta se pueden encontrar:

- FPGA Xilinx xc3s700a 700k
- Plataforma de 4 MBit de configuración flash PROM
- LCD de 16 líneas y 2 renglones
- Puerto PS/2 (soporta mouse y teclado al tiempo)
- Puerto VGA de 12bit de color
- 10/100 puerto Ethernet PHY (requiere Ethernet MAC en la FPGA)
- Dos puertos RS232
- Reloj oscilador de 50MHZ
- Espacio de ocho pines para agregar un oscilador adicional
- Conector SMA para otro entrada o salida de otro reloj
- Conector para expansión de 100 FX2 para hasta 43 entradas salidas de FPGA
- Conectores diferenciales de entrada salida de alta velocidad
- Dos puertos de expansión de seis pines para módulos periféricos
- DAC basado en SPI de cuatro salidas
- ADC basado en SPI de dos entradas con un pre-amplificador de ganancia programable
- Conector de audio externo usando pines entradas salidas digitales

[16] XILINX, Spartan 3a fpga starter kit board user guide UG334 (v 1.1) June 19 de 2008

- Puerto Chipscope
- Perilla con codificador de rotación y botón de pulsado
- Ocho LEDs discretos
- Cuatro switch de desplazamiento
- Cuatro pulsadores

Entre las características con las que cuenta esta tarjeta podemos encontrar las siguientes:

- Programación por puerto USB
- Programación directa en el sistema mediante el SPI serial Flash
- 64 Mbyte(512Mbit) de DDR2 SDRAM, 32Mx16 interfaz de datos
- Mbyte de memoria paralela Nor flash
- Almacenamiento de la configuración de la FPGA (con característica Multiboot)
- Almacenamiento de código Microblaze
- Interfaz de datos x8 o x16 después de la configuración
- Dos 16Mbits SPI flash seriales
- Arquitecturas seriales STMicroelectronics y Atmel de dataflash

Como características extras de este modelo de tarjeta se puede contar con:

- Varias formas para configurar la FPGA
- Voltaje para todas las aplicaciones gracias a un regulador incluido

9.2 DESCRIPCIÓN DE LOS COMPONENTES DE LA TARJETA DE DESARROLLO SPARTAN3A

La tarjeta de desarrollo posee diversos elementos periféricos como puertos VGA, PS2, RS-232, integrados como conversores análogo a digital, digital a análogo, memoria flash y algunos medios de interacción físicos con el usuario con Switches, pulsadores y display LCD. Cada uno de estos elementos está conectado a los puertos de entrada o salida del chip de la FPGA lo que permite realizar un control de dichas unidades.

A lo largo del texto se utilizaran algunos de los elementos contenidos dentro de la tarjeta de desarrollo Spartan3A, con el propósito de construir conocimientos sobre la tarjeta y el lenguaje de programación Verilog.

Aunque las conexiones físicas entre los elementos periféricos y la FPGA ya existen es necesario hacer que la FPGA reconozca que unidad se encuentra conectada a que pin de entrada o salida para así contar con un camino de control. Esta tarea de ubicación de los puertos I/O de la FPGA se logra mediante el uso de un archivo UCF (User Constraints File).

El archivo UCF será el que permita que la FPGA tenga conexión con los elementos periféricos de la tarjeta que se quieran utilizar, dicho archivo contiene líneas especificas de código diferentes para cada una de las unidades que permiten la activación de los puertos de entrada o salida de la FPGA

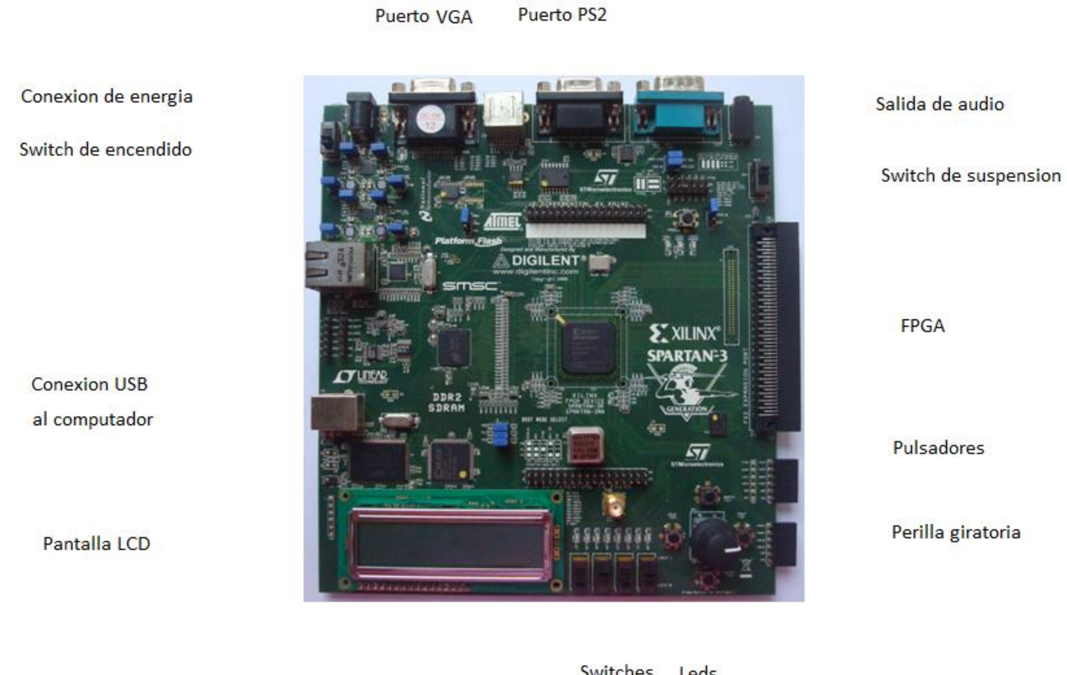
conectados a los periféricos además de describir características como nivel de voltaje que utilizaran y tipo de conexiones de resistencias, los códigos correspondientes a la tarjeta de desarrollo Spartan3A de cada uno de los elementos utilizados serán dados junto con una breve descripción.

En caso tal de que se quiera conocer la totalidad de las líneas de código del archivo UCF, Xilinx lo ofrece en su página web en el siguiente enlace: http://www.xilinx.com/products/boards/s3astarter/files/s3astarter.ucf

En las siguientes secciones se describirán los componentes que se utilizaran a lo largo de todo el texto para crear un conocimiento previo a su uso y hacer más sencillo las explicaciones futuras.

La distribución de cada uno de los elementos de la tarjeta de desarrollo Spartan3A a utilizar a lo largo del presente texto se puede observar en la siguiente figura.

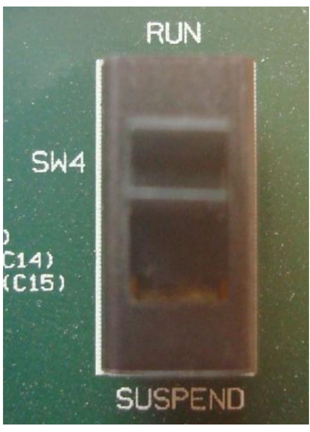
Figura 11 Elementos a utilizar en la tarjeta



9.2.1 Switch de suspensión

A continuación se observa el switch de suspensión que posee la tarjeta.

Figura 12 Switch de suspensión



Este switch tiene dos opciones en su operación, RUN y SUSPEND.

Al estar activada la opción de ahorro de energía, y si el switch se encuentra en estado RUN, la FPGA continua realizando la acción que debe ejecutar, sin embargo si el switch es posicionado en estado SUSPEND la FPGA entra en modo de suspensión. Cuando la opción de ahorro de energía no está activada el switch no interrumpe el funcionamiento normal del programa consignado en la FPGA.

Para activar o desactivar la función de este switch se debe recurrir al archivo UCF. Ya que a lo largo del texto no se utiliza esta característica de la tarjeta su función debe ser desactivada, el comando que se debe registrar en el archivo UCF para cumplir con esta condición es:

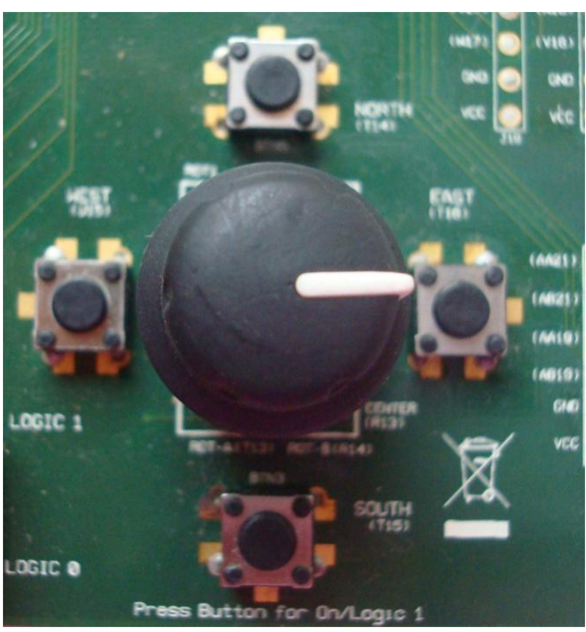
`CONFIG ENABLE_SUSPEND = ”NO”`[17]

9.2.2 Pulsadores

Se pueden observar los pulsadores en la siguiente figura.

[17] XILINX, Spartan 3a fpga starter kit board user guide UG334 (v 1.1) June 19 de 2008

Figura 13 Pulsadores



Son cuatro los pulsadores con los que se puede trabajar en la tarjeta, cada uno de ellos necesita en su configuración en el archivo UCF una resistencia pull-down para asegurar una conexión a tierra cuando el pulsador no se esté activando, para ello la configuración de cada uno debe ser la siguiente:

```
NET "BTN_EAST" LOC = "T16" | IOSTANDARD = LVCMOS33 | PULLDOWN ;
NET "BTN_NORTH" LOC = "T14" | IOSTANDARD = LVCMOS33 | PULLDOWN ;
NET "BTN_SOUTH" LOC = "T15" | IOSTANDARD = LVCMOS33 | PULLDOWN ;
NET "BTN_WEST" LOC = "U15" | IOSTANDARD = LVCMOS33 | PULLDOWN ;[18]
```

En la configuración anterior el indicador NET constituye el nombre que recibirá el elemento al momento de declararlo como entrada o salida dentro del programa en Verilog, el indicador LOC corresponde al pin de la FPGA al cual está conectado el elemento, el indicador IOSTANDAR se refiere al nivel de voltaje que manejara el dispositivo –en este caso la FPGA solo maneja niveles de 3.3 v- y por último la declaración PULLDOWN describe que el elemento al no estar activo estará conectado a tierra mediante una resistencia lo que mantendrá su nivel de salida en cero.

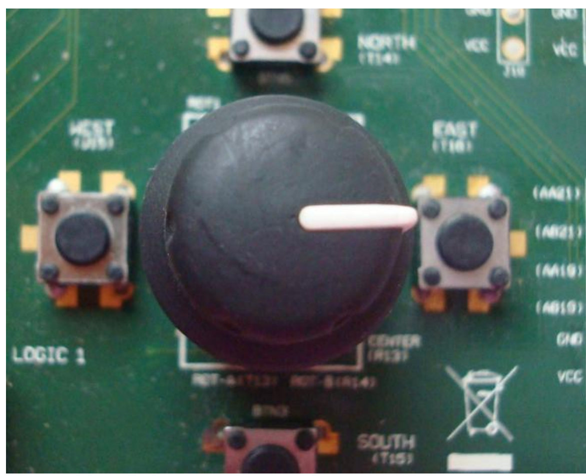
Estas instrucciones del archivo UCF por lo general siguen el mismo estándar para los demás dispositivos.

9.2.3 Perilla de rotación

La perilla de rotación puede observarse en la siguiente figura

[18] XILINX, Spartan 3a fpga starter kit board user guide UG334 (v 1.1) June 19 de 2008

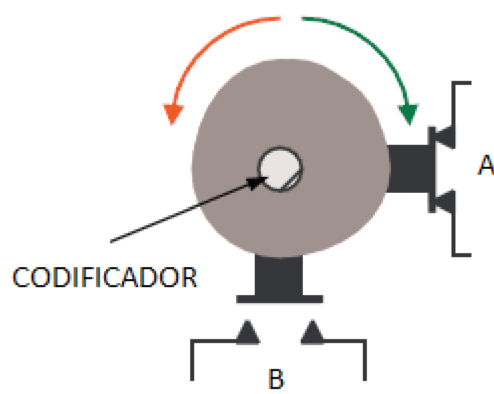
Figura 14 Perilla de rotación



Este dispositivo produce tres salidas, dos provenientes de un codificador de eje que posee denominadas ROT_A y ROT_B y la salida del pulsador ROT_CENTER, dicho pulsador presenta las mismas características de los pulsadores mencionados anteriormente. El funcionamiento del codificador se explica a continuación.

La estructura física del codificador es como lo muestra la siguiente figura, esta cuenta con una leva que activa unos pulsadores con su movimiento.

Figura 15 Codificador de eje[19]



El funcionamiento del codificador está basado en una leva conectada a su eje. Rotando el eje, la leva activa dos pulsadores A y B, la dirección de giro se puede determinar fácilmente dependiendo del orden de activación de los pulsadores.

Se debe definir una resistencia pull-up o pull-down para las conexiones de los dos pulsadores A y B, y mantener la resistencia pull-down para la función de pulsador.

```
NET "ROT_A" LOC = "T13" | IOSTANDARD = LVCMOS33 | PULLUP ;
```

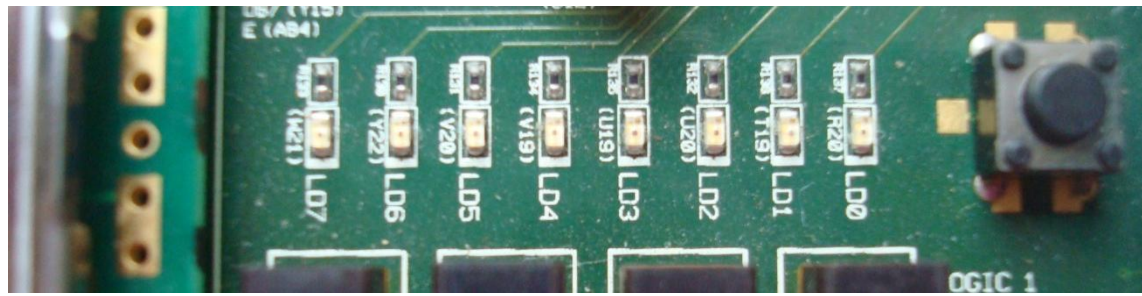
[19] XILINX, Spartan 3a fpga starter kit board user guide UG334 (v 1.1) June 19 de 2008

```
NET "ROT_B" LOC = "R14" | IOSTANDARD = LVCMOS33 | PULLUP ;
NET "ROT_CENTER" LOC = "R13" | IOSTANDARD = LVCMOS33 | PULLDOWN ;[20]
```

9.2.4 LEDs discretos

La tarjeta posee ocho LEDs con sus respectivas resistencias mostrados en la siguiente figura.

Figura 16 LEDs discretos



Los ocho LEDs son nombrados LED [7:0] y para su activación se requiere voltaje alto en el terminal del LED correspondiente.

En su configuración se describen dos nuevos elementos, la velocidad de cambio de voltaje (SLEW) y la corriente a la salida del pin de la FPGA.

Debido a la configuración con resistencia pull-up que poseen los LEDs, es posible que de no agregar ninguna instrucción en el archivo UCF se presente una leve iluminación de los LEDs.

La configuración de los LEDs es como se muestra:

```
NET "LED<7>" LOC = "W21" | IOSTANDARD = LVCMOS33 | SLEW = SLOW | DRIVE = 8 ;
NET "LED<6>" LOC = "Y22" | IOSTANDARD = LVCMOS33 | SLEW = SLOW | DRIVE = 8 ;
NET "LED<5>" LOC = "V20" | IOSTANDARD = LVCMOS33 | SLEW = SLOW | DRIVE = 8 ;
NET "LED<4>" LOC = "V19" | IOSTANDARD = LVCMOS33 | SLEW = SLOW | DRIVE = 8 ;
NET "LED<3>" LOC = "U19" | IOSTANDARD = LVCMOS33 | SLEW = SLOW | DRIVE = 8 ;
NET "LED<2>" LOC = "U20" | IOSTANDARD = LVCMOS33 | SLEW = SLOW | DRIVE = 8 ;
NET "LED<1>" LOC = "T19" | IOSTANDARD = LVCMOS33 | SLEW = SLOW | DRIVE = 8 ;
NET "LED<0>" LOC = "R20" | IOSTANDARD = LVCMOS33 | SLEW = SLOW | DRIVE = 8 ;[21]
```

Existen dos LEDs opcionales que pueden o no estar disponibles para su uso como salidas definidas por el usuario. Estos son:

- Awake LED: Este LED estará disponible para ser asignado por el usuario si en el momento de su uso la opción de suspensión se encuentra deshabilitada, de lo contrario el LED indicara el estado RUN o SUSPEND en el que se encuentra la FPGA.
- Init_B LED: Cuando la FPGA esté lista para su uso- es decir haya cargado la configuración correctamente-, este LED podrá ser manejado

[20] XILINX, Spartan 3a fpga starter kit board user guide UG334 (v 1.1) June 19 de 2008
[21] XILINX, Spartan 3a fpga starter kit board user guide UG334 (v 1.1) June 19 de 2008

como una salida asignada por el usuario. Fuera de esta condición, el Init_B, realiza las siguientes funciones:

1. Parpadea indicando que la FPGA se esta inicializando o cuando el pulsado prog_B está siendo activado para clarear la memoria de configuración.
2. Se encenderá en el caso de presentar un error durante el proceso de configuración de la FPGA.
3. Se encenderá para indicar posibles errores de CRC.

La configuración de estos dos LEDs si se desea utilizarlos es:

```
NET "FPGA_INIT_B" LOC= "V13" | IOSTANDARD= LVCMOS33 | SLEW= SLOW | DRIVE= 8 ;
NET "FPGA_AWAKE" LOC= "AB15" | IOSTANDARD= LVCMOS33 | SLEW= SLOW | DRIVE= 8 ;[22]
```

9.2.5 Fuentes de reloj

En diseños digitales es por lo general necesario contar con una señal de reloj, para suplir esta necesidad la tarjeta ofrece tres opciones para escoger la más adecuada a la aplicación a realizar. Estas son:

- Un reloj de 50MHz, cuyo chip está incluido en la tarjeta de desarrollo.
- Un conector SMA, el cual puede funcionar como un puerto para recibir una señal de reloj e igualmente para enviarla.
- Un espacio para insertar un oscilador compatible de ocho pines, lo que permite obtener fácilmente el valor de reloj deseado.

Además de las tres opciones ya mencionadas existe otra opción indirecta disponible para generar más señales de reloj, se trata del DCM (Digital Clock Manager) incluido dentro de la FPGA y que permite generar señales de otras frecuencias a partir de una dada y agregar así flexibilidad al diseño.

Para el archivo UCF, deben definirse dos características del reloj, la primera corresponde a los parámetros usuales como el puerto de entrada o salida, y el nivel de voltaje utilizado. La segunda característica que debe definirse cuando se crea el UCF de un reloj es su periodo y su Duty Cycle.

Teniendo en cuenta las nuevas características que requieren los componentes de reloj el UCF será:

```
NET "CLK_50MHZ" LOC = "E12"| IOSTANDARD = LVCMOS33 | PERIOD = 20 ns HIGH 40%;
NET "CLK_AUX" LOC = "V12"| IOSTANDARD = LVCMOS33 | PERIOD = 20 ns HIGH 40%
NET "CLK_SMA" LOC = "U12"| IOSTANDARD = LVCMOS33 | PERIOD = 20 ns HIGH 40%
```

9.2.6 Opciones de configuración de la FPGA

Las siguientes son las diferentes opciones de configuración que ofrece la tarjeta para la FPGA.

[22] XILINX, Spartan 3a fpga starter kit board user guide UG334 (v 1.1) June 19 de 2008

- Descargar los diseños directamente a la FPGA a través del cable JTAG usando la interfaz USB que se encuentra en la tarjeta. Este configura directamente la FPGA, la plataforma Flash PROM o la CPLD
- Programar la plataforma serial flash PROM de 4Mbit (XCF04S), después configurar la FPGA a partir de la imagen almacenada en la memoria usando el modo maestro serial (Master Serial Mode)
- Programar el SPI flash PROM serial de STMicroelectronics de 16Mbit que se encuentra en la tarjeta o, el SPI-based Dataflash PROM de 16 Mbit de Atmel, después configurar la FPGA con la imagen guardada en la SPI flash PROM serial usando el modo SPI
- Programar la NOR FLASH PROM paralela de 32 MBit de STMicroelectronics incluida en la tarjeta, después configurar la FPGA desde la imagen guardada utilizando el modo BPI up[23]

Si se desea aplicar alguna de las configuraciones nombradas, es necesario conocer cuál debe ser la ubicación correcta del jumper que se encuentran en la tarjeta, ya que estos son los encargados de definir el modo (Master Serial Mode, SPI o BPI) que tendrá la tarjeta al iniciarse.

9.2.7 Display LCD

El display LCD puede observarse en la figura siguiente:

Figura 17 Display LCD

La tarjeta posee una pantalla LCD de 16 columnas y 2 renglones, que está conectada a la tarjeta para tener una interfaz de ocho bits para su control, aunque podría configurarse para utilizar solo cuatro bits. Esta opción da un mayor grado de flexibilidad al diseño porque lo hace compatible con otros modelos de tarjetas de desarrollo. En este caso, sin embargo, el estudio estará enfocado a la configuración de ocho bits.

[23] XILINX, Spartan 3a fpga starter kit board user guide UG334 (v 1.1) June 19 de 2008

La pantalla LCD, posee un controlador grafico ST7066U, el cual consta de tres regiones de memoria que permiten la visualización de los caracteres deseados en el LCD, además brindar la opción de personalizar algunos de ellos. Los sectores en los que se divide son los siguientes.

- DDRAM (Display Data RAM): Es el sector encargado de almacenar el código del carácter que se va a desplegar en la pantalla.
- CG ROM (Character Generator ROM): Aquí se guardan los datos de todos los posibles caracteres que se pueden visualizar en la pantalla.
- CG RAM (Character Generator RAM): Provee de un espacio que permite personalizar algunos caracteres, pudiendo llegar a almacenar hasta ocho nuevos caracteres que cumplan con el tamaño de 5 puntos por 8 líneas.

Al momento de utilizar la pantalla, se debe tener en cuenta que los niveles de voltaje de la FPGA son del tipo LVCMOS33, y por tanto el display LCD debe configurarse para aceptar dichos niveles.

```
NET "LCD_E" LOC = "AB4" |IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = SLOW;
NET "LCD_RS" LOC = "Y14" |IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = SLOW;
NET "LCD_RW" LOC = "W13" |IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = SLOW;
NET "LCD_DB<7>" LOC = "Y15" |IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = SLOW;
NET "LCD_DB<6>" LOC = "AB16" |IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = SLOW;
NET "LCD_DB<5>" LOC = "Y16" |IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = SLOW;
NET "LCD_DB<4>" LOC = "AA12" |IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = SLOW;
NET "LCD_DB<3>" LOC = "AB12" |IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = SLOW;
NET "LCD_DB<2>" LOC = "AB17" |IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = SLOW;
NET "LCD_DB<1>" LOC = "AB18" |IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = SLOW;
NET "LCD_DB<0>" LOC = "Y13" |IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = SLOW;[24]
```

9.2.8 Puerto VGA

El conector del puerto VGA se puede apreciar en la figura

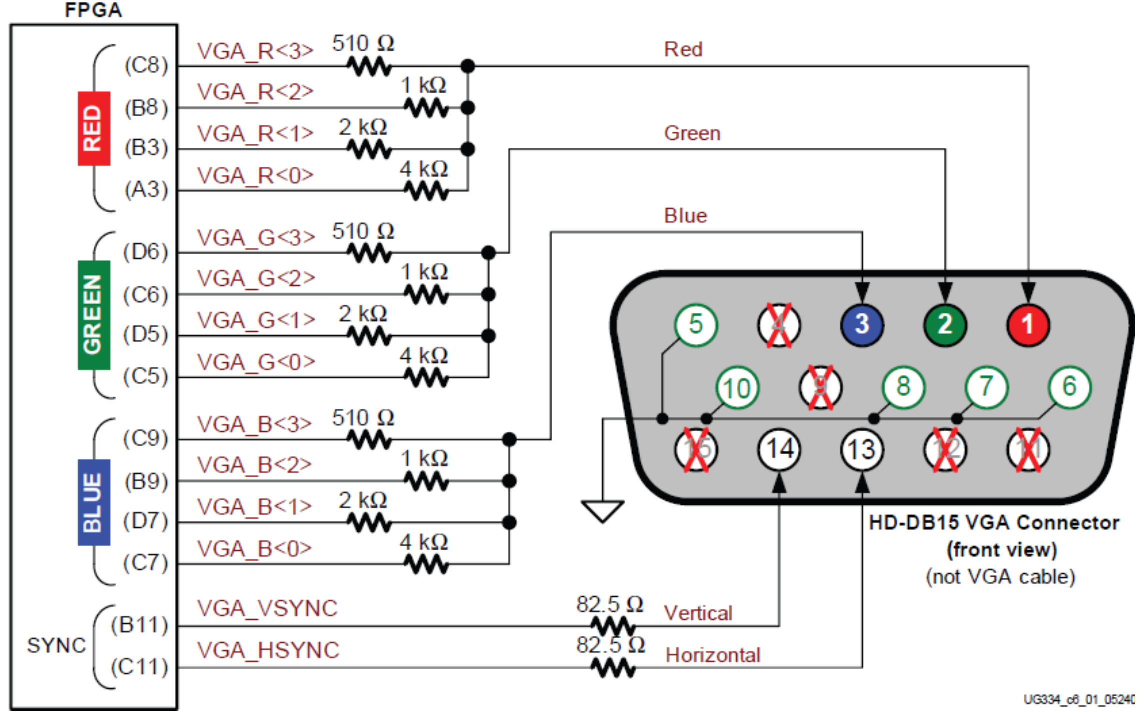
[24] XILINX, Spartan 3a fpga starter kit board user guide UG334 (v 1.1) June 19 de 2008

Figura 18 Puerto VGA

El puerto VGA que posee la tarjeta es un conector hembra HD-DB15. Cuando el conector indicado es acoplado, la FPGA directamente dirige señales a través de las resistencias que se observan en la grafica, ya que la especificación de voltaje para los puertos VGA, está estipulada entre los 0 y 0,7v, las resistencias tienen la función de mantener los voltajes en los niveles requeridos.

Ya que la especificación del puerto es de 12 bits por color, la FPGA utiliza cuatro salidas por cada color primario (azul, verde o rojo) como lo muestra la figura siguiente, esto genera una palabra de cuatro bits por color, lo que da como resultado una palabra de 12 bits en la salida del puerto.

Figura 19 Conexiones entre la FPGA y el puerto VGA[25]



[25] XILINX, Spartan 3a fpga starter kit board user guide UG334 (v 1.1) June 19 de 2008

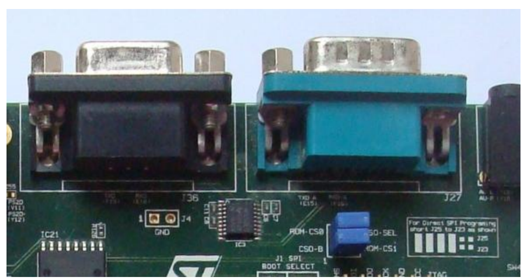
Como ya se sabe, el puerto VGA debe configurarse a niveles LVCMOS33, por lo tanto su configuración UCF debe incluir lo siguiente.

```
NET "VGA_R<3>" LOC = "C8" | IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = FAST ;
NET "VGA_R<2>" LOC = "B8" | IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = FAST ;
NET "VGA_R<1>" LOC = "B3" | IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = FAST ;
NET "VGA_R<0>" LOC = "A3" | IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = FAST ;
NET "VGA_G<3>" LOC = "D6" | IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = FAST ;
NET "VGA_G<2>" LOC = "C6" | IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = FAST ;
NET "VGA_G<1>" LOC = "D5" | IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = FAST ;
NET "VGA_G<0>" LOC = "C5" | IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = FAST ;
NET "VGA_B<3>" LOC = "C9" | IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = FAST ;
NET "VGA_B<2>" LOC = "B9" | IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = FAST ;
NET "VGA_B<1>" LOC = "D7" | IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = FAST ;
NET "VGA_B<0>" LOC = "C7" | IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = FAST ;
NET "VGA_HSYNC" LOC = "C11" |IOSTANDARD = LVCMOS33 |DRIVE = 8 | SLEW = FAST ;
NET "VGA_VSYNC" LOC = "B11" |IOSTANDARD = LVCMOS33 |DRIVE = 8 | SLEW = FAST ;
```
[26]

9.2.9 Puerto serial RS-232

El puerto serial cuenta con dos conectores diferentes que pueden apreciarse en la figura.

Figura 20 Conectores puerto RS-232



Para ampliar las posibilidades de aplicaciones que se le podrían dar a la tarjeta de desarrollo, esta incluye dos tipos de puertos seriales, el primero es el DB9-DCE, el cual permite establecer una conexión entre la FPGA y la gran mayoría de computadores o estaciones de trabajo; el segundo tipo es el DB9-DTE, que hace posible realizar una conexión entre la FPGA y otros dispositivos periféricos que manejen el protocolo RS-232 (una impresora por ejemplo).

Para funcionar, la FPGA suministra voltaje al dispositivo de comunicación serial, estos voltajes en niveles LVCMOS, el trabajo del puerto es convertir estos niveles en voltajes compatibles con el estándar RS-232 para poder enviar la información necesaria, de igual manera, al recibir algún tipo de información, este debe adecuar la señal al nivel apropiado para la FPGA.

La configuración de los puertos seriales se hace aparte una de otra y debe incluir lo siguiente.

El archivo UCF para el conector DCE es:

```
NET "RS232_DCE_RXD" LOC = "E16" | IOSTANDARD = LVCMOS33 ;
```

[26] XILINX, Spartan 3a fpga starter kit board user guide UG334 (v 1.1) June 19 de 2008

```
NET "RS232_DCE_TXD" LOC= "F15" | IOSTANDARD= LVCMOS33 | DRIVE= 8 | SLEW= SLOW
```
[27]

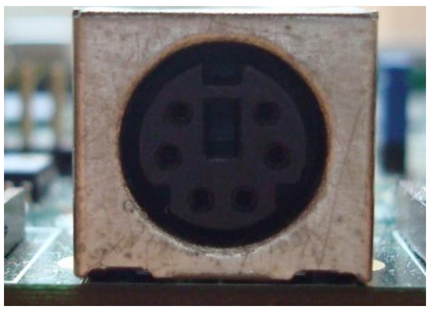
El archivo UCF para el conector DTE es:

```
NET "RS232_DTE_RXD" LOC = "F16" | IOSTANDARD = LVCMOS33 ;
NET "RS232_DTE_TXD" LOC="E15" | IOSTANDARD=LVCMOS33 | DRIVE= 8 | SLEW= SLOW ;
```
[28]

9.2.10 Puerto PS2 para mouse y teclado

El conector para el puerto PS2 incluido en la tarjeta puede observarse en la siguiente figura.

Figura 21 Vista frontal puerto PS2



El conector PS/2 estándar con el que cuenta la tarjeta, permite a la FPGA comunicarse con mouse y teclado de ser necesario. Además cuenta con la característica de permitir agregar un cable Y para conectar ambos dispositivos al tiempo.

Ya que ambos, el mouse y el teclado, utilizan protocolos de comunicación muy similares entre sí, su control mediante este conector es posible. Por ejemplo, ambos necesitan señales de reloj e indicadores de inicio y final de datos, además de poseer la característica de ser bidireccionales, lo que permite un control sobre ellos mediante la FPGA.

Para su configuración, se debe tener en cuenta que las entradas y salidas de este puerto deben estar configuradas como pull-up para evitar daños en los pines del puerto, además de especificar los niveles LVCMOS33.

```
NET "PS2_CLK1" LOC = "W12" |IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = SLOW ;
NET "PS2_DATA1" LOC = "V11" |IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = SLOW;
```
[29]

9.2.11 Circuito de captura análoga (ADC)

Para lograr una conversión análogo a digital, la tarjeta cuenta con un circuito de captura que está constituido por un amplificador de ganancia programable que

[27] XILINX, Spartan 3a fpga starter kit board user guide UG334 (v 1.1) June 19 de 2008
[28] XILINX, Spartan 3a fpga starter kit board user guide UG334 (v 1.1) June 19 de 2008
[29] XILINX, Spartan 3a fpga starter kit board user guide UG334 (v 1.1) June 19 de 2008

permite adecuar los voltajes de entrada, y un conversos análogo digital (ADC) de dos entradas.

El circuito toma la señal análoga de su entrada y mediante la siguiente expresión convierte la señal en un valor digital de 14 bits.

Figura 22 Palabra de salida del ADC[30]

$$D[13:0] = GAIN * \frac{V_{IN} - 1.65v}{1.25v} * 8192$$

En donde GAIN representa la ganancia que ha sido programada en el amplificador, Vin el voltaje aplicado a la entrada del circuito de captura, 1.65v es el voltaje de referencia del ADC y del amplificador, ±1.25v es el voltaje medio del conversor.

El valor 8192 corresponde al valor final que tendrá el resultado, es decir si la palabra es de 14 bits con complemento a dos, los valores estarán entre-2^{13} y 2^{13}-1, 2^{13} corresponde a 8192

Por separado cada componente del circuito funciona de la siguiente manera:

1. **Pre-amplificador (LTC6912-1):** El objetivo del amplificador, es el de adecuar los niveles de voltaje para el ADC. Para manejar el amplificador, la FPGA debe controlar las señales indicadas en la figura de la manera indicada.

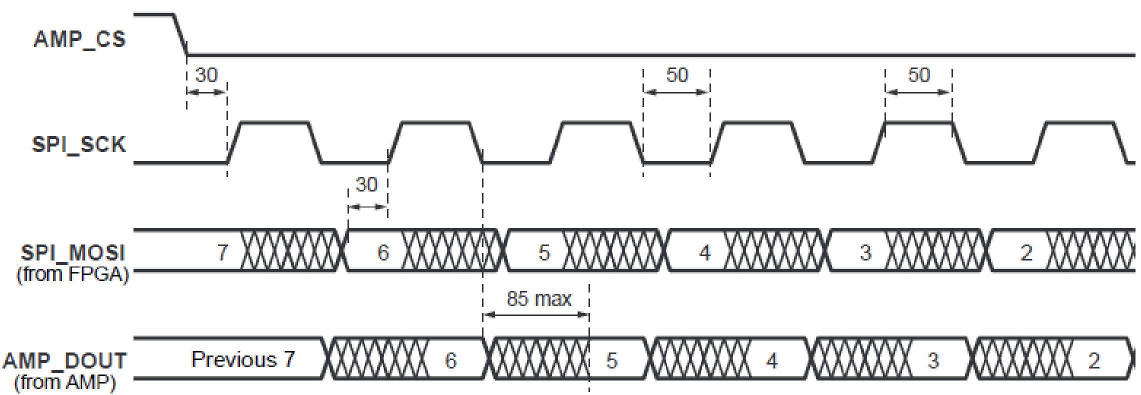
Figura 23 Señales de control del amplificador[31]

Señal	Puerto de la FPGA	Dirección de la señal	Descripción
SPI_MOSI	AB14	FPGA→AMP	Datos seriales: ocho bits de ganancia programable
AMP_CS	W6	FPGA→AMP	Activo en bajo; la ganancia del amplificador se fija en nivel alto.
SPI_SCK	AA20	FPGA→AMP	Señal de reloj
AMP_SHDN	W15	FPGA→AMP	RESET; activo en nivel alto
AMP_DOUT	T7	FPGA←AMP	Datos seriales de confirmación del dato enviado

[30] XILINX, Spartan 3a fpga starter kit board user guide UG334 (v 1.1) June 19 de 2008
[31] XILINX, Spartan 3a fpga starter kit board user guide UG334 (v 1.1) June 19 de 2008

Explicando brevemente, la comunicación SPI entre la FPGA y el amplificador, ajusta el valor requerido de la ganancia, sin embargo para lograr esto, primero la señal AMP_CS debe tomar un valor de nivel bajo, durante este intervalo se debe enviar el dato de la ganancia y una vez terminado el envío, AMP_CS debe retornar al nivel alto para que así el amplificador ajuste la nueva ganancia. Estas señales pueden verse en la siguiente figura.

Figura 24 Señales de control del amplificador[32]



La configuración que debe tener el amplificador es la siguiente.

```
NET "SPI_MOSI" LOC = "AB14"| IOSTANDARD = LVCMOS33 | SLEW = SLOW | DRIVE = 8 ;
NET "AMP_CS" LOC = "W6" | IOSTANDARD = LVCMOS33 | SLEW = SLOW | DRIVE = 8 ;
NET "SPI_SCK" LOC = "AA20"| IOSTANDARD = LVCMOS33 | SLEW = SLOW | DRIVE = 8 ;
NET "AMP_SHDN" LOC = "W15" | IOSTANDARD = LVCMOS33 | SLEW = SLOW | DRIVE = 8 ;
NET "AMP_DOUT" LOC = "T7" | IOSTANDARD = LVCMOS33 ;
```
[33]

2. **ADC (LTC1407A-1):** Se encarga de convertir simultáneamente, el valor análogo de sus dos entradas a valores digitales. El control que debe aplicar la FPGA sobre sus entradas para que funcione de manera correcta es como se observa en la figura siguiente

Figura 25 Señales de control del ADC[34]

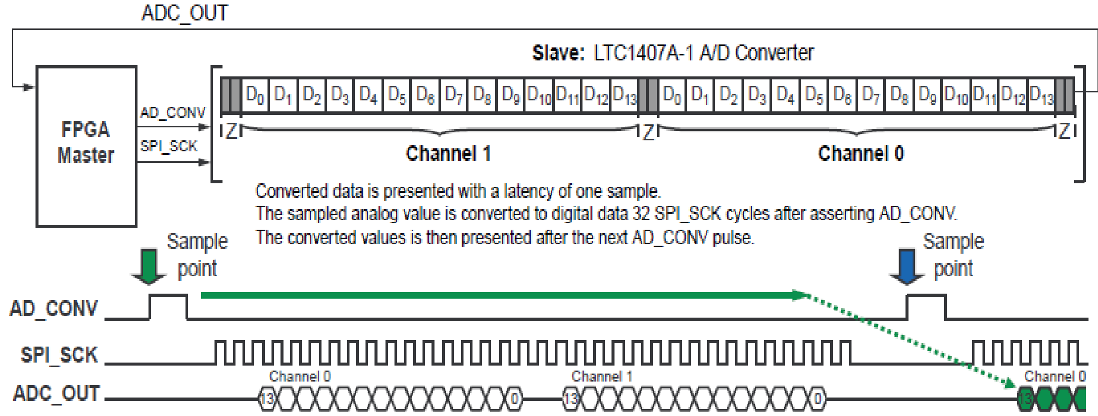
Señal	Puerto de la FPGA	Dirección de la señal	Descripción
SPI_SCK	AA20	FPGA➔ADC	Reloj
AD_CONV	Y6	FPGA➔ADC	Activo en alto, inicializa el proceso de conversión.

[32] XILINX, Spartan 3a fpga starter kit board user guide UG334 (v 1.1) June 19 de 2008
[33] XILINX, Spartan 3a fpga starter kit board user guide UG334 (v 1.1) June 19 de 2008
[34] XILINX, Spartan 3a fpga starter kit board user guide UG334 (v 1.1) June 19 de 2008

Señal	Puerto de la FPGA	Dirección de la señal	Descripción
ADC_OUT	D16	FPGA←ADC	Dato serial, presenta el valor digital de las entradas

Para iniciar su funcionamiento se debe dar un pulso en la señal AD_CONV, en este punto el ADC comenzara a muestrear los datos de sus entradas pero el resultado de esto no será obtenido sino hasta que otro pulso del AD_CONV se presente. Sin embargo hay que tener en cuenta que el ADC necesita un tiempo mínimo de 34 pulsos de reloj para completar el proceso de conversión como se ve en la figura, debe tenerse en cuenta que este requisito limita la frecuencia del dispositivo a 1.5MHz

Figura 26 Funcionamiento del ADC[35]



La configuración para el ADC es:

```
NET "AD_CONV" LOC = "Y6" | IOSTANDARD = LVCMOS33 | SLEW = SLOW | DRIVE = 8 ;
NET "SPI_SCK" LOC = "AA20" | IOSTANDARD = LVCMOS33 | SLEW = SLOW | DRIVE = 8 ;
NET "AD_DOUT" LOC = "D16" | IOSTANDARD = LVCMOS33 ;
```
[36]

9.2.12 Conversor digital análogo (DAC)

Para realizar la operación inversa del ADC, la tarjeta cuenta también con un conversor digital a análogo (DAC) LTC2625 de cuatro canales y resolución de 12 bit sin signo.

[35] XILINX, Spartan 3a fpga starter kit board user guide UG334 (v 1.1) June 19 de 2008
[36] XILINX, Spartan 3a fpga starter kit board user guide UG334 (v 1.1) June 19 de 2008

La relación que determina el voltaje de salida del conversor viene dada por los términos en la siguiente figura.

Figura 27 Voltaje de salida del DAC[37]

$$V_{OUT} = \frac{D[11:0]}{4096} * V_{referencia}$$

Donde Vout es el valor equivalente de voltaje análogo, el voltaje de referencia es 3.3v y al igual que en el ADC, 4096 corresponde al valor de la resolución es decir 2^{12}.

El DAC incluido en la tarjeta, también es compatible con la comunicación SPI, por lo tanto su funcionamiento se asemeja en gran medida al que ya se conoce.

La FPGA controla los siguientes puertos del DAC enumerados en la figura.

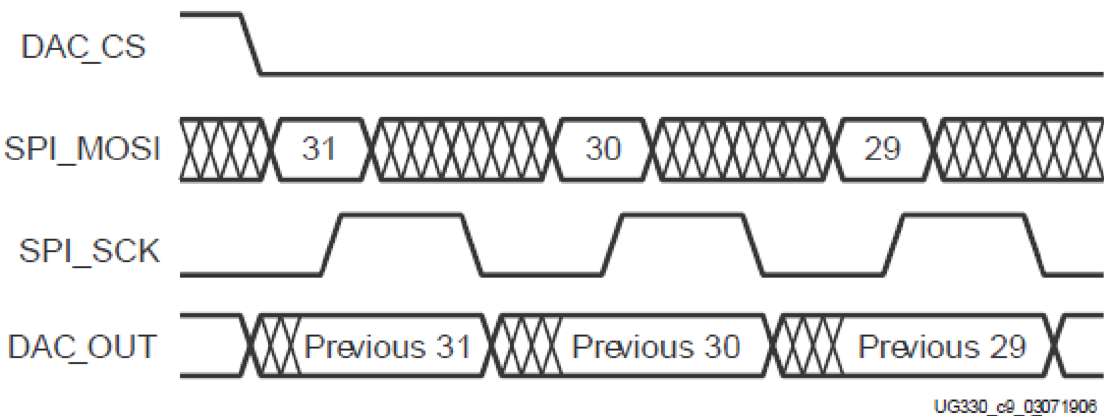
Figura 28 Señales de control del DAC[38]

Señal	Puerto de la FPGA	Dirección	Descripción
SPI_MOSI	AB14	FPGA→DAC	Datos seriales
DAC_CS	W7	FPGA→DAC	Activo en nivel bajo, la conversión comienza cuando esta señal retorna al valor alto.
SPI_SCK	AA20	FPGA→DAC	Reloj
DAC_CLR	AB13	FPGA→DAC	RESET asíncrono, activo en nivel bajo
DAC_OUT	V7	FPGA←DAC	Datos seriales

Cuando un nivel bajo es dado a la señal DAC_CS como se observa en la figura 29, la FPGA puede comenzar a transmitir los datos por medio de la señal SPI_MOSI, la cual es captada por el DAC cada vez que se presenta un flanco de subida de su señal de reloj. Una vez que todos los datos hayan sido enviados, se debe retornar a alto del valor DAC_CS para así dar inicio al proceso de conversión.

