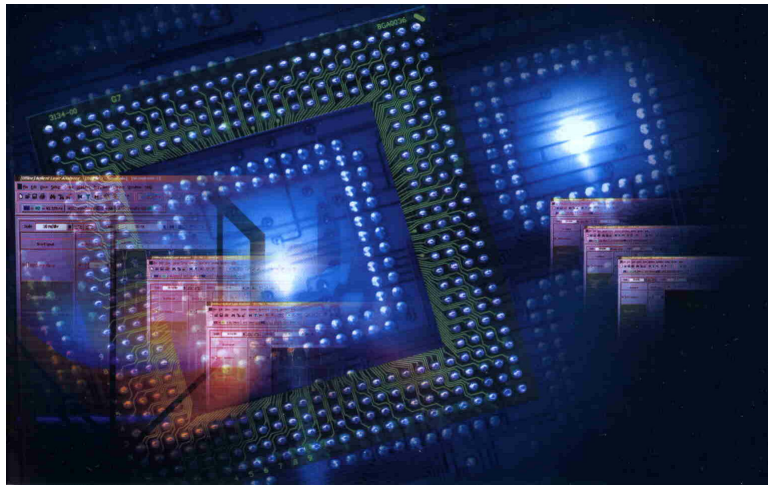




Facultad de Ingeniería
Universidad Nacional de la Plata
Departamento de Electrotecnia
Cátedra de Trabajo Final

Introducción a los Dispositivos FPGA. Análisis y ejemplos de diseño



Autor: Bozich, Eduardo Carlos 49989
Director: Noriega, Sergio

La Plata, año 2005



**Cátedra de Trabajo Final – Electrónica
Departamento de Electrónica
Universidad Nacional de La Plata**

Denominación del Trabajo Final:

***Introducción a los dispositivos FPGA.
Análisis y ejemplos de diseño***

Grupo de Trabajo:

N° de alumno	Nombre y Apellido	Calificación
49989	Eduardo Bozich	

Director (es): Sergio Noriega

Lugar de realización: CIOp (Centro de Investigaciones Ópticas)

Fecha: 18 de Marzo, 2005

Firma de los integrantes de la mesa examinadora:

Concepto general del Trabajo (*):

Comentarios sobre el trabajo:

(*) Calificar con: regular (**R**) – Bueno (**B**) – Muy Bueno (**MB**) – Excelente (**EX**):

Agradecimientos

Deseo expresar mi agradecimiento a todas las personas e instituciones que hicieron que este trabajo pueda ser llevado a cabo:

- A mi familia, que me apoyó incondicionalmente, y que me permitió llegar a esta instancia.
- A mis compañeros y a mis amigos, los cuales me brindaron su ayuda y apoyo durante toda ésta etapa.
- Al Ing. Sergio Noriega, por haberme ofrecido el proyecto y confiado en mi capacidad, brindando su dirección y asesoramiento además de su apoyo y estrecha colaboración durante todo éste tiempo.
- Al Centro de Investigaciones Ópticas (CIOp) y a todo su personal, por brindarme las herramientas e instrumental necesarios para llevar a cabo todo el proyecto.
- A todos los que de una forma u otra hicieron posible que este proyecto se concrete.

A mis padres y amigos

Resumen

El siguiente proyecto tuvo como finalidad el estudio en general de los dispositivos lógicos programables por hardware, en particular de los de tipo FPGA (Arreglo de Compuertas Programables por el Usuario); para luego realizar dos implementaciones de circuitos lógicos en ellos.

Una de la implementaciones es un medidor de frecuencias y períodos para señales de lógica TTL de hasta 100MHz, con una interfaz física con el usuario a través de una plaqueta y con una comunicación con PC para ser controlado a través de la misma.

La otra implementación es un adquisidor autónomo de datos, mediante el uso del conversor analógico digital ADC0820, con comunicación a PC; para señales comprendidas entre 0 y 2.5 Volts de hasta 250kHz de frecuencia.

El diseño de la lógica de ambos circuitos se realizó en AHDL (Lenguaje de Descripción de Hardware de Altera) y se implementaron en Altera con el dispositivo FLEX10K10.

El software para la interfaz a través de PC de ambas implementaciones se desarrollo en Visual Basic.

Abstract

The following project had as purpose the study in general of the programmable logical devices by hardware, in particular of those of type FPGA (Field Programmable Gate Array); to carry out two implementations of logical circuits then in them.

One of the implementations is a meter of frequencies and periods for signals of logical TTL up to 100MHz, with a physical interface with the user through a board and with a communication with PC to be controlled through the same one.

The other implementation is an autonomous data keeper, by the use of the analogic to digital converter ADC0820, with communication to PC; for signals include between 0 and 2.5 Volts of up to 250kHz of frequency.

The design of the logic of both circuits was carried out in AHDL (Altera Hardware Description Language) and they were implemented in Altera with the FLEX10K10 device.

The software for the interface through PC of both implementations was development in Visual Basic.

Prefacio

El objetivo de éste proyecto fue el de proporcionar a la cátedra de 'Introducción a los Sistemas Lógicos y Digitales' de la Facultad de Ingeniería de la UNLP, de un kit para ejemplificar una aplicación que conduzca al diseño a base de un dispositivo lógico programable del tipo FPGA (Arreglo de Compuertas Programables por el Usuario).

El proyecto se realizó, básicamente, en cuatro etapas, en una primera parte se comenzó por la recopilación de información sobre la distintas familias de lógica programable, para luego realizar una monografía (*Capítulo I*) haciendo hincapié en las FPGAs y dentro de ellas en la FLEX10K10, desarrollada por Altera; y en la Spartan II, su equivalente desarrollada por Xilinx.