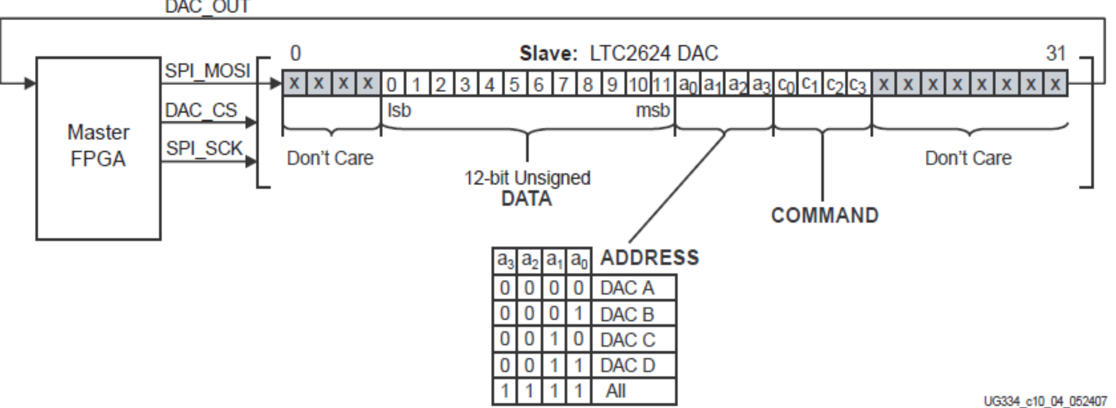
[37] XILINX, Spartan 3a fpga starter kit board user guide UG334 (v 1.1) June 19 de 2008
[38] XILINX, Spartan 3a fpga starter kit board user guide UG334 (v 1.1) June 19 de 2008

Figura 29 Señales de control del DAC[39]



Hay que tener en cuenta que la señal SPI_MOSI, no solo envía los datos que van a ser convertidos, sino que también envía datos de control al DAC en donde se indica la dirección del puerto de entrada en donde se encuentra la señal y la instrucción que deberá realizar con esos datos. Esta palabra está constituida en su totalidad por 32 bits incluyendo 12 bits "don't care", que indican el inicio y el fin del ciclo. Un diagrama de esta señal puede observarse a continuación.

Figura 30 Señal SPI_MOSI[40]



La configuración para el DAC es:

```
NET "SPI_MOSI" LOC = "AB14" | IOSTANDARD = LVCMOS33 | SLEW = SLOW | DRIVE = 8;
NET "SPI_SCK" LOC = "AA20" | IOSTANDARD = LVCMOS33 | SLEW = SLOW | DRIVE = 8 ;
NET "DAC_CS" LOC = "W7" | IOSTANDARD = LVCMOS33 | SLEW = SLOW | DRIVE = 8 ;
NET "DAC_CLR" LOC = "AB13" | IOSTANDARD = LVCMOS33 | SLEW = SLOW | DRIVE = 8 ;
NET "DAC_OUT" LOC = "V7" | IOSTANDARD = LVCMOS33 ;
```
[41]

[39] XILINX, Spartan 3a fpga starter kit board user guide UG334 (v 1.1) June 19 de 2008
[40] XILINX, Spartan 3a fpga starter kit board user guide UG334 (v 1.1) June 19 de 2008
[41] XILINX, Spartan 3a fpga starter kit board user guide UG334 (v 1.1) June 19 de 2008

10 LENGUAJE DE PROGRAMACIÓN HDL

10.1 INTRODUCCIÓN AL HDL

Para comenzar a explotar las diversas opciones que ofrecen las FPGAs, se necesita conocer las formas que tienen estas de programarse. Las FPGA pueden ser programadas utilizando software que permita crear diseños esquemáticos con la función deseada o diagramas de bloque, para después convertirlos en hardware, sin embargo cuando se pretenden realizar diseños complejos o extensos, es tedioso -por no decir muy poco probable- implementar este tipo de procedimiento. Para estos casos, existen lenguajes de programación de hardware que son útiles para describir funciones digitales complejas que después se convertirán en hardware, con diferentes niveles de abstracción, lo que permite un control detallado del hardware final.

Los lenguajes de programación utilizados para programar los PLDs se conocen como HDL (Hardware Description Languages) por sus siglas en ingles. Su aparición en escena se debió al desarrollo de PLDs que resistían diseños cada vez más complejos como los CPLDs o FPGAs. El HDL es una herramienta que permite describir funciones digitales sin tener que recurrir al uso de diagramas esquemáticos. De esta manera se simplifica considerablemente la tarea de definir un circuito; además de esta ventaja, el HDL permite manejar los detalles del circuito final de la manera en que el diseñador lo requiera, es decir, cuenta con diferentes niveles de abstracción que van desde describir a un nivel de compuertas lógicas el producto final, definir los tiempos de propagación de las señales manejadas o especificar la lista de sensibilidad, hasta implementar funciones ya definidas por el software utilizado o por otro diseñador sin tener que necesariamente entenderlas.

Actualmente existen dos lenguajes HDL muy populares, el primero es el VHDL (Very High Speed HDL) creado por el departamento de defensa de los Estados Unidos como una manera de verificar el comportamiento de los circuitos que empleaban, el segundo es el Verilog HDL creado por Philip Moorby. Ambos lenguajes están actualmente disponibles al dominio público y debido a la creciente popularidad de los PLDs han sido estandarizados por la IEEE.
Centraremos la atención en el Verilog, ya que este es el lenguaje en el cual se enfoca este texto.

10.2 INTRODUCCIÓN AL VERILOG

Verilog fue creado por Philip R Moorby cuando hacia parte de Gateway Design en 1984 para la verificación y simulación de productos. Más tarde Verilog pasó

a propiedad de Cadence Desing System cuando esta adquirió a Gateway Design[42].

Al irse popularizando el uso de los PLDs e ir apareciendo herramientas para el análisis y simulación, y otros nuevos lenguajes de descripción de hardware, Cadence decidió convertir Verilog al dominio público en 1990, lo que dió paso a que se convirtiera en el lenguaje HDL más popular entre los diseñadores y surgió la necesidad de su estandarización; por eso en 1995 se convirtió en estándar de la IEEE #1364-1995, y más recientemente en estándar IEEE #1364-2001, el cual incluye unas leves modificaciones al lenguaje original y el que se estudiara en este texto.

Verilog guarda una gran relación con los lenguajes de programación C y pascal. Además, permite al diseñador dar gran cantidad de detalles en su descripción del circuito, así como también omitir algunos, cuenta con algoritmos que permiten hacer referencia a circuitos con lógica combinacional o secuencial definiendo claramente el hardware.

[42] Verilog. internet: http://es.wikipedia.org/wiki/Verilog [consulta: 29 septiembre 2009]

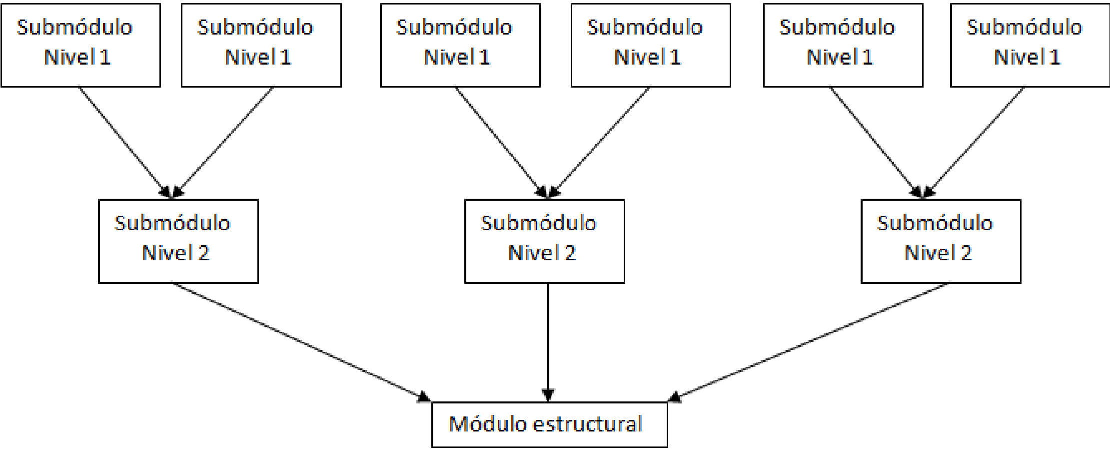
11 DISEÑO EN VERILOG

Para realizar programas en Verilog se debe tener en mente que estos diseños deben ser modulares, es decir, el producto final debe estar dividido en módulos, donde cada módulo realice una pequeña parte del trabajo sin estructuras muy complicadas, de ser así, el código creado puede ser implementado en otros dispositivos sin tener que modificarlo o ser reutilizado para otro diseño diferente. Esta es sin duda una de las características más importantes de Verilog.

Para empezar a realizar un diseño es recomendable tener el producto final bien definido antes de comenzar a programar y decidir el tipo de diseño que se implementara, sea este Bottom-up o top-down.

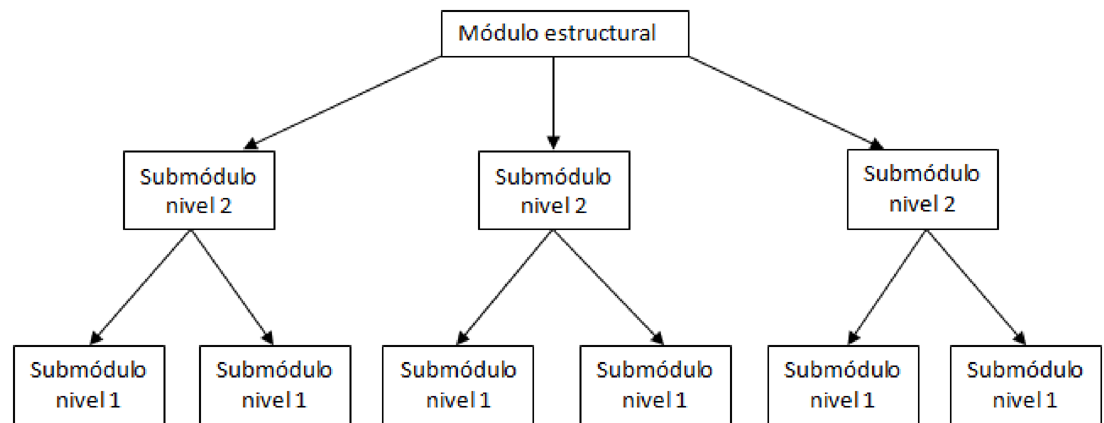
El modelo Bottom-up consiste en especificar primero módulos de bajo nivel, es decir que realicen acciones básicas diferentes y sin relación entre sí, para después incluirlos en un módulo estructural en donde se entrelacen y realicen la función deseada como lo ilustra el siguiente diagrama.

Figura 31 Diseño bottom-up



El modelo top-down es la versión inversa del anterior, en este se crea primero el módulo de alto nivel y desde ahí se comienzan a crear los submódulos, como lo muestra la figura.

Figura 32 Diseño top-down



Una vez se haya cumplido con los puntos anteriores es importante mantener presente que debido a que Verilog fue inicialmente creado como una herramienta para la verificación y simulación de productos, aunque todas las instrucciones pueden ser utilizadas en simulación, no todas son aceptadas por las herramientas de sintetización, por lo tanto no pueden ser traducidas a elementos de hardware, casi el 80% de las instrucciones utilizadas para la simulación no son sintetizables.

11.1 CONCEPTOS BASICOS

Se definirán a continuación ciertos términos para tenerlos presentes a lo largo del texto y evitar posibles confusiones en el momento de emplearlos.

11.1.1 Módulos

Este es sin duda el termino que más se debe entender cuando se habla de Verilog, ya que la modularidad es la clave del éxito en la mayoría de los diseños que se realizan. Los módulos son las unidades básicas de funcionamiento del programa, en ellos por medios de software, esta descrito un hardware que realiza una función específica. Para asegurar un diseño que sea lo más portable posible, los módulos no deben realizar más de una operación o función, esto asegura que puedan ser fácilmente instanciados creando modelos estructurales

11.1.2 Módulo estructural

Un módulo estructural es la agrupación de varios módulos para conformar operaciones complejas.

Este módulo estructural o principal deberá tener comunicación con los periféricos de la tarjeta que serán utilizados; es por eso que se debe crear un archivo UCF (User's Constraints File) por sus siglas en ingles, que contiene las instrucciones que activan o desactivan los periféricos de la tarjeta con la

configuración necesaria. En la sección de prácticas se puede observar el proceso de creación del UCF.

11.1.3 Sintetización

En términos sencillos, el proceso de sintetización de un diseño es la acción de convertir a términos de hardware las instrucciones especificadas en el software diseñado. En este paso es importante tener en cuenta que la sintetización de un programa variará dependiendo del software utilizado y de las librerías que este contenga – es decir podría no contar con todos los elementos de circuitos especificados en el programa-, es por esta razón que el diseño del producto final puede variar entre las diferentes herramientas de sintetización, sin embargo esto no implica que su función varíe ya que se el circuito será distinto pero su funcionamiento será equivalente. Teniendo en cuenta que la tecnología de las herramientas de sintetización aún no está del todo perfeccionada debido a su gran complejidad, es de gran importancia no dejar en el programa realizado espacios que no contengan una especificación concisa que conlleve a una mala interpretación por parte de la herramienta de sintetización o a optimizaciones no requeridas, ya que esto sería causa de circuitos que varíen su salida final con cada sintetización.

11.1.4 Lógica combinacional

En este tipo de lógica la salida del diseño depende únicamente de la entrada actual, es decir no tiene elementos de memoria y cualquier cambio en alguna variable tendrá efecto inmediato en la salida, es por esto que durante la especificación del circuito la salida debe ser asignada cada vez que ocurra un cambio en alguna de las variables.

11.1.5 Lógica secuencial

La lógica secuencial se diferencia de la combinacional en el hecho que su salida no solo depende de la entrada sino también de la entrada anterior, es decir, este tipo de circuitos si poseen elementos de memoria y son bastante utilizados para describir máquinas de estado.

11.1.6 Glitch

Un glitch se genera cuando un solo cambio en una variable genera cambios en más de un camino o cuando ocurren cambios simultáneos en varias variables, lo que ocasiona que durante un corto periodo de tiempo las señales adquieran valores diferentes a los esperados. Sin embargo, esto se resuelve cuando las señales han tenido suficiente tiempo para estabilizarse, que por lo general no es más de un ciclo de reloj.

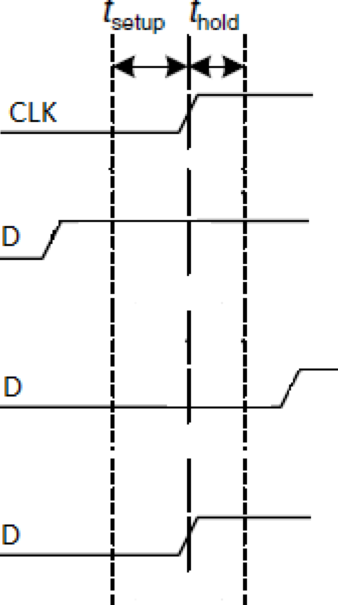
Por lo general es imposible evitar que el diseño presente glitches pero estos se deben tener presentes al momento de elegir los tiempos de un diseño para evitar que causen problemas.

11.1.7 Metaestabilidad

El estado metaestable ocurre cuando a la salida de un elemento secuencial, como un flipflop se presenta un nivel de entre uno y cero, esto en sistemas digitales constituye un error que ocasiona un mal funcionamiento de los caminos que se vean afectados por dicha salida errónea.

La razón para llegar a dicho estado se encuentra en la violación de los tiempos de setup y hold de un flipflop, que son aquellos tiempos en los cuales la señal a la entrada de un flipflop debe permanecer estable antes y después durante la transición del reloj.

Figura 33 Tiempo de setup y hold en las entradas de un flipflop



En la grafica anterior se muestran tres casos de transición a la entrada D de un flipflop. En los dos primeros se observa que la transición de estado ocurre por fuera de los tiempos de setup y hold del flipflop por lo tanto la salida tendrá un valor de uno o cero, sin embargo en el tercer caso se presenta una transición que viola los tiempos de asentamiento del flipflop, en este escenario pueden ocurrir tres resultados: el primero que el flipflop continue a su salida con un estado anterior, que el flipflop adquiera en un salida un valor aleatorio de uno o cero o que se entre a un estado metaestable.

En la mayoría de los casos la razón principal para presentar este problema se debe a la interacción de señales que no tienen relación de periodo o fase entre ellas. Por esto, cuando una varia su estado no hay forma de prevenir que esto no suceda durante el cambio de estado de la otra.

El tiempo requerido para salir de este estado metaestable y adquirir un valor de cero o uno es desconocido, pero debido a la estructura interna de un flipflop, este estado tenderá a estabilizarse en uno o cero a medida que avanza el tiempo. Por esto, aunque no se sepa con certeza cuándo terminará el estado metaestable si se puede asegurar que entre más tiempo pase, más grande será la probabilidad de que el estado metaestable desaparezca.

12 ESCRITURA DEL LENGUAJE VERILOG HDL

Verilog cuenta con ciertos elementos de lenguaje, operadores y operandos que conforman el conjunto de herramientas que permiten crear los códigos de control para manejar las diferentes características de, en este caso, las unidades periféricas de la tarjeta de desarrollo Spartan 3A.

Para lograr una buena comprensión de los programas en código Verilog se deben tener en cuenta de las diferentes instrucciones que acompañan este lenguaje.

Como recomendación, también es importante tener presente que todas las instrucciones dentro de un programa serán muy seguramente traducidas a hardware en algún momento, por lo que pensar en términos de hardware cuando se escribe el código facilitara en gran medida su entendimiento, además de evitar errores en la síntesis.

12.1 ELEMENTOS DEL LENGUAJE

Una parte importante del lenguaje Verilog, son los diferentes elementos que le otorgan diferentes funciones a ciertas palabras y símbolos

12.1.1 Comentarios

El uso de comentarios dentro de un programa es bastante recomendado ya que ayuda en gran medida al entendimiento del mismo cuando este es tratado de interpretar por otra persona diferente a su creador.

Un comentario puede ser de una sola línea

//comentario de una sola línea

O puede contener varias líneas

*/*Estos son comentarios distribuidos*
A lo largo de varias líneas
De programa/*

12.1.2 Identificadores

Los identificadores son los nombres, como su nombre lo dice, que identifican las diferentes variables de un programa y que permitir así su uso en distintas partes del código

Los identificadores pueden contener dentro de sus caracteres letras, números, _ (underscore) y el símbolo $. Sin embargo el primer carácter que se debe utilizar debe ser siempre una letra o el underscore.

Debe tenerse en cuenta que Verilog diferencia entre caracteres en minúscula y en mayúscula por lo que la variables clk, es completamente diferente a Clk.

12.1.3 Palabras clave

Verilog posee un conjunto de identificadores predeterminados que permiten describir acciones propias del lenguaje.

Entre las palabras clave de uso más común se pueden encontrar las siguientes

Figura 34 Tabla palabra clave en Verilog

Categorías	Palabra Clave	Descripción
Compuertas Lógicas	and nor xor not not	Son las compuertas lógicas predeterminadas que posee Verilog. Para su uso se deben instanciar de la siguiente manera: **and** num1 (output, input1,input2,..,inputn);
Asignaciones continuas	assign	Si la variable cambia, asigna el valor del operando derecho al izquierdo. Ejemplo: **assign** valor1 = valor2;
Tipo de datos	wire reg	Existen aun mas tipos de datos pero estos son los más utilizados. wire: Representa una conexión física reg: Representa un almacenamiento en el modelo
Declaración de módulos	module … endmodule	Delimitan los limites de un módulo
Evento	@	Simboliza un evento regular de control, es decir el monitoreo del cambio de valor de una variable cada flanco, se suele utilizar junto a un always

Categorías	Palabra Clave	Descripción
Parámetros	parameter	Su valor no puede ser modificado dentro del módulo y funcionan como una manera de transferir información al módulo cuando este es instanciado module [nombre](); #(**parameter** [nombre]=[valor]);
Declaración de puertos	inout	Simbolizan las conexiones de un módulo con el entorno externo. inout: Puerto entrada/salida y debe ser declarado tipo wire
Declaración de puertos	input output	input: Puerto de entrada y debe ser declarado tipo wire output: Puerto de salida y puede declararse como tipo wire o reg
Construcciones de Procedimientos	always	Este bloque es la base de la programación concurrente, comienza a ejecutarse en un tipo cero y nunca sale del ciclo el cual se repite cada vez que una variable en la lista de sensibilidad cambie su valor always @ ([lista de sensibilidad]) begin … [ciclo de control] … end

12.1.4 Representación numérica

Verilog acepta varios bases numéricas y formatos de representación, en forma general la representación de un valor numérico en Verilog debe verse así:

[signo] [# bits] ‘ [base] [valor]

El término de signo puede ser omitido a menos de que se quiera declarar un número negativo.

El número de bits determina el tamaño total de la palabra, este término puede ser omitido y el programa asumirá que el tamaño será de 32 bits

La base de la expresión numérica puede ser:
Binaria (b)

Octal (o)
Decimal (d)
Hexadecimal (h)

El termino valor corresponderá al valor número que se quiera utilizar, en caso de que dicho valor sea menor al valor máximo determinado por el número de bits se agregaran ceros en los bits más significativos

Algunos ejemplos de representaciones numéricas pueden encontrarse en la figura.

Figura 35 Ejemplos de representaciones numéricas

Representación	Valor almacenado en el programa
2’b10	10
4’b10	0010
5’d10	01010
8’h0F	00001111

Aunque la representación numérica puede tener variaciones en su escritura, es recomendable utilizar el formato completo de expresión al momento de realizar operaciones en el programa

12.1.5 Tipos de variables

Verilog acepta tres tipos de definición de variables las cuales son: **reg, wire** e **integer.**

La declaración **wire** se utiliza en variables que posean asignaciones continuas, es decir, que no tengan registros o elementos de memoria

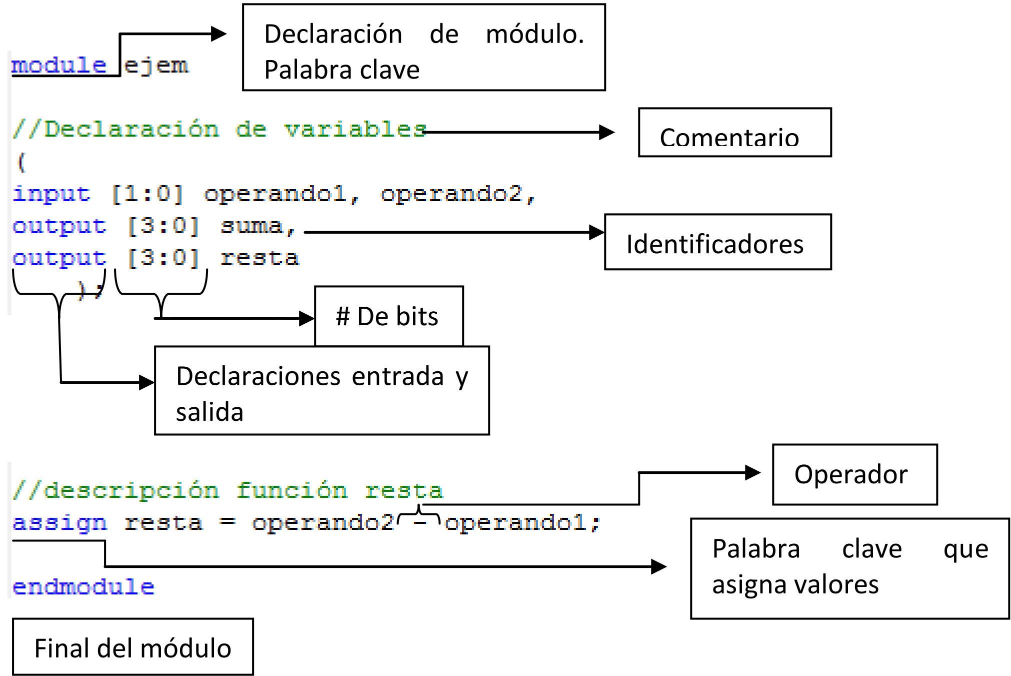
La declaración **reg** se utiliza en variables que posean elementos de memoria como los registros

La declaración **integer** se utiliza para variables con un número fijo de bits con formato de complemento a 2.

Existen otros tipos de variables, sin embargo solo los tres anteriores pueden ser sintetizados razón por la cual no se mencionaran.

Para ilustrar el uso de los elementos del lenguaje declarados hasta el momento y su uso dentro de un programa de Verilog se muestra en la figura a continuación cada uno de ellos.

Figura 36 Elementos del programa



12.1.6 Operadores

Verilog cuenta con aproximadamente 24 operadores que permiten realizar funciones aritméticas, lógicas, de igualdad y bit por bit entre otras.

Como se menciono anteriormente Verilog HDL tiene muchos aspectos similares al lenguaje C, entre algunos de ellos están los operadores, los cuales son bastante similares en función y escritura.

Dividiendo los operadores en los diferentes tipos, las clasificaciones serian las siguientes

12.1.6.1 Operadores de tipo aritmético

Existen seis operadores de tipo aritmético, estos son los expuestos en la siguiente figura.

Figura 37 Operadores de tipo aritmético

Operador	Función	Ejemplo
+	Suma	C= a+b, +a
-	Resta	C=a-b, -b
*	Multiplicación	C= a*b
/	División	C= a/b
%	Módulo	C=a%b
**	Exponenciación	C= a**2

Todas estas funciones, se realizan correctamente en el ambiente de simulación, sin embargo las únicas que se pueden sintetizar correctamente y sin problemas son las funciones de suma y resta. La función de multiplicación es bastante compleja es por ello que la herramienta de sintetización tiene problemas desarrollándola, sin embargo su uso puede depender de la tecnología disponible en la sintetización. Para el caso especifico de la Spartan 3A, que posee módulos específicos de multiplicación, esta función no debería representar un problema en el momento de la síntesis, sin embargo, se debe tener precaución en su uso.

12.1.6.2 Operadores de desplazamiento

Existen dos tipos de operadores de desplazamiento que acepta Verilog, el lógico y el aritmético

Las características de dichos operadores se muestran en la siguiente figura.

Figura 38 Operadores de desplazamiento

Operador	Función	Ejemplo
<<	Desplazamiento lógico a la izquierda	1101**<<**2 = 0111
>>	Desplazamiento lógico a la derecha	1101**>>**2 = 0011
>>>	Desplazamiento aritmético a la derecha	1101**>>>**2 = 1011
<<<	Desplazamiento aritmético a la izquierda	1101**<<<**2 = 0111

Sin embargo, al aplicarlos en el código, ambos realizan la misma función de corrimiento lógica, es decir, Verilog no realiza distinción entre la función << y <<<.

12.1.6.3 Operadores de relación

Estos operadores se encargan de evaluar dos operandos, que pueden ser conformados por uno o más bits, y dan como resultado un valor verdadero – número uno -, o un valor falso –número cero-.
Su operación y escritura es prácticamente igual a sus equivalentes en el lenguaje C y se muestra a continuación.

Figura 39 Operadores de relación

Operación	Función	Ejemplo
<	Menor que	010 **<** 110 = 1
>	Mayor que	010 **>** 110 = 0
<=	Menor o igual que	110 **<=** 110 = 1
>=	Mayor o igual que	001 **>=** 110 = 0

Si estos operadores llegaran a usarse con valores dont'care - x - el resultado también sería un valor don't care, sin embargo se debe recordar que en el mundo real no se deben hacer comparaciones con valores que no se conocen o que no importan.

12.1.6.4 Operadores de igualdad

Al igual que los operadores de relación, los operadores de igualdad son cuatro y presentan el mismo estilo de escritura que los utilizados en el lenguaje C.

En la figura se presentan estos operadores y su explicación.

Figura 40 Operadores de igualdad

Operador	Función	Ejemplo
==	Igual a	001 **==** 001 = 1
!=	Diferente a	001 **!=** 001 = 0
===	Igualdad bit por bit	0x1 **===** 011 = 1
!===	Desigualdad bit por bit	0x1 **!=** 011 = 0

Las funciones de igual a (==) y diferente a (!=), función de manera similar a los operadores de relación antes discutidos, ambas evalúan los dos operandos que conforman la función y dan como resultado una expresión verdadera – número uno- o falsa – número cero- dependiendo si se cumple o no la igualdad.
La función igualdad bit por bit, evalúa uno por uno los bits de los operandos, si todos los bits son iguales, esto incluyendo las posiciones de bits en donde aparezcan valores **x,** el resultado será un valor verdadero – uno-.

La función desigualdad bit por bit realiza una operación similar, aunque esta es utilizada para determinar si dos operadores son diferentes.
Es de gran importancia tener en cuenta que las únicas funciones sintetizables son == y !=, las funciones === y !== **no son sintetizables,** es decir no tienen una traducción a hardware.

12.1.6.5 Operaciones bit a bit

Estos operadores realizan operaciones lógicas bit a bit entre dos operandos, estas son AND, NOT, XOR, o un operando como es el caso de la función NOT. Como una combinación de las anteriores funciones se puede obtener la operación XNOR, para nombrar un ejemplo.

Tomando a = 1101 y b=0101 en la siguiente figura se explican este tipo de operaciones.

Figura 41 Operadores bit a bit

<table>
<tr><th>Operador</th><th>Función</th><th>Ejemplo</th></tr>
<tr><td>&</td><td>AND</td><td>a&b = 0101</td></tr>
<tr><td>|</td><td>OR</td><td>a|b = 1101</td></tr>
<tr><td>~</td><td>NOT</td><td>~a = 0010</td></tr>
<tr><td>^</td><td>XOR</td><td>a^b = 1000</td></tr>
<tr><td>^~</td><td>XNOR</td><td>a~^b = 0111</td></tr>
</table>

Si en algún caso un operando llegase a ser de mayor número de bits, inmediatamente el de menor número de palabas se extiende, rellenando los bits adicionales con ceros.

12.1.6.6 Operadores de reducción

Aunque las funciones que se pueden realizar con los operadores de reducción son básicamente las mismas logradas con las funciones bit a bit anteriormente nombradas, estas tienen la diferencia que solo trabajan con un solo operando.

Es decir que tomando un vector de cierta longitud de palabras, estas funciones retornaran un valor de uno o cero únicamente.

Para la figura siguiente se toma el siguiente vector como guía: a = 1010

Figura 42 Operadores de reducción.

<table>
<tr><th>Operador</th><th>Función</th><th>Ejemplo</th></tr>
<tr><td>&</td><td>AND</td><td>&a = 0</td></tr>
<tr><td>|</td><td>OR</td><td>|a = 1</td></tr>
<tr><td>^</td><td>XOR</td><td>^a = 0</td></tr>
</table>

Para ejemplificar de manera un poco más clara las operaciones que se realizaron, se tiene lo siguiente

Si a = 1010, la función ^a correspondería a la operación:

a[3] & a[2] & a[1] & a[0];

En este tipo de operaciones como en las anteriores se pueden realizar combinaciones entre las funciones principales para crear nuevas, a manera de ejemplo se podrían encontrar las funciones NAND (~&) o NOR (~|).

12.1.6.7 Operadores lógicos

A continuación se muestran los diferentes operadores de Verilog.

Tomando a a= 1111, b= 0101, c= 1001 y d= 0000 se obtienen los resultados de la siguiente figura.

Figura 43 Operadores lógicos.

<table>
<tr><th>Operador</th><th>Función</th><th>Ejemplo</th></tr>
<tr><td>&&</td><td>AND lógica</td><td>a&&b = 1
a&&d = 0</td></tr>
<tr><td>||</td><td>OR lógica</td><td>c||b = 1
d||a = 1</td></tr>
<tr><td>!</td><td>NOT lógica</td><td>!a = 0
!d = 1</td></tr>
</table>

Aunque Verilog podría aceptar estos operadores para realizar operaciones bit a bit, es recomendable utilizarlos únicamente como conexiones de expresiones como:

([expresio1]==valor1) && ([expresion2] <valor2);

La función NOT lógica se puede implementar también para negar palabras de un bit sin ningún problema, sin embargo, durante las pruebas que se realizan al código podría presentarse el caso de un cambio en la longitud de la palabra a la cual se le aplico la función, es por esto que para evitar errores y posibles molestias por el cambio de operadores una buena práctica utilizar siempre la función **~**.

12.1.6.8 Operadores de concatenación y replicación.

El operador de concatenación permite la creación de un nuevo vector mediante el uso de datos separados o vectores más pequeños.

El operador de replicación permite la creación de vectores a partir de la repetición de un dato un número determinado de veces.

Tomando para la figura siguiente los datos y vectores: a= 101, b= 010, c = 1.

Figura 44 Operadores de concatenación y replicación.

Operador	Función	Ejemplo
{ }	Concatenación	D = {a,c,b} => D= 1011010
{N{ } }	Replicación	D = {3{2'b10}} => D = 101010

12.1.6.9 Operador condicional

El operador condicional trabaja con tres expresiones:

[señal] = [expresión_booleana] ? [True] : [False];

Su operación se puede relacionar con la de una operación if – else, si la expresión booleana es verdadera la señal tomara el valor del operando True de lo contrario tomara el valor de False

Tomando a = 1010 y b=0100

D = (a>b) ? 2'b10 : 2'b01; => D=2'b10.

D = (a==b) ? 2'b00 : 2'b11; => D=2'b11.

12.1.6.10 Parámetros y constantes

Las constantes son valores fijos no modificables dentro de un programa que son útiles para facilitar la lectura y comprensión del mismo.
En Verilog una constante se declara con la palabra clave **localparam**

localparam N = 4'd8;

En esta declaración el valor de 8 se le ha asignado a la constante N la cual podrá ser utilizada a lo largo del programa

Los parámetros poseen casi el mismo funcionamiento que las constantes, sin embargo su diferencia radica en que el valor de un parámetro puede ser variado al momento de instanciar el módulo en donde se encuentra, lo que lo convierte en una buena herramienta para transmitir información a los módulos.

Un parámetro se declara con la palabra clase **parameter.**

#(parameter N = 32'd100;)

En este caso a N se le asigno un valor de 100, sin embargo al momento de instanciar el módulo este valor puede ser modificado.

Ejercicio propuesto

1. Utilizando como base el programa sencillo utilizado anteriormente se recomienda realizar una práctica de cada una de las operaciones explicadas para observar el resultado que generan en los operandos y así comprenderlas mejor.

   A manera de recordatorio se vuelve a mostrar el ejemplo referido en la figura siguiente.

Figura 45 Elementos del programa

```
module ejem

//Declaración de variables
(
input [1:0] operando1, operando2,
output [3:0] suma,
```

```
output [3:0] resta
   );

//Descripcion de función suma
assign suma = operando1 + operando2; //implementar aquí las
                                      //nuevas operaciones

//descripción función resta
assign resta = operando2 - operando1;

endmodule
```

2. Con las bases hasta el momento obtenidas y teniendo como base el programa anterior, diseñe ejercicios que implementen diferentes funciones a la vez utilizando varias de las operaciones vistas hasta el momento.

12.1.7 Construcciones en Verilog

12.1.7.1 IF-ELSE

La construcción if-else permite analizar señales y generar acciones de acuerdo a los valores de dichas señales de manera secuencial.

Su estructura básica es como sigue en la figura.

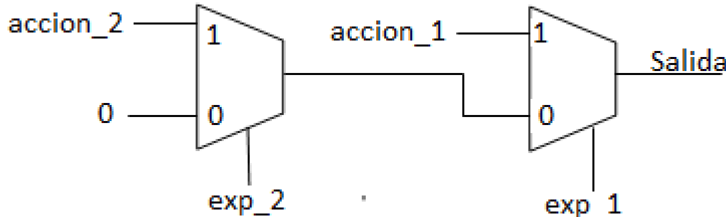
Figura 46 Estructura if-else

```
If (exp1)
 begin
        [acción 1];
 end
Else  //no se cumple la primera condición
 begin
        [acción 2];
 end
```

En este caso si la expresión evaluada (exp) tiene nivel lógico alto se realizara la acción 1, de lo contrario se realizara la acción 2.

Como se debe tener presente que Verilog es un lenguaje de programación en hardware, el código descrito en la figura anterior debe ser traducido en términos de elementos físicos, en este caso el código anterior describe el circuito mostrado en la siguiente figura.

Figura 47 Estructura if-else en hardware



La estructura if-else se representa como se observa en la figura anterior con multiplexores que tienes como entrada de selección la expresión por la cual se pregunta, eligiendo así la salida correcta, esta construcción puede contener aun más multiplexores permitiendo así la evaluación de mas expresiones, es decir puede utilizarse en cascada.

El uso de la estructura if-else en cascada, genera una ruta de prioridad en donde la primera expresión tiene más prioridad que la segunda y así sucesivamente. En la figura a continuación se puede observar esta sintaxis.

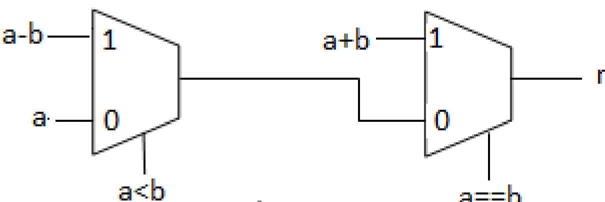
Figura 48 Ejemplo de construcción if-else en cascada

```
if (a==b)
        r = a+b;

else if (a<b)

        r = a-b;
else
        r = a;
```

Los indicadores begin…end no fueron utilizados aquí debido a que solo se realiza una acción en opción.

Siguiendo el concepto básico de la estructura en hardware de la construcción if-else vista anteriormente, el código descrito en se traduciría a hardware de la manera en que lo ilustra la grafica siguiente.

Figura 49 Ejemplo de construcción if-else traducido a hardware



Un segundo ejemplo del uso de la construcción if-else se ilustra en el siguiente ejemplo en donde se implementa un encoder de prioridad

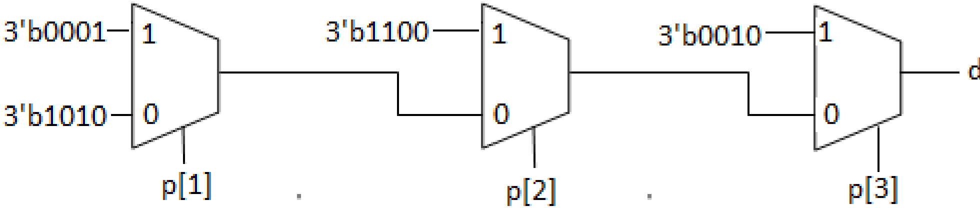
Figura 50 Encoder de prioridad construcción if-else

```
module encoder_prioridad
(
input [3:0] p,                    //entrada de 4 bits
output reg [2:0] d                //salida de tres bits
);

always @ *
begin
if (p[3])                         //si el bit 3 de la entrada se encuentra en uno
d <= 3'b0010;
else if (p[2])                    //si el bit 2 de la entrada se encuentra en uno
d <= 3'b1100;
else if (p[1])                    //si el bit 1 de la entrada se encuentra en uno
d <= 3'b0001;
else                              //si el bit 0 de la entrada se encuentra en uno
y <= 3'b1010;
end
```

El anterior código preguntaba por el estado lógico de los bits de la palabra de entrada "p" en orden y dependiendo de cual estaba acertada primero se asignaba el valor a la salida "d".

Si traducimos el código a un diagrama esquemático a para observar la interacción de los diferentes elementos se obtendría el mostrado en la siguiente figura.

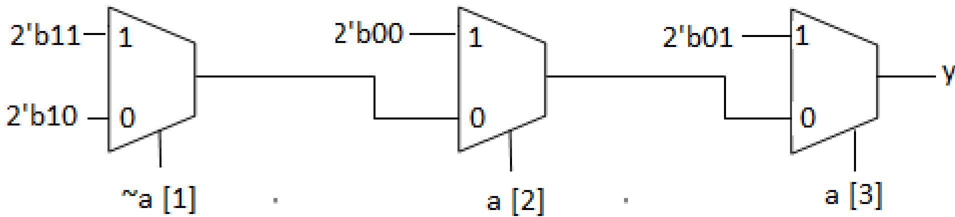
Figura 51 Diagrama esquemático del encoder de prioridad



Ejercicios propuestos

1. De la misma manera en que se realizo el encoder de prioridad, realice un decodificador binario que encienda un solo bit de la palabra de salida con cada combinación de entrada. Las palabras de entrada y salida deben tener un mínimo de tres bits.
2. Teniendo los como base los ejercicios realizados transforme el diagrama esquemático de la siguiente figura a código Verilog.

Figura 52 Diagrama esquemático ejercicio 12.1.7.1



3. Transforme el siguiente código en un diagrama esquemático

```
module v_to_esq
(
input a, b, c, d
output reg e1, e2
);

always @ *
begin
if (a==c) begin
e1 <= 1'b0;
e2 <= 1'b0;
end
else if (a<b) begin
e1<= 1'b0;
e2<= 1'b1;
end
else if (b<c) begin
e1<= 1'b1;
e2<= 1'b0;
end
else if (c<d) begin
e1<= 1'b1;
e2<= 1'b1;
end

end
endmodule
```

12.1.7.2 CASE

La función de la estructura case es evaluar el valor de una variable y de acuerdo a dicho valor realizar una o más acciones, por lo general el case representa a un multiplexor.
Su estructura básica puede apreciarse en la figura siguiente.

Figura 53 Estructura CASE

```
case  (var)

valor1 : acción_1;

valor2 : acción_2;
…
Default: acción_n

endcase
```

La opción default utilizada al final de la lista de opciones del case, reemplaza todos los posibles valores que no fueron especificados.

Un ejemplo del uso de esta estructura se puede ver a continuación.

Figura 54 Ejemplo del uso de la estructura CASE

```
case (estado)

2’00: begin
        C = 4’d8;
        D = 3’d2;
        end
2’01: begin
        C = 4’d1;
        D = 3’d0;
        end
default: begin
        C = 4’d9;
        D = 3’d1;
        end
endcase
```

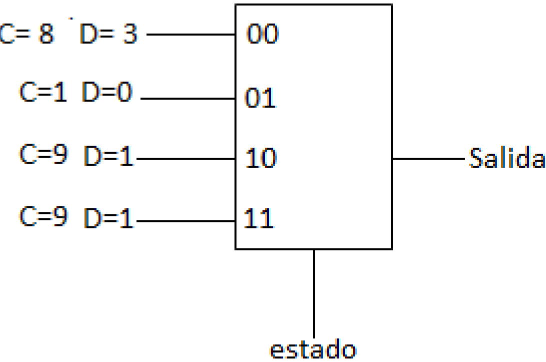
En la anterior figura se observa que dependiendo del valor de estado, 2’b00 o 2’b01, se le asignara un valor específico a D y C. las palabras clave begin…end fueron utilizadas en este caso debido a que cada opción contenía más de una instrucción para realizar.

Ya que los demás posible valores de estado no fueron especificados (2’b10 y 2’b11) se utiliza la palabra clave default para darles valor a las variables D y C. el uso de default es recomendable para evitar la generación de latches en el circuito al momento de la implementación.

En términos de hardware el circuito que representa la construcción case es un multiplexor, como ejemplo se demuestra en la siguiente figura el código descrito anteriormente en términos de elementos físicos en donde la entrada

selectora es la variable estado y cada una de las entradas al multiplexor son las especificadas dentro de la construcción.

Figura 55 Construcción Case en hardware



Para ejemplificar el uso del case dentro de otro diseño se utilizara el encoder de prioridad antes ilustrado.

Figura 56 Encoder de prioridad constrcción case

```
module prior_enc
(
input [3:0] p,
output reg [2:0]
);

always @ *
case (p)
4'b1000:d <= 3'b0010;
4'b1100: d<= 3'b1100;
4'b1110: d<= 3'b0001;
default: d<= 3'b1010;
endcase

endmodule
```

Ejercicios propuestos

1. Implemente un encoder de prioridad similar al del ejemplo anterior pero en este caso añada una señal de enable que cuando este activa permita el funcionamiento del case y cuando este inactiva lo evite.
2. Implemente en Verilog un multiplexor que tenga como valores la combinación del primer y último bit de dos entradas de dos bits cada una, además que el valor que se asigna a la salida en cada opción se decida mediante la comparación de mayor o igual que de los bits restantes de las señales de entrada.

3. Diseñe una memoria ROM de cinco espacios en base a la construcción case, la entrada de la memoria deberán ser la dirección y el enable.

12.1.7.3 Bloque always

El bloque always es una de las estructuras más utilizadas para describir lógica secuencial en Verilog.

Al utilizar un bloque always es recomendable realizar una sola operación y no mezclar demasiadas variables

La siguiente figura expone la estructura del bloque always.

Figura 57 Estructura básica del bloque always

```
always @ (lista de sensibilidad)
begin
        instrucción
        instrucción
        …
end
```

Un bloque always ejecutara las instrucciones que se encuentran en el cada vez que ocurra un cambio en el estado de las variables que estén declaradas en la lista de sensibilidad. Es importante recordar que cuando se define lógica combinacional todas las variables deben estar declaradas en la lista de sensibilidad.

Una solución para evitar que alguna variable quede por fuera de la lista de sensibilidad es escribir un asterisco en ella, esto hará que todas las variables queden automáticamente incluidas.

always @ (*)

Dentro de un bloque always se pueden encontrar dos tipos de asignaciones las asignaciones de bloqueo (blocking assignments) y las asignaciones sin bloqueo (nonblocking assignments).

En las asignaciones de bloqueo la expresión se evalúa y se establece su nuevo valor inmediatamente, mientras esto sucede ninguna otra expresión puede ser evaluada, es decir bloquea las demás asignaciones.

En las asignaciones sin bloqueo ocurre lo contrario, la expresión se evalúa pero su valor no es asignado sino hasta el final de la ejecución del bloque always, mientras tanto otras instrucciones pueden ser evaluadas.

Figura 58 Ejemplo de lógica combinacional utilizando el bloque always

```
always @ (a , b, c)
begin
if (b>c)
        out = b;
else
        out = a;
end
```

En la figura anterior a, b, y c son todas las entradas que posee el módulo, como se observa siempre que una variable cambie su estado el bloque always se ejecutara y su valor será asignado inmediatamente, requisitos necesarios para considerarse lógica combinacional.

Con el bloque always se pueden definir también elementos secuenciales como los flipflop, la estructura básica de un flipflop en Verilog es como se muestra a continuación:

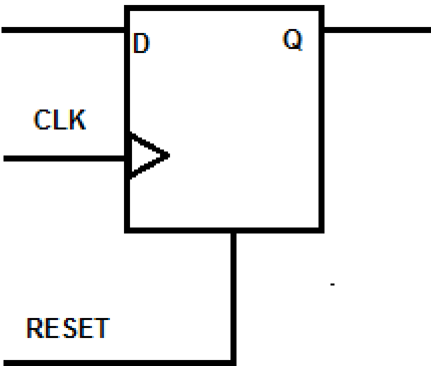
Figura 59 Definición de un flipflop con el bloque always

```
always @ (posedge clk)
begin
if (reset)
        out <= 0;
else
        out <= valor_actual;
end
```

El anterior bloque always será sintetizado como un flipflop tipo D con reset síncrono, este bloque se ejecutara cada vez que la variable clk presente una transición de cero a uno, flanco positivo, y el valor de la entrada será asignado a la salida cuando el reset no se encuentre activo

El código corresponde a la figura siguiente

Figura 60 Flipflop tipo D



Un ejemplo de un flipflop con reset asíncrono es como se muestra a continuación:

Figura 61 Flipflop tipo D con reset asíncrono.

```
always @ (posedge clk or negedge reset)
begin
if(reset)
        out<= 0;
else
        out <= valor_actual;
end
```

En este caso el bloque se ejecutara cada flanco positivo de la variable clk, o cada flanco negativo de la variable reset.

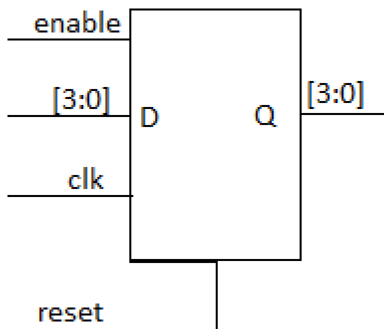
Durante el trabajo con bloques always es bueno tener presentes las siguientes recomendaciones:

- No asignar un valor a una misma variable en distintos bloques always
- Revisar que la lista de sensibilidad se encuentre completa
- Si el bloque always asigna valores a mas de una variable se debe asegurar que ambas variables sean asignadas en todas los posibles caminos del bloque
- De dejar caminos de control incompletos en las estructuras if-else o case.

Ejercicios propuestos

1. Implemente un FlipFlop tipo T con reset sincrono mediante el uso del bloque always
2. Implemente un FlipFlop tipo T con reset asíncrono mediante el uso del bloque always
3. Transforme el siguiente diagrama esquemático en código Verilog

Figura 62 FlipFlop tipo D con enable y reset síncrono



4. Del código Verilog de la siguiente figura dibuje el diagrama esquemático

Figura 63 Codigo Verilog ejercicio 4

```
module ff
(
input [3:0]  a,
output reg [3:0] f
);
reg temp [3:0];

always @ (posedge clk)
begin
if (reset)
temp <=0;
else
temp<= a;
end

always @ (posedge clk)
begin
if (reset)
f <= 0;
else
f <=temp;
end
endmodule
```

12.1.8 Estructura básica de un programa en Verilog

Un módulo Verilog por lo general sigue una estructura básica que está dividida en cuatro partes que son:

- Declaración de entradas y salidas: Justo después de haber sido declarado el módulo todas las entradas y salidas que tendrá el mismo deben ser especificadas.

- Declaración de variables internas: Para tener un mayor orden en el programa es recomendable declarar en un solo espacio todas las variables internas que serán utilizadas, ya que aunque estas pueden ser declaradas en cualquier momento de la programación esto podría ser una dificultad al momento de comprender el programa.
- Cuerpo del programa: Después de haber realizado todas las declaraciones de entrada, salida y variables necesarias se puede continuar al modelamiento de la función que realizara el módulo

Un módulo sencillo que realiza la función de un contador binario de 4 bits ejemplificara la distribución del programa. Este módulo puede observarse en la figura a continuación.

Figura 64 Ejemplo módulo contador

```
module contador

( input clk, reset;                 //Declaración de entradas y salidas
output [3:0] contador
);

reg [3:0]  valor_actual;            //Declaración de las variables internas

always @ (posedge clk)              //Cuerpo del programa
begin
if (reset)
        valor_actual <= 0;
else
        valor_actual <= valor_actual + 1;
end

assign contador = valor_actual;

endmodule
```

12.2 MÁQUINAS DE ESTADO

Las máquinas de estado son utilizadas para modelar sistemas que realicen un número finito de acciones específicas en un determinado orden.

Una máquina de estados está conformada por una combinación entre elementos secuenciales y lógica combinacional, de acuerdo al valor de la entrada y del estado actual en que se encuentra la máquina se define el valor de la salida.

Existen dos tipos de máquinas de estado denominadas tipo Moore o tipo Mealy, en el primer caso la salida de control depende únicamente del estado actual en que se encuentra el proceso, en el segundo caso la salida no

dependerá únicamente del estado actual sino también del estado de las entradas al sistema.

La elección del tipo de máquina a utilizar se debe basar en el tipo de subsistema en que se está trabajando y sus condiciones, por lo general una tipo Mealy requiere menor cantidad de estados pero a cambio de esto se pueden presentar glitches en la salida que generalmente una máquina tipo Moore no tendría.

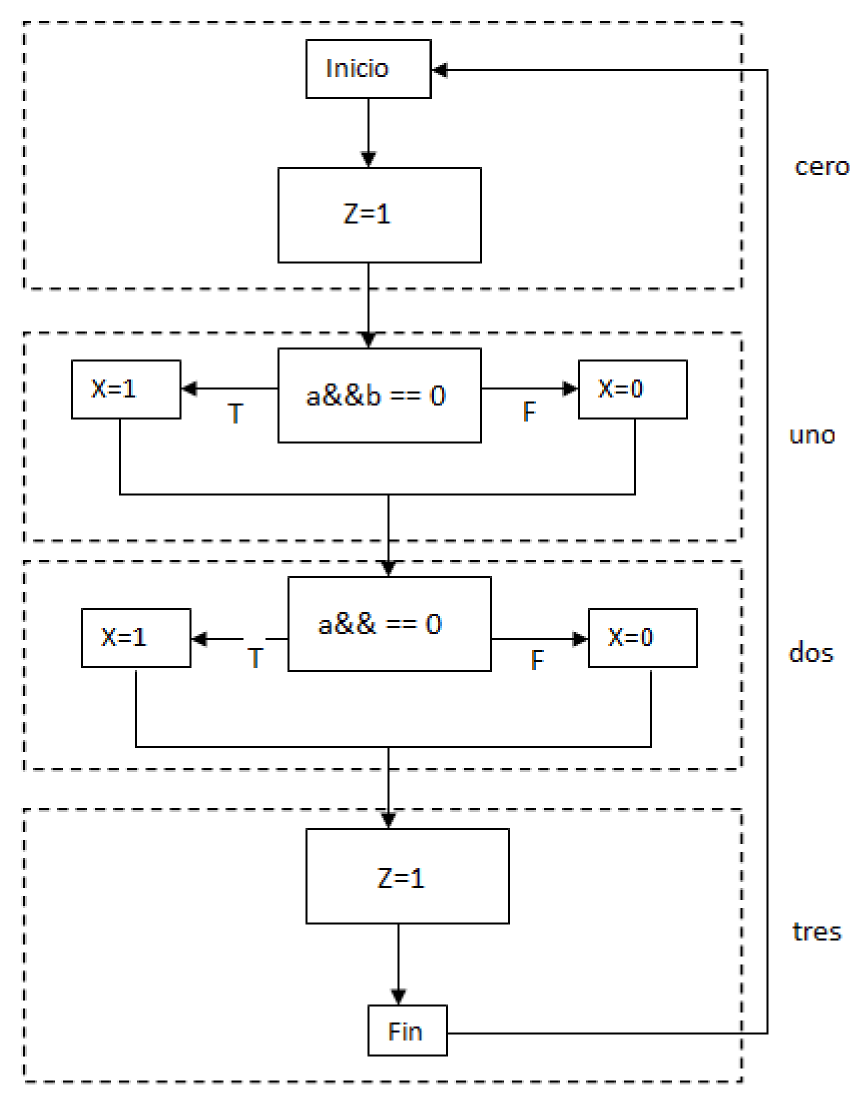
Cuando se quiere realizar una máquina de estados para ejercer un control sobre algún tipo de proceso, es recomendable realizar antes un diagrama de transición de estados que contenga las acciones a seguir en cada estado y las condiciones necesarias para el cambio de estado, esto facilita en gran medida el trabajo de convertir la idea de control a un código Verilog.

Estos diagramas de estados contendrán bloques que corresponderán a las preguntas sobre la toma de decisiones dentro del proceso y a las acciones de control que se tomen a partir de la respuesta a dichas preguntas, igualmente deberá contener un indicador de flujo explicito y bloques de estado que indicaran el estado en que se encuentra el proceso.

Para ejemplificar el diseño de una máquina de estados y las diferencias entre el tipo Moore y el tipo Mealy se realizará un ejemplo que ilustre las dos diferentes salidas.

El diseño constara de dos entradas, a y b, y dos salidas, z y x. La primera salida se activara cuando la máquina se encuentre en el primer y último estado, mientras que la segunda salida se activara en los dos estados intermedios si las dos entradas se encuentran desactivadas. Para iniciar con el diseño se realizara el diagrama de transición de estados como se ve en la siguiente figura.

Figura 65 Diagrama de transición de estados para una máquina tipo Mealy y Moore



Después de tener definido el diagrama de transición de estados se puede proceder a traducirlo a lenguaje Verilog, utilizando lógica del estado siguiente como se observa en la siguiente figura.

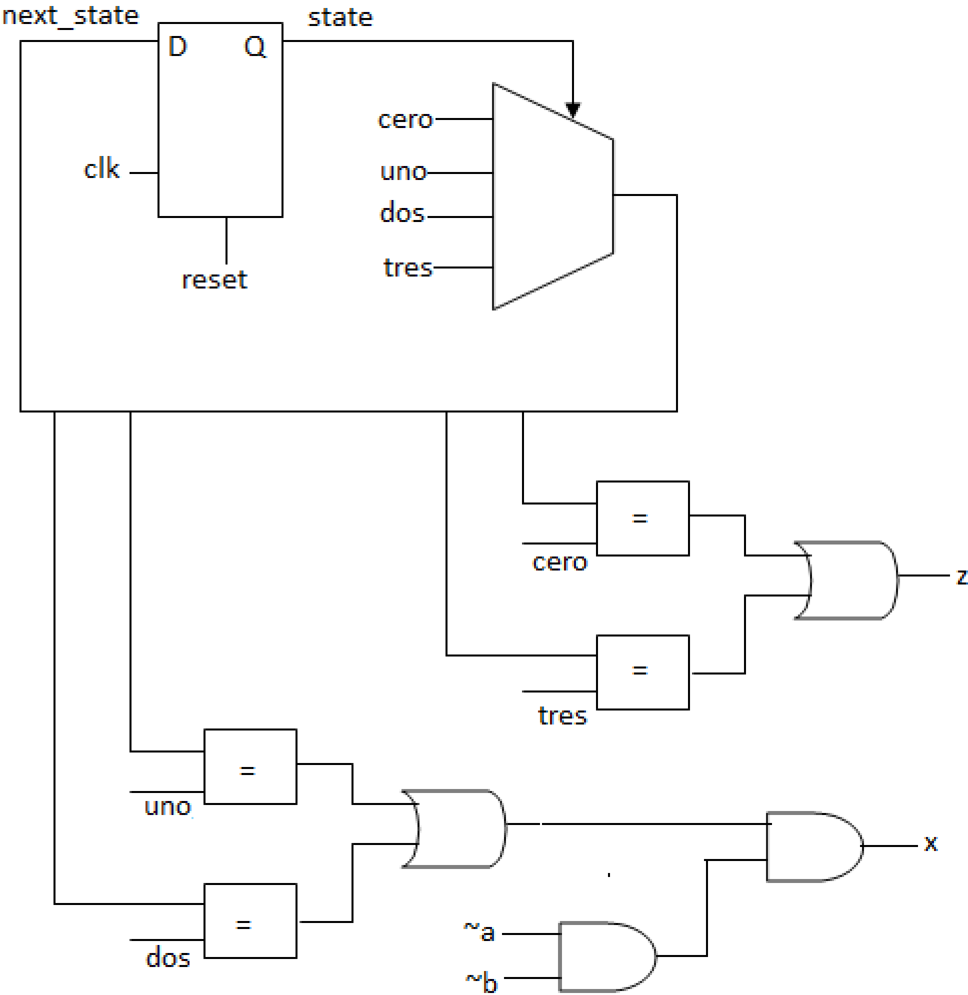
Figura 66 Módulo FSM, salidas Moore y Mealy

```
module FSM(
input a, b, clk, reset,          //Declaración de entradas y salidas
output x, z
   );

reg [1:0] state, next_state;          //variables internas del programa

localparam [1:0]       //Declaración de las constantes del programa
        cero= 2'b00,
```

```
        uno= 2'b01,
        dos= 2'b10,
        tres= 2'b11;

always @ (posedge clk)    //flipflop tipo D con reset sincrono
if (reset) state <=0;
else
state <= next_state;      //cada flanco positivo se actualiza state

always @ *
begin

next_state = state;       //valor inicial

case (next_state)          //Inicio de la máquina de estados
        cero:  next_state = uno;    //Los estados avanzan automaticamente

        uno:   next_state = dos;

        dos: next_state = tres;

        tres: next_state = cero;

        default : next_state = cero;
endcase

end

//Salida Mealy
assign x = (((state==uno)||(state==dos))&&((a==0)&&(b==0)));
//Salida Moore
assign z = ((state==cero)||(state==tres));

endmodule
```

El código al estar escrito en lenguaje de descripción en hardware puede ser traducido a un diagrama esquemático, para así comprender mejor su funcionamiento y enfocarse en el pensamiento en hardware. En la siguiente figura se puede observar el circuito descrito por el ejemplo anterior.

Figura 67 Diagrama esquemático módulo FSM

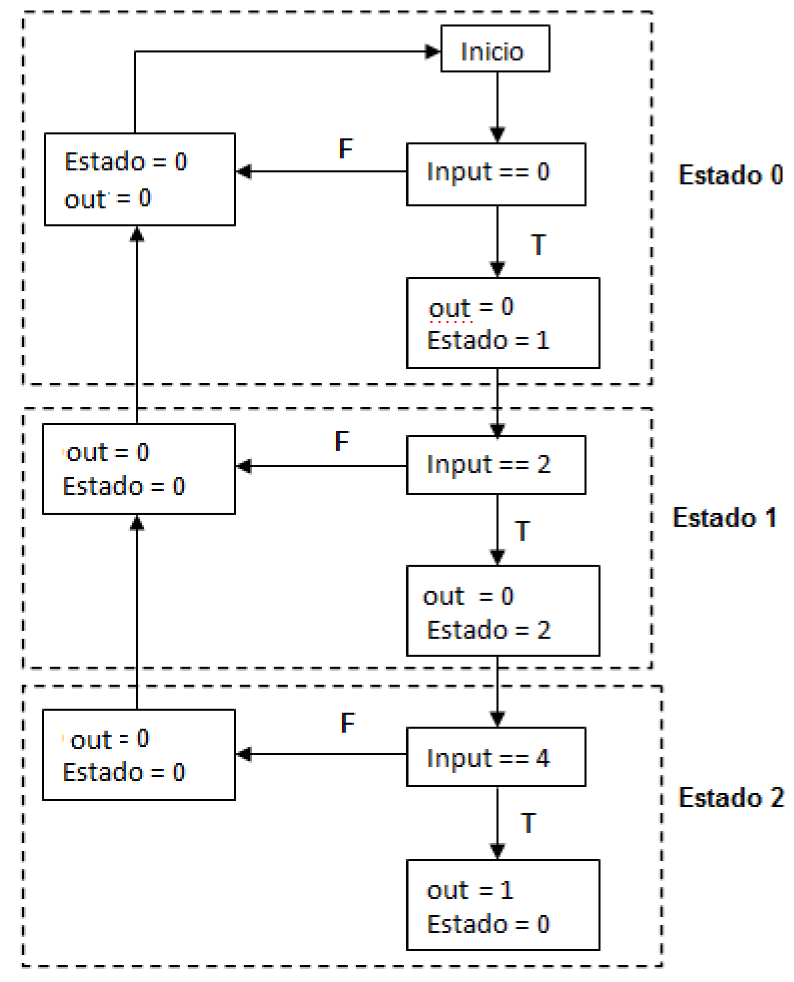


Continuando con la metodología del ejemplo anterior y para reforzar más los conceptos sobre máquinas de estado, se realizara otro diseño que contenga una aplicación enfocada a la solución de un problema. Se creara un detector de secuencia.

El funcionamiento será el siguiente: el módulo recibirá un valor a su entrada el cual comparará con el primer valor de la secuencia; en caso de concordar, se proseguirá a evaluar el segundo valor. En caso de ser un valor diferente, el sistema volverá a su punto de inicio y el código deberá ser accedido nuevamente.

El diagrama de estados de este detector de secuencia se observa en la figura:

Figura 68 Diagrama de estados del módulo de detección se secuencia.



En el diagrama anterior se observan las diferentes partes que contienen los diagramas de estado en donde las flechas indican la dirección del flujo del proceso y los rectángulos de líneas discontinuas corresponden a los bloques de estado.

Una vez realizado el diagrama de estados y de entender el funcionamiento del sistema, pasar a código Verilog es una tarea sencilla.

El código Verilog que describe el sistema mostrado en la figura anterior corresponde al observado en la figura siguiente:

Figura 69 Código Verilog detector de secuencia.

```
module example(              //declaración del módulo y sus entradas y salidas
input [3:0] codigo,
input clk, reset,
output reg pulse
   );
```

```
reg [1:0]estado = 0;          //variables del programa
reg [1:0] estado_siguiente;

localparam [1:0]              //constantes que se utilizaran en el programa para
        cero = 2'b00,         //los estados
        uno = 2'b01,
        dos = 2'b10;

always @ (posedge clk)              //FF que actualiza el valor actual de estado
if (reset) estado <= 0;
else
        estado<= estado_siguiente;


always @ ( * )                //inicio de la máquina de estados
begin
estado_siguiente <= estado;             //condiciones iniciales de las variables
pulse <= 0;

        case (estado)

        cero: begin
                if (codigo == 4'd0)           //si la entrada concuerda con cero siga
                begin                         //al siguiente estado
                        estado_siguiente <= uno;
                        pulse <=0;
                end
                else
                begin
                estado_siguiente<= cero;  //de lo contrario se reinicia el proceso
                pulse <= 0;
                end
                   end

        uno: begin
                if (codigo == 4'd2)           //si la entrada concuerda con dos siga
                begin                         //al siguiente estado
                        estado_siguiente <= dos;
                        pulse <=0;
                end
                else
                begin
                estado_siguiente<= cero;  //de lo contrario se reinicia la máquina
                pulse <= 0;
                end
                   end

        dos: begin
                if (codigo == 4'd4)           //si concuerda con cuatro se finalizo la
                begin                         //la clave con exito
```

```
                estado_siguiente <= cero; //se reinicia el proceso
                pulse <=1;           //se genera un pulso en la variable
            end                      //pulse
            else
            begin
            estado_siguiente<= cero;         //de lo contrario se reinicia el
            pulse <= 0;              //proceso sin cambios en la salida
            end
              end
endcase
end
endmodule
```

Ejercicios propuestos

1. De la misma manera en la que se ha estado trabajando, realice el diagrama esquemático de la máquina de estados de la figura anterior.
2. Diseñe un detector de flancos positivos utilizando una máquina de estados, la señal de salida deberá ser alta cuando exista una transición de cero a uno y bajo cuando se presente la transición contraria. Recuerde realizar el diagrama de transferencia de estados antes de escribir el código
3. Realice el diagrama esquemático del ejercicio anterior.
4. Realice un detector de secuencia que detecte la palabra 10011, realice el diagrama esquemático

12.3 PICOBLAZE

Debido a que la programación a base de hardware es más compleja que el diseño en software y existen aplicaciones muy complejas de describir físicamente, Xilinx incluyo en sus tarjetas Spartan3 y VirtexII un microcontrolador embebido de 8 bits nombrado Picoblaze.

Picoblaze es un microcontrolador capaz de realizar un procesamiento y control sencillo de señales, es una herramienta compacta que añade flexibilidad a las FPGA y que puede ser integrado en otro sistema. Su uso es una buena elección cuando se tienen máquinas de estados muy largas o complejas que no requieran tiempos críticos de ejecución.

La razón por la que Picoblaze es una buena elección para reemplazar máquinas de estado complejas es porque su constitución es la misma que la de una máquina de estados.

Una máquina de estados contiene tres partes fundamentales:

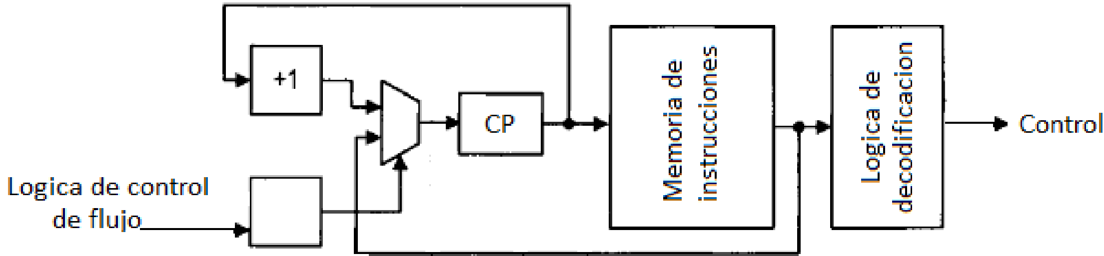
- Un registro que almacena el estado actual de la máquina de estados
- Una lógica que activa los valores de salida correspondientes

- Una lógica que determina el estado siguiente.

Un microcontrolador no es más que una máquina de estados programable cuyos componentes principales reemplazan las partes de una máquina de estados en hardware.

- Posee un contador de programa que lleva el registro de la instrucción que se está ejecutando actualmente.
- Contiene una memoria de instrucciones que almacena en distintas direcciones las acciones que debe realizar el microcontrolador
- Después de ejecutada la instrucción actual, el contador de programa aumenta automáticamente su valor en uno pasando así a ejecutar la siguiente instrucción. Sin embargo este orden secuencial puede ser alterado con instrucciones de salto provenientes desde la memoria del programa.

Figura 70 Diagrama simplificado de un microcontrolador[43]



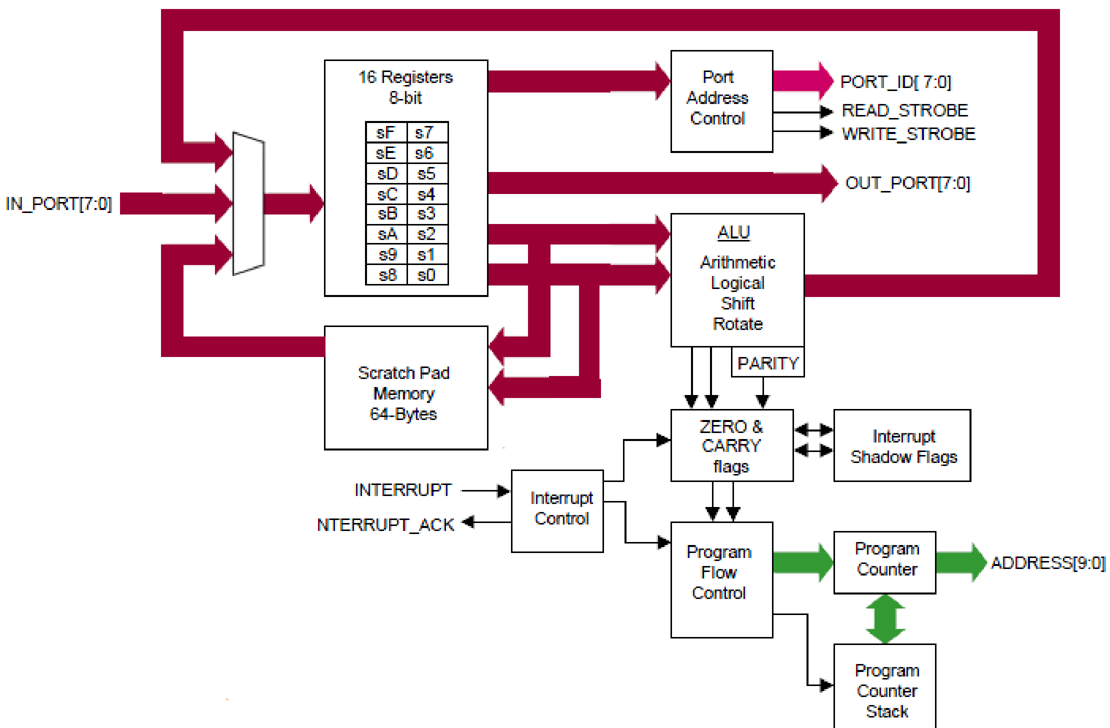
En la figura anterior se observa el diagrama simplificado de un microcontrolador, en este se pueden observar el orden de ejecución de las acciones que maneja. Cada uno de los elementos mostrados en figura anterior realiza la siguiente función:

- La lógica de control de flujo corresponde a los estados de las banderas (carry, zero) y instrucciones de control como los saltos condicionales o llamadas a rutinas, estas variables seleccionan el orden de ejecución del programa, es decir, permiten que el programa continúe ejecutando las instrucciones en orden o que se realice un salto en el orden mediante la asignación de un nuevo valor en el contador del programa (CP)
- El contador del programa (CP) como se observa recibe el resultado de la lógica de control de flujo y puede aumentar su valor en uno para continuar una ejecución de instrucciones en orden o adoptar otro valor para realizar saltos en el programa
- La memoria de instrucciones recibe del contador del programa la dirección en donde se encuentra la instrucción que se debe ejecutar en el momento.
- Una vez identificada la dirección donde se encuentra la instrucción deseada, la misma se procesa en la lógica de codificación en donde el resultado es la salida del microcontrolador.

[43] Pong, Chu. FPGA prototyping by Verilog examples. Estados Unidos: WILEY

Como ya se había mencionado el diagrama presentado en la figura anterior es solamente una aproximación a la estructura completa del microcontrolador, ya que este cuenta con más componentes que permiten su funcionamiento, un acercamiento más detallado su arquitectura se observa en la siguiente figura

Figura 71 Diagrama de bloques de Picoblaze.[44]



Los nuevos elementos que se observan en el diagrama anterior son los 16 registros de 8 bits, el control de la dirección de los puertos, la unidad aritmético lógica (ALU), la memoria RAM (Scratch Pad Memory), el control de interrupciones y las banderas de control (zero y carry). En este diagrama hacen aparición también nuevas señales de control de proceso que corresponden a:

- IN_PORT: corresponde al valor de la señal en el puerto de entrada del microcontrolador
- OUT_PORT: corresponde al valor de la señal en el puerto de salida del microcontrolador
- PORT_ID: es el identificador de la dirección del puerto de entrada o salida que se utilizara
- READ_STROBE: esta señal produce un pulso cuando el microprocesador realiza una acción de lectura en IN_PORT.
- WRITE_STROBE: esta señal produce un pulso cuando el microprocesador realiza una acción de escritura en OUT_PORT.
- INTERRUPT: esta señal de entrada activa las interrupciones del programa

[44] CHAPMAN, Ken. KCPSM3 8bits Microcontroller for spartan3, virtex-II and virtex-IIPRO. Internet: www.xilinx.com/picoblaze [consulta: 15 enero 2010]

- INTERRUPT_ACK: esta señal de salida se produce al terminar de realizar la acción de interrupción

Para la realización de un proceso completo dentro del microprocesador mostrado en la figura anterior seguiría el siguiente orden.

Los registros utilizados obtienen el valor presente en el puerto de entrada, en la memoria RAM o del resultado del ALU. Los valores almacenados en los registros pueden ser almacenados en la memoria RAM para su posterior uso, o introducidos en el ALU en caso de ser necesaria cualquier operación lógica o aritmética, también pueden ser reflejados en el puerto de salida.

Una vez el ALU haya realizado su operación, esta modificara los valores de las banderas carry y zero dependiendo de las cuales se maneja el flujo del programa. En esta etapa del proceso entran también a contar las banderas y señales correspondientes a las interrupciones.

El contador del programa recibe la dirección de la instrucción que se ejecutara en el ciclo actual y a su vez la almacena en una memoria temporal para posteriores referencias.

Una vez comprendido el funcionamiento del microcontrolador Picoblaze es importante recordar que se está trabajando con un elemento sencillo cuyo funcionamiento, por tratarse de software, es secuencial lo cual facilita su uso dependiendo de la aplicación, además es muy útil para realizar tareas que no requieran un tiempo crítico.

La programación es sencilla y se realiza en base al lenguaje de programación assembler.

Picoblaze acepta un conjunto de 57 instrucciones, dos banderas de estado denominadas zero (Z) y carry (c) y cuenta con 16 registros de s0 a sF.

12.3.1 Instrucciones lógicas.

Este set de instrucciones realiza operaciones lógicas bit a bit entre dos registros o entre un registro y una constante. El primer operando deberá ser siempre un registro y en este se guardara el resultado de la operación.

El estado de la bandera carry será siempre cero y la bandera zero se activara en caso cual que la operación de cero.

Una tabla con las operaciones lógicas permitidas en Picoblaze se muestra en la siguiente figura.

Figura 72 Instrucciones lógicas en Picoblaze

<table>
<tr><th>Instrucción</th><th>Operación</th><th>Explicación</th></tr>
<tr><td>and sx, sy
and sx, k</td><td>sx ← sx & sy
sx ← sx & k</td><td>Operación and entre registros
Operación and entre registro y constante</td></tr>
<tr><td>or sx, sy
or sx, k</td><td>sx ← sx | sy
sx ← sx | k</td><td>Operación or entre registros
Operación or entre registro y constante</td></tr>
<tr><td>xor sx, sy
xor sx, k</td><td>sx ← sx ^ sy
sx ← sx ^ k</td><td>Operación xor entre registros
Operación xor entre registro y constante</td></tr>
<tr><td>test sx, sy
test sx, k</td><td>t ← sx & sy
z ← 1 si t==0
c ← ^t</td><td>Operación test entre registros o entre un registro y una constante, si el resultado es cero se activa la bandera zero, en la bandera carry se almacena el resultado de la operación xor realizada.</td></tr>
</table>

12.3.2 Instrucciones de corrimiento y rotación.

Existen cinco operaciones de corrimiento y rotación de registro a la derecha y otras seis a la izquierda. Estas operaciones involucran solamente a un registro y pueden afectar el estado de la bandera de carry.

La bandera zero solo se activara en el caso en que el registro adquiera un valor de cero.

La tabla con el contenido de las operaciones de corrimiento y rotación se muestra a continuación en la figura.

Figura 73 Instrucciones de corrimiento y rotación para Picoblaze.

Instrucción	Operación	Carry	Explicación
sl0 sx sr0 sx	sx ←{sx[6:0], 0} sx ←{0, sx[7:1]}	c ← sx [7] c ← sx [0]	Corrimiento a la izquierda, se reemplaza el LSB por cero. Corrimiento a la derecha, se reemplaza el MSB por cero
sl1 sx	sx ←{sx[6:0], 1}	c ← sx [7]	Corrimiento a la izquierda, se reemplaza el LSB por uno.
Instrucción	Operación	Carry	Explicación
sr1 sx	sx ←{1, sx[7:1]}	c ← sx [0]	Corrimiento a la derecha, se reemplaza el MSB por uno
slx sx srx sx	sx ←{sx[6:0], sx[0]} sx ←{sx[7], sx[7:1]}	c ← sx [7] c ← sx [0]	Corrimiento a la izquierda, se reemplaza el LSB por sx[0]. Corrimiento a la derecha, se reemplaza el MSB por sx[7]

Instrucción	Operación	Carry	Explicación
sla sx sra sx	sx ←{sx[6:0], c} sx ←{c, sx[7:1]}	c ← sx [7] c ← sx [0]	Corrimiento a la izquierda, el MSB reemplaza al LSB utilizando el carry Corrimiento a la derecha, el LSB reemplaza al MSB utilizando el carry
rl sx rr sx	sx ←{sx[6:0], sx[7]} sx ←{sx[0], sx[7:1]}	c ← sx [7] c ← sx [0]	Rotación del registro un bit a la izquierda, se conserva la información Rotación del registro un bit a la derecha, se conserva la información

12.3.3 Instrucciones aritméticas.

Las operaciones aritméticas permitidas son la suma y la resta, la cual puede realizarse teniendo o no en cuenta el carry.

Estas operaciones involucran dos registros o un registro y una constante, el resultado se guardara en el registro sx.

Las operaciones aritméticas se muestran en la tabla de la figura siguiente.

Figura 74 Instrucciones aritméticas para Picoblaze.

Instrucción	Operación	Explicación

add sx, sy
 	
add sx, k
	sx ← sx + sy
	
	sx ← sx + k
		Suma de dos registros sin el carry
		
		Suma de un registro con una constante k

addcy sx, sy
 	
addcy sx, k
	sx ← sx + sy + c
	
	sx ← sx + k + c
		Suma de dos registros con carry
		
		Suma de un registro y una constante con carry

sub sx, sy
 	
sub sx, k
	sx ← sx – sy
	
	sx ← sx – k
		Resta de dos registros sin carry
		
		Resta de un registro con una constante k

subcy sx, sy
 	
sub sx, k
	sx ← sx – sy- c
	
	sx ← sx – k – c
		Resta de dos registros con carry
		
		Resta de un registro con una constante con carry

compare sx, sy compare sx, k	z ←1 si sx == sy c ← 1 si sy > sx z ←1 si sx == k c ←1 si k > sx	Compara dos registros o un registro y una constante, si son iguales se activara la bandera zero, si sy es mayor a sx se activara la bandera carry, de lo contrario las banderas permanecerán inactivas. Con esta operación los datos de los registros no se modifican

12.3.4 Instrucciones de control de flujo del programa

En la ejecución del programa el contador del programa (pc) lleva el registro de la instrucción que se está ejecutando y al finalizar continua a la siguiente instrucción. Las instrucciones de control de flujo modifican el orden de ejecución del programa ya que varían el valor del contador de programa.

Al utilizar la instrucción **jump** una nueva dirección en el contador del programa (pc) borrando la anterior, el programa continúa ejecutándose normalmente en la nueve dirección después de esto.

La instrucción **call** por el contrario carga una nueva dirección en el contador del programa (pc) para que el programa siga ejecutándose desde un nuevo punto, pero a diferencia de la instrucción jump, en esta caso la antigua dirección es almacenada en memoria.

Cuando se utiliza **return** se obtiene la dirección que fue almacenada en memoria por la instrucción call, esta dirección es aumentada en uno y se almacena en el contador del programa (pc) para continuar desde el punto en que se había dejado.

Las instrucciones pueden observarse en la tabla de la figura siguiente.

Figura 75 Instrucciones de control de flujo para Picoblaze.

Instrucción	Operación	Explicación
jump AAA	pc ← AAA	La dirección AAA se almacena en pc.
jump c AAA jump nc AAA	si c== 1 pc ← AAA si c==0 pc ← AAA	Si el valor del carry es uno se almacena AAA en pc, de lo contrario pc = pc+1 Si el valor del carry es cero se almacena AAA en pc, de lo contrario pc = pc+1
jump z AAA jump nz AAA	si z== 1 pc ← AAA si z==0 pc ← AAA	Si el valor de zero es uno se almacena AAA en pc, de lo contrario pc = pc+1 Si el valor de zero es cero se almacena AAA en pc, de lo contrario pc = pc+1
call AAA	pc ← AAA	La dirección AAA se almacena en pc.
call c AAA call nc AAA	si c== 1 pc ← AAA si c==0 pc ← AAA	Si el valor del carry es uno se almacena AAA en pc, de lo contrario pc = pc+1 Si el valor del carry es cero se almacena AAA en pc, de lo contrario pc = pc+1
call z AAA call nz AAA	si z== 1 pc ← AAA si z==0 pc ← AAA	Si el valor de zero es uno se almacena AAA en pc, de lo contrario pc = pc+1 Si el valor de zero es cero se almacena AAA en pc, de lo contrario pc = pc+1
return		Se carga la dirección existente antes de la ejecución de la instrucción call
return c return nc		Si el carry es uno se carga la dirección existente antes de la ejecución de la instrucción call Si el carry es cero se carga la dirección existente antes de la ejecución de la instrucción call
return z return nz		Si zero es uno se carga la dirección existente antes de la ejecución de la instrucción call Si zero es cero se carga la dirección existente antes de la ejecución de la instrucción call

12.3.5 Instrucciones de movimiento de datos.

Los datos almacenados en registros pueden moverse en otros registros, entre el registro y la RAM o entre registros y puertos de entrada o salida.

Las instrucciones de movimiento de datos pueden observarse en la tabla de la figura siguiente.

Figura 76 Instrucciones de movimiento de datos para Picoblaze.

Instrucción	Operación	Explicación
load sx, sy load sx, k	sx ← sy sx ← k	Mueve el dato del registro sy al registro sx Mueve el dato de K al registro sx
fetch sx, sy fetch sx, DR	sx ← RAM[sy] sx ← RAM[DR]	Mueve el dato entre la RAM y un registro, la dirección de la RAM se da de manera indirecta utilizando un registro Mueve el dato entre la RAM y un registro, la dirección de la RAM se da de manera directa (DR)
store sx, sy store sx, DR	RAM[sy] ← sx RAM[DR] ← sx	Mueve el dato entre un registro y la RAM, la dirección de la RAM se da de manera indirecta utilizando un registro Mueve el dato entre un registro y la RAM, la dirección de la RAM se da de manera directa (DR)
input sx, sy input sx, k	port_id ← sy sx ← in_port port_id ← k sx ← in_port	El registro sy identifica de dirección del Puerto en donde se recibirá la información, la cual se guarda en sx La constante k identifica de dirección del Puerto en donde se recibirá la información, la cual se guarda en sx
output sx, sy output sx, k	port_id ← sy out_pot ← sx port_id ← k out_pot ← sx	El registro sy identifica de dirección del Puerto en donde se enviara la información, la cual está almacenada en sx La constante k identifica de dirección del Puerto en donde se enviara la información, la cual esta almacenada en sx

12.3.6 Instrucciones de interrupción.

Picoblaze cuenta con un espacio especial de memoria dedicada la ejecución de interrupciones, el cual comienza desde la dirección 3FF.

Cuando las interrupciones son activadas, la normal ejecución del programa se interrumpe para dar paso a la ejecución de las instrucciones dentro de la interrupción, la dirección actual del contador del programa es almacenada y se reemplaza con el valor 3FF, se almacenan también los valores actuales de las banderas zero y carry, y se deshabilita la bandera de interrupción. Una vez se haya terminado la ejecución de la interrupción se regresa al punto anterior en el programa.

Las instrucciones de interrupción pueden observarse en la tabla de la siguiente figura.

Figura 77 Instrucciones de interrupciones para Picoblaze.

Instrucción	Operación	Explicación
returni disable	i ← 0	Regresa de la interrupción desactivando la bandera de interrupción
returni enable	i ← 1	Regresa de la interrupción dejando activada la bandera de interrupción
enable interrupt	i ←1	Activa la bandera de interrupción, lo que habilita las interrupciones del programa
disable interrupt	i ← 0	Deshabilita las interrupciones del programa

12.3.7 Declaraciones en Picoblaze.

Las declaraciones utilizadas en Picoblaze son.

- **address:** Especifica la dirección que será utilizada en la memoria del programa
- **namereg**: Asigna nombres especiales a los registros de Picoblaze para facilita la lectura del programa
- **constant**: Asigna nombres representativos para valores utilizados en el programa, esto facilita la lectura del mismo.

13 PRÁCTICAS

El texto contiene cinco prácticas propuestas que buscan generar habilidades y conocimientos sobre el lenguaje de programación Verilog y la tarjeta de desarrollo Spartan3A, construyendo así las bases necesarias para el manejo no solo a un nivel básico sino también más complejo de las características y utilidades del lenguaje Verilog.

Cada una de las prácticas cuenta con objetivos específicos que tienen como finalidad dar un ejemplo del modo de empleo de algunos de los elementos contenidos en la tarjeta de desarrollo como los pulsadores, Switches, LEDs, perilla de rotación, de puertos como el VGA y PS2, de características de la FPGA como el Picoblaze, además de ejemplificar estructuras del lenguaje Verilog como las maquinas de estado.

Aunque las prácticas no toquen todos y cada uno de los elementos de la tarjeta de desarrollo o del lenguaje Verilog si aseguran su habilidad de manejo ya que presentan bases de funcionamiento similares.

Una breve explicación de cada una de las prácticas y su propósito se dará a continuación. Una explicación más extensa de ellas se estará disponible en secciones posteriores.

13.1 PRÁCTICA 1: CONCEPTOS BASICOS.

Esta es la práctica introductoria al lenguaje de programación en hardware Verilog, al uso de algunos elementos de la tarjeta de desarrollo, al manejo del software ISE Project navigator el cual realizara la síntesis e implementación de los programas creados y al uso del software iMPACT el cual ejecuta la programación en la FPGA.

Los elementos utilizados a lo largo de esta práctica corresponden a los pulsadores, los Switches, los LEDs, la salida de audio y la perilla giratoria, cuyo control es logrado a través de la implementación de seis módulos que contienen funciones sencillas y que sirven como practica a muchas de las estructuras y operaciones de Verilog vistas en capítulos anteriores.

Presenta además el uso de un módulo estructural que realiza la función de interfaz entre todos los módulos y la tarjeta de desarrollo.

13.2 PRÁCTICA 2: MAQUINAS DE ESTADO

Como se había explicado anteriormente las maquinas de estado son importantes en el desarrollo de control de procesos que requieran un orden determinado, por esta razón ellas se constituyen como la base de control de muchos de los elementos periféricos de la tarjeta de desarrollo. Con la realización de la presente práctica se buscan sentar las bases de los códigos en Verilog correspondientes a maquinas de estado mediante la realización de dos sencillos procesos.

El primer proceso a realizar consiste en diseñar un sistema de control de entrada y salida de personas en un determinado punto mediante la ayuda de sensores.

En el segundo proceso se implementará un sistema anti rebote para entradas mecánicas que eliminará los glitches antes y después de la transición de la señal.

13.3 PRÁCTICA 3: PUERTO PS2/ PICOBLAZE/ LCD

El objetivo principal de esta práctica es el de realizar una aplicación dirigida a uno de los elementos periféricos de la tarjeta utilizando maquinas de estado, además de realizar un proceso sencillo en Picoblaze para ejemplificar su uso y funcionamiento enfocado al manejo del display LCD, otro de los elementos contenidos en la tarjeta de desarrollo.

En la primera aplicaciones se ejemplificará la creación de un módulo que realice un proceso de recepción de señales mediante una máquina de estados para poder obtener los datos enviados a través del puerto PS2, sentando así las bases de creación de módulos que sirvan como interfaz entre la FPGA y los elementos periféricos de la tarjeta de desarrollo.

En la segunda aplicación se utilizará el microcontrolador embebido que posee la FPGA para realizar el proceso de control del display LCD de la tarjeta de desarrollo para ejemplificar así el uso de ambos elementos.

Finalmente, se realizará una unión entre todas estas aplicaciones mediante un módulo estructural para lograr que funcionen en conjunto.

13.4 PRÁCTICA 4: PUERTO VGA

El propósito de esta práctica es reforzar las habilidades obtenidas en actividades anteriores sobre el manejo de elementos periféricos, demostrando que una vez se conozcan las bases de funcionamiento de estos, la creación de un módulo Verilog que realice su control no es una tarea compleja.

En este caso el elemento controlado corresponde al puerto VGA, mediante el cual se buscará crear elementos visibles en una pantalla compatible con

tecnología VGA los cuales posean tamaños y formas manipulables, y que además realicen movimientos dentro de un área previamente especificada.

13.5 PRÁCTICA 5: CORE GENERATOR

Como objetivo principal, esta práctica plantea la forma de uso de una de las características de las tarjetas de XILINX, el Core Generator para crear módulos de manera sencilla y rápida que realicen funciones que van desde un nivel de baja hasta uno de alta complejidad

Para ejemplificar el uso del Core Generator, se creará una memoria de tipo FIFO a la cual se le podrán controlar los momentos de escritura y lectura a deseo del usuario, y que brindara una visualización de su estado y salida mediante la ayuda de los LEDs.

14 PRÁCTICA 1: CONCEPTOS BASICOS

14.1 OBJETIVOS

- Conocer la sintaxis del lenguaje orientado a hardware Verilog
- Obtener conocimientos acerca de la estructura que manejan los programas en lenguaje Verilog
- Relacionarse con las interfaces básicas que ofrece la tarjeta de desarrollo Spartan3A
- Familiarizarse con el entorno del software ISE de Xilinx
- Exponer el proceso necesario para crear un proyecto en el software ISE de Xilinx

14.2 RECURSOS UTILIZADOS

- Tarjeta de desarrollo Spartan3A
- Computadora con software ISE Project Navigator
- LEDs
- Switches
- Pulsadores
- Perilla giratoria (Rotary Knob)

14.3 PREREQUISITOS

Para el desarrollo de la siguiente práctica se deberá contar con conocimientos básicos sobre conceptos de digitales y Verilog, de manera que se puedan comprender a totalidad las actividades realizadas.

14.4 EXPLICACIÓN DE LA PRÁCTICA.

El principal objetivo que presenta esta práctica es el de familiarizarse tanto con el lenguaje Verilog, como con el software utilizado para su síntesis e implementación y con algunas características incluidas en la tarjeta de desarrollo Spartan3A.

La práctica contará con varios módulos que realizarán las siguientes funciones:

- Se realizará un control de la frecuencia del reloj maestro de la FPGA, para lograr así obtener diferentes frecuencias para propósitos diferentes, este módulo se llamará variador_frecuencia.

- Otro módulo que será llamado contador, estará encargado de realizar un conteo binario con control de conteo ascendente o descendente.
- Un generador de una secuencia pseudo-aleatoria de 7 bits, logrado a través del uso de un Language Template (plantilla predefinida en lenguaje Verilog) que se llamara LFSR_7bits.
- Una secuencia de corrimiento llamada rotación_leds
- Se realizará el control del Rotary Knob incluido en la tarjeta, diferenciando entre la dirección de giro y la acción de oprimir del switch central referenciado como control_rot_knob.

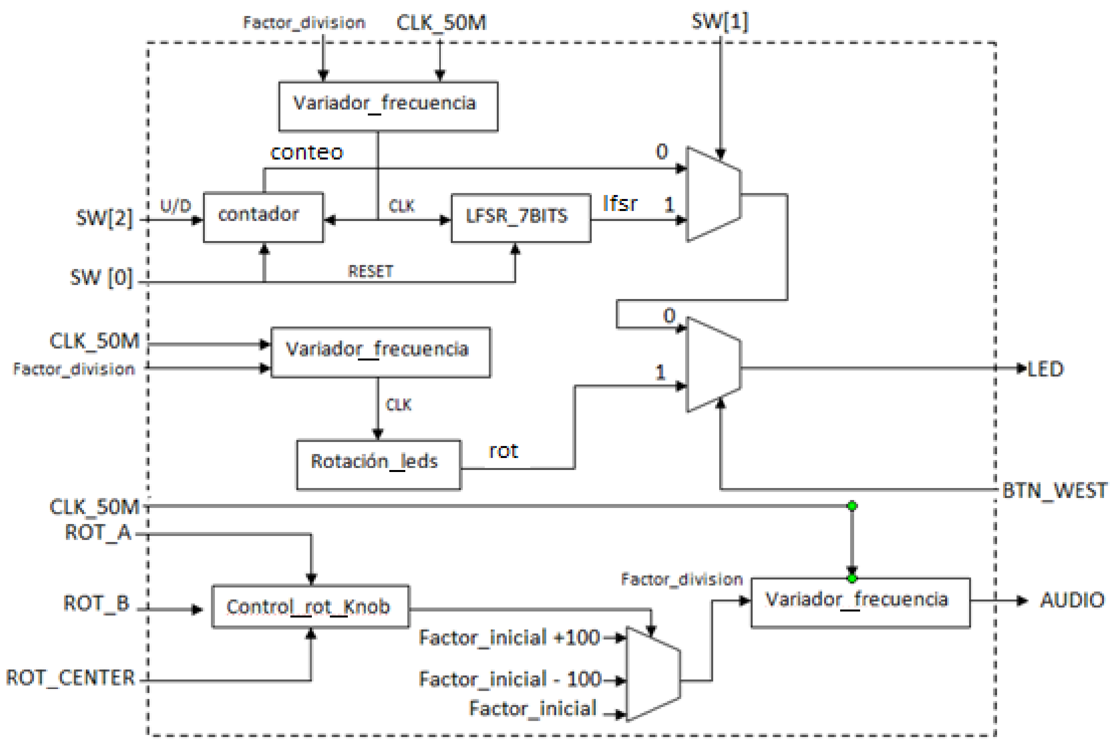
Una vez se tengan los módulos anteriores funcionando, se unirán a un módulo estructural que instanciara cada módulo para realizar su respectiva función a la vez que manejara las entradas y salidas de y hacia la FPGA.

EL funcionamiento del módulo estructural será como se explica a continuación:

- El módulo realizará continuamente las tareas de generación de la frecuencia adecuada para los módulos LFSR_7bits, contador y rotación_leds. También mantendra activado el módulo control_rot_knob para detectar cualquier actividad de la entrada de la perilla giratoria.
- Por medio del switch SW[0], se podrá controlar la entrada reset de los módulos LFSR_7bits y contador. Es decir al encontrarse en nivel alto ninguno de los dos módulos podrá realizar su función, y al encontrarse en nivel bajo ambos módulos realizaran sus respectivas secuencias.
- Utilizando el switch SW[1], se seleccionará la secuencia que será mostrada en los LEDs de la tarjeta Spartan3A. En nivel alto se visualizará la secuencia pseudo-aleatoria de bits y estando en nivel bajo se podrá observar el conteo binario.
- El switch SW[2] estará encargado del control up/down del contador, en nivel alto deberá realizarse una suma ascendente.
- Mientas el pulsador BTN_WEST este activo, se visualizará en los LEDs el funcionamiento del módulo rotación-Leds, omitiendo las dos secuencias anteriores.
- Mediante el uso de variador_frecuencia, se generará una frecuencia que podrá ser escuchada mediante audífonos o parlantes en el conector de audio de la tarjeta, esta frecuencia podrá ser aumentada o disminuida dependiendo de la dirección de giro del Rotary Knob en la tarjeta y deberá regresar a una frecuencia inicial cuando el Rotary Knob sea presionado.

El diagrama de bloques que describe el funcionamiento en conjunto de la práctica descrita se puede observar en la siguiente figura, donde cada módulo corresponde a un bloque con sus respectivas entradas y salidas. El módulo estructural es representado por el rectángulo de líneas discontinuas que encierra todo como conjunto, exponiendo las conexiones entre módulos y las operaciones que determinan la salida.

Figura 78 Diagrama de bloques de la práctica 1.

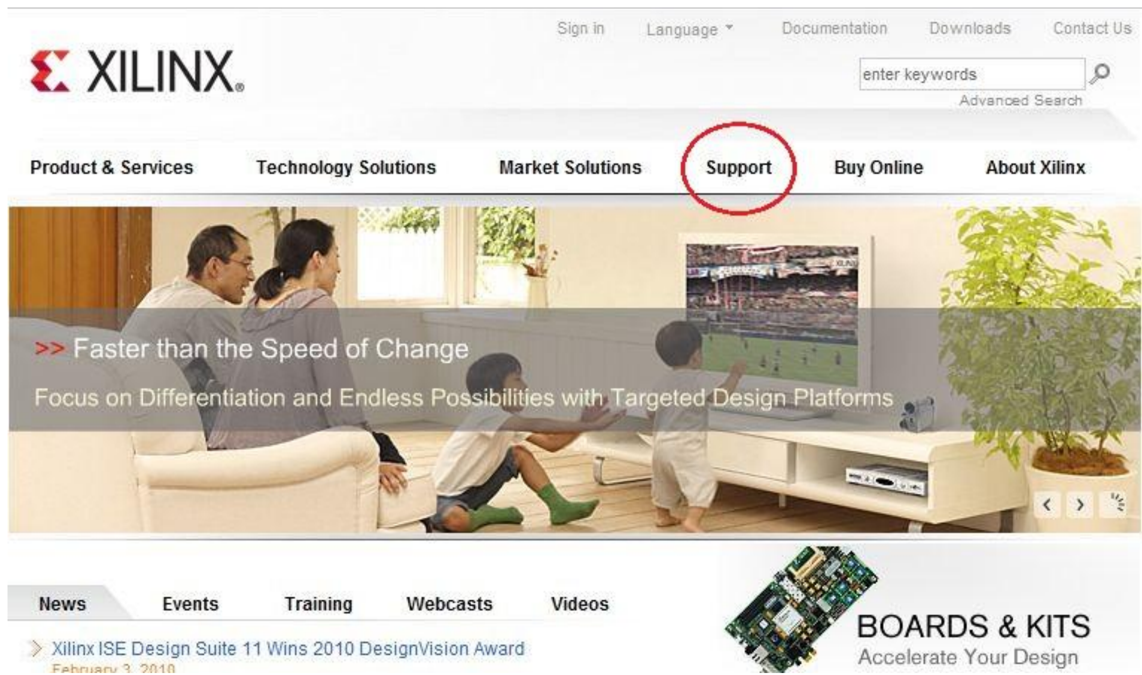


Una explicación más detallada de cada una de las operaciones presentes en esta práctica se encuentra en la siguiente sección.

14.5 DESARROLLO DE LA PRÁCTICA.

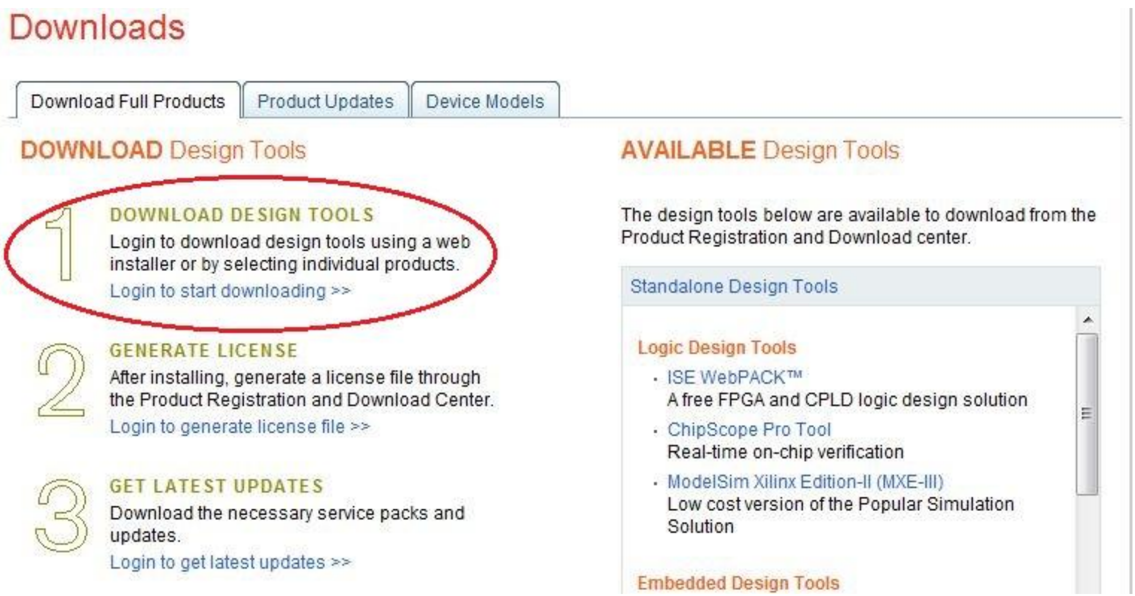
Para poder trabajar con Verilog, se necesita un software que sintetice e implemente el diseño que se esté realizando, el programa que se utilizara será el ISE project Navigator, de la empresa Xilinx, versión 11.1. Este puede conseguirse en la página web de Xilinx (www.xilinx.com), en la sección Support como se muestra en la figura siguiente.

Figura 79 Website de xilinx[45]



En la sección **Downloads** existen dos maneras de descargar Ise WebPack, en este caso se irá a la opción **download design tools** como se muestra a continuación.

Figura 80 Descarga ISE Webpack.[46]



Después de esto aparecerán dos ventanas como las mostradas en las siguientes dos figuras, se debe crear una cuenta y rellenar el formulario en cada una respectivamente.

---

[45] XILINX. Internet: www.xilinx.com [consulta: 18 enero 2010]
[46] XILINX. Internet: www.xilinx.com [consulta: 18 enero 2010]

Figura 81 Creación de un nuevo registro.[47]

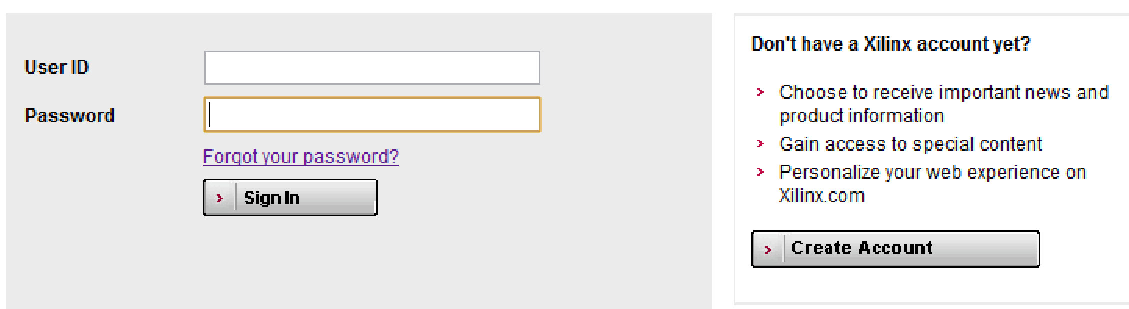


Figura 82 Formulario para descargas[48]

Product Download and Licensing

US export regulations require that your shipping address be verified bef
complete information for immediate processing. Sorry, **addresses with**

Fields marked with an asterisk * are required.

Department

Industry *

Address 1 *

Address 2

City *

Postal/Zip Code *

Country *

State/Province — Select One — help

Phone (include area code) *

Corporate Email

Fax (include area code)

Company/Organization *

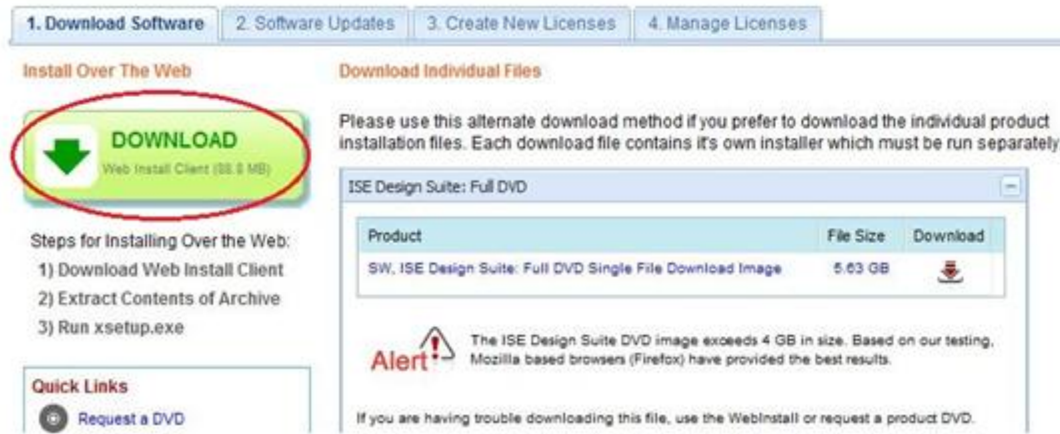
Update profile

Next

[47] XILINX. Internet: www.xilinx.com [consulta: 18 enero 2010]
[48] XILINX. Internet: www.xilinx.com [consulta: 18 enero 2010]

Una vez el registro se haya terminado, la siguiente pagina aparecerá y se debe seleccionar la opción descargar Web Install como se indica en la figura, ya que esto hará más ágil el proceso de descarga.

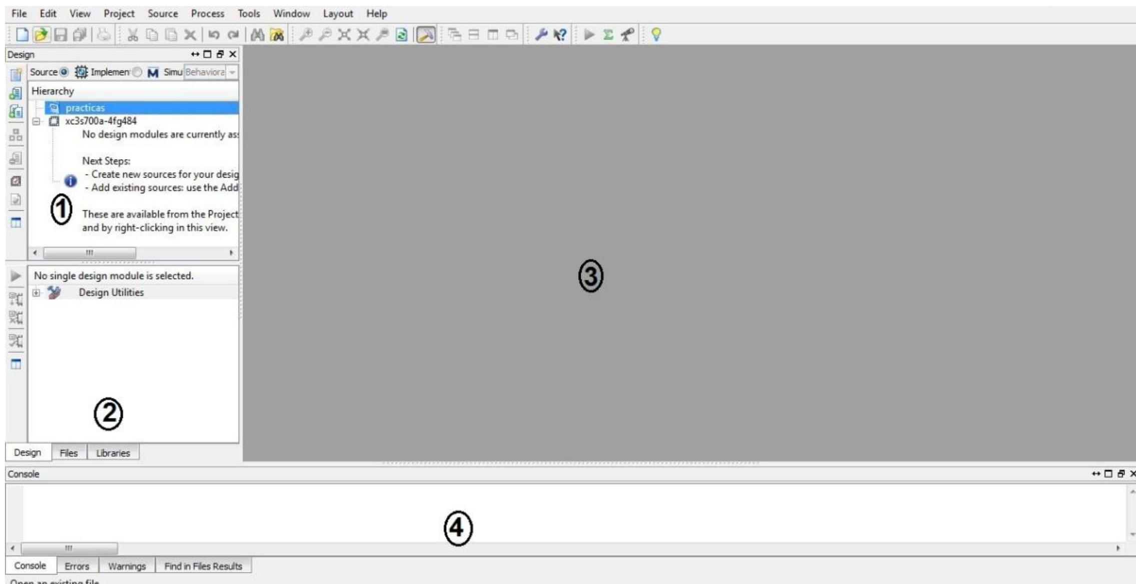
Figura 83 Descarga del Web Install[49]



Una vez la descarga haya finalizado los contenidos del archivo se extraen y se ejecuta el archivo **Xsetup.**

Cuando el proceso de instalación haya terminado y se ejecute el programa, la ventana que se muestra deberá verse como en la figura siguiente.

Figura 84 ISE Project Navigator



En la primera zona se encuentra la ventana **SOURCE** en donde se visualizaran los módulos en los que se están trabajando y donde también se podrán administrar la inclusión de nuevos módulos o su remoción.

[49] XILINX. Internet: www.xilinx.com [consulta: 18 enero 2010]

En la segunda zona se observa la ventana **PROCESSES**, en esta sección se encuentran ubicadas las opciones para realizar los procesos de síntesis, implementación, creaciones de limitaciones para el diseño y generación del archivo de programación entre otras cosas.

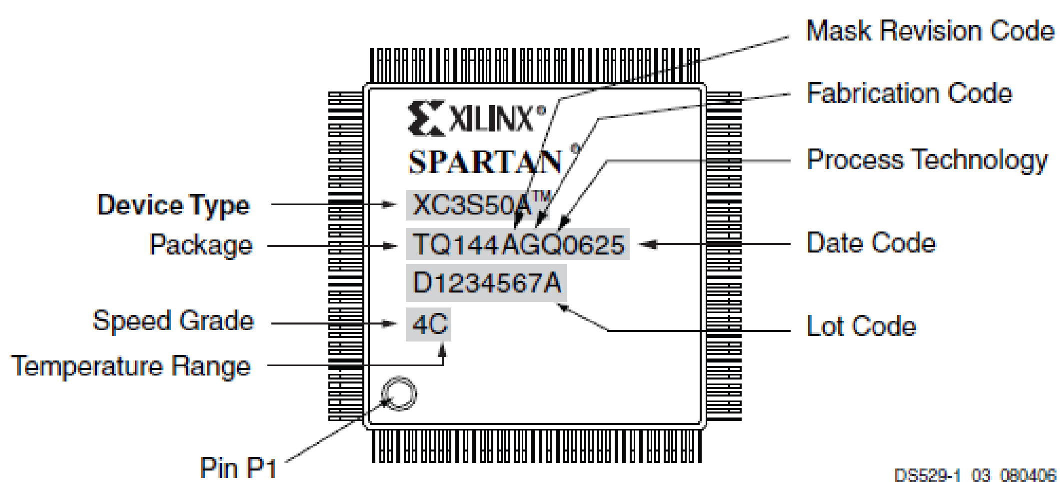
La zona tres es el área en el que se trabaja en el programa en sí, en esta área se visualizaran y editaran por ejemplo los códigos de los módulos y podrá observarse un resumen del proceso.

En la cuarta y última zona podrán observarse los mensajes de error y advertencia generados por el programa, además de otros procesos.

14.5.1 Creación de un nuevo proyecto.

Para comenzar con el uso del software aplicado a la tarjeta Spartan 3A, se debe primero crear un proyecto con las especificaciones propias del chip, estas se encuentran en el circuito integrado correspondiente a la FPGA y se ven de la siguiente manera.

Figura 85 Especificaciones de la FPG[50]



De acuerdo a esto, el integrado con el que se trabajara tiene las especificaciones expuestas en la siguiente figura.

Figura 86 Especificaciones Spartan3A

XC3S700A -4 FG848 C

Tipo de dispositivo

Grado de velocidad

Rango de temperatura

Tipo de integrado/Numero de pines

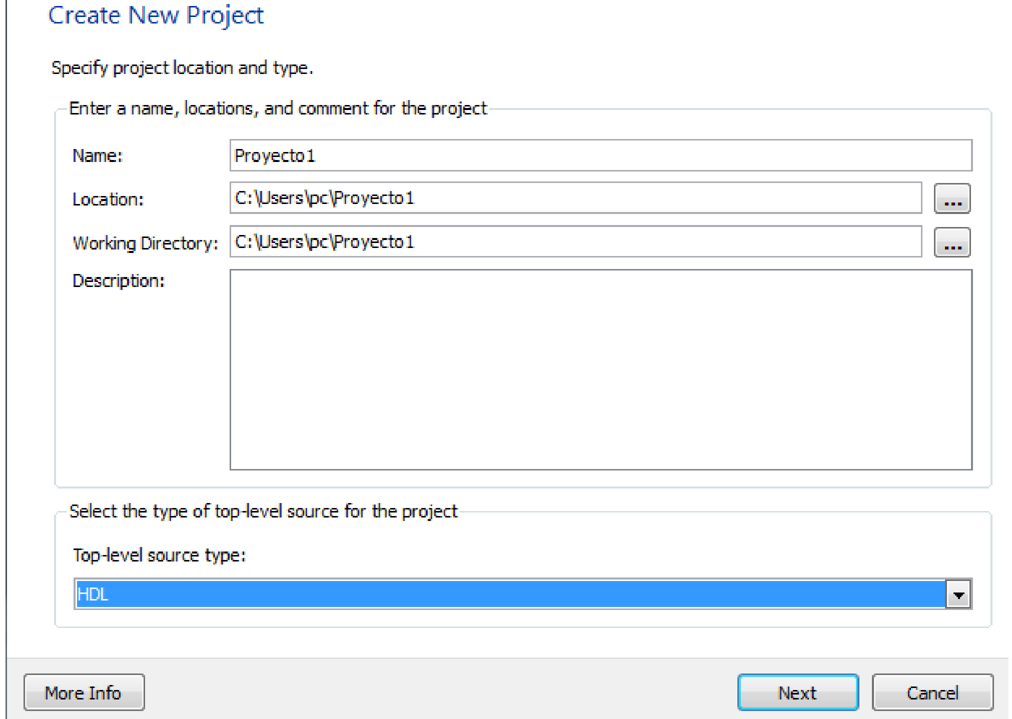
[50] XILINX, Spartan 3a FPGA Family: Datasheet. March 2006. Pag 6

Los datos que nos proporciona el integrado de la FPGA son los que utilizaremos en el momento de crear un proyecto nuevo.

Para la creación del nuevo proyecto se deben seguir los siguientes pasos:

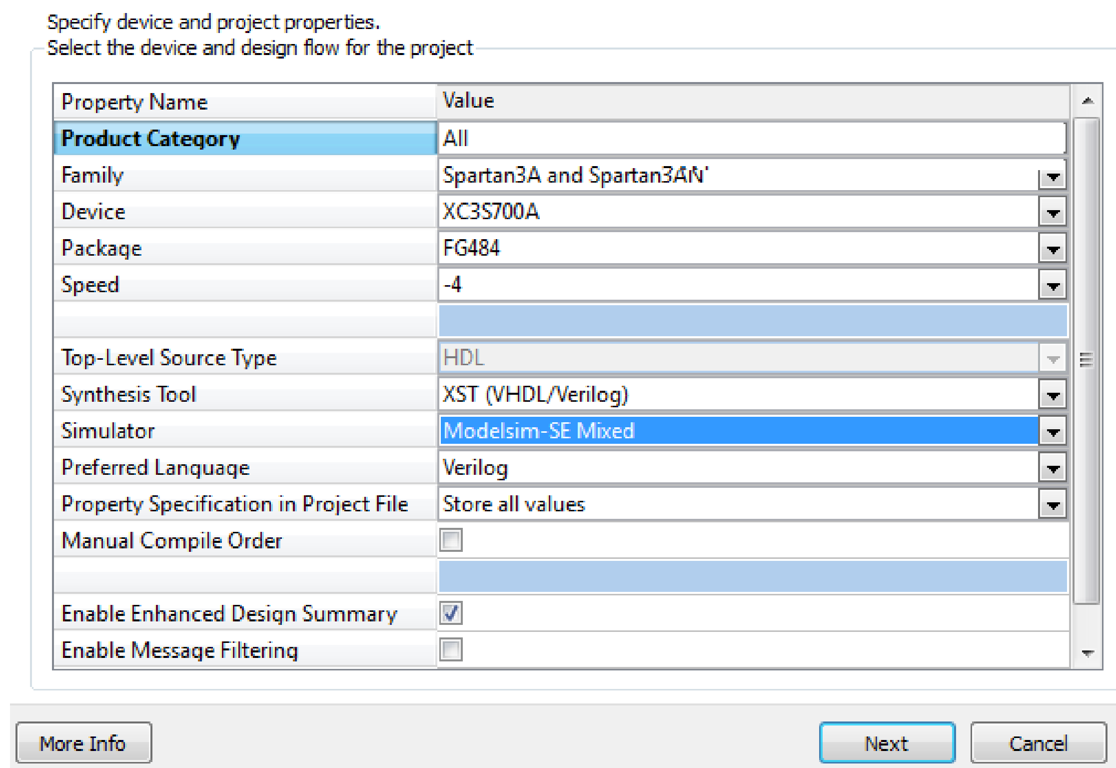
1. Se debe tener abierto el software ISE Project Navigator
2. Se sigue el camino **File >> New Project** y se abrirá la ventana que se observa en la siguiente figura.

Figura 87 Ventana para la creación de un nuevo proyecto



3. Después de verificar que el lenguaje fuente sea HDL y de llenar los espacios requeridos se hace clic en **next**.
4. Se especifican las propiedades de la FPGA antes mencionadas como se muestra en la siguiente figura.

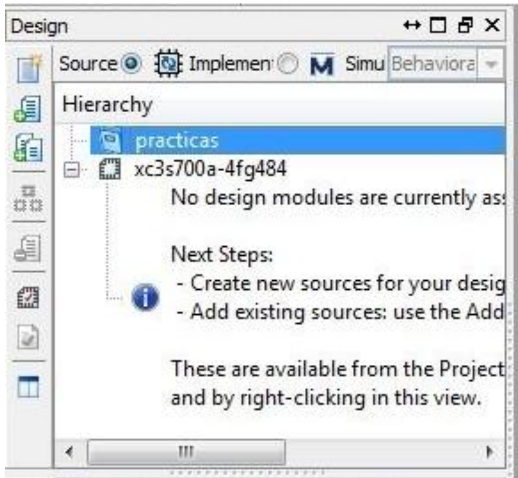
Figura 88 Ingreso de las propiedades de la FPGA



Después de haber rellenado todos los campos de la manera correcta, se puede continuar en el proceso de creación del proyecto.

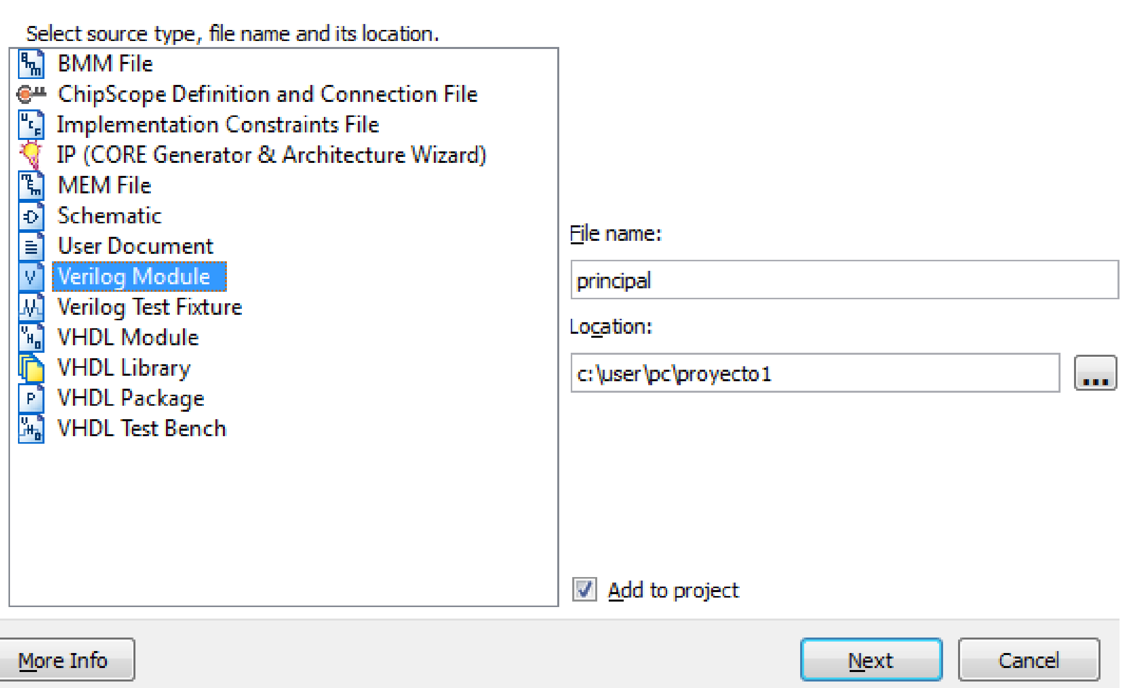
5. Las dos siguientes ventanas que aparecen, son para añadir fuentes nuevas o ya existentes al proyecto creado, se continuara pulsando **next** ya que esto se realizara manualmente más adelante.
6. La ventana final brinda un informe sobre el proyecto, al pulsar el botón **finish,** se habrá terminado el proceso de creación de un nuevo proyecto, la ventana **SOURCE** se verá como lo demuestra la siguiente figura.

Figura 89 Ventana source



Para crear un módulo en el cual comenzar a trabajar, se debe hacer clic derecho en la ventana SOURCE y elegir la opción ***new source,** aparecerá la ventana que se muestra en la figura siguiente.

Figura 90 Ventana de creación de nuevo módulo Verilog

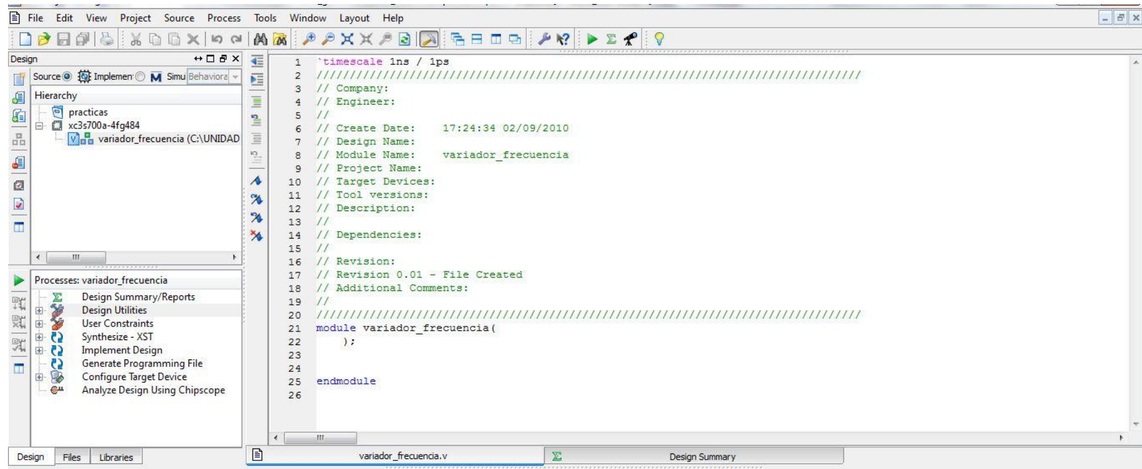


En esta ventana se debe seleccionar la opción **Verilog Module**, insertar el nombre del módulo –en este caso principal- y oprimir **Next**, **Next, Finish.**

Una vez terminados los pasos anteriores el módulo está listo para trabajar con él, en el área de trabajo se debe abrir el módulo creado, de no ser así se puede hacer doble clic en el nombre del módulo en la ventana SOURCE.

Una vez se completen los pasos anteriores, una vista general de ISE Project Generator se vería como lo muestra la siguiente figura.

Figura 91 Nuevo módulo



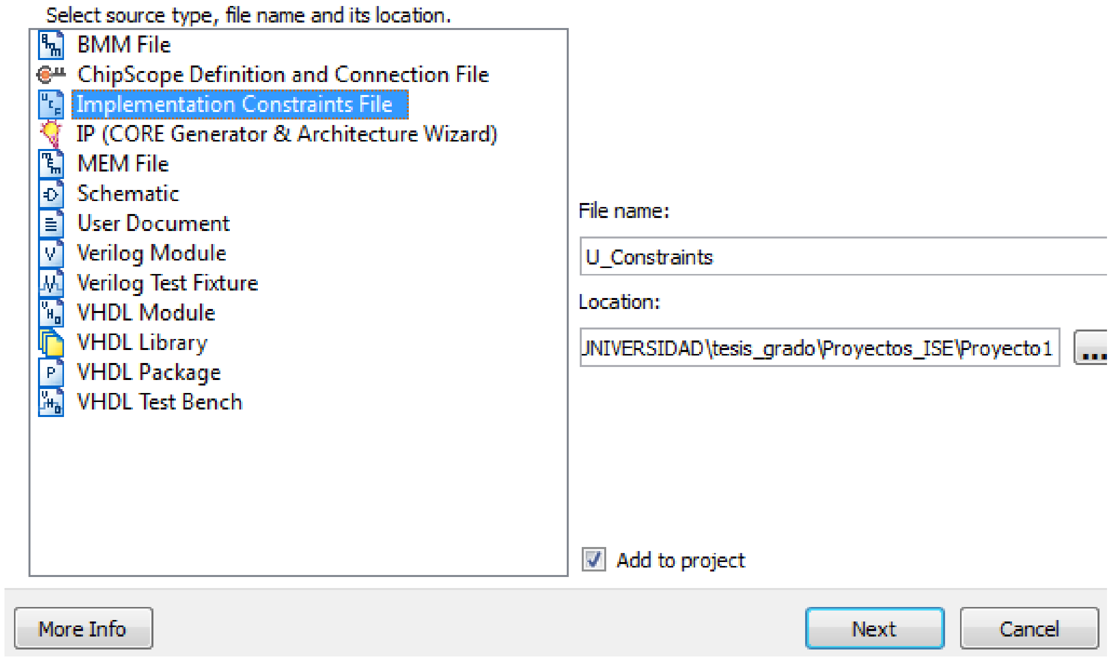
En este caso se ha creado inicialmente el módulo estructural, dicho modulo debe poseer un archivo UCF (User´s Constraints File) para permitir que la FPGA tenga conexión con todos los elementos periféricos de la tarjeta de desarrollo que se quieran utilizar. Este código ya fue explicado y definido en las paginas anteriores del libro pero si se desea, Xilinx tiene disponible el archivo completo del UCF en la siguiente dirección:
http://www.xilinx.com/products/boards/s3astarter/files/s3astarter.ucf

Debe recordarse que todo elemento que se nombre en el UCF, debe tener sus respectivas entradas y salidas en el módulo estructural, de lo contrario se generaran errores en el proyecto que impedirán la sintetización de códigos.

Para la creación de este UCF se siguen los siguientes pasos:

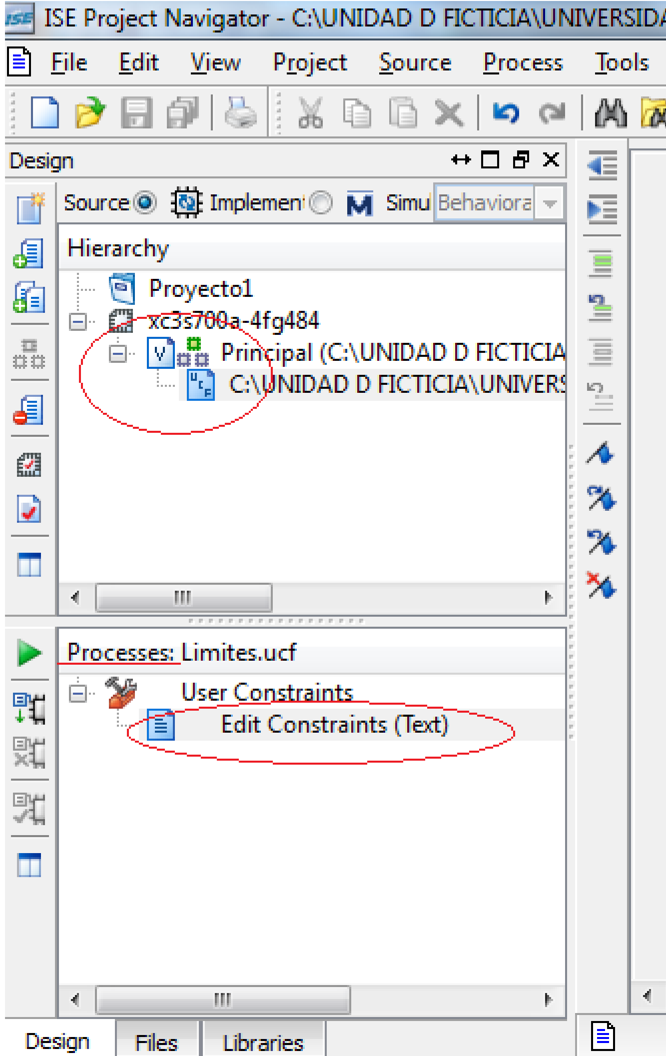
1. Con el módulo estructural seleccionado se deben seguir de nuevo los pasos para crear una nueva fuente. La diferencia es que en este caso se selecciona la opción **Implementation Constaint File** como se aprecia en la siguiente figura.

Figura 92 Ventana para la creación del UCF



2. En el menú de procesos se selecciona la opción **Edit constraints** como se indica en la figura.

Figura 93 Edición del archivo UCF



Al abrir la opción Edit Constraints, se abre en el espacio de trabajo una página en blanco, allí se colocaran las líneas de instrucción necesarias para cada uno de los elementos periféricos que se utilizaran, para este caso se debe habilitar el nivel correcto de voltaje, el reloj de 50M, los LEDs, la perilla giratoria y los Switches.

En este caso se utilizaron solo las secciones necesarias del archivo brindado por Xilinx, una visión de un fragmento del archivo UCF para esta práctica es como lo muestra la siguiente figura.

Figura 94 Fragmento del archivo UCF

```
Tools  Window  Layout  Help
47  ################################################################################
48  ## Copyright (c) 2006, 2007 Xilinx, Inc.
49  ## This design is confidential and proprietary of Xilinx, All Rights Reserved.
50  ################################################################################
51
52  # On this board, VCCAUX is 3.3 volts.
53
54  CONFIG VCCAUX = "3.3" ;
55
56  # Configure SUSPEND mode options.
57
58  CONFIG ENABLE_SUSPEND = "FILTERED" ;
59
60  # FILTERED is appropriate for use with the switch on this board. Other allowed
61  # settings are NO or UNFILTERED.  If set NO, the AWAKE pin becomes general I/O.
62  # Please read the FPGA User Guide for more information.
63  ################################################################################
64  # Discrete Indicators (LED)
65  ################################################################################
66
67  NET "LED<0>"       LOC = "R20"  | IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = SLOW ;
68  NET "LED<1>"       LOC = "T19"  | IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = SLOW ;
69  NET "LED<2>"       LOC = "U20"  | IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = SLOW ;
70  NET "LED<3>"       LOC = "U19"  | IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = SLOW ;
71  NET "LED<4>"       LOC = "V19"  | IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = SLOW ;
72  NET "LED<5>"       LOC = "V20"  | IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = SLOW ;
73  NET "LED<6>"       LOC = "Y22"  | IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = SLOW ;
74  NET "LED<7>"       LOC = "W21"  | IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = SLOW
```

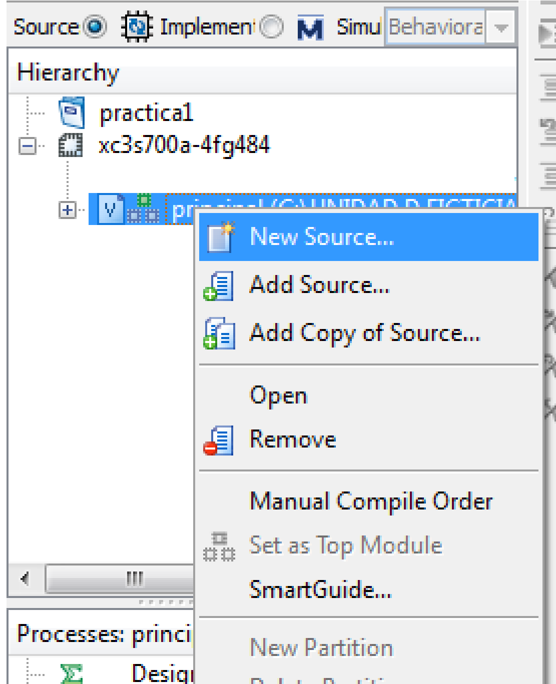
Una vez se hayan creado correctamente el módulo principal y el archivo UCF se puede comenzar con el desarrollo de la práctica.

14.5.2 Funcionamiento del módulo contador

Para comenzar a trabajar con este módulo es necesario crearlo, esto se realiza siguiendo unos pasos similares a los anteriormente explicados.

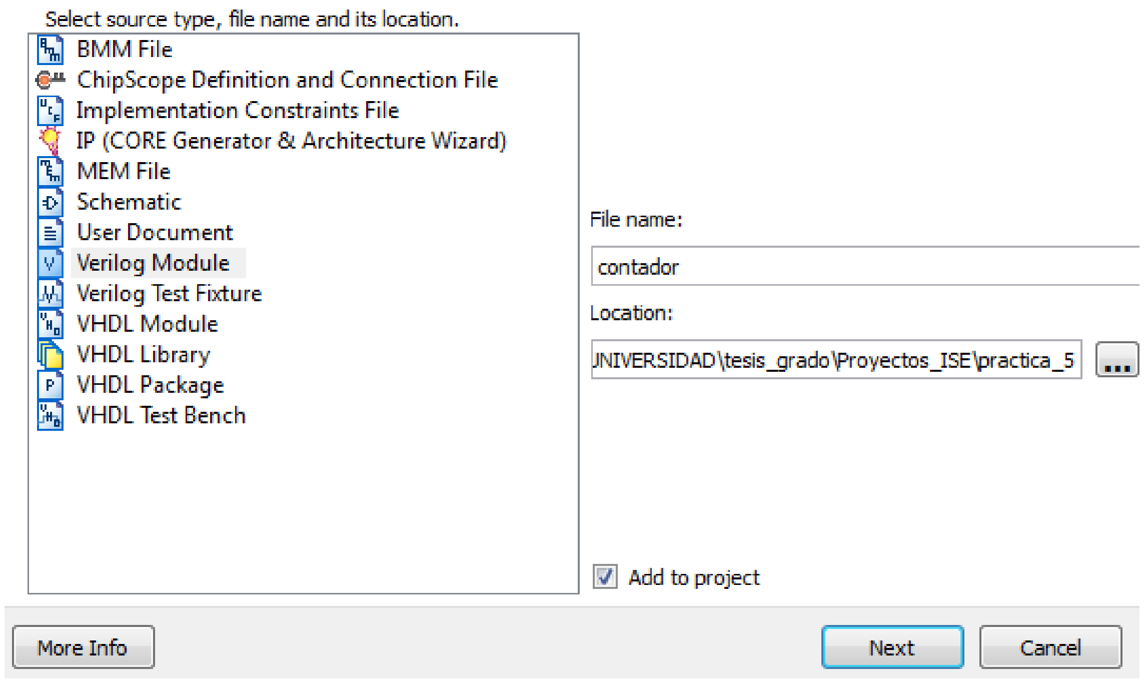
1. En la ventana **source** de debe hacer clic derecho y seleccionar la opción **New Source** en el menú desplegable como se indica en la siguiente figura

Figura 95 Inserción de nuevo módulo al proyecto



2. Seleccionar la opción **Verilog Module** y añadir el nombre del módulo como lo muestra la figura siguiente.

Figura 96 Nuevo módulo contador

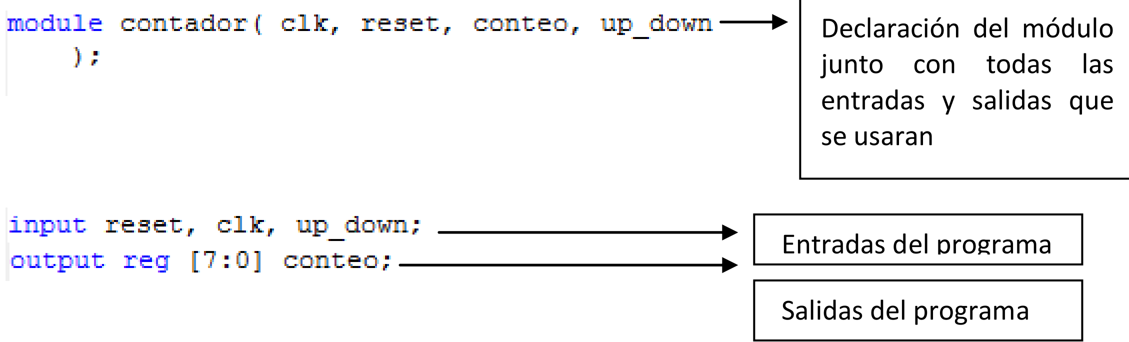


3. Presionar **next** y **finish.**

Con estos pasos completados se debe abrir en ISE el nuevo módulo

Este es un módulo sencillo que realiza la función de un contador binario de 7 bits, cuenta con tres entradas correspondientes a la señal de reloj (clk), el reset (reset) y el control up/down (up_down) que maneja la dirección de conteo del módulo.

El código utilizado para su realización y la explicación de cada una de las instrucciones que contiene se puede encontrar a continuación.



```
always @ (posedge clk)
begin
if (reset) conteo<=0;

else if (up_down)
begin
    if (conteo>8'd127) conteo<=0;
       else conteo<=conteo+1;
end

else if (~up_down)
begin
    if (conteo<=8'd0) conteo<= 8'd128;
     else conteo<=conteo-1;
end
end

endmodule
```

Definición de un Flipflop con reset síncrono

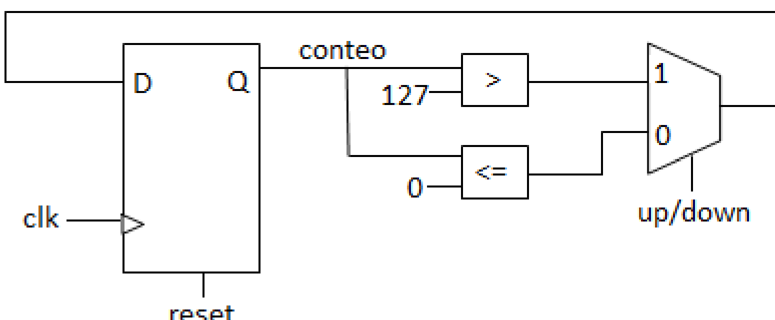
Si up_down tiene nivel alto, realizara un conteo ascendente hasta 127, al llegar a ese valor se reiniciara la variable conteo

Si up_down tiene nivel bajo, realizara un conteo descendente hasta 0, en cero la variable conteo se reiniciara en 128

Aunque el número de bits debería permitir que el conteo se realizara entre 255 y cero, se debe tener en cuenta que la variable conteo puede dar valores positivos y negativos, por esta razón se toman como limites los números cero y 128.

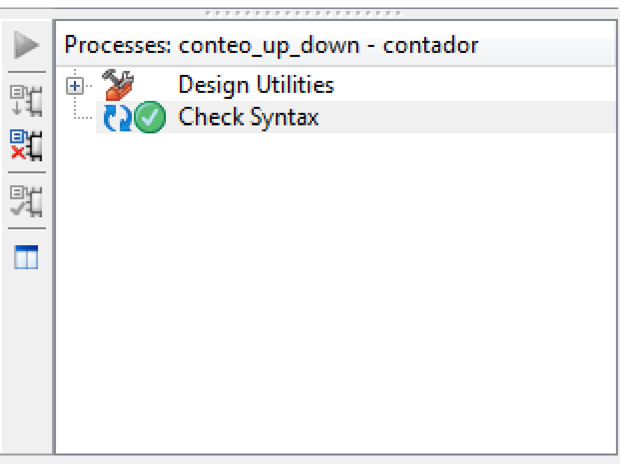
Para enfatizar en la importancia de pensar en términos de hardware al momento de utilizar Verilog, se tradujo el código del módulo anterior a un circuito de acuerdo a la descripción dada en donde las operaciones como comparación se muestran como cajas negras por motivo de facilidad como se observa en la figura siguiente.

Figura 97 Módulo contador en términos de hardware



Después de terminar la escritura de un módulo se debe verificar que la sintaxis este correcta, para esto en la ventana de procesos se debe dar doble clic en la opción **check syntax** visible en la siguiente grafica.

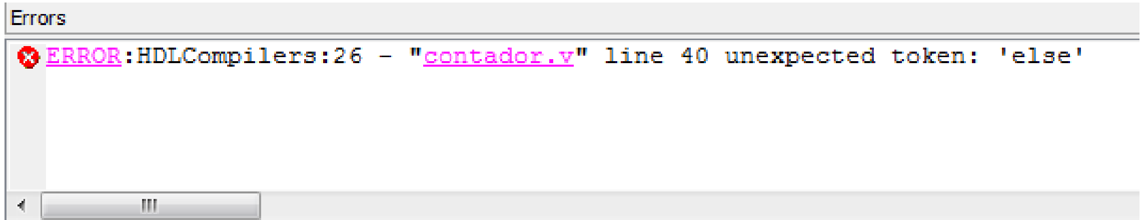
Figura 98 Verificación de la sintaxis del programa.



Si el programa presenta algún error de sintaxis aparecerá una X y los errores se mostraran en la barra de errores donde se indicara el tipo de error y la línea donde se presento.

La barra de errores se encuentra en la parte inferior de la zona de trabajo de ISE, cuando el programa detecta algún error este se muestra como se observa en la figura siguiente

Figura 99 Barra de errores de ISE



En la figura anterior, se presento un error en la línea 40 del módulo contador.v, en este caso el programa no reconoce la palabra clave else debido a que en la línea anterior no se escribió el punto y coma (;). Si se requiere más información sobre el error puede encontrarse haciendo clic en el hipervínculo ERROR.

El proceso de revisión de sintaxis debe realizarse después de realizar cambios a cualquier módulo para evitar problemas posteriores.

14.5.3 Módulo variador_frecuencia

Inicialmente debe crearse un nuevo módulo dentro del proyecto que lleve por nombre variador_frecuencia, una vez creado se puede proceder a editar el código.

La función que realiza el módulo es la de un divisor de frecuencia, el resultado final deberá ser una señal de reloj de una frecuencia menor o igual a la del reloj de entrada.

El código contara con tres entradas, el reloj de entrada (clk), la señal de reset (reset) y el factor de división (factor_division) que corresponderá a la frecuencia por la cual se dividirá el reloj de entrada para dar como resultado la salida del módulo (frecuencia_resultante)

El código para lograr dicho funcionamiento sería el siguiente.

```
module variador_frecuencia
```

Declaración del nombre del modulo

```
(clk, reset, frecuencia_resultante, factor_division);

input clk, reset;
input [31:0] factor_division;

output reg frecuencia_resultante;
```

Declaración de todas las variables input o output que se utilizaran en el modulo.

```
reg [31:0]contador =0;
wire [31:0] factor_medio;
```

Variables internas del programa

```
assign factor_medio = factor_division>>1;
```

La operación >> realizada a factor_division constituye una división por dos, el resultado será almacenado en factor_medio. Esto corresponderá al Duty Cycle del reloj de salida

```
always @ (posedge clk or posedge reset)
begin

if (reset) frecuencia_resultante<=0;
```

El bloque always se activara cada flanco de subida de clk o reset.
Se creara un Flipflop con reset asíncrono

```
else if (contador<factor_medio)begin
 frecuencia_resultante<=0;
 contador <= contador+1;
 end
```

El control de la frecuencia se logra con un contador, el valor del contador determina el nivel de la salida

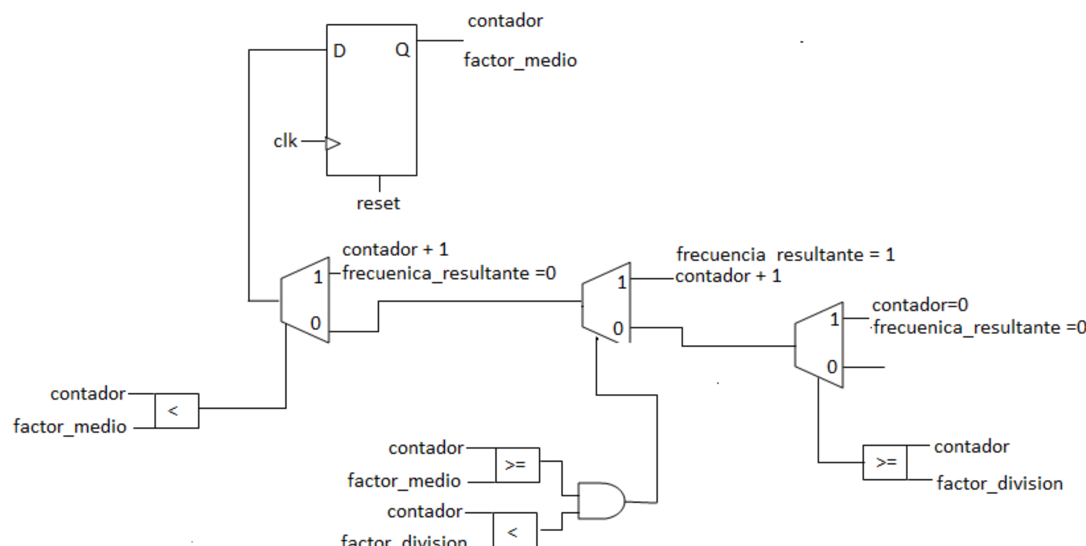
```
else if ((contador>=factor_medio)&&(contador<factor_division))begin
frecuencia_resultante <= 1;
contador<= contador +1;
end
```

Se debe cubrir todo el rango de valores de la variable contador para asegurar un funcionamiento adecuado

```
else if (contador>=factor_division) begin
contador<=0;
frecuencia_resultante<=0;
end
end

endmodule
```

Se debe recordar utilizar los indicadores begin y end si se tiene más de una línea de instrucción

Figura 100 Módulo variador_frecuencia en circuito hardware



Nuevamente al igual que se hizo con el módulo contador, la anterior figura representa el módulo variador_frecuencia en términos de hardware con el fin de brindar al lector una mejor noción de lo que es “pensar en hardware” y así entender el funcionamiento de Verilog. Se recomienda realizar la misma acción con los demás módulos para ganar mayor práctica y comprensión.

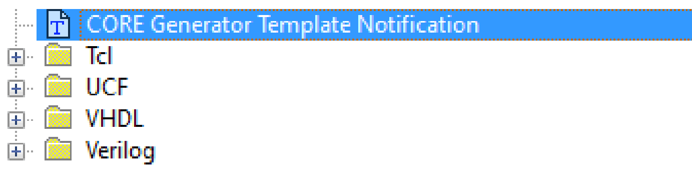
14.5.4 Módulo LFSR_7bits

ISE Project Navigator ofrece la opción de utilizar módulos que ya han sido realizados por otras personas dentro del proyecto personal, estos módulos son conocidos como Language Template.

Para este módulo se utilizara un Language Template, cuya adición al proyecto se realiza siguiendo los siguientes pasos:

1. Se crea un nuevo módulo que se llamara LFSR_7bits que tendrá una entrada de reloj (clk), una entrada para reset (reset) y una salida de 8 bits (lfsr).
2. Se irá a Edit >> Language Templates. Se abrirá la pestaña que se ve en la siguiente figura

Figura 101 Pestaña de Languages Templates



3. Se selecciona **Verilog >> Synthesis Constructs >> Coding Examples>> Counters >> LFSR >> 8bit/w CE & Sync Active High Reset**.
4. En 8bit/w CE & Sync Active High Reset con clic derecho se selecciona la opción **Use in file**. Esto hará que el código del Language Template aparezca en el módulo que se creó.
5. Los siguientes cambios deben realizarse en el código para adaptar el Language Template al módulo creado

<reg_name> = lfsr
<clock> = clk
<reset> = reset

Una vez modificados el nombre de las variables el código de módulo se vera de la siguiente manera.

```
module LFSR_7bits( lfsr, clk, reset
    );

input clk, reset;
output reg  [7:0]lfsr;



   always @(posedge clk)
      if (reset)
         lfsr <= 8'h01;
      else  begin
         lfsr[7:1] <= lfsr[6:0];
         lfsr[0] <= ~^{lfsr[7], lfsr[6:4]};
      end


endmodule
```

Declaración de un Flipflop con reset síncrono

Se realizo un cambio en el número de bit, en vez de 8 se coloca 7

14.5.5 Módulo rotación_leds

La tarea de este módulo será realizar una secuencia para ser visualizada en 8 LEDs, para cumplir este propósito se utilizara el operador de corrimiento >> y <<.

La secuencia funcionara de la siguiente manera: dos LEDs deben estar siempre encendidos, al inicio se encenderá un LED en cada extremo y conforme llegue el pulso del reloj se encenderá el LED adyacente hasta que ambos LEDs se encuentren en la mitad en donde deberán realizar el mismo proceso en la dirección contraria.

El código propuesto utilizado para realizar la función requerida junto con su explicación es el siguiente.

```
module rotacion_leds( clk, reset, rot_led_a , rot_led_b
    );

input clk, reset;
output reg [3:0]rot_led_a =4'd1 ;
output reg [3:0]rot_led_b = 4'd8;

reg init_a=1;
reg init_b=1;
```

Declaración del modulo y variables de entrada y salida

Variables internas que actuaran como bandera para indicar el estado de los bits extremos de rot_led_a y rot_led_b

Secuencia para los primeros cuatro Leds

```
////////////////////////////////////////////////////
///primer segmento
////////////////////////////////////////////////////
always @ (posedge clk)
begin
   if (rot_led_a [1] || rot_led_a [0]) init_a <= 1'b1;
   else if (rot_led_a [2]) init_a <= 1'b0;
   else init_a <= init_a;
end
```

Estructura de un Flipflop

Si el primer o el segundo bit de rot_led_a esta activado, init_a se activa.
Si el segundo led se activa, init_a se desactiva, en cualquier otra condición init_a conserva su valor anterior. Esto indica los extremos

```
always @ (posedge clk)
begin

   if (reset)
   begin
       rot_led_a <= 4'd1;

   end

   else
      begin
      if (init_a) rot_led_a <= rot_led_a << 1;
      else rot_led_a <= rot_led_a >> 1;
      end
end
```

Siempre deberán existir dos Leds encendidos por esto se inicializa en uno

Si init_a esta activo el corrimiento será a la izquierda, de lo contrario correrá a la derecha

La segunda parte del programa contendrá esencialmente el mismo principio de funcionamiento, invirtiendo solamente la dirección en la pregunta final.

```
//////////////////////////////////////////////////////
//segundo segmento
//////////////////////////////////////////////////////

always @ (posedge clk)
begin
   if (rot_led_b [2] || rot_led_b [3]) init_b <= 1'b1;
   else if (rot_led_b [1]) init_b <= 1'b0;
   else init_b <= init_b;
end

always @ (posedge clk)
begin

   if (reset)
   begin
       rot_led_b <= 4'd8;
   end

   else
      begin
      if (init_b) rot_led_b <= rot_led_b >> 1;
      else rot_led_b <= rot_led_b << 1;
      end
end
endmodule
```

Pregunta para asignar el nivel a init_b

Inicializa en uno

El valor de init_b determina la dirección de corrimiento de rot_led_b

14.5.6 Módulo control_rot_knob

Una vez conocido el principio de funcionamiento de la perilla giratoria incluida en la tarjeta, entender el código para su control es sencillo.

El Rotary Knob proporciona tres entradas, una de cada pulsador de dirección rot_a y rot_b y una para el pulsador del centro rot_center, realizar el control para la pulsación de la perillla es sencillo debido a que es básicamente igual a

la de uno de los pulsadores ya utilizados, para determinar la dirección de rotación se parte del conocimiento que esta se obtiene mediante el orden de activacion de los pulsadores rot_a y rot_b.

El control se realizo mediante la implementación del siguiente código

```
module control_rot_knob

( rot_a, rot_b, rot_center, clk, rot_left, rot_right, rot_press);


input rot_a, rot_b, rot_center, clk;
output reg rot_left, rot_right, rot_press;

reg  estado;
reg dir;
reg estado_ant;
reg dir_ant;
```

Indica si el giro comienza o termina

Estados anteriores de estado y giro

Diferencia entre la dirección del giro

```
always @ (posedge clk)
begin
case ({rot_a, rot_b})
2'b00: begin estado <= 1'b0; dir<=dir; end
2'b01: begin estado <= estado; dir<=1'b1; end
2'b10: begin estado <= estado; dir<=1'b0; end
2'b11: begin estado <= 1'b1; dir<=dir; end
default: begin estado <= estado; dir<= dir; end
endcase
```

Se concatenan rot_a y rot_b para utilizarlas como variables en la función **case**

Hay más de una instrucción se utiliza begin...end

En cada opción una variable mantiene su valor actual mientras la otra adquiere un nuevo valor.
Si estado es 1, y dir es 1 se estará en la mitad del giro a la derecha

```
estado_ant <= estado;
dir_ant <= dir;

rot_right<= dir_ant&& !estado_ant && estado;
rot_left <= !dir_ant && !estado_ant && estado;
rot_press <= rot_center;

end


endmodule
```

Estado_ant y dir_ant estarán un ciclo de reloj retrasados

De acuerdo al estado de las variables, las salidas son asignadas

14.5.7 Módulo principal

El módulo principal será el módulo estructural del proyecto, dentro de él se debe procurar reducir al máximo las líneas de instrucciones y preferir la instanciación (llamadas) a otros módulos para que realicen la función deseada.

Dichas instanciaciones o llamados a otros módulos se realizan de la siguiente manera.

Nombre_modulo identificador (.entrada(entrada1), .salida (salida1));

Nombre_modulo corresponde al nombre que posee el módulo que se quiere instanciar, identificador es el nombre que recibirá esta instancia o llamado, seguidamente se deben enunciar todas las entradas y salidas que posea el módulo instanciado precedidas de un punto y separadas cada una por coma, cada una de las variables deberán poseer un nuevo nombre para funcionar en el modulo al cual han sido instanciadas, en este caso los nuevos nombre para las variables entrada y salida corresponderán respectivamente a entrada1 y salida1. Ejemplos que brinden más claridad sobre el concepto de instancia se verán más adelante en la explicación de funcionamiento del módulo.

Las entradas y salidas de este módulo corresponderán a los elementos periféricos a usar de la tarjeta de desarrollo, los cuales deben coincidir con los declarados en el archivo UCF.

El código utilizado en el módulo es el siguiente.

```
module Plantilla_Lab_1_Sem_2_2009
(LED, CLK_50M, ROT_CENTER, ROT_A, ROT_B,
 AUD_L, AUD_R, SW, BTN_SOUTH, BTN_WEST,
   BTN_NORTH, BTN_EAST);

input CLK_50M;
input ROT_CENTER;
input ROT_A;
input ROT_B;
input [3:0] SW;
input BTN_SOUTH;
input BTN_WEST;
input BTN_NORTH;
input BTN_EAST;

output reg [7:0] LED;
output reg AUD_L;
output reg AUD_R;
```

Declaración de las entradas y salidas que posee la tarjeta de desarrollo Spartan3A y que serán utilizadas en esta práctica

```
reg [31:0] variacion_frecuencia = 32'd25000000;

always @ ( posedge CLK_50M)
begin
   if (BTN_NORTH) variacion_frecuencia <= variacion_frecuencia+20;
   else if (BTN_EAST) variacion_frecuencia <= variacion_frecuencia-20;
   else variacion_frecuencia<=variacion_frecuencia;
end
```

Variación_frecuencia aumentara o disminuirá su valor en 20 cada vez que el pulsador BTN_NORTH o BT_EAST se opriman de lo contrario mantendrá su valor actual

```
wire clk_cont_lfsr;

variador_frecuencia reloj_cont_lfsr (
    .clk(CLK_50M),
    .reset(1'b0),
    .frecuencia_resultante(clk_cont_lfsr),
    .factor_division(variacion_frecuencia)
    );
```

Salida de la instancia

Se instancia el modulo variador_frecuencia. El nombre reloj_cont_lfsr la diferenciara en caso de que se necesite instanciarlo otra vez

Variación_frecuencia se convierte en entrada

```
wire [7:0] lfsr;

LFSR_7bits generador (
    .lfsr(lfsr),
    .clk(clk_cont_lfsr),
    .reset(SW[0])
    );
wire [7:0] conteo;

contador conteo_up_down (
    .clk(clk_cont_lfsr),
    .reset(SW[0]),
    .conteo(conteo),
    .up_down(SW[2])
    );
wire clk_05h;

variador_frecuencia reloj_rot (
    .clk(CLK_50M),
    .reset(1'b0),
    .frecuencia_resultante(clk_05h),
    .factor_division(32'd25000000)
    );
```

Todas las entradas y salidas del modulo deben estar en la instancia

Switch SW[0] controlara el reset del modulo

El modulo variador_frecuencia se vuelve a instanciar, pero con el identificador reloj_rot y una salida diferente

```
wire [3:0] rot_led_a, rot_led_b;

rotacion_leds rebota (
    .clk(clk_05h),
    .reset(1'b0),
    .rot_led_a(rot_led_a),
    .rot_led_b(rot_led_b)
    );
 always @ ( posedge CLK_50M)
 begin
  if (BTN_WEST)
  begin
    LED [3:0] <= rot_led_a;
    LED [7:4] <= rot_led_b;
  end

    else
       begin
       if(SW[1]) LED [7:0] <= lfsr [7:0];
       else if (~SW[1]) LED [7:0] <= conteo [7:0];
       end
 end
```

Instrucciones para determinar la secuencia visualizada en los Leds

Si BTN_WEST está activo la secuencia de rotación será vista

Si BTN_WEST no se encuentra activo la secuencia la decidirá el switch SW[1]

```
wire rot_left, rot_right, rot_press;

control_rot_knob giro_eje (
    .rot_a(ROT_A),
    .rot_b(ROT_B),
    .rot_center(ROT_CENTER),
    .clk(CLK_50M),
    .rot_left(rot_left),
    .rot_right(rot_right),
    .rot_press(rot_press)
    );
reg [31:0]factor_inicial = 32'd71633;


always @ (posedge CLK_50M)
begin
   if (rot_right) factor_inicial <= factor_inicial - 1000;
   else if (rot_left) factor_inicial <= factor_inicial + 1000;
   else if (rot_press) factor_inicial <= 32'd71633;

end
```

Se instancia el modulo de control del Rotary Knob, sus entradas deben ser los pulsadores que contiene la perilla

Se realiza la nueva operación que con variación_frecuencias, en este caso una vuelta del Rotary Knob aumenta o disminuye a factor_inicial y si se oprime se obtendrá un valor definido

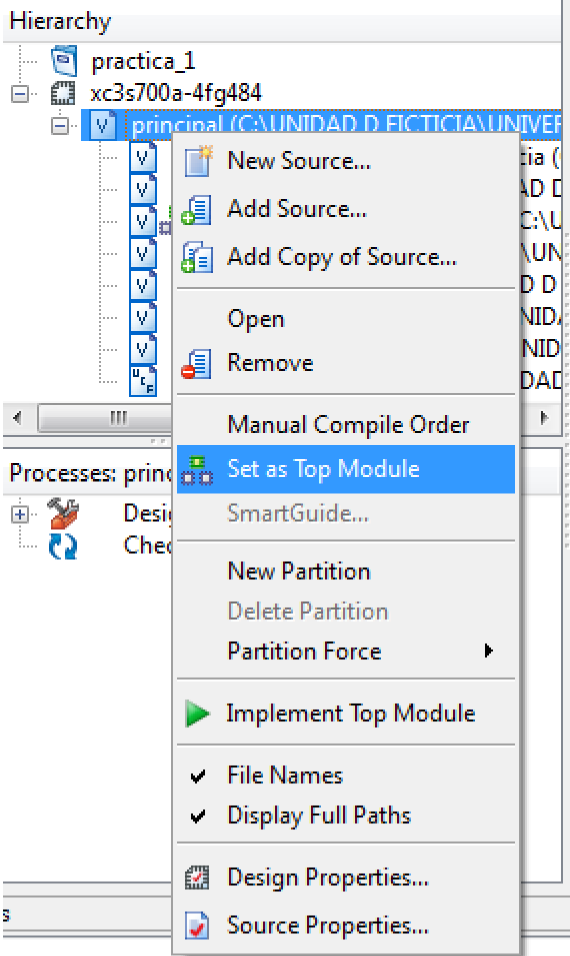
```
wire audio_sig;

variador_frecuencia audio (
    .clk(CLK_50M),
    .reset(1'b0),
    .frecuencia_resultante(audio_sig),
    .factor_division(factor_inicial)
    );

always @ (posedge CLK_50M)
begin
AUD_L <= audio_sig;
AUD_R <= audio_sig;
end

endmodule
```

Se instancia de nuevo el modulo variador_frecuencia, su salida tendrá una frecuencia variable debido a factor_inicial

Las salidas AUD_L y AUD_R conectan directamente al conector de audio en donde podrán escucharse las variaciones de frecuencia

Para asegurar que ISE Project Navigator reconozca a principal como el módulo estructural, se debe ir a la ventana de fuentes y hacer clic derecho sobre el nombre de este módulo, seguidamente elegir la opción **set as top module** como se muestra en la siguiente figura.

Figura 102 Asignación del módulo estructural

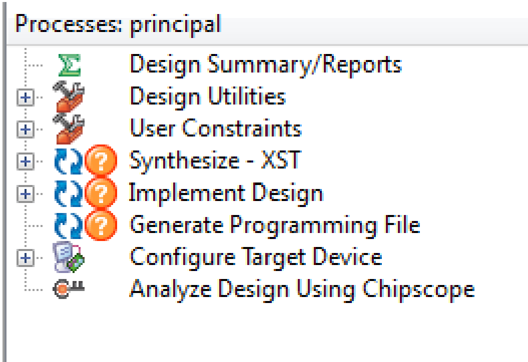


Si la opción se encuentra desactivada es porque el módulo ya está asignado como estructural.

Una vez se tenga el módulo estructural indicado, en la ventana de procesos se debe realizar la síntesis, implementación y generación del archivo de programa para permitir que el código escrito sea traducido a términos de hardware y este pueda programarse en la FPGA, para realizar dichas acciones se debe hacer doble clic en cada uno de los procesos mientras el módulo estructural este seleccionado.

Los procesos y su ubicación se encuentran en la siguiente figura.

Figura 103 Procesos de síntesis, implementación y generación del archivo de programa

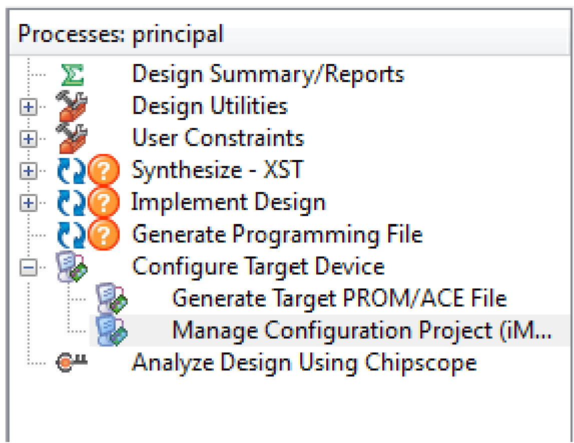


La realización de cada uno de estos procesos puede demorar cierto tiempo dependiendo de la complejidad del diseño.

Una vez estos procesos este terminados y sin ningún error la programación de la FPGA se lleva a cabo siguiendo los siguientes pasos

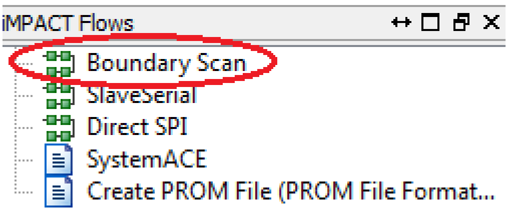
1. Abrir iMPACT desde el proyecto con el proceso **Manage Configuration Project** indicado en la figura siguiente.(esto también puede realizarse abriendo el programa externamente)

Figura 104 Ejecución del iMPACT



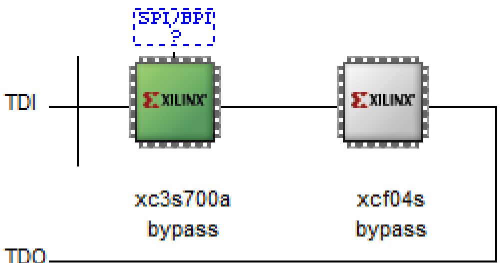
2. Cuando la ventana del iMPACT este abierta, se debe seleccionar la opción **Boundary Scan** como se indica en la figura

Figura 105 Selección del tipo de programación



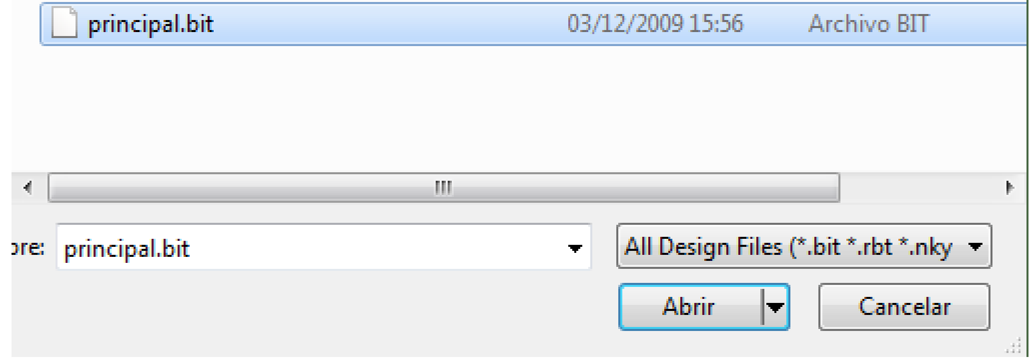
3. Teniendo la FPGA conectada al computador se sigue el camino **file >> Initialize chain** para establecer una conexión entre el computador y la FPGA**.** Deberá aparecer en la ventana lo mostrado en la figura siguiente.

Figura 106 Conexión establecida entre computador y FPGA.



4. Se debe hacer clic derecho en el bloque identificado como xc3s700a y seleccionar la opción **assign new configuración file** y en el directorio en donde se encuentra guardado el proyecto escoger el archivo .bit como se muestra en la siguiente figura.

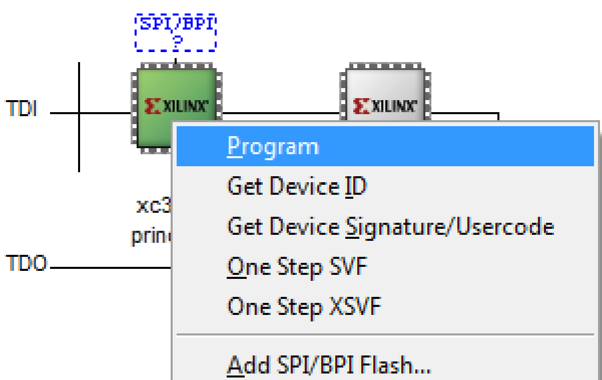
Figura 107 Selección del archivo .bit



5. Al bloque del lado derecho no se le debe asignar ningún tipo de archivo y por el contrario debe permanecer en bypass.

6. Una vez cargado el archivo de configuración se vuelve a hacer clic derecho en el bloque en donde se encuentra el proyecto y se selecciona la opción **progamm** como se ilustra en la siguiente figura.

Figura 108 Programación en la FPGA



7. Si después de aparece otra ventana se le debe dar clic en OK
8. La programación se realiza y el programa informa que se realizo con éxito.

Después de haber completado los pasos anteriores el programa creado debe comenzarse a ejecutar en la tarjeta de desarrollo.

14.5.8 Ejercicios propuestos

- Realizar los diagramas esquemáticos de los módulos restantes.
- Utilizando el mismo método de la práctica, implementar un piano que genere las frecuencias de las notas Do, Re, Mi, Fa, Sol, La, Si, Do de acuerdo al estado de los Switches, estas notas deben ser audibles por medio del conector de audio.
- Genere una nueva secuencia de rotación de LEDs cuya velocidad aumente al girar el Rotary Knob a la derecha, disminuya al girar a la izquierda y regrese a la velocidad inicial al presionarlo.
- Implemente un circuito que visualice el estado del pulsador BTN_NORTH y BTN_SOUTH en los LEDs awake e Init.

15 PRÁCTICA 2: MÁQUINAS DE ESTADO

15.1 OBJETIVOS

- Conocer el funcionamiento y la estructura básica de las máquinas de estado en Verilog
- Realizar una aplicación utilizando máquinas de estado

15.2 RECURSOS A UTILIZAR

- Tarjeta de desarrollo Spartan3A
- LEDs
- Switches
- Pulsadores
- Computador con software ISE Project Navigator

15.3 PREREQUISITOS

Para el desarrollo de esta práctica se debe contar con conocimiento básicos sobre las máquinas de estado, conceptos de digitales y conceptos básicos de Verilog

15.4 EXPLICACIÓN DE LA PRÁCTICA

La práctica constará de dos módulos que deberán realizar la función de un conteo sencillo de peatones mediante el uso de sensores y un sistema de anti rebote para las entradas mecánicas de la tarjeta respectivamente, esto mediante la utilización de máquinas de estado finitas (FSM por sus siglas en ingles).

Los módulos que hacen parte de esta práctica son los siguientes.

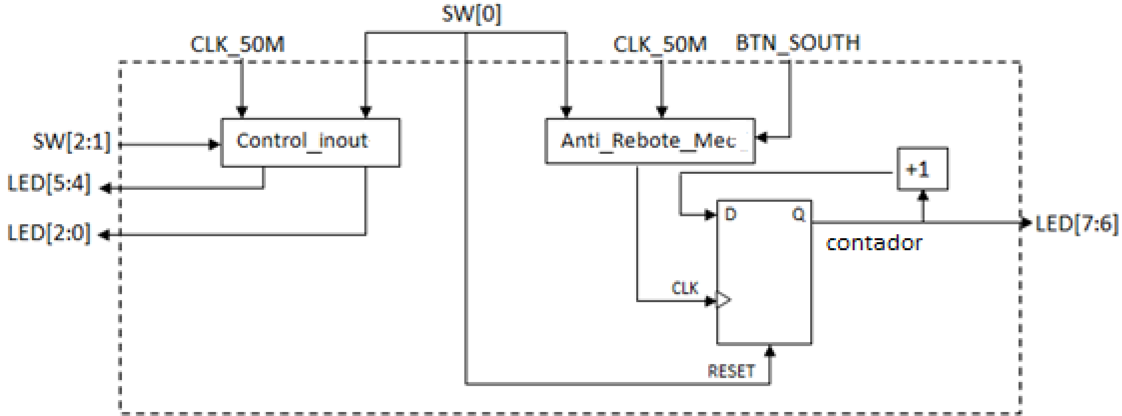
- control_inout: este modulo tendrá la función de evaluar el estado de la salida de los sensores y determinar si una persona entra o sale por el área vigilada.
- Anti_Rebote_Mec: la función de este módulo es la de evaluar la señal de salida proporcionada por un pulsador e imitarlos omitiendo los glitches propios de los elementos mecánicos.

Una vez terminados los módulos que hacen parte de la práctica, el modulo estructural deberá conectarlos con cada una de las señales externas de la siguiente manera.

- SW[2:1] simularan las salidas de los sensores
- SW[0] actuara como la entrada reset del módulo de conteo de peatones y el sistema anti rebote
- El pulsador BTN_SOUTH actuara como la entrada del módulo anti rebote los LEDs [2:0] deberán llevar el conteo de las personas que entran, los LEDs [5:4] de quienes salen y los LEDs [7.6] un conteo que tendrá como reloj la señal de salida del módulo anti rebote

El diagrama de bloques que describe el funcionamiento de esta práctica se muestra en la siguiente figura.

Figura 109 Diagrama de bloques para la práctica 2.



En el diagrama se aprecian los dos módulos que conformaran esta práctica, control_inout y Anti_Rebote_Mec, cada uno con sus respectivas señales realizando su función correspondiente sin la necesidad de interactuar entre sí.
El modulo estructural, en este caso el área dentro del rectángulo de líneas discontinuas, solo deberá servir como interfaz con la FPGA y realizar una pequeña función de conteo.

15.5 DESARROLLO DE LA PRÁCTICA

Para esta práctica se debe tener en cuenta que se utilizaran menos elementos de la tarjeta, por lo cual se debe modificar el archivo UCF solo para los Switches, el reloj y los LEDs.

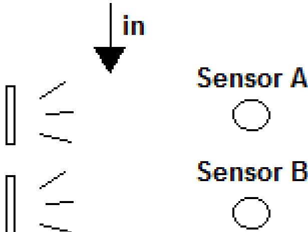
15.5.1 Módulo control_inout

Este módulo realiza una función de conteo sencilla de personas que entran y salen por un área específica mediante el uso de sensores.

Por medio de dos sensores el sistema debe diferenciar si una persona está entrando o saliendo y debe indicar esto mediante un contador.

En la siguiente figura se puede apreciar la distribución de los sensores

Figura 110 Sensores del módulo control_inout



El sensor envía una señal de cero cuando no hay nada entre el emisor y el receptor, y una señal de uno cuando se está obstruyendo el camino. Por consiguiente la secuencia de sensores que se obtendría para una persona entrando seria la descrita en la tabla de la siguiente figura.

Figura 111 Entrada de una persona.

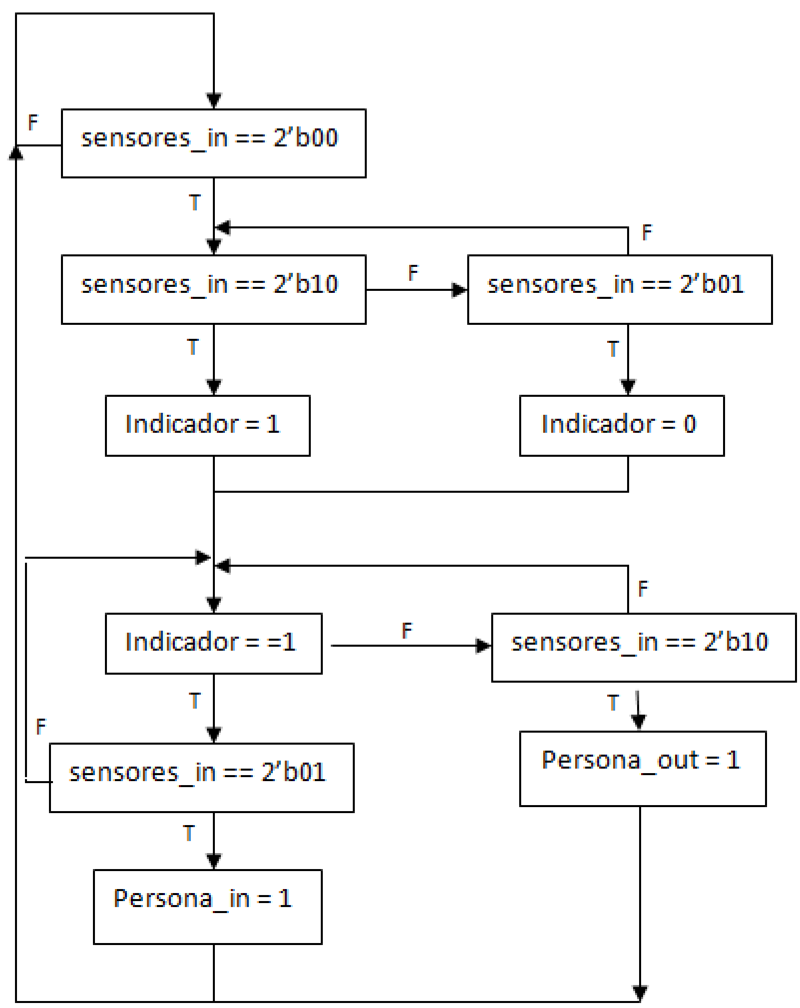
Sensor A	Sensor B
0	0
1	0
0	0
0	1

De igual manera se puede deducir la secuencia para la salida de personas.

Debido a que se trabajaran con máquinas de estado, es recomendable realizar un diagrama de estados que por cada una para asegurar una total comprensión de su función y facilitar por consiguiente su escritura en lenguaje Verilog.

En la siguiente figura se podrá observar el diagrama de estados correspondiente al módulo actual.

Figura 112 Diagrama de estados del módulo control_inout



Con el diagrama de estados completado se pueden observar claramente las acciones que se realizaran en cada estado, asumiendo que cualquier variable no mencionada deba mantener su estado anterior.

La transcripción a código Verilog del diagrama anterior se presenta a continuación.

```
module ctrl_in_out
(clk, reset, sensores_in, entrada, salida);

input clk, reset;
input [1:0] sensores_in;
output reg [3:0] entrada, salida;

reg  person_in, person_out;
```

Bandera para activar suma en salidas

```
reg [1:0]estado=0, estado_siguiente=0;
reg indicador;
```

Variables para la transición entre estados

Bandera para indicar dirección del peatón

```
always @(posedge clk)
begin
if (reset) begin
entrada<= 0;
salida<=0;
end
else  begin
if(person_in)
entrada <= entrada+1;
if(person_out)
salida<= salida+1;
end
end
```

Estructura Flipflop

La bandera person_in activa la suma para la salida entrada.
La bandera person_out activa la suma para salida

```
always @ (posedge clk)
if (reset) estado<=0;
else
estado<= estado_siguiente;
```

Actualiza el valor de estado con estado_siguiente

```
always @ *
begin
estado_siguiente = estado;

person_in =0;
person_out = 0;

case (estado)

2'b00: begin

    if (sensores_in == 2'b0)
      estado_siguiente = 2'b01;
    else estado_siguiente = 2'b00;
    end
```

Always corre con el cambio de cualquier señal

Valores por defecto de estas variables

Valor de estado determina que bloque del case se ejecuta

Primer bloque. Si los dos sensores están en valor cero se puede continuar al siguiente bloque, de lo contrario se mantiene el valor de la variable estado

```
2'b01: begin

   if (sensores_in == 2'b10) begin
   estado_siguiente = 2'b10;
   indicador = 1;
   end
   else if (sensores_in == 2'b01) begin
   estado_siguiente = 2'b10;
   indicador = 0;
   end
   else estado_siguiente = estado_siguiente;
   end
```

Se continúa buscando la secuencia de los sensores, el primer sensor en ser bloqueado indica entrada o salida, esto se almacena en indicador.

Si alguna condición se cumple avanza al siguiente estado, de lo contrario permanece en el bloque

```
2'b10: begin

   if (sensores_in == 2'b00)
      estado_siguiente = 2'b11;
   else
   estado_siguiente = 2'b10;
   end
```

Cuando se compruebe que la secuencia sigue como debería se puede avanzar al siguiente estado

```
2'b11: begin

   if (indicador) begin
    if (sensores_in== 2'b01)begin
      person_in =1;
      estado_siguiente = 2'b00;
      end
      else estado_siguiente= estado_siguiente;
   end
   else if (~indicador) begin
    if (sensores_in == 2'b10) begin
      person_out =1;
      estado_siguiente= 2'b00;
      end
      else estado_siguiente = estado_siguiente;
   end

   end
endcase
end

   endmodule
```

Se cumple el fin de la secuencia, la variable indicador enseña cual variable de salida se modifica

Este módulo representa un contador sencillo debido a que no tiene en cuenta modificaciones a la secuencia de los sensores, sin embargo cumple su propósito de ejemplificar el uso de las máquinas de estado en Verilog

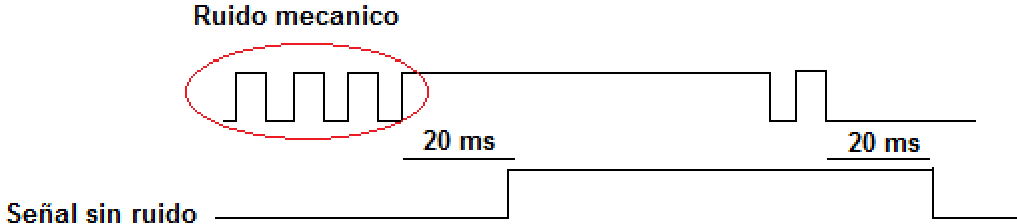
15.5.2 Módulo Anti_Rebote_Mec

El propósito de este módulo es el de brindar una señal libre de los rebotes propios de los interruptores mecánicos, para esto una máquina de estados

debe verificar que la señal de entrada permanezca estable por un tiempo especificado después de cada flanco, este tiempo suele ser por lo general de 20ms.

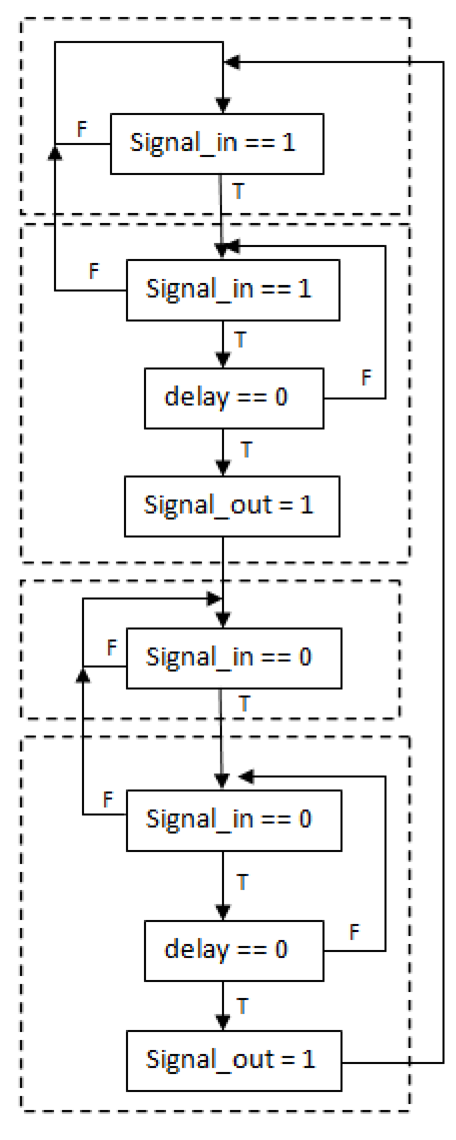
En la siguiente figura, se observa la señal de entrada que recibiría el módulo y seguidamente la señal de salida sin ruido y con un retraso de 20ms que se obtendría a su salida.

Figura 113 Entrada y salida del modulo Anti_Rebote_Mec



La máquina de estados que verificara la señal de entrada y los tiempos mínimos para desechar glitches tendrá el diagrama de transición de estados presentado en la siguiente figura, en donde lo rectángulos de líneas discontinuas se refieren a cada uno de los estados de la máquina.

Figura 114 Diagrama de estados para el módulo Anti_Rebote_Mec



Ya con el diagrama de transición de estados terminado se conoce que preguntas deben realizarse y las acciones que tendrá cada variable en cada estado.

El código utilizado para transcribir el diagrama de estados a Verilog se muestra a continuación.

(Tener en cuenta el tiempo de referencia de 20ms, es decir 1^{6} ciclos del reloj de 50 MHz es el mínimo tiempo a utilizar)

```
module Anti_Rebote_Mec

clk, signal_in, signal_out,reset);

 input clk,  signal_in, reset;
 output reg signal_out;

 reg [31:0] delay;
 reg [31:0] delay_next;
 reg [1:0] estado_siguiente =0;
 reg reset_delay, en_delay;
 reg [1:0] estado;


  localparam [1:0]
  uno = 2'b00,
  espera_1 = 2'b01,
  cero = 2'b10,
  espera_2 = 2'b11;


  localparam [31:0] valor_inicial = {20{1'b1}};
```

El identificador localparam se utiliza para declarar constantes dentro de un modulo

Declaración de varias constantes con el mismo número de bits

Declaración de la constante valor_inicial de 20 bits en alto. Los demás bits se asumen cero

```
 always @ (posedge clk)
 begin

 if (reset) begin
    delay <= valor_inicial;
    estado<= 0;
    end
 else begin
    delay <= delay_next;
    estado<= estado_siguiente;
    end
 end
```

Bloque always que permite la actualización cada ciclo de reloj, de las variables delay y estado

```
always @ (posedge clk)
begin
if (reset_delay)
delay_next <= valor_inicial;
else begin
if (en_delay)
   delay_next <= delay_next-1'b1;
   else
      delay_next <= delay_next;
   end
end
```

Al estar activado reset_delay se mantiene en el valor inicial la variable delay_next, de lo contrario se mantiene en el valor actual a menos que se active en_delay que permite la resta de una unidad cada ciclo

```
always @ *
begin
estado_siguiente <= estado;
en_delay <= 1'b0;
reset_delay<= 1'b0;
```

Inicio máquina de estados

Valores por defecto

```
case (estado)
uno: begin
      signal_out<=1'b0;
         if (signal_in)
            begin
               estado_siguiente<=espera_1;
               reset_delay<= 1'b1;
            end
         else estado_siguiente<=uno;
      end
```

Primer estado. Si la señal de entrada esta en nivel alto, se sigue al estado de espera, de lo contrario se debe permanecer en este estado.
La salida es cero y el contador de delay esta desactivado

```
espera_1: begin
            signal_out<= 1'b0;
               if (signal_in) begin
                  en_delay<=1'b1;
                  if (delay==0)
                     estado_siguiente<= cero;
                  end
               else estado_siguiente<=uno;
            end
```

Si la entrada continua en 1 se activa el contador de delay

Si la entrada conserva el nivel alto, se permanecerá en el estado hasta que delay sea cero, si por el contrario la entrada pasa a cero antes de terminar el conteo se regresa al estado inicial.

```
cero: begin
         signal_out<= 1'b1;
            if(~signal_in) begin
               estado_siguiente<=espera_2;
               reset_delay<=1'b1;
            end
            else estado_siguiente<=cero;
      end
```

Señal de salida continúa en nivel alto en este estado

Si la señal de entrada llega a nivel bajo se procederá al siguiente estado.
De lo contrario se continúa esperando el cambio de nivel

```
espera_2: begin
            signal_out<=1'b1;
               if(~signal_in) begin
                  en_delay<=1'b1;
                   if (delay==0)
                     estado_siguiente <= uno;
               end
               else estado_siguiente<=cero;
            end
endcase
end
 endmodule
```

Mientras la entrada continúe en cero, se activara el conteo de delay.
Cuando delay llegue a cero se retornara al estado inicial donde la señal de salida tomara el valor bajo

Si la entrada toma nivel bajo en este estado se asumirá como ruido y regresara al estado cero

15.5.3 Módulo principal

Este modulo actuara como módulo estructural del proyecto.

```
module principal

(SW, LED, CLK_50M, BTN_SOUTH);

output [7:0] LED;

input [3:0] SW;
input BTN_SOUTH;
input CLK_50M;
```

Recursos que se utilizaran en la tarjeta

```
wire clk_manual;
```

Declaración de la variable de salida del modulo

```
Anti_Rebote_Mec reloj (
    .clk(CLK_50M),
    .signal_in(BTN_SOUTH),
    .signal_out(clk_manual),
    .reset(SW[0])
    );
```

Instancia del modulo anti rebote Anti_Rebote_Mec.

```
ctrl_in_out control_inout (
    .clk(CLK_50M),
    .reset(SW[0]),
    .sensores_in({SW[2],SW[1]}),
    .entrada(LED[2:0]),
    .salida(LED[5:3])
    );
```

Instancia del modulo ctrl_inout

Los Switches SW[2:1], se concatenan para formar el vector de entrada.
La salida se visualiza en los Leds [5:0]

```
reg [1:0] contador;

 always @ (posedge clk_manual)
 if (SW[0]) contador <= 0;
 else contador <= contador +1;

 assign LED [7:6] = contador;

endmodule
```

Declaración variable interna contador

Con cada flanco de subido de la señal de salida del modulo Anti_Rebote_Mec se aumentara contador y se mostrara en los Leds [7:6]

15.5.4 Ejercicios propuestos:

- Realice los diagramas esquemáticos de cada uno de los módulos utilizados en esta práctica y del diagrama de estados.
- Implemente un control de entrada y salida de peatones similar al mostrado en la práctica pero que en este caso sea capaz de soportar variaciones en la secuencia de los sensores, por ejemplo que una persona decida regresar al interior a mitad de camino hacia la salida.
- Diseñe una máquina de estados que posee dos salidas X y Y, y dos entradas A y B. La salida X debe adquirir valor lógico alto solo si la entrada A ha permanecido en el mismo valor durante tres pulsaciones y la salida Y debe adquirir el valor lógico ato si B es cero cuando se cumpla la condición de valor alto en X.

16 PRÁCTICA 3: PUERTO PS2/ PICOBLAZE/ LCD

16.1 OBJETIVOS

- Conocer el funcionamiento del puerto PS2 de la tarjeta Spartan3A para el manejo del teclado
- Conocer el manejo de la pantalla LCD integrada en la tarjeta
- Entender el uso del microprocesador embebido que posee la FPGA Spartan3A aplicado al control de la pantalla LCD de la tarjeta

16.2 RECURSOS A UTILIZAR

- Tarjeta de desarrollo Spartan3A
- Teclado
- LEDs
- Puerto PS2 (teclado)
- Pantalla LCD
- Picoblaze
- Computador con software ISE Project Navigator

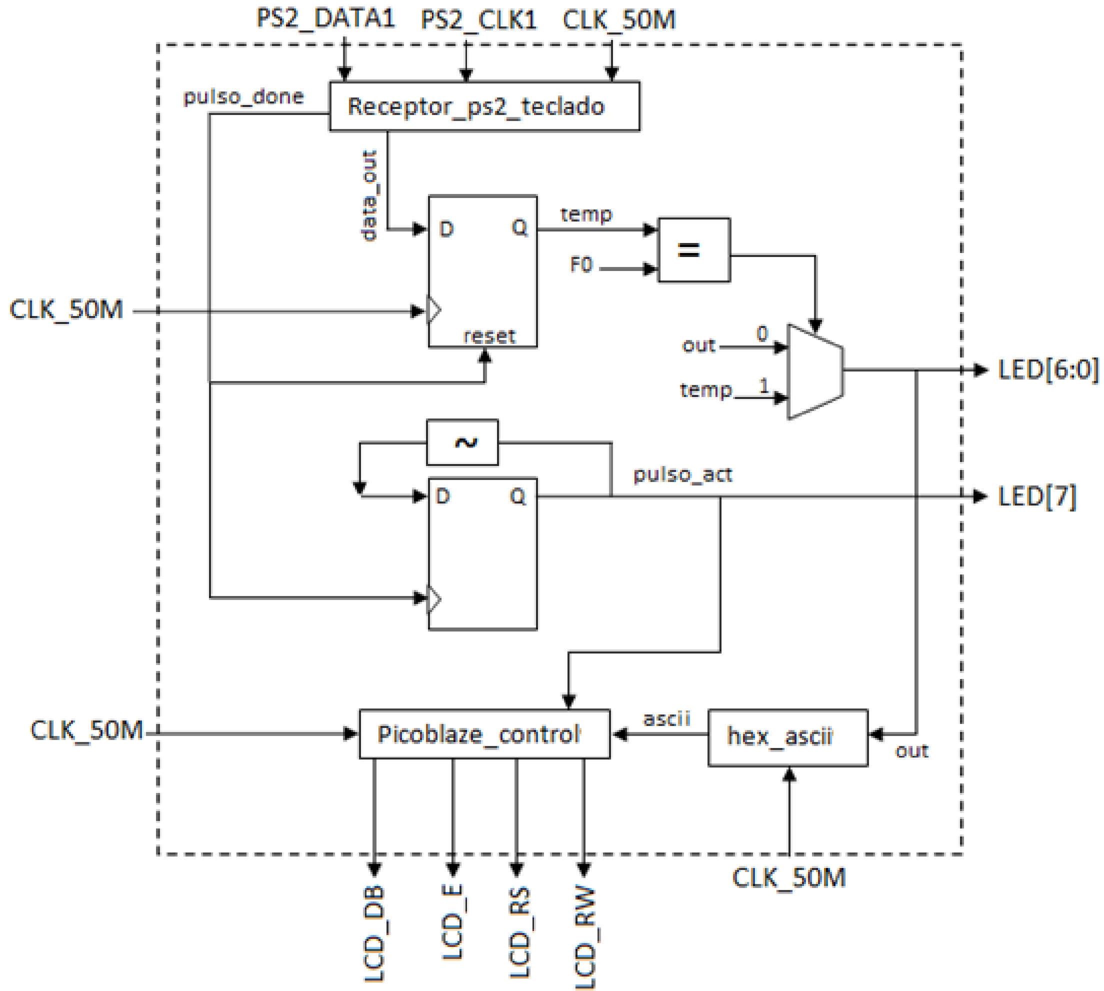
16.3 PREREQUISITOS

Para el desarrollo de esta práctica se deben contar con conocimientos de programación en Asembler, comunicación serial y conceptos de digitales.

16.4 EXPLICACIÓN DE LA PRÁCTICA

La práctica constara de tres módulos, además del principal como se observa en el diagrama de bloques de la siguiente figura en donde el módulo estructural está representado por el rectángulo de líneas discontinuas.

Figura 115 Diagrama de bloques para la práctica 3.



Cada módulo recibe sus respectivas señales de entrada y se encargara de controlar el receptor del puerto PS2, la conversión a código ASCII del valor enviado por el teclado a través del puerto y de crear una interfaz para el código Picoblaze según se enumeran a continuación.

- El módulo receptor_ps2_teclado mediante una máquina de estados, deberá recibir los datos que envía el teclado y almacenarlos en un registro
- El módulo hex_ascii tomara el valor recibido desde el teclado y lo convertíra a un valor ASCII necesario para su visualización en el display LCD.
- El módulo Picoblaze_control, deberá ser la interfaz que permita comunicar el código Picoblaze con las señales de entrada y salidas necesarias.

Después de tener cada una de las funciones por separado, el modulo estructural se encargara de recibir el valor desde el puerto PS2, de crear los caminos de comunicaciones entre los diferentes módulos y de efectuar

sencillas operaciones de selección para asignar los valores de salida de las variables correspondientes.

16.5 DESARROLLO DE LA PRÁCTICA

16.5.1 Módulo receptor_ps2_teclado

Antes de explicar el funcionamiento del módulo, es necesario entender el funcionamiento del conector ps2.

El conector ps2, es una interfaz bastante utilizada para la comunicación de teclados y mouse con un controlador principal, un puerto sencillo cuenta con 4 líneas, correspondientes a: línea de datos, línea de reloj, línea de energía, línea de tierra. En la tarjeta de desarrollo Spartan3A, el puerto cuenta con la opción de manejar otra entrada de datos y de reloj mediante el uso de un cable Y para manejar al tiempo dos periféricos.

En este caso se implementara el receptor para el teclado, sin embargo este puede adaptarse fácilmente para el manejo de un mouse ya que su funcionamiento es similar debido a que ambos utilizan dos señales para comunicarse, los mismos tiempos de funcionamiento y paquetes de datos de 11 bits.

El teclado cuenta con un microcontrolador embebido que controla la actividad de las teclas y envía los datos acordes a la actividad registrada:

- Si ninguna tecla está siendo activada las señales de datos y reloj permanecerán estables en nivel alto
- Cuando una tecla es oprimida la señal de reloj comenzara a generar transiciones, en cada flanco de bajada de la señal reloj se emitirá un bit de dato hasta completar 11 bits: un bit de inicio, 8 bits de datos, un bit de paridad y un bit de parada.
- Al final de la transmisión las señales regresaran a su estado de reposo.

Los 8 bits de datos generados por el teclado corresponden al valor hexadecimal de cada tecla (el cual es único). Se debe tener en cuenta que el proceso de envío de datos se realizara un total de tres veces ya que al oprimir una sola vez alguna tecla, se enviara el valor correspondiente a la tecla seguido del “break code” (F0) y nuevamente el valor de la tecla, es decir el receptor del controlador principal recibiría los siguiente datos en caso de que la tecla B sea oprimida una ves

32 F0 32

Si una tecla se mantiene presionada, el microcontrolador del teclado enviara cada 100ms el valor correspondiente a la tecla y al ser esta liberada se generara el break code.

32 32 … 32 F0 32

En las siguientes figuras se pueden observar los valores correspondientes a cada tecla en un teclado estándar y el diagrama de tiempos del teclado respectivamente.

Figura 116 Códigos del teclado

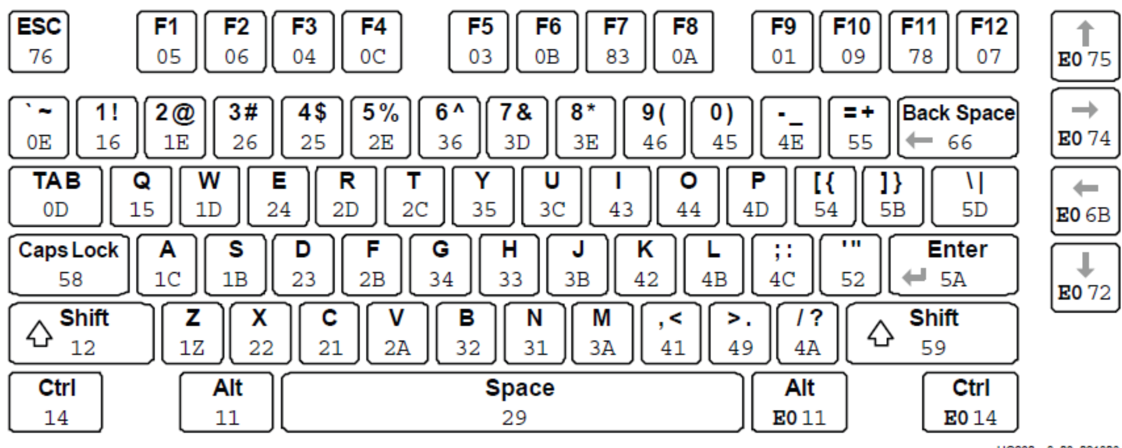
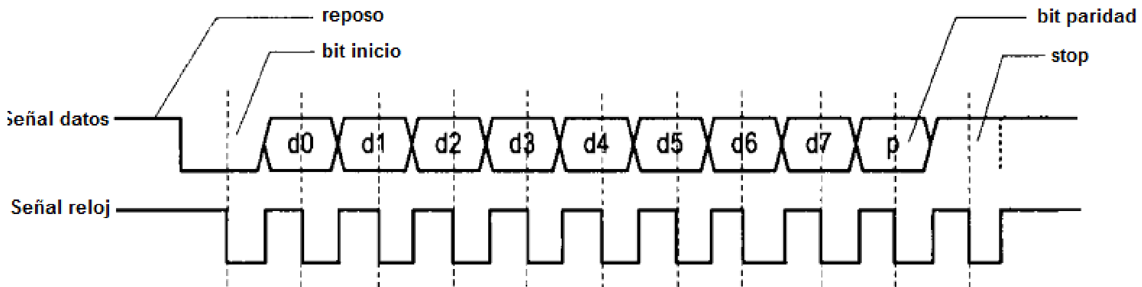


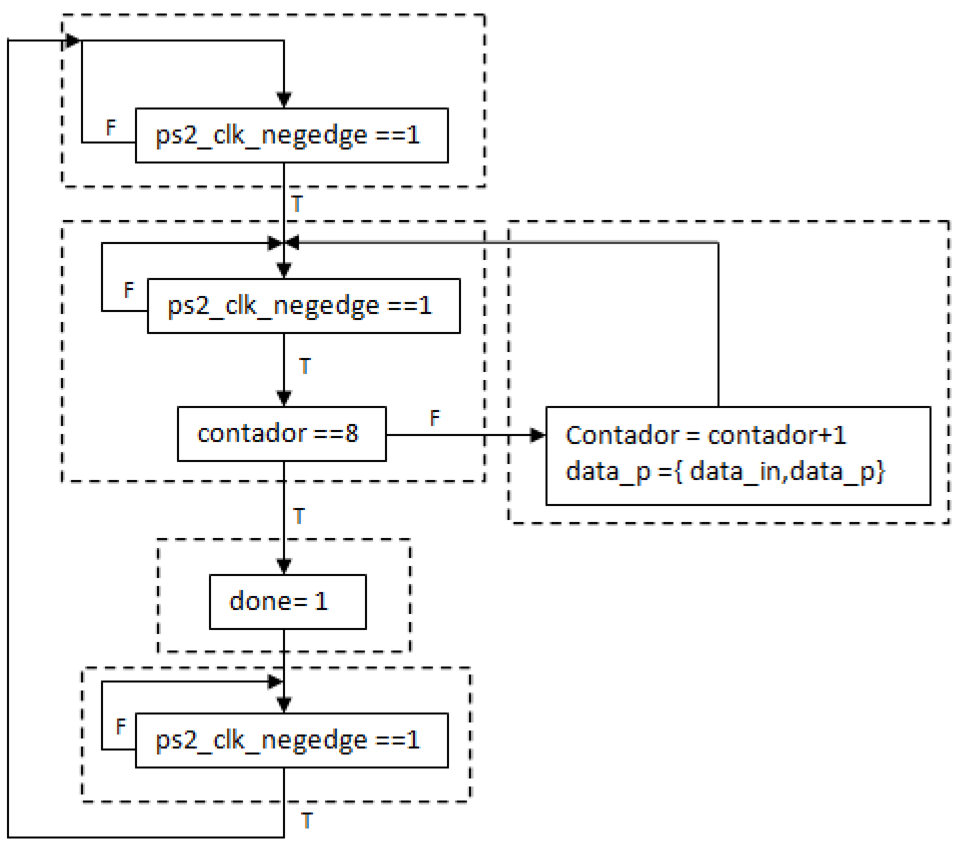
Figura 117 Protocolo de comunicación del teclado



Ya que se conoce el funcionamiento básico del teclado y su protocolo de comunicación la creación de módulo receptor será más sencilla.

Para recibir los datos en serie que envía el teclado, se implementara una máquina de estados que presentara el flujo expuesto en la siguiente grafica.

Figura 118 Diagrama de transición de estados para el módulo ps2_receptor_teclado



El control realizado en la figura anterior permite la lectura de los 8 bits requeridos, basándose en los flancos de bajada del reloj de teclado.

El módulo que se utilizo para implementar el receptor es el siguiente:

```
module receptor_ps2_teclado
(clk, ps2_data, ps2_clk, ps2_data_out, pulso_done);
```

```
input clk;
input ps2_clk, ps2_data;
output [7:0]ps2_data_out;
output reg pulso_done;
```

Señales de entrada desde el teclado

Concatenación de los 8 bits de datos recibidos

Fin del paquete de datos

```
//variables del programa
reg [7:0] filtro_antirebote;
reg ps2_clk_negedge;
reg [2:0] estado=0;
reg [7:0] contador =0;
reg [7:0] data_p;
reg data_in;

localparam [2:0]
   idle = 3'b000,
   uno = 3'b001,
   dos = 3'b010,
   tres = 3'b011,
   cuatro = 3'b100;
```

Estados

El modulo se encargara de detectar los flancos de bajada de la señal de reloj y almacenar el dato recibido

```
always @ (posedge clk)
begin
filtro_antirebote [7:0]<= {ps2_clk,filtro_antirebote[7:1]};
end
```

El valor de la señal filtro_antirebote va variando cada flanco de reloj según el valor de ps2_clk, si el valor de ps2_clk esta constante en 1 el valor de filtro_antirebote cambiaria su valor en el tiempo de la siguiente manera: 00000000, 10000000, 11000000, 11100000, 11110000,...., 11111111.

```
always @ (posedge clk)
begin
if (filtro_antirebote == 8'b00001111)
   ps2_clk_negedge <= 1'b1;
else
   ps2_clk_negedge <= 1'b0;
end
```

El valor 00001111 indica una transición de valor alto a bajo. La función de esta operación es evitar posibles glitches por medio de un tiempo de espera.
Esto es posible debido a que clk (50M) es de mayor frecuencia que ps2_clk

Ps2_clk en alto cada flanco negativo

```
always @ (posedge clk)
begin
if (ps2_clk_negedge)
data_in<= ps2_data;
else
data_in <= data_in;
end
```

El registro data_in almacena el valor de entrada ps2_data para evitar su cambio antes de que este pueda ser almacenado en el registro de salida ya que esta acción no se realiza en el mismo ciclo de reloj.

```
always @ (posedge clk)
begin
case (estado)
idle: begin

      data_p <= data_p;
      contador <= 4'd0;
      pulso_done <= 1'b0;

      if(ps2_clk_negedge)

            estado<= uno;

      else

         estado <= idle;

      end
```

Valores de cada variable en el estado

Cuando se presenta un flanco de bajada en él se puede seguir al siguiente estado, de lo contrario se debe seguir esperando.
Aquí se recibe el bit de inicio pero no se toma en cuenta.

```
uno: begin
         data_p <= data_p;
         contador <= contador;
         pulso_done <= 1'b0;

          if (ps2_clk_negedge)
         begin
            if (contador == 4'd8)

               estado<= tres;

         else
                  estado<= dos;

         end

            else
               estado<= uno;

      end
```

En cada flanco negativo se preguntara por el valor de contador si es igual a 8 todos los bits han sido transmitidos y se pasara al estado tres de lo contrario se irá al estado dos

Si no hay flanco negativo permanece en el estado uno

```
dos: begin
      data_p [7:0] <= {data_in, data_p [7:1]};
      contador <= contador +1;
      pulso_done <= 1'b0;
      estado <= uno;
      end
```

El registro data_p se irá llenando con el valor de data_in, el contador aumentara su valor con cada bit recibido y se regresara al estado uno

```
tres: begin
      estado <= cuatro;
      pulso_done <= 1'b1;
      data_p <= data_p;
      contador <= 4'd0;
      end

cuatro: begin
      data_p <= data_p;
      contador <= 4'd0;
      pulso_done <= 1'b0;

      if (ps2_clk_negedge)
         estado <= idle;

      end

endcase
end

assign ps2_data_out = data_p;

endmodule
```

Al terminar de recibir los 8 bits de datos, el valor de pulso_done será 1 y se reinicia el contador

Pulso_done tendrá valor de cero, creando un pulso de un ciclo de reloj de duración

Se regresa al estado inicial ignorando el bit de paridad

Se asigna el valor del registro data_p al puerto de salida

Una vez terminado el funcionamiento del módulo se puede contar con el código hexadecimal de las teclas para diferentes usos.

16.5.2 Módulo hex_ascii

Debido a que el dato enviado por el teclado será utilizado para escribir en la pantalla LCD y a que el valor hexadecimal de las teclas no corresponde a su valor ASCII, se utilizara el módulo hex-ascii para convertir cada valor recibido del teclado en el código ASCII correspondiente a cada tecla.

El tamaño del módulo es bastante extenso, aunque su funcionamiento es sencillo, en el utiliza un multiplexor que dependiendo del valor de entrada dará el valor de salida.

```
module hex_ascii
(key_code, ascii_code, clk);

    input wire [7:0] key_code;
    output reg [7:0] ascii_code;
    input clk;


always @(posedge clk)
   case(key_code)
        8'h45: ascii_code <= 8'h30;    // 0
        8'h16: ascii_code <= 8'h31;    // 1
        8'h1e: ascii_code <= 8'h32;    // 2
        8'h26: ascii_code <= 8'h33;    // 3
        8'h25: ascii_code <= 8'h34;    // 4
        8'h2e: ascii_code <= 8'h35;    // 5
        8'h36: ascii_code <= 8'h36;    // 6
        8'h3d: ascii_code <= 8'h37;    // 7
        8'h3e: ascii_code <= 8'h38;    // 8
        8'h46: ascii_code <= 8'h39;    // 9

        8'h1c: ascii_code <= 8'h41;    // A
        8'h32: ascii_code <= 8'h42;    // B
        8'h21: ascii_code <= 8'h43;    // C
        8'h23: ascii_code <= 8'h44;    // D
        8'h24: ascii_code <= 8'h45;    // E
        8'h2b: ascii_code <= 8'h46;    // F
        8'h34: ascii_code <= 8'h47;    // G
        8'h33: ascii_code <= 8'h48;    // H
        8'h43: ascii_code <= 8'h49;    // I
        8'h3b: ascii_code <= 8'h4a;    // J
        8'h42: ascii_code <= 8'h4b;    // K
        8'h4b: ascii_code <= 8'h4c;    // L
        8'h3a: ascii_code <= 8'h4d;    // M
        8'h31: ascii_code <= 8'h4e;    // N
        8'h44: ascii_code <= 8'h4f;    // O
        8'h4d: ascii_code <= 8'h50;    // P
        8'h15: ascii_code <= 8'h51;    // Q
        8'h2d: ascii_code <= 8'h52;    // R
        8'h1b: ascii_code <= 8'h53;    // S
        8'h2c: ascii_code <= 8'h54;    // T
        8'h3c: ascii_code <= 8'h55;    // U
        8'h2a: ascii_code <= 8'h56;    // V
        8'h1d: ascii_code <= 8'h57;    // W
        8'h22: ascii_code <= 8'h58;    // X
        8'h35: ascii_code <= 8'h59;    // Y
        8'h1a: ascii_code <= 8'h5a;    // Z
```

La estructura case funciona como un multiplexor, donde la entrada determina la salida

```
        8'h0e: ascii_code <= 8'h60;   // `
        8'h4e: ascii_code <= 8'h2d;   // -
        8'h55: ascii_code <= 8'h3d;   // <=
        8'h54: ascii_code <= 8'h5b;   // [
        8'h5b: ascii_code <= 8'h5d;   // ]
        8'h5d: ascii_code <= 8'h5c;   // \
        8'h4c: ascii_code <= 8'h3b;   // ;
        8'h52: ascii_code <= 8'h27;   // '
        8'h41: ascii_code <= 8'h2c;   // ,
        8'h49: ascii_code <= 8'h2e;   // .
        8'h4a: ascii_code <= 8'h2f;   // /

        8'h29: ascii_code <= 8'h20;   // (space)
        8'h5a: ascii_code <= 8'h0d;   // (enter, cr)
        8'h66: ascii_code <= 8'h08;   // (backspace)
        default: ascii_code <= 8'h2a; // *
    endcase

endmodule
```

Una vez se tenga el código ASCII correspondiente, este se puede utilizar en la pantalla LCD para obtener el carácter deseado.

16.5.3 Módulo Picoblaze_control

El módulo picoblaze_control es el encargado de generar una interfaz para permitir la comunicación entre el la memoria ROM de instrucciones y el procesador KCPSM3.

Como primer paso se debe crear el programa de control de Picoblaze, este puede ser escrito en cualquier procesador de texto básico como block de notas, y debe ser guardado con la extensión .psm.
El programa en Picoblaze debe realizar las siguientes funciones:

- Inicializar el display LCD de la tarjeta.
- Recibir la salida del módulo hex_ascii y presentarlas en el display

Para realizar la inicialización del display LCD, se deben seguir los siguientes pasos los cuales especificaran una interfaz de comunicaciones de 8 bits, seguidamente se debe configurar el display. Los pasos son:

Para configurar la interfaz de comunicación

1. Esperar 15ms o más.
2. Enviar el comando 0x38
3. Esperar 4.1ms o más.
4. Enviar el comando 0x38
5. Esperar 100us o más.
6. Enviar el comando 0x38
7. Esperar 40us o más.

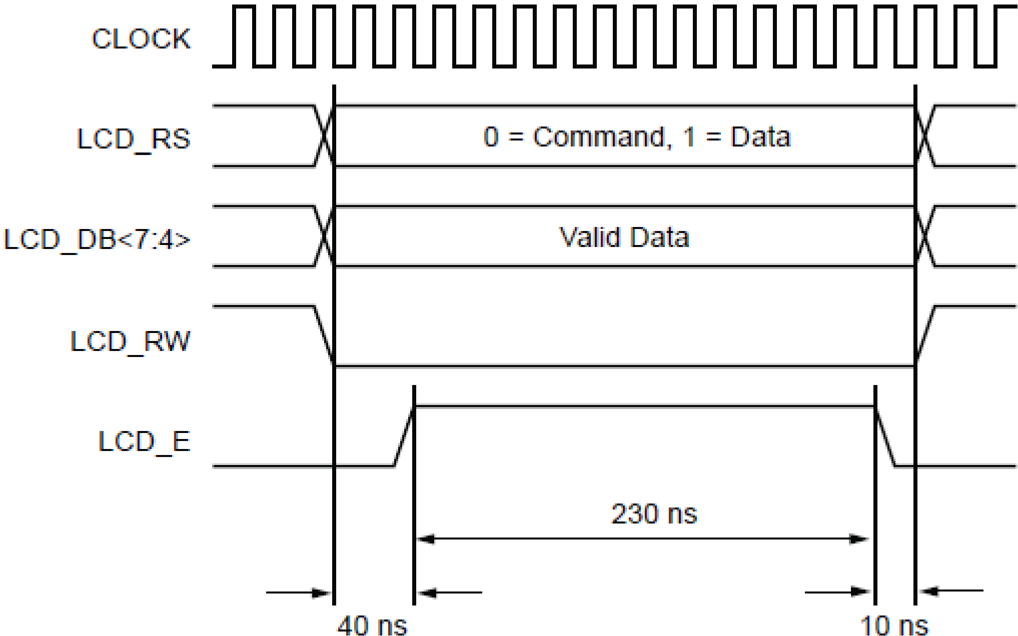
Después de establecida la comunicación mediante interfaz de 8 bits se prosigue a configurar el display

1. Enviar el comando 0x38
2. Esperar 40us o más.
3. Enviar el comando 0x06

4. Esperar 40us o más.
5. Enviar el comando 0x0C
6. Esperar 40us o más.
7. Enviar comando 0x01
8. Esperar 1.64ms o más.

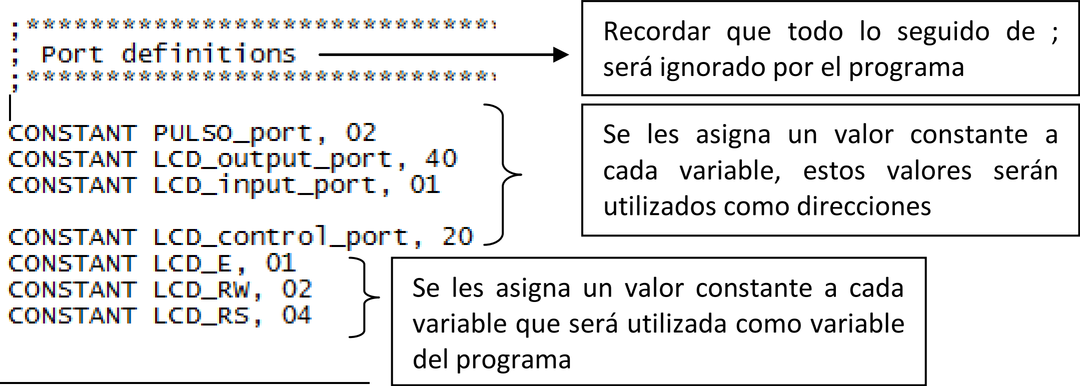
Se debe recordar que el LCD posee unos tiempos específicos para recibir comandos y datos, como se puede ver en las figura a continuación.

Figura 119 Tiempos de funcionamiento en la pantalla LCD[51]



Como se muestra en la figura 3, las señales LCD_DB, LCD_RW y LCD_RS deben estar en valor estable por lo menos 40us antes del cambio de la señal LCD_E, este enable debe estar en estado alto por lo menos 230ns antes de cambiar su estado para que el controlador del LCD pueda procesar la información y esperar al menos 40us antes de enviar algún otro dato.

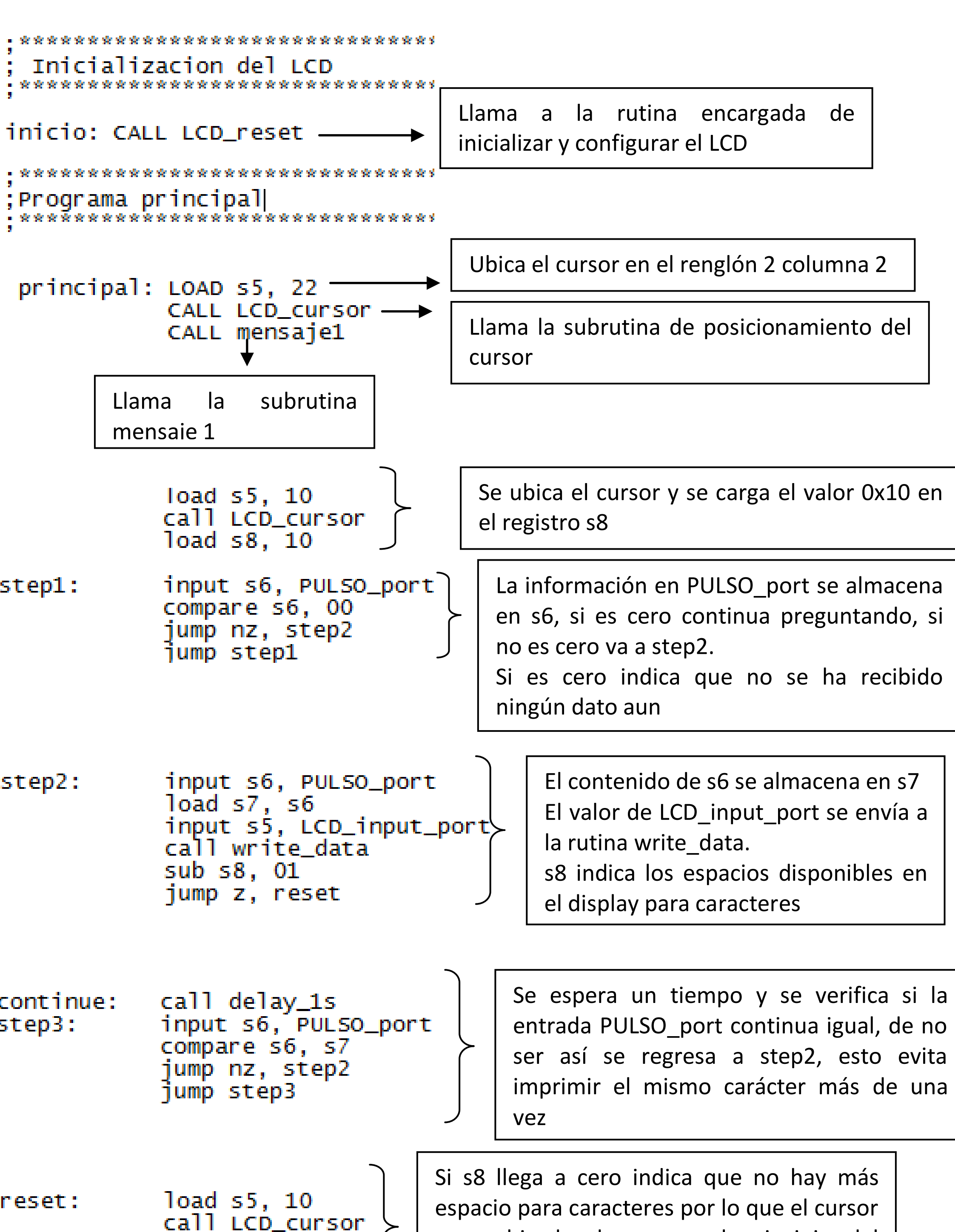
Una vez entendido el manejo necesario del LCD, se puede continuar al programa Picoblaze explicado a continuación.

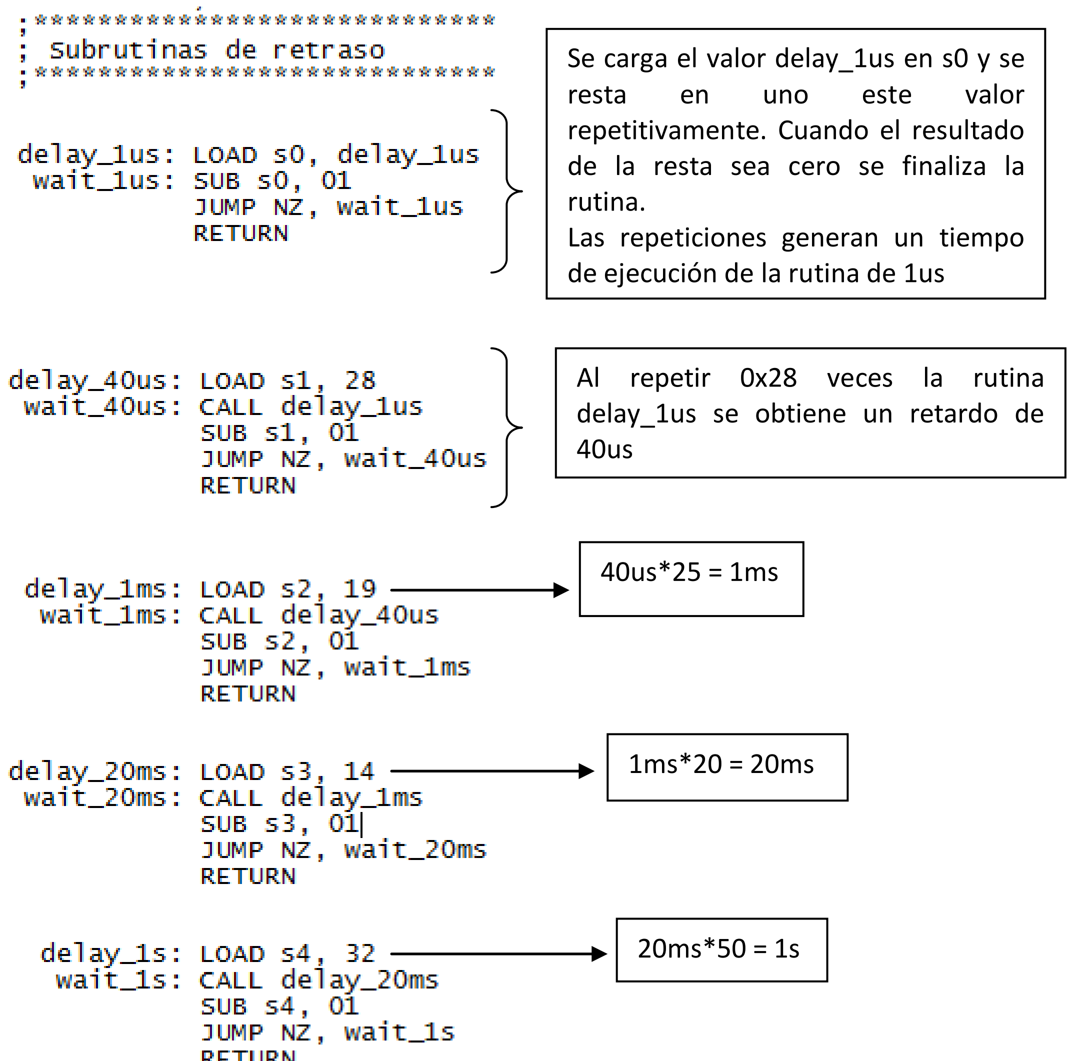


[51] XILINX, Spartan 3a fpga starter kit board user guide UG334 (v 1.1) June 19 de 2008

```
CONSTANT delay_1us, 0B
```

Esta variable será utilizada para crear un delay, ya que se está utilizando un reloj de 50MHz (20ns) y el número de instrucciones del delay es de 6 (como se verá más adelante), es fácil calcular el número necesario de repeticiones necesarias




;*******************************
; Subrutinas de retraso
;*******************************
delay_1us: LOAD s0, delay_1us
wait_1us: SUB s0, 01
JUMP NZ, wait_1us
RETURN
Se carga el valor delay_1us en s0 y se resta en uno este valor repetitivamente. Cuando el resultado de la resta sea cero se finaliza la rutina.
Las repeticiones generan un tiempo de ejecución de la rutina de 1us
delay_40us: LOAD s1, 28
wait_40us: CALL delay_1us
SUB s1, 01
JUMP NZ, wait_40us
RETURN
Al repetir 0x28 veces la rutina delay_1us se obtiene un retardo de 40us
delay_1ms: LOAD s2, 19
wait_1ms: CALL delay_40us
SUB s2, 01
JUMP NZ, wait_1ms
RETURN
40us*25 = 1ms
delay_20ms: LOAD s3, 14
wait_20ms: CALL delay_1ms
SUB s3, 01
JUMP NZ, wait_20ms
RETURN
1ms*20 = 20ms
delay_1s: LOAD s4, 32
wait_1s: CALL delay_20ms
SUB s4, 01
JUMP NZ, wait_1s
RETURN
20ms*50 = 1s

```
;***************************************
; Subrutinas LCD
;***************************************

mensaje1: LOAD s5, 50 ;character_p
          CALL write_data
          LOAD s5, 72 ;character_r
          CALL write_data
          LOAD s5, 61 ;character_a
          CALL write_data
          LOAD s5, 63 ;character_c
          CALL write_data
          LOAD s5, 74 ;character_t
          CALL write_data
          LOAD s5, 69 ;character_i
          CALL write_data
          LOAD s5, 63 ;character_c
          CALL write_data
          LOAD s5, 61 ;character_a
          CALL write_data
          LOAD s5, 20 ;character_space
          CALL write_data
          LOAD s5, 33 ;character_3
          CALL write_data

          RETURN
```

La subrutina carga en s5 los caracteres necesarios para formar la frase "practica 3" en la posición actual del cursor, estos datos se envían a write_data

Envía comandos al LCD

```
write_com: OUTPUT s5, LCD_output_port
           LOAD s4, 00
           OUTPUT s4, LCD_control_port
           CALL LCD_pulso_E
           CALL delay_40us
           RETURN
```

Se envían los datos en s5 al puerto de salida del LCD
Las señales RW, E, RS se envían con valor cero (si RS=0 el dato es un comando)
Se llama el LCD_pulso_E
Delay de 40us para el siguiente dato

Envía datos al LCD

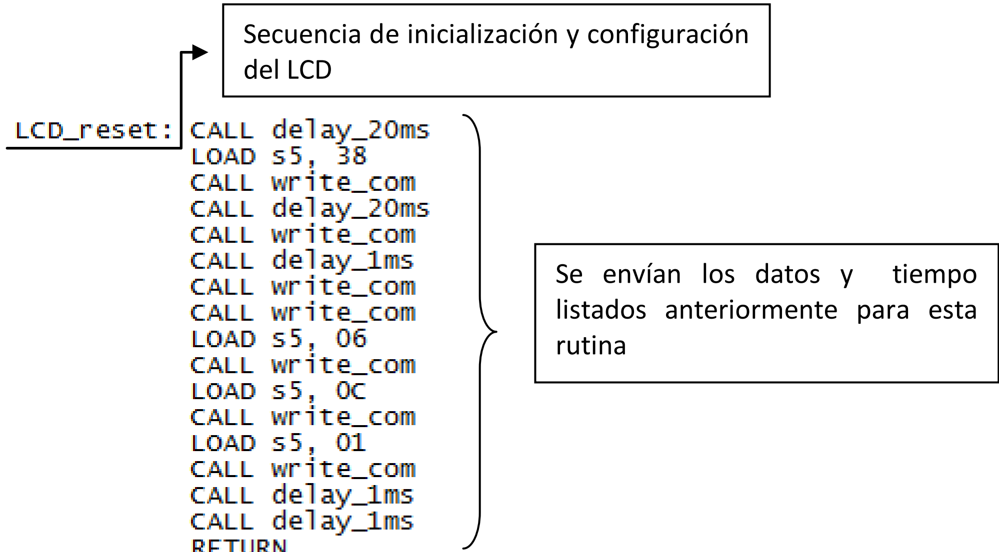
```
write_data: OUTPUT s5, LCD_output_port
            LOAD s4, 04
            OUTPUT s4, LCD_control_port
            CALL LCD_pulso_E
            CALL delay_40us
            RETURN
```

Los datos en s5 se envían al puerto lcd_output_port
Se envía RW Y E en cero, RS en uno (si RS=1 el valor es un dato)
Se llama LCD_pulso_E
Se esperan 40us para el siguiente dato

Genera el pulso del Enable

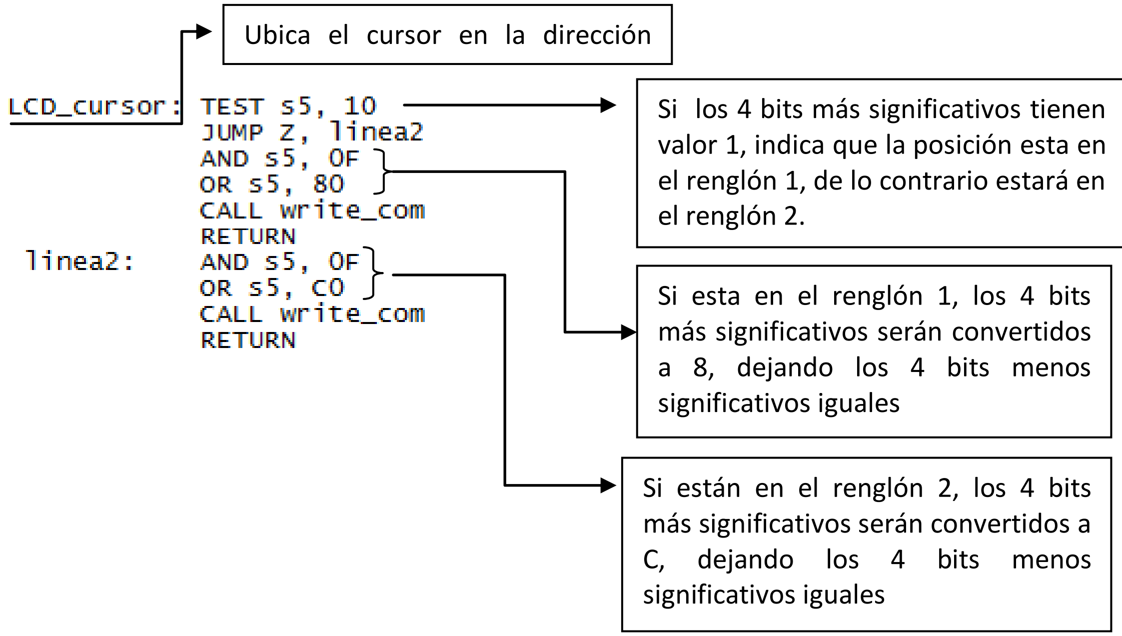
```
LCD_pulso_E: XOR s4, LCD_E
             OUTPUT s4, LCD_control_port
             CALL delay_1us
             XOR s4, LCD_E
             OUTPUT s4, LCD_control_port
             RETURN
```

S4^LCD_E se asegura de que el valor del enable (bit 0) sea uno.
Se envía a la salida de control
Se espera más de 230ns
S4^LCD_E se asegura de que el valor del enable (bit 0) sea cero.



Las direcciones de cada espacio en el display LCD son las siguientes

línea	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Renglón 1	80	81	82	83	84	85	86	87	88	89	8A	8B	8C	8D	8E	8F
Renglón 2	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	CA	CB	CC	CD	CE	CF

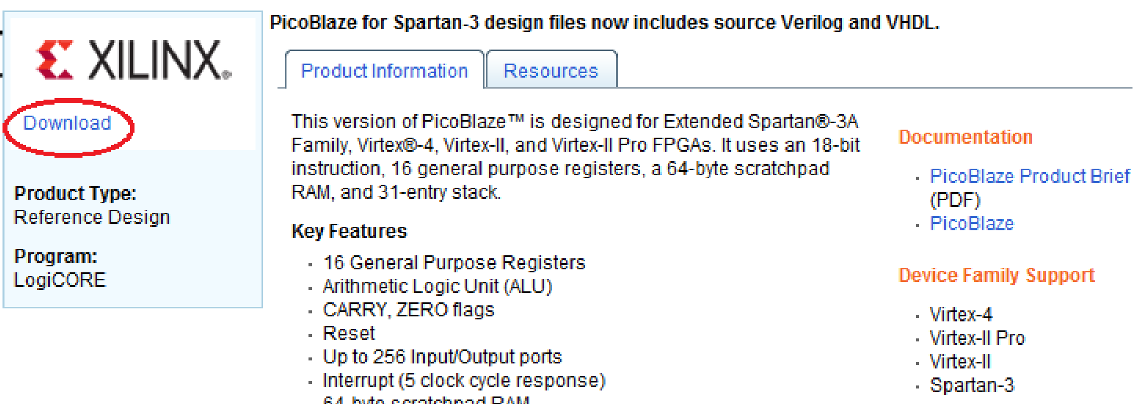


Una vez el código este terminado, se debe recordar guardarlo con extensión .psm.

Para verificar que el código esta sin errores se deben realizar los siguientes pasos.

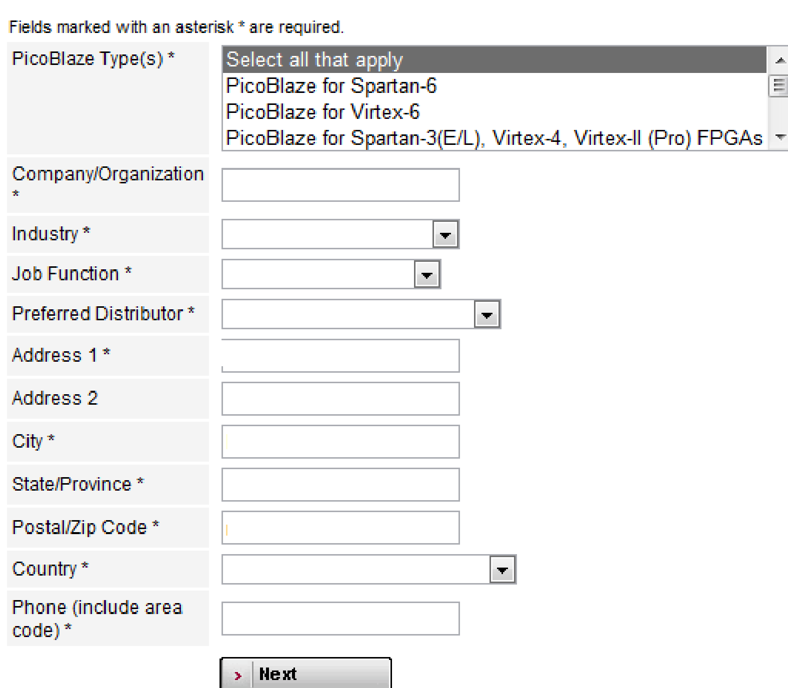
1. Primero debe descargarse el archivo KCPSM3 desde la pagina web http://www.xilinx.com/products/ipcenter/picoblaze-S3-V2-Pro.htm como indica la siguiente figura.

Figura 120 Descarga del archivo KCPSM3



2. En caso de no estar registrado se debe crear una cuenta y responder al formulario mostrado en la siguiente figura.

Figura 121 Formulario de descarga



3. Una vez se hayan aceptado las condiciones de uso aparecerá una pantalla como la mostrada en la siguiente figura, donde se debe hacer clic en el enlace Picoblaze Lounge

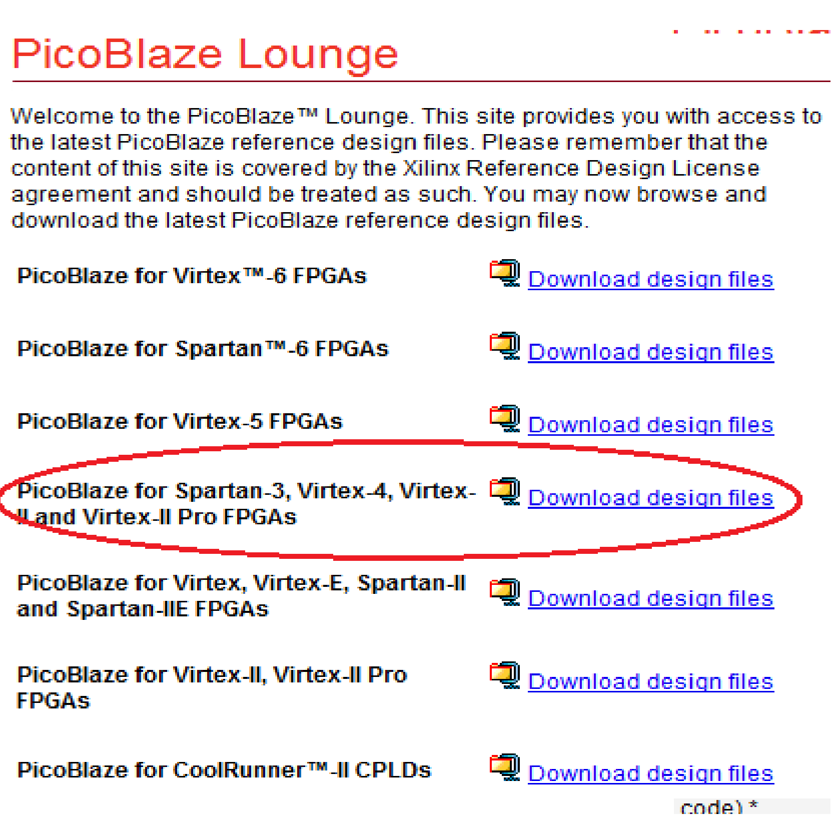
Figura 122 Fin del registro para descarga

PicoBlaze Lounge Registration Complete

Thank you for completing the Xilinx PicoBlaze Lounge registration process.
This PicoBlaze Lounge provides you with access to the latest PicoBlaze reference design files.
To browse and download the latest PicoBlaze reference design files, visit the PicoBlaze Lounge.

4. En la pantalla mostrada en la siguiente figura se debe seleccionar el tipo de tarjeta utilizada, en este caso la tarjeta es Spartan3A. es recomendable guardar el archivo KCPSM3 en el directorio c\:

Figura 123 Pantalla de descarga del archivo KCPSM3



6. Una vez se tenga la carpeta kcpsm3 se debe copiar el archivo psm creado a la carpeta Assembler, la cual se encuentra dentro de kcpsm3.

Un archivo Verilog debe ser creado a partir del código Picoblaze para poderlo incorporar al proyecto de trabajo, para realizar esta acción se debe continuar con los pasos siguientes.

7. Se ejecuta el programa “símbolo del sistema” y se ingresa al directorio donde está almacenado la carpeta kcpsm3. En caso de que haya sido guardada en c:\ se vería como lo muestra la siguiente figura.

Figura 124 Direccionamiento para ingresar a kcpsm3

```
Símbolo del sistema
Microsoft Windows [Versión 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. Reservados todos los derechos.

C:\Users\pc>cd..

C:\Users>cd..

C:\>cd kcpsm3

C:\KCPSM3>
```

8. Debe ingresarse a la carpeta assembler, donde se guardo el archivo .psm y escribir las instrucciones dadas en la figura a continuación, en este caso particular el archivo se llama lcd_control.psm

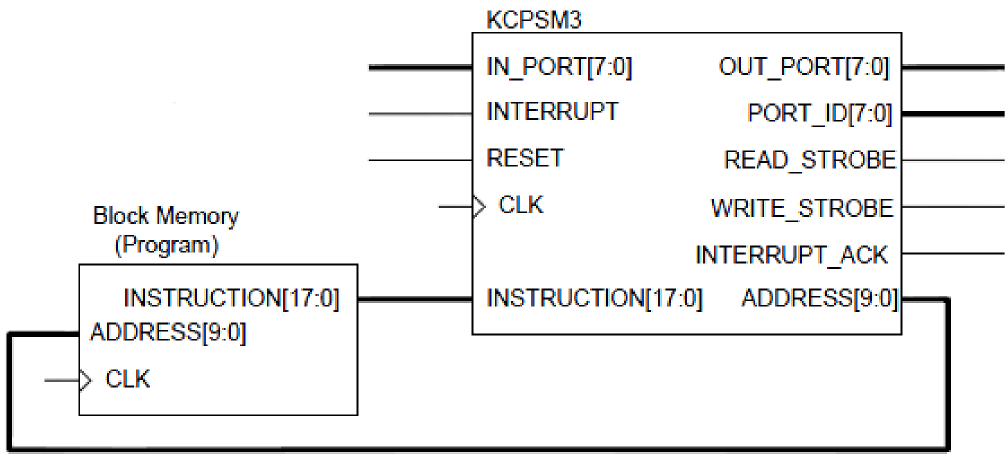
Figura 125 Instrucciones de ejecución para kcpsm3

```
C:\KCPSM3>cd assembler

C:\KCPSM3\Assembler>kcpsm3 lcd_control.psm
```

9. Se ejecutara así kcpsm3.exe lo que generara un archivo con extensión .V, que llevara el mismo nombre que el archivo .psm, LCD_CONTROL.V debe copiarse en el directorio donde se encuentra el proyecto Verilog que se está trabajando.
10. El archivo kcpsm3.V, que se encuentra en la carpeta Verilog de kcpsm3 también debe ser copiado en el directorio del proyecto.

Picoblaze consta de dos módulos que permiten su funcionamiento, la memoria ROM de instrucciones y el procesador kcpsm3 como se puede observar en la figura.

Figura 126 Conjunto del diagrama de Picoblaze.



El archivo LCD_CONTROL corresponde a la memoria ROM de instrucciones y el archivo kcpsm3 corresponde al procesador. Estos dos archivos poseen las entradas y salidas necesarias para poder interactuar con bloques externos y deben agregarse al proyecto en el que se está trabajando

Contando con estos dos archivos, el módulo Verilog que sirve de interfaz con el módulo Picoblaze puede crearse de la siguiente manera.

```
module picoblaze_control(

            inout [7:0] lcd_d,
            output reg lcd_rs,
            output lcd_rw,
            output reg lcd_e,
            input clk,
            input [7:0] input_data,
            input done
              );
```

Señales de control del LCD (lcd_d, lcd_rs, lcd_rw, lcd_e)

input_data → Entrada del código del carácter a visualizar

done → Señal que indica el cambio de dato en la entrada input_data

```
reg[7:0]    lcd_output;
```

lcd_output → Variable del programa para el display LCD

```
wire[9:0]  address;
wire[17:0]  instruction;
wire[7:0]  port_id;
wire[7:0]  out_port;
reg[7:0]  in_port;
wire  write_strobe;
wire  read_strobe;
reg  interrupt;
wire  interrupt_ack;
wire  kcpsm3_reset;
```

Variables internas para el manejo del Picoblaze

```
kcpsm3 lcd_ctrl(.address(address),
               .instruction(instruction),
                   .port_id(port_id),
              .write_strobe(write_strobe),
                  .out_port(out_port),
               .read_strobe(read_strobe),
                   .in_port(in_port),
                 .interrupt(interrupt),
             .interrupt_ack(interrupt_ack),
                     .reset(kcpsm3_reset),
                       .clk(clk));
```

Se instancia el archivo kpsm3 que se había incluido al proyecto

```
lcd_control rom (
    .address(address),
    .instruction(instruction),
    .clk(clk)
    );
```

Se instancia el archivo lcd_control que se había agregado al proyecto

```
assign kcpsm3_reset = 0;
```

El reset no se utiliza

```
 always @ (posedge clk)
 begin
     case (port_id[7:0])
        8'h01:    in_port <= input_data;
        8'h02:    in_port <= done;
        default: in_port <= 8'bx;
    endcase

end
```

Selección de la entrada a recibir de acuerdo al valor de port_id

```
always @ (posedge clk)
begin

      if (write_strobe & port_id[6])
        lcd_output <= out_port;

      if (write_strobe & port_id[5])
   begin
           lcd_rs <= out_port[2];
           lcd_rw <= out_port[1];
           lcd_e <= out_port[0];
      end

end
```

Selección del puerto de salida mediante el valor de port_id y la disponibilidad del dato (dado por write_strobe)

```
assign lcd_d =  lcd_output;
```

Se asigna la señal de salida

```
endmodule
```

16.5.4 Módulo estructural picoblaze_practica

Debido a que el teclado envía el código F0 junto con el código de las teclas, se debe crear un circuito que omita este código de la entrada al conversor hex_ascii, dicho filtro se encuentra dentro de este módulo.

```
module picoblaze_practica

(LED, LCD_DB, LCD_RS, LCD_E,
LCD_RW, CLK_50M, PS2_CLK1, PS2_DATA1);
```

Puertos utilizados por la FPGA

```
inout [7:0] LCD_DB;
output LCD_RS;
output LCD_E;
output LCD_RW;
input CLK_50M;
output [7:0] LED;
input PS2_CLK1, PS2_DATA1;
```

Se declara como in/out porque el LCD contiene la opción de lectura de su memoria

```
reg [7:0] temp;
reg [7:0] out;
reg pulso_act=0;
```

Variables internas

```
wire pulso_done;
wire [7:0] data_out;
receptor_ps2_teclado TECLADO (
     .clk(CLK_50M),
     .ps2_data(PS2_DATA1),
     .ps2_clk(PS2_CLK1),
     .ps2_data_out(data_out),
     .pulso_done(pulso_done)
     );
```

Instancia del receptor del teclado

```
always @(posedge CLK_50M)
if (pulso_done)
   temp<= data_out;
else
   temp<= temp;
```

Cuando el teclado finalice el envío de información (pulso_done=1), se actualiza temp

```
always @(posedge CLK_50M)
begin
if(temp==8'hF0)
   out <= out;
else
   out<=temp;
end
```

Se evalúa el valor en temp, si es igual a F0 no se actualiza la señal out

```
always @ (posedge pulso_done)
begin
pulso_act <= ~pulso_act;
end
```

El cambio del valor lógico de pulso_act indica de una manera más estable el cambio en el valor de la variable out.

```
wire [7:0] ASCII;
hex_acsii ASCII_code (
     .key_code(out),
     .ascii_code(ASCII),
     .clk(CLK_50M)
     );
```

Se instancia el conversor hex_ascii.

```
picoblaze_control cod_pico(
                .lcd_d(LCD_DB),
                .lcd_rs(LCD_RS),
                .lcd_rw(LCD_RW),
                .lcd_e(LCD_E),
                .clk(CLK_50M),
                .input_data(ASCII),
                .done (pulso_act)
                  );
```

Se instancia la interfaz para el código Picoblaze, con los respectivos puertos

```
   assign LED  = out;
```

Se visualiza en los Leds el valor de las teclas activadas

```
endmodule
```

16.5.5 Ejercicios propuestos

- Realice los diagramas esquemáticos de cada uno de los módulos utilizados en esta práctica

- Realice el control del receptor del teclado mediante código PicoBlaze
- Utilizando código PicoBlaze inicialice el display LCD como se mostro en la práctica, pero a diferencia del proyecto de la práctica en este caso implemente el control sobre el display utilizando código Verilog.
- Mediante un código en Picoblaze implemente una calculadora sencilla que sume y reste dos valores de dos bits ingresados mediante los Switches y que visualice el resultado en los LEDs

17 PRÁCTICA 4: PUERTO VGA

17.1 OBJETIVOS

- Conocer el funcionamiento del conector VGA incluido en la tarjeta, para utilizarlo en el control de una pantalla compatible.
- Familiarizarse con el diseño de objetos
- Conocer las bases de la animación de objetos

17.2 RECURSOS A UTILIZAR

- Tarjeta de Desarrollo Spartan3A
- Pantalla con tecnología VGA
- Puerto VGA
- Computador con el software ISE Project Navigator

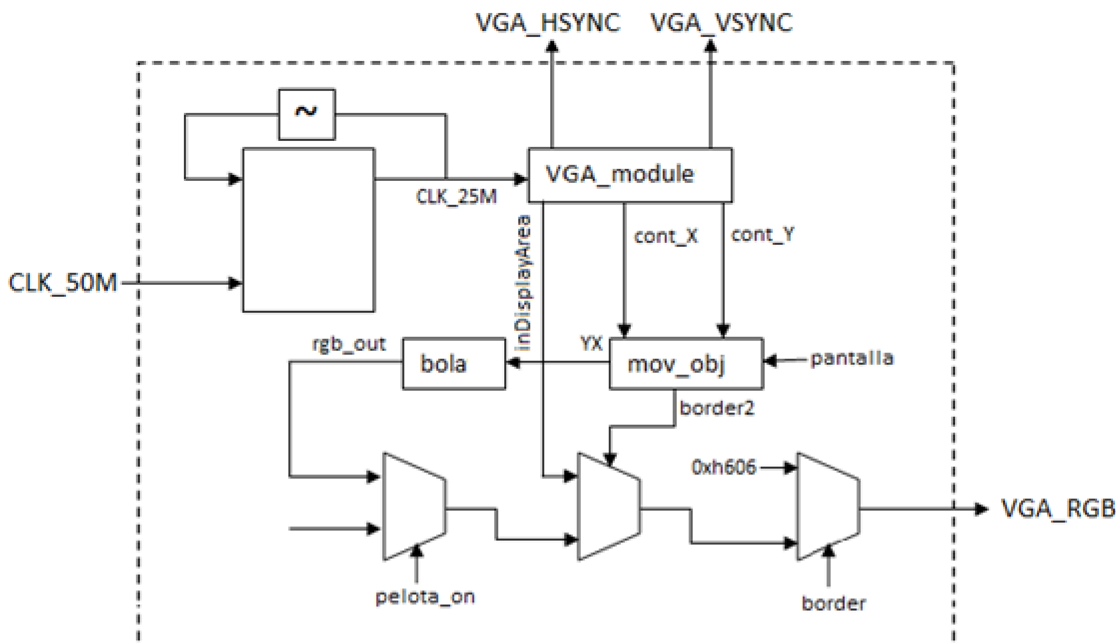
17.3 PREREQUISITOS

Para el desarrollo de esta práctica se deben contar con los conocimientos sobre el funcionamiento de una pantalla de tubos de rayos catódicos, la tecnología VGA y el sistema de colores RGB.

17.4 EXPLICACIÓN DE LA PRÁCTICA

Como se observa en el diagrama de bloques de la siguiente figura el presente módulo no contara con entradas diferentes a la señal de reloj y solo deberá controlar las variables de manejo del puerto VGA.

Figura 127 Diagrama de bloques para la practica 4.



Los módulos deberán realizar las siguientes funciones.

- El módulo VGA_module debe realizar un barrido en toda el área correspondiente a la pantalla VGA e indicar el área de visualización.
- El módulo mov_obj debe crear un objeto cuadrado y seguir una secuencia de movimiento dentro de un área determinada
- El módulo bola debe generar un objeto no cuadrado.

Como se observa en la figura anterior, el módulo estructural creara la comunicación entre los tres submódulos anteriores y como producto final se obtendrá lo siguiente.

- Se realizara un barrido en toda el área de la pantalla que imprimirá un área cuadrada de fondo morado, sobre la cual se encontraran dos objetos, un objeto cuadrado y un objeto creado a partir de un mapa de bits.
- Realizara un movimiento dentro del área de color morado de ambos objetos

17.5 DESARROLLO DE LA PRÁCTICA

Antes de poder realizar cualquier módulo para el control de una pantalla VGA, es necesario entender el funcionamiento de este dispositivo.

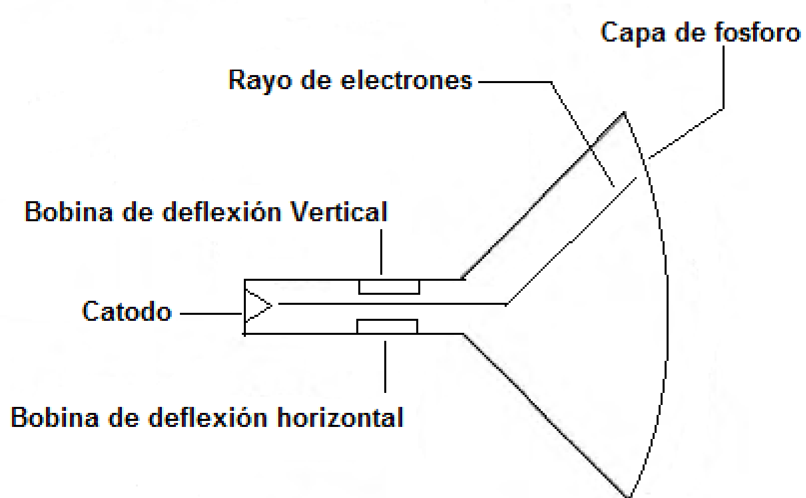
Las pantallas de tubo de rayos catódicos (dispositivo que se utilizara en la práctica), tienen un principio de funcionamiento sencillo. Inicialmente un cátodo genera un rayo de electrones que viajan a través del tubo de vacio hasta

impactar con una capa de fosforo, este impacto produce un punto luminoso en la pantalla denominado pixel. Con el fin de crear una imagen en la totalidad de la capa de fosforo el rayo de electrones debe recorrer la misma, para esto se cuenta con bobinas de deflexión cuyo campo magnético afecta la dirección del rayo brindando un control sobre el punto de impacto.

En una pantalla a color, el funcionamiento es idéntico, la única diferencia es que cuenta con tres fuentes de rayo de electrones que corresponden a los colores verde, azul y rojo, su combinación lograda a través del manejo de voltajes crea un pixel del color deseado en la capa de fosforo.

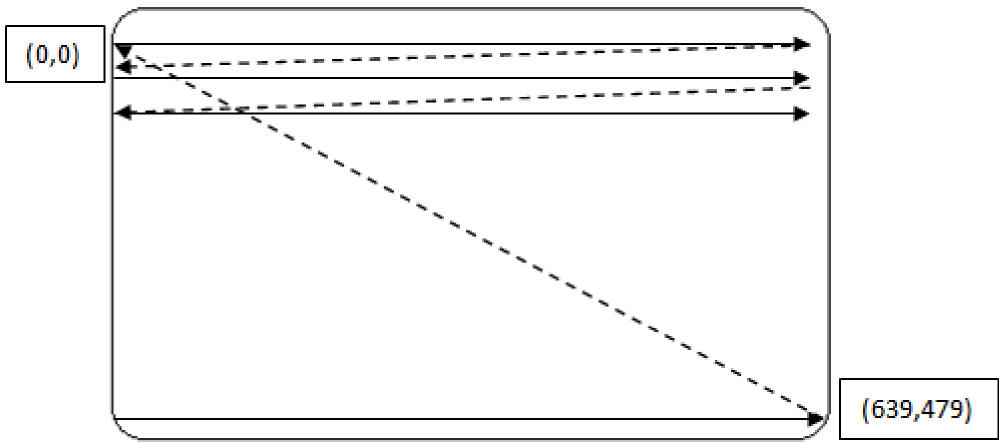
Un diagrama de la estructura de un monitor de rayos catódicos para comprender mejor sus bases funcionamiento se encuentra en la siguiente figura.

Figura 128 Diagrama de un monitor de tubo de rayos catódicos.



El barrido que debe realizar el rayo de electrones deberá recorres toda la pantalla de izquierda a derecha y de arriba abajo como se ilustra en la figura a continuación.

Figura 129 Barrido en una pantalla VGA



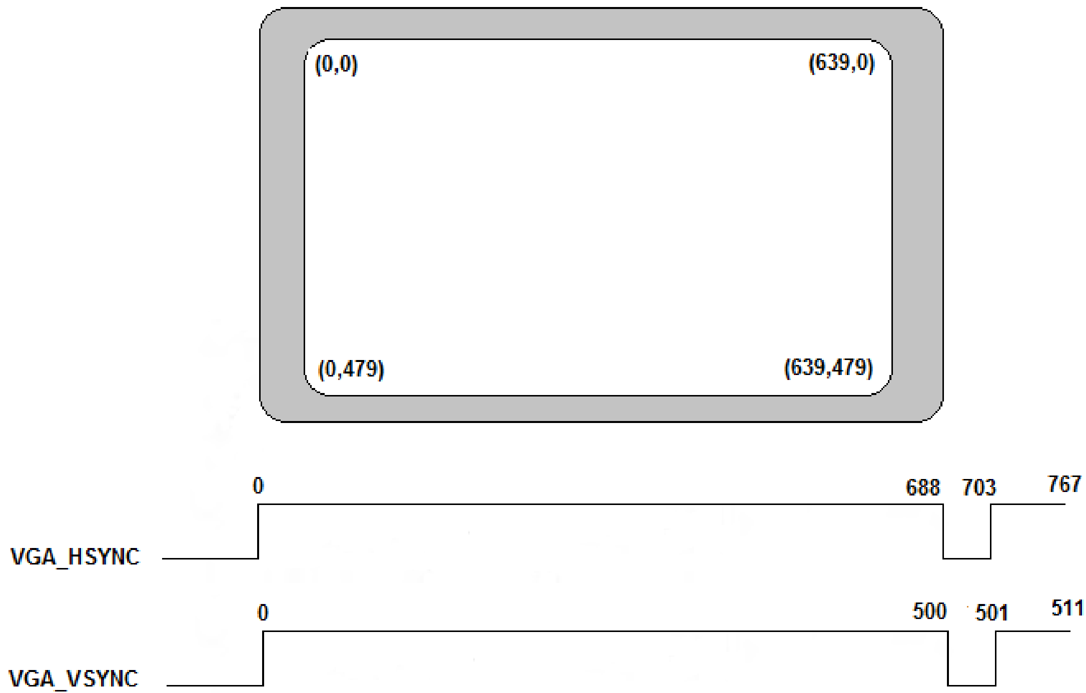
En la figura anterior los números (0,0) y (639,479) corresponden a las coordenadas de la pantalla que en este caso tendrá una resolución de 640x480.

En la figura las líneas continuas representan el movimiento en X y las líneas punteadas el movimiento en Y, realizar este barrido es sencillo utilizando dos contadores, cada uno para un eje.

El control de deflexión de las bobinas son las entradas digitales de sincronización VGA_HSYNC y VGA_VSYNC del puerto VGA las cuales generan en el interior del una señal diente de sierra que controla el campo magnético de las bobinas.

Con respecto a la posición del barrido en la pantalla, las señales de sincronización deben verse como lo indica la siguiente figura.

Figura 130 Señales VGA_HSYNC y VGA_VSYNC



Los valores sobre cada grafica especifican el valor de la coordenada en el cual se produce el cambio de estado, se observa que dichos valores superan los valores del tamaño de la pantalla visible esto debido a que cada pantalla posee bordes donde se puede guardar información que no será visualizada, dichos bordes pueden variar su tamaño entre distintos monitores, sin embargo en general el tamaño completo de la pantalla será de 800x525 si se trabaja con resolución de 480x640.

17.5.1 Módulo VGA_module

Este módulo genera un control del barrido en pantalla y asigna los valores deseados a las señales de sincronización

```
module VGA_module
( clk, h_sync, v_sync,
Display_on, cont_x, cont_y);

input clk;
output  h_sync, v_sync;
output reg Display_on;
output reg [9:0] cont_x=0, cont_y=0;

reg lim_x, lim_y;

wire cont_x_max = (cont_x == 10'd767);
```

Control del barrido

Los contadores cont_x y cont_y llevaran el control de la posición del barrido

Lim_x y lim_y controlaran las señales de sincronización

Especifica el valor máximo del barrido en X

```
always @ (posedge clk)
begin
if (cont_x_max)
cont_x <= 0;
else
cont_x <= cont_x+10'd1;
end
```

Mientras no se haya llegado al límite del barrido en X, el contador aumentara el valor en 1.
Así se cubre el área horizontal de la pantalla

```
always @ (posedge clk)
begin
if (cont_x_max)
   begin
      if (cont_y>=10'd511) cont_y <=10'd0;
      else cont_y <= cont_y +10'd1;
   end
end
```

Cada vez que termina un barrido en X se varía el cont_y

Cont_y aumenta hasta ser mayor o igual a 511, lo que indica el límite de la pantalla, en este punto se reinicia
Así se cubre el área vertical de la pantalla

```
always @ (posedge clk)
begin
lim_x <= (cont_x[9:4] == 6'h2B);
lim_y <= (cont_y == 10'd500);
end
```

Las coordenadas se utilizan para dar valor a lim_x y lim_y.
Obsérvese que el valor de estas dos variables corresponde al valor lógico negado de las señales de sincronización

```
always @ (posedge clk)
begin
if ((cont_x <= 10'd640)&&(cont_y<10'd480))
Display_on <= 1'b1;
else
Display_on <= 1'b0;
end
```

Display_on indicara el momento en el cual el barrido se encuentre dentro del área visible de la pantalla

```
assign h_sync = ~lim_x;
assign v_sync = ~lim_y;

endmodule
```

Se asignan los valores a las señales de sincronización

17.5.2 Módulo estructural top_vga

Para facilitar la comprensión del funcionamiento de los demás módulos, es necesario entender primero las funciones del módulo estructural.

Además de recoger todas las funciones de los demás módulos, este módulo asigna el valor de las variables VGA_R, VGA_G, y VGA_B dependiendo del valor de las coordenadas de barrido.

```
module top_vga
(CLK_50M, VGA_H_SYNC,
VGA_V_SYNC, VGA_R, VGA_G, VGA_B,
);

input CLK_50M;
output VGA_H_SYNC, VGA_V_SYNC;
output reg [3:0]VGA_R, VGA_G, VGA_B;
```

Modulo principal con todas las entradas y salidas que se utilizaran

```
reg CLK_25M;

always @ (posedge  CLK_50M)
begin
  CLK_25M <= ~CLK_25M;
end
```

Se divide la frecuencia del reloj de 50M para obtener 25MHz

El reloj se escoge de 25MHz ya que es el valor recomendado para una pantalla de resolución 640x480 (800x525 con los bordes) con una velocidad de 60 imágenes por segundo (velocidad en que el ojo humano ve movimientos continuos)

800x525x60 ≈ 25M

```
wire [9:0]CounterX;
wire [8:0]CounterY;
wire inDisplayArea;

VGA_module barrido (
    .clk(CLK_25M),
    .h_sync(VGA_H_SYNC),
    .v_sync(VGA_V_SYNC),
    .Display_on(inDisplayArea),
    .cont_x(CounterX),
    .cont_y(CounterY)
    );
```

Se instancia el modulo de control del barrido

```
wire pantalla =((CounterX[9:0]==10'd639) &&  (CounterY [8:0]==9'd479)) ;
```

La variable pantalla tendrá un valor lógico alto cuando las coordenadas del barrido estén en el extremo inferior derecho. Por lo tanto esta variable indicara el final de un ciclo completo de barrido.

```
wire border =((CounterX[9:0]>=10'd15) &&  (CounterY [8:0]>=9'd15)) &&
             ((CounterX[9:0]<=10'd625) &&  (CounterY [8:0]<=9'd465));
```

La variable border estará activa cuando se encuentre dentro de los parámetros dados, en este caso estará activa en un área cuadrada entre (15,15) y (625,465)

```
wire [11:0] rgb_out;
wire [18:0] YX;

bola instance_name (
     .RGB(rgb_out),
     .YX({1'b0,YX})
     );
```

Se instancia el modulo en donde se encuentra el mapa de bits del objeto pelota

```
wire pelota_on = ((YX[18:10]>= 9'd0) && (YX[18:10]<9'd10) &&
                  (YX[9:0]>10'd0) && (YX[9:0]<=10'd10));
```

Esta variable deberá estar encendida cuando las coordenadas YX descritas en el modulo mov_obj se encuentren en el área indicada, justo como antes, se define un cuadrado en donde se visualizara la pelota

```
wire boder2;
mov_obj rebote (
     .pantallazo(pantalla),
     .CounterY(CounterY),
     .CounterX(CounterX),
     .border2(border2),
     .YX(YX)

     );
```

Instancia del modulo que controla el movimiento de los objetos

La salida de este modulo son las coordenadas en movimiento de un cuadrado y los valores YX que conforman la pelota

Codificador con prioridad

```
always @(posedge CLK_50M)
begin
if (pelota_on)
begin
  vga_R <= rgb_out[11:8];
  vga_G <= rgb_out[7:4];
  vga_B <= rgb_out[3:0];
 end
else if (border2)
begin
  vga_R <= 4'b1111;
  vga_G <= 4'b1111;
  vga_B <= 4'b1111;
end
```

Si el área de la pelota esta activa se enviaran los colores correspondientes de acuerdo al modulo bola

Si el área del cuadrado definido por border2 esta activa se enviara el color blanco a la pantalla

```
else if (border)
begin
  vga_R <=  4'd6;
  vga_G <= 4'd0;
  vga_B <= 4'd6;
end
```

En el área de border se envía el color morado a la pantalla

```
else
begin
   vga_R <= 4'b1111;
   vga_G <= 4'b1111;
   vga_B <= 4'b1111;
end
end

endmodule
```

Cualquier área de la pantalla que no haya sido asignada se verá blanca

17.5.3 Módulo mov_obj

Este módulo tiene como función detectar los extremos del área permitida de movimiento (en este caso el cuadrado formado por border en el módulo estructural) y cambiar las coordenadas de un cuadrado dibujado y de la pelota definida por el módulo bola.

```
module mov_obj
( pantallazo, CounterY ,
CounterX, border2, YX);

input pantallazo;
input [9:0] CounterX;
input [8:0] CounterY;
output border2;
output [18:0] YX;

reg [9:0] move_x=10'd300;
reg [8:0] move_y = 9'd300;
```

Posición inicial del cuadrado

```
reg [9:0] move_x2=10'd200;
reg [8:0] move_y2 = 9'd150;
```

Posición inicial de la pelota

```
wire border2 =((CounterX[9:0]>=move_x) &&  (CounterY [8:0]>=move_y)) &&
((CounterX[9:0]<=(move_x+10'd10)) &&  (CounterY [8:0]<=move_y+9'd10));
```

Al igual que se hizo con border se define border2 como un cuadrado de inicio (move_x, move_y) hasta (mov_x+10, mov_y+10).
Como se ve las coordenadas del cuadrado son variables, característica que le permite moverse por la pantalla

Se realiza el mismo proceso con YX

```
wire [18:0] YX = {{CounterY-move_y2},{CounterX-move_x2}};

reg dirX= 1'b0;
reg dirY= 1'b0;
reg hit;
always @(posedge pantallazo)
begin
if((move_x<=10'd15))
 dirX<=1'b1;
else if (move_x >= 10'd615)
dirX<=1'b0;
else
dirX<= dirX;
end

always @ (posedge pantallazo)
begin
if (move_y <=9'd15)
dirY <= 1'b1;
else if (move_y>=9'd455)
dirY <= 1'b0;
else
dirY<= dirY;
end

always @ (posedge pantallazo)
begin

      if (dirX) move_x <= move_x+10'd1;
      else  move_x <= move_x - 10'd1;

end

always @ (posedge pantallazo)
begin

      if (dirY) move_y <= move_y+9'd1;
      else  move_y <= move_y - 9'd1;

end
```

El estado de dirX y dirY indicara la dirección de los contadores move_x y move_y

Si se cumple la condición se llego al extremo izquierdo del área border y se cambiara la dirección de conteo

Si se cumple la condición se llego al extremo derecho de border y se cambiara la dirección

Si se cumple la condición se llego al extremo superior de border y se cambiara la dirección

Si se cumple la condición se llego al extremo inferior de border y se cambiara la dirección

El cambio se realiza cada fin de ciclo de barrido para crear la ilusión de movimiento continuo

Dependiendo del estado de dir, el conteo será ascendente o descendente

```
reg dirX2= 1'b0;
reg dirY2= 1'b0;
reg hit2;
always @(posedge pantallazo)
begin
if((move_x2<=10'd15))
 dirX2<=1'b1;
else if (move_x2 >= 10'd615)
dirX2<=1'b0;
else
dirX2<= dirX2;
end
```

Se repite de nuevo el mismo proceso para darle movimiento al objeto pelota

Se definen los extremos del movimiento en el eje x

```
always @ (posedge pantallazo)
begin
if (move_y2 <=9'd15)
dirY2 <= 1'b1;
else if (move_y2>=9'd455)
dirY2 <= 1'b0;
else
dirY2<= dirY2;
end
```

Se definen los extremos del movimiento en el eje y

```
always @ (posedge pantallazo)
begin
	if (dirX2) move_x2 <= move_x2+10'd1;
	else  move_x2 <= move_x2 - 10'd1;
end

always @ (posedge pantallazo)
begin
	if (dirY2) move_y2 <= move_y2+9'd1;
	else  move_y2 <= move_y2 - 9'd1;
end
endmodule
```

Se realiza el control del conteo

17.5.4 Módulo bola

A diferencia del cuadrado que se ha dibujado, el diseño de la pelota que se tiene aquí es más complejo.
La pelota cuenta con forma no cuadrada y con varios matices de colores lo que conlleva que cada pixel debe describirse de manera individual, tarea que es bastante extensa
El módulo se agrego al proyecto para ilustrar la manera de describir figuras con diferentes matrices de colores y formas no cuadradas, como se observa según cada valor de entrada YX se genera un color específico.
El módulo puede observarse en el proyecto.

17.5.5 Ejercicios propuestos

- Implemente un código Verilog que visualice en la pantalla dos objetos cuadrados de diferentes tamaños con movimientos contrarios.
- Genere un código Verilog que visualice en la pantalla VGA cuatro rectángulos de diferentes colores sobre un fondo blanco, dentro de cada uno de los rectángulos deberá existir un objeto cuadrado que realice un movimiento dentro del área del rectángulo donde se encuentra.
- Implemente un código Verilog que permita manejar el movimiento de un objeto en pantalla mediante el uso de los pulsadores de la tarjeta de desarrollo Spartan3A

18 PRÁCTICA 5: CORE GENERATOR

18.1 OBJETIVOS

- Familiarizarse con la forma de uso, las aplicaciones y los alcances del CORE Generator de Xilinx.
- Ejemplificar el uso de un CORE mediante la realización de un sistema de conversión de paralelo a serial con memoria.

18.2 RECURSOS UTILIZADOS

- Tarjeta de desarrollo Spartan3A
- Core Generator
- Switches
- LEDs
- Pulsadores
- Computador con el software ISE Project Navigator

18.3 PREREQUISITOS

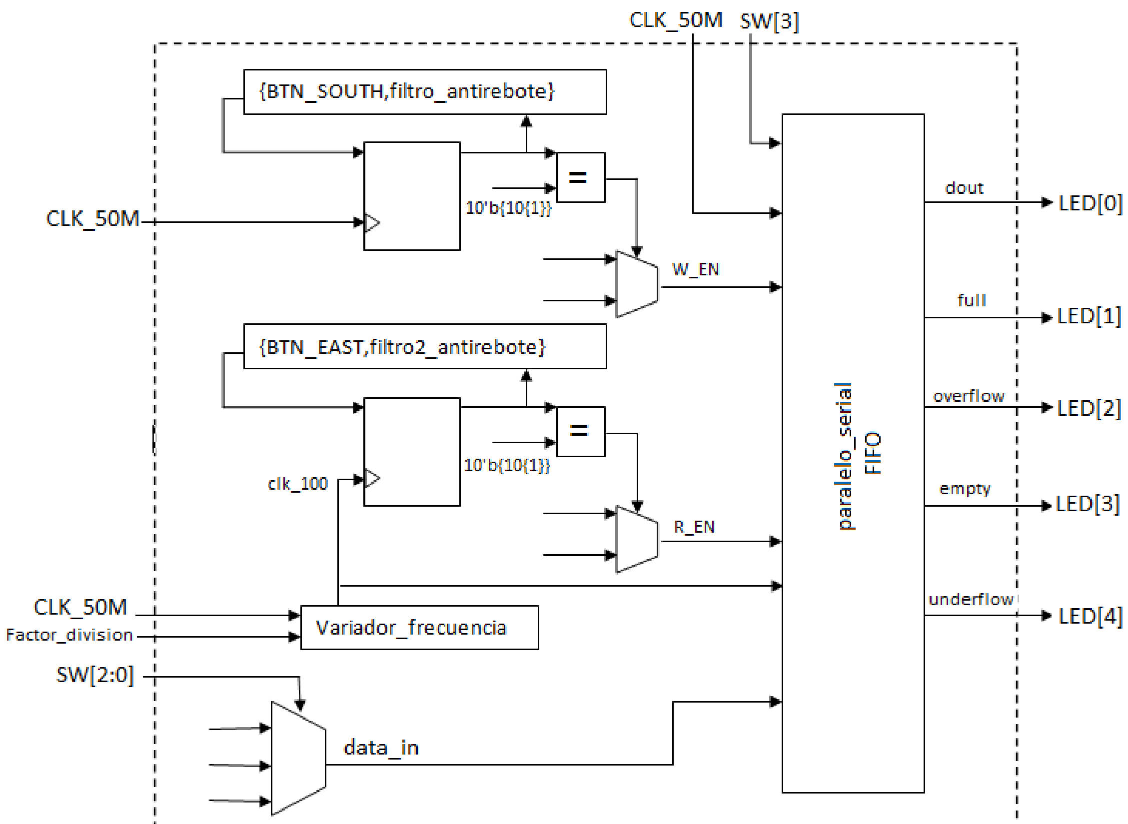
Para el desarrollo de esta práctica se requiere un conocimiento previo sobre características de memorias y conceptos básicos de digitales para lograr comprender el funcionamiento del FIFO.

18.4 EXPLICACIÓN DE LA PRÁCTICA

Mediante el uso del CORE Generator, se creara una memoria first-in first-out (FIFO) que tendrá como objetivo almacenar los valores indicados por el usuario y después leerlos de manera serial.

Un diagrama de bloques que modela el funcionamiento en general de la práctica se puede observar en la siguiente figura.

Figura 131 Diagrama de bloques para la práctica 5



Como se observa en el diagrama anterior el proyecto cuenta solamente con dos módulos que realizaran las siguientes funciones.

- Módulo paralelo_serial: este será el módulo que se generara con el Core Generator y deberá efectuar todas las funciones propias de una memoria tipo FIFO.
- Módulo variador_frecuencia: este módulo deberá generar una frecuencia de reloj distinta para manejar las señales de lectura de la memoria FIFO.

Con las funciones de los módulos que constituyen la práctica terminadas el módulo estructural deberá realizar las siguientes funciones.

- Implementar filtros que eliminen los glitches de las señales de entrada de los pulsadores.
- Implementar un circuito que seleccione el valor que recibirá la memoria FIFO a su entrada

Además de esto deberá manejar Las señales externas que influyen en el manejo de la práctica cada una con la siguiente función.

- Los Switches [2:0] definirán el valor que será guardado en la memoria FIFO
- El switch [3] funcionara como el reset de la memoria.

- El LED [0] será la salida serial de la memoria
- El LED [1] será el indicador de que la memoria está llena
- El LED [2] indicara un error en la acción de lectura de la memoria por estar esta vacía.
- El LED [3] indicara que la memoria está vacía
- El LED [4] indicara un error en la acción de escritura por estar llena la memoria.
- Después de asignado el valor mediante los Switches, el pulsador South guardara el valor en la memoria, esta acción puede realizarse varias veces.
- Una vez haya un valor guardado en la memoria, el pulsador East será la acción que indique que el primer valor almacenado se leerá de manera serial y así hasta que la memoria este vacía.

18.5 DESARROLLO DE LA PRÁCTICA

El CORE Generator, es una interfaz grafica cuya función es generar módulos de alto nivel para maximizar el rendimiento y la flexibilidad de un diseño además de reducir los riesgos de error para las FPGA Xilinx, reduciendo considerablemente el tiempo de desarrollo.

Los diseños ofrecidos por Xilinx para el CORE Generator garantizan sistemas robustos con más de 400 opciones personalizables.

El CORE utilizado es el FIFO Generator, una memoria first-in first-out, útil para aplicaciones que requieran un almacenamiento ordenado, alto rendimiento y optimización de recursos.

El FIFO puedo ser personalizado en las siguientes características:

- Amplitud de las palabras de entrada y salida
- Cantidad de espacios de almacenamiento
- Relación de reloj entre comando de escritura y lectura
- Banderas de estados.

Para este práctica se implementara un FIFO con relojes independientes, RAM en bloque, una amplitud del dato de entrada de 8 bits, una amplitud del dato de salida de 1 bit, una capacidad de almacenamiento de 1024 palabras y una interfaz capaz de trabajar en dos periodos de reloj diferentes al mismo tiempo. Estas características deberán ser elegidas al momento de crear el CORE.

18.5.1 Módulo paralelo_serial

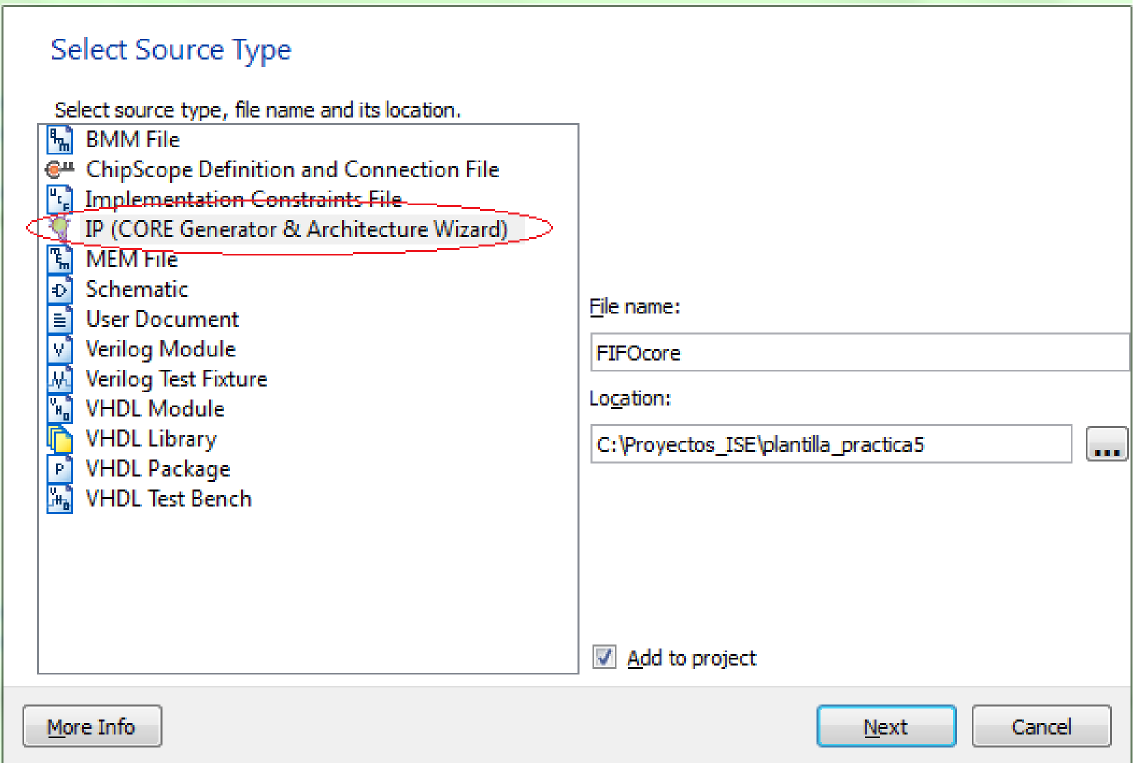
Este módulo tendrá la tarea de almacenar una o más palabras de 8 bits, guardarlas en el orden en que fueron ingresadas y mostrar esos datos de manera serial, esto sincronizando dos periodos de reloj diferentes ya que la escritura se realizara sobre el reloj de 50 Mhz y la lectura utilizando un reloj de 100 Hz.

Realizar estas tareas resulta en un diseño complejo y extenso como para ser incluido en un mismo módulo, sin embargo estas tareas las puede realizar un diseño ya creado y comprobado por Xilinx, el FIFO Generator

Para generar el Core se deben seguir los siguiente pasos.

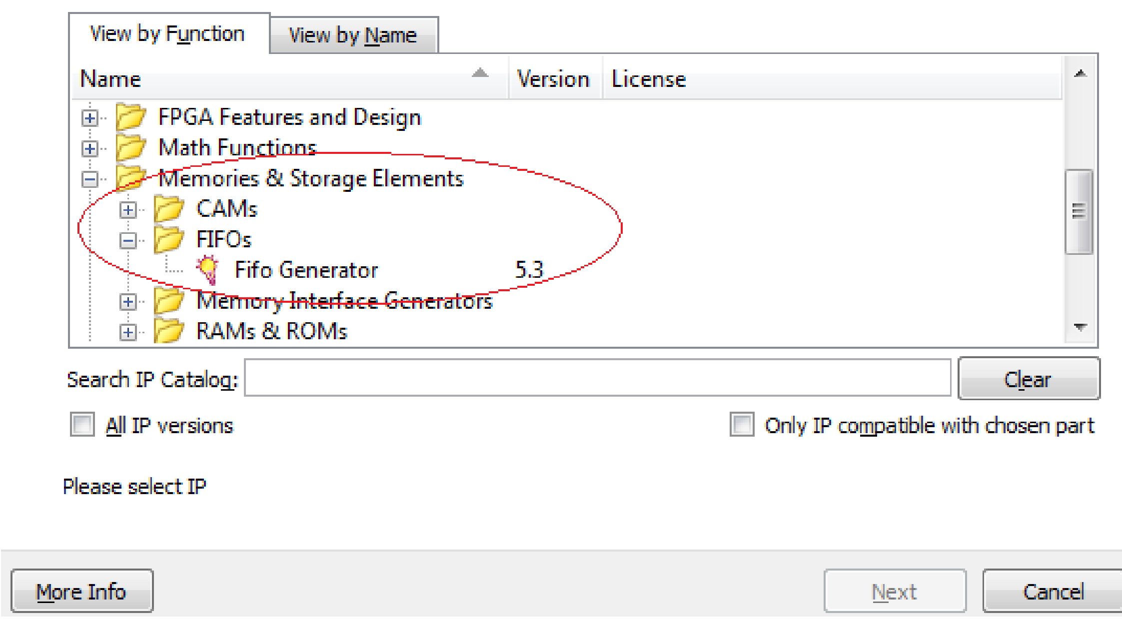
1. En el nuevo proyecto se debe ir a la opción **new source** igual como se haría al crear un nuevo módulo Verilog.
2. se debe elegir la opción de fuente **IP (CORE & Architecture Wizard)**, se asigna el nombre a la fuente y se oprime la opción next como se muestra en la figura siguiente.

Figura 132 Ventana para la selección de una fuente CORE.



3. En la siguiente ventana que se abre, se debe elegir el Fifo Generator como se observa en la figura siguiente.

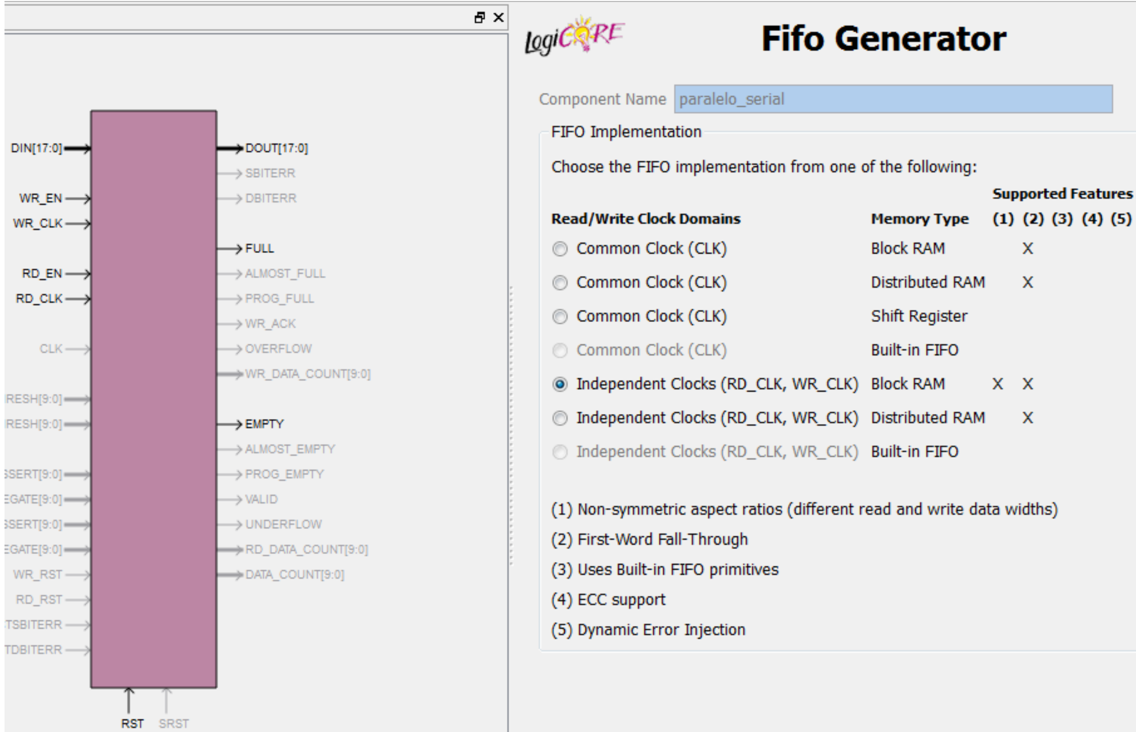
Figura 133 Selección de la herramienta CORE a utilizar



4. Una vez seleccionado el Fifo Generator, se debe oprimir el botón next y seguidamente finish.

Completados estos pasos se abrirá la ventana mostrada en la siguiente figura en donde se podrá comenzar a personalizar el módulo para que se acople a las necesidades del diseño.

Figura 134 Implementación del FIFO: tipo de memoria

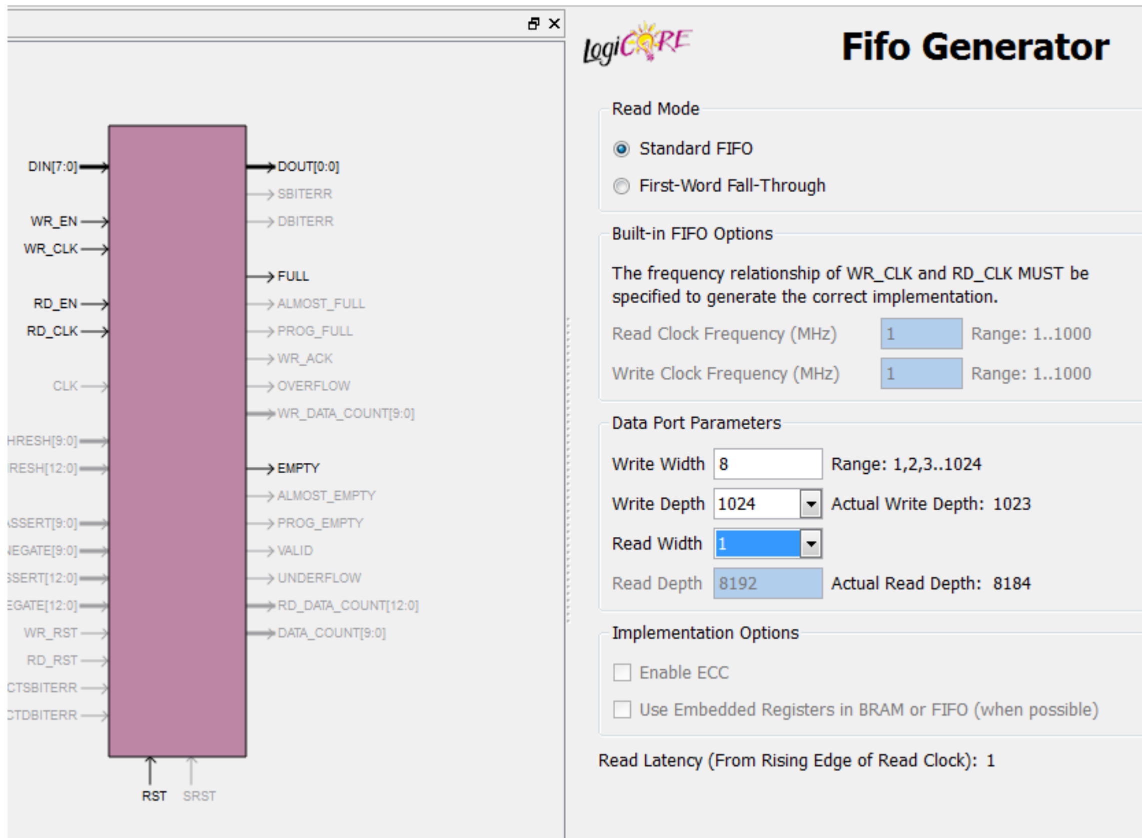


1. Se debe elegir en la ventana correspondiente a la figura anterior la opción **independent cloks block RAM**, seguidamente hacer clic en next.
2. En la siguiente ventana que se muestra en la figura a continuación se deben elegir las siguientes opciones y se da clic en next

- En la sección read mode se elige **estándar FIFO**
- Write width = 8
- Write depth = 1024
- Read width = 1

Esto permitirá datos de entrada de 8 bits, capacidad de almacenamiento de 1024 palabras y datos de salida de 1 bit.

Figura 135 Implementación del FIFO: parámetros de puertos y tipo de lectura

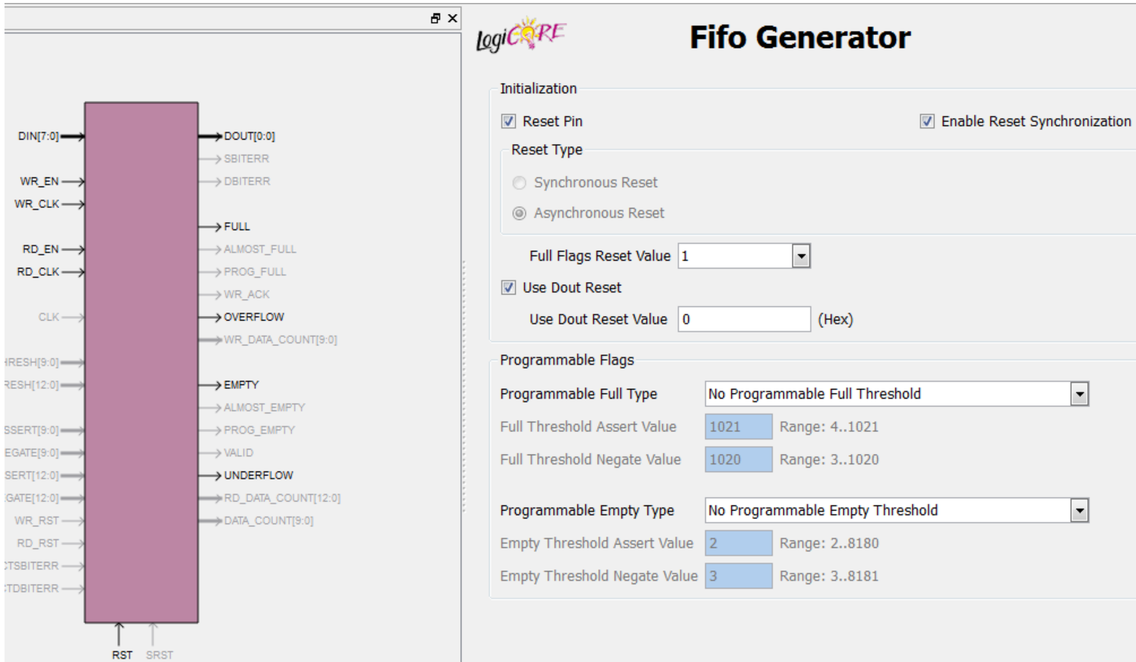


3. En la ventana que se muestra a continuación se deben elegir las opciones de banderas **underflow** y **overflow** con estado activo en alto y se da clic en next.

Figura 136 Implementación del FIFO: selección de banderas

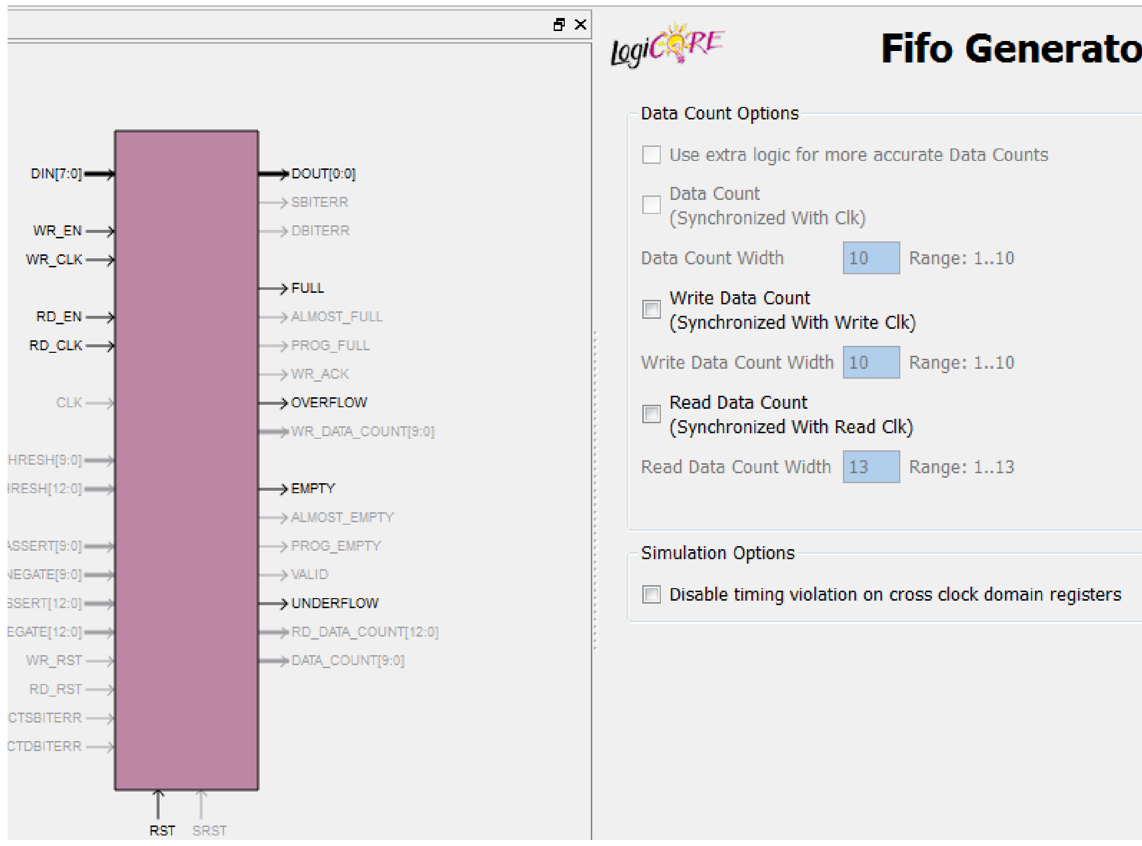
4. En la siguiente ventana se verifican que las siguientes opciones estén seleccionadas como se observa en la figura siguiente y se da clic en next.
- Reset pin
- Enable Reset Synchronization
- Use Dout Reset
- Programmable Full Type = No programmable Full Threshold.
- Programmable Empty Type = No programmable Full Threshold.

Figura 137 Implementación del FIFO: opciones de inicialización.



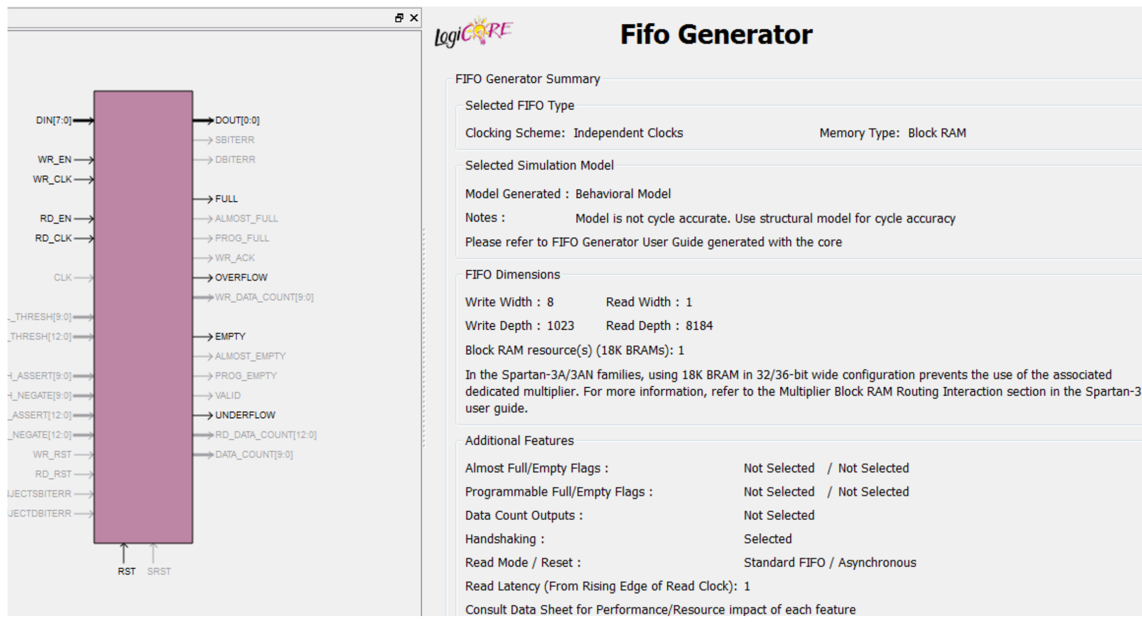
5. En la siguiente ventana que se muestra en la figura a continuación se debe verificar que ninguna opción este seleccionada y se da clic en next.

Figura 138 Implementacón del FIFO: opciones de conteo de datos



6. La última ventana que se muestra en la figura a continuación corresponde a un resumen de todas las opciones seleccionadas. Después de verificar que estén correctas se da clic en generar.

Figura 139 Implementación del FIFO: resumen



Con estos pasos completados el módulo FIFOcore se agregara al proyecto de trabajo con las siguientes entradas y salidas.

- rst (entrada) : señal de reset del módulo
- din (entrada) : entrada de datos al módulo
- wr_clk (entrada): entrada de la señal de reloj con el cual se sincronizaran las operaciones de escritura del módulo
- rd_clk (entrada): entrada de la señal de reloj con el cual se sincronizaran las operaciones de lectura del módulo.
- wr_en (entrada): señal que permite la escritura de un dato dentro de la memoria, cuando se encuentra en alto el dato en la entrada din se almacena en la memoria.
- rd_en (entrada): señal que permite la lectura de un dato desde la memoria, cuando se encuentra en alto el dato que primero fue guardado en la memoria se transporta al puerto de salida dout.
- dout (salida): puerto en donde saldrán los datos que sean leídos de la memoria
- full (salida): bandera que indica que la memoria se encuentra llena
- overflow (salida): bandera que indica que ocurrió un error en la escritura causado por estar la memoria llena
- empty (salida): bandera que indica que la memoria se encuentra vacía
- underflow (salida): bandera que indica que ocurrió un error en la lectura causado por estar la memoria vacía.

Se debe tener en cuenta que ya que se trabajaran con dos frecuencias de reloj diferentes cada señal debe trabajarse dentro de su frecuencia correspondiente, es decir, todas la señales de escritura deberán realizarse con el reloj en wr_clk y todas las señales de lectura deberán realizarse con el reloj rd_clk.

18.5.2 Módulo variador_frecuencia

Este módulo ya había sido utilizado como una opción para generar relojes de menor frecuencia a partir de alguno de los relojes incluidos dentro de la tarjeta de desarrollo en la práctica 1.
En este caso se generara una frecuencia de 100 Hz con este módulo.

18.5.3 Módulo estructural ctrl_FIFO

Este módulo se encargara de manejar las señales que requiere el FIFO Core para su funcionamiento.

```
module ctrl_FIFO (LED, CLK_50M,
SW, BTN_SOUTH, BTN_EAST);

input CLK_50M;
input [3:0] SW;
input BTN_SOUTH;
input BTN_EAST;
output  [7:0] LED;
```

Declaración del modulo y de las entradas y salidas a utilizar

```
reg [7:0] data_in;
```

Registro que guarda el dato de acuerdo al estado de los Switches

```
always @ (posedge CLK_50M)
begin
case (SW[2:0])
4'd0: data_in <= 8'b10101010;
4'd1: data_in <= 8'b11001100;
4'd2: data_in <= 8'b00000100;
4'd3: data_in <= 8'b11000000;
4'd4: data_in <= 8'b11000011;
4'd5: data_in <= 8'b00100101;
4'd6: data_in <= 8'b10010010;
4'd7: data_in <= 8'b11001111;
default: data_in <= data_in;
endcase
end
```

El estado de los Switches [0:2] determinan el valor asignado a data_in

```
wire clk100;
variador_frecuencia clk2 (
    .clk(CLK_50M),
    .reset(1'b0),
    .frecuencia_resultante(clk100),
    .factor_division(32'd500000)
    );
```

Instancia del modulo variador_frecuencia para generar un reloj de 100Hz

```
reg [9:0] filtro_antirebote, filtro2_antirebote;
```

Señales que reducirán el ruido en los pulsadores

```
reg W_EN, R_EN;
```

Control de las señales enable del FIFO

```
always @ (posedge CLK_50M)
begin
filtro_antirebote [9:0]<= {BTN_SOUTH,filtro_antirebote[9:1]};
end
```

El vector filtro_antirebote se irá llenando con los valores de BTN_SOUTH

```
always @ (posedge CLK_50M)
begin
if (filtro_antirebote == 10'b111111111)
   W_EN<= 1'b1;
else
   W_EN <= 1'b0;
end
```

Cuando el pulsador BTN_SOUTH permanezca en un valor estable de 1 por 10 ciclos de reloj se asigna el valor a la señal W_EN.
Esto disminuye un poco los glitches presentados por ruido en el pulsador

```
always @ (posedge clk100)
begin
filtro2_antirebote [9:0]<= {BTN_EAST, filtro2_antirebote[9:1]};
end

always @ (posedge clk100)
begin
if (filtro2_antirebote == 10'b111111111)
   R_EN<= 1'b1;
else
   R_EN <= 1'b0;
end

wire rst;
assign rst = SW[3];

wire dout, full, overflow, empty, underflow;
FIFOcore paralelo_serial (
   .rst(rst),
   .wr_clk(CLK_50M),
   .rd_clk(~clk100),
   .din(data_in), // Bus [7 : 0]
   .wr_en(W_EN),
   .rd_en(R_EN),
   .dout(dout), // Bus [0 : 0]
   .full(full),
   .overflow(overflow),
   .empty(empty),
   .underflow(underflow));

assign LED[0] = dout;
assign LED[1] = full;
assign LED[2] = overflow;
assign LED[3] = empty;
assign LED[4] = underflow;

assign LED[7:5] = 0;
endmodule
```

Se realiza la misma acción para filtro2_antirebote y R_EN pero esta vez con el reloj de 100Hz ya que con este estarán sincronizadas las señales de lectura del FIFO

SW[3] manejara el estado de la señal rst

Se instancia el modulo FIFOcore y se asignan las respectivas señales de control a los puertos de entrada y salida del modulo.

Se asigna una señal a cada Led para poder observar su estado

Se les asigna un valor a los Leds que no están siendo utilizados para evitar errores

18.5.4 Ejercicios propuestos

- Utilizando un Core, implemente una memoria RAM que almacene los datos dados y que realice un módulo que permite leer dichos datos desde el ultimo hasta el primero.

19 CONCLUSIONES

Las FPGA constituyen una buena opción en el momento de buscar una solución eficaz para realizar diseños físicos debido a sus características de programación, velocidad, fiabilidad y economía.

La decisión de utilizar el lenguaje de programación Verilog se dio debió a su facilidad de comprensión de los programas resultado de la cercana relación entre sus instrucciones y directivas con el lenguaje C.

La tarjeta de desarrollo SPARTAN3A representa un buen punto de inicio para comenzar el uso de algún lenguaje de programación HDL, debido a la variedad de elementos periféricos que requieren para su uso diferentes construcciones, lo que proporciona una buena práctica para mejorar las habilidades de programación y comprensión

Verilog HDL cuenta con una gran cantidad de operadores, instrucciones y construcciones que le concede flexibilidad al momento de diseñar circuitos en hardware y varias vías de especificación para asegurar que el resultado final que se obtenga sea igual o lo más cercano posible al diseño inicial ideado por el programador.

Las habilidades que se buscan desarrollar con cada una de las prácticas corresponden a las bases que sirven como punto de arranque para diferentes aplicaciones que se quieran diseñar en ocasiones futuras

El uso de Picoblaze dentro de los diseños en hardware, ofrece una gran alternativa para la simplificación de las rutinas de programación gracias a la integración de software dentro del diseño, aunque deba utilizarse solo en tareas que no requieran tiempo de funcionamiento demasiado cortos.

El texto brinda los conocimientos fundamentales y necesarios para todo aquel que desee aprender sobre el lenguaje de programación Verilog y sobre las FPGA de una manera sencilla, abriendo así el camino para que el lector después avance en su conocimiento sobre temas más avanzados y desarrolle diseños más complejos.

20 RECOMENDACIONES

La realización de las prácticas propuestas en el texto debe complementarse tratando de expandir los alcances de cada una de ellas, añadiendo funciones extras, igualmente hacer el esfuerzo de utilizar algún otro elemento periférico de los disponibles dentro de la tarjeta de desarrollo para obtener un mejor manejo y confianza en el lenguaje de programación Verilog.

Leves diferencias se pueden presentar entre las diferentes marcas y modelos de las FPGA, es recomendable utilizar otros modelos de tarjetas de desarrollo para expandir los conocimientos y experiencias.

Existen diversas aplicaciones que podrían realizarse haciendo uso de las FPGA's pero que están siendo implementadas utilizando otras herramientas, teniendo en cuenta el rápido surgimiento que tuvieron las FPGA, dichas aplicaciones deberían tratar de ser construidas utilizándolas en lo más posible para lograr tener una noción de los avances que se vayan generando en esta área.

21 BIBLIOGRAFÍA

CAVANAGH, Joseph. Verilog HDL: Digital Design and Modeling. Estado Unidos: CRC Press 2007

Carpio Prado. Fernando, VHDL lenguaje para descripción y modelado de circuitos, universidad de valencia

CHAPMAN, Ken. KCPSM3 8bits Microcontroller for spartan3, virtex-II and virtex-IIPRO. Internet: www.xilinx.com

FLOYD, Thomas L. Fundamentos de Sistemas Digitales. 6a.ed. Madrid: Prentice Hall, c1997

GUISADO, José Luis. Lenguajes de Descripción de Hardware, Universidad de Extremadura, España. 2002

HARRIS, David Money. HARRIS, Sarah L. Digital design and computer architecture. Estados Unidos: Morgan Kaupmann. 2007

MAXFIELD, Clive Max. The design warrior guide to FPGA: Devices tools and flows. Ámsterdam 2004

P. CHU, PONG, FPGA prototyping by Verilog examples. Estados Unidos: WILEY, 2008

PLANITKAR, SAMIR, Verilog HDL. Prentice hall, 2003

THOMAS, Donal E. MOORBY, Philip R. The Verilog Hardware Description Language, 5 ed. Boston: Kluwer Academic Publishers, 2002

TOCCI, Ronald J. Circuitos y dispositivos electrónicos. México: McGraw-Hill, 1986

WAKERLY, John F.Diseño digital principios y prácticas. México: Prentice Hall, 1992

XILINX, Spartan 3A/3AN Starter kit board user guide, UG334 (v1.1) June 19, 2008. Internet: www.xilinx.com

XILINX, Spartan 3a fpga starter kit board user guide UG330 (v 1.3) June 21 de 2007. Internet: www.xilinx.com

XILINX. Spartan-II 2.5V FPGA Family: Functional Description, DS001-2 (v2.2) September 3, 2003. Internet: www.xilinx.com