En una segunda parte (*Capítulo II*) se confeccionó un tutorial del software MAX+plus II de Altera, dentro del cual se explican los diseños a partir del Editor Gráfico y del Editor de Texto en lenguaje AHDL (Lenguaje de Descripción de Hardware de Altera).

En la tercer etapa (*Capítulo III*) se realizó un estudio sobre las técnicas posibles para la implementación de la lógica de los siguientes circuitos en lenguaje AHDL:

- *Medidor de frecuencias y períodos*: con las siguientes características
 - Entradas de medición compatibles con lógica TTL.
 - Rango en modo frecuencia: 1Hz a 100MHz.
 - Rango en modo período: 100ns a 10s.
 - Representación mediante 6 dígitos.
 - Comunicación con PC a través de puerto paralelo en modo SPP.
- *Adquisidor autónomo de datos*: con las siguientes características
 - Controlado desde PC a través de puerto paralelo en modo SPP.
 - Control directo del proceso de adquisición de datos del conversor analógico digital ADC0820.
 - Capacidad de almacenamiento de hasta 500 muestras.

Se diseñaron para ambos circuitos, los programas en el Lenguaje de Descripción de Hardware de Altera, con sus respectivas simulaciones, en una forma detallada para su correcto entendimiento. También se estudió la forma de comunicar dichos circuitos con la PC a través de puerto paralelo en modo SPP (*Standard Parallel Port*).

Finalmente en la cuarta y última etapa (*Capítulo IV*) se procedió a la implementación de ambos circuitos en el dispositivo FLEX10K10 de Altera, mediante la utilización de la plaqueta experimental Upx10K10 (propiedad de la cátedra de 'Introducción a los Sistemas Lógicos y Digitales') y el diseño de tres plaquetas más. Se procedió también al desarrollo de los programas en Visual Basic para la comunicación con PC y se optimizaron los programas en AHDL para un correcto funcionamiento en los circuitos impresos diseñados.

Contenido

Capítulo 1	Lógica Programable	1
1.1	Introducción	1
1.2	ROM	2
1.2.1	ROM	2
1.2.2	PROM, EPROM, EEPROM, Flash	4
1.3	PLD	5
1.3.1	SPLD (Simple PLD)	6
1.3.1.1	PAL	6
1.3.1.2	PLA	9
1.3.1.3	GAL	10
1.3.2	CPLD (Complex PLD)	12
1.3.2.1	Matriz de Interconexiones Programables	12
1.3.2.2	Bloques Lógicos	13
1.3.2.3	Distribución de Productos	13
1.3.2.4	Macrocelda	14
1.3.2.5	Celda de entrada/salida	16
1.3.2.6	Tiempos de propagación	16
1.3.2.7	Otras características de las CPLD	16
1.3.3	FPGAs	17
1.3.3.1	Arquitectura general de un FPGA	17
1.3.3.2	Bloques Lógicos	19
1.3.3.3	Interconexión entre bloques programables	21
1.3.3.4	Bloques entrada/salida	22
1.3.3.5	Tiempos de propagación	23
1.4	Arquitectura de los Dispositivos FPGAs de Xilinx	23
1.4.1	Bloques Lógicos (CLBs)	24
1.4.2	Tecnología de programación	25
1.4.3	Bloques de entrada/salida (IOB)	25
1.4.4	Descripción de las principales familias	25
1.4.4.1	Familias XC4000 y Spartan	26
1.4.4.2	Familia Virtex	29
1.4.4.3	Familia Spartan IIE	30
1.5	Arquitectura de los Dispositivos de Altera	42
1.5.1	Familia FLEX 10K	43
1.5.1.1	Introducción	43
1.5.1.2	Arquitectura de los dispositivos FLEX 10K	44
1.5.1.3	Embedded Array Block (bloques de arreglos integrados)	46
1.5.1.4	Logic Array Block (bloques de arreglos lógicos)	47
1.5.1.5	FastTrack Interconnect (pista rápida de interconexión)	52
1.5.1.6	I/O Element	54
1.5.2	LA FAMILIA APEX 20K	56
1.5.3	LA FAMILIA ACEX 1K	57
1.5.4	Familia Cyclone	59
1.5.5	Familia Stratix	61
1.6	Arquitectura de los Dispositivos FPGAs de Actel	63
1.6.1	Módulos lógicos	64
1.6.1.1	Módulos Lógicos Simples	64
1.6.1.2	Módulos Lógicos Combinacionales	65
1.6.1.3	Módulos Lógicos Secuenciales	66
1.6.1.4	Módulo Lógico de Decodificación Amplia (<i>Wide Decode Logic Module</i>)	67
1.6.1.5	Integrado SRAM de doble puerto (<i>Embedded Dual-Port SRAM</i>)	67
1.6.1.6	Módulo Lógico Secuencial Mejorada (<i>Enhanced Sequential Logic Module</i>)	68
1.6.2	Interconexión por canal (<i>Channeled Interconnect</i>)	68
1.6.3	Recursos de reloj	68
1.6.3.1	Relojes ruteados (<i>Routed Clocks</i>)	69
1.6.3.2	Arreglo de reloj especializado (<i>Dedicated Array Clock</i>)	69
1.6.3.3	Reloj Entrada/Salida Especializado (<i>Dedicated I/O Clock</i>)	69

1.6.3.4	Relojes de Cuadrantes (<i>Quad Clocks</i>)	69
1.6.4	Módulos E/S	70
1.6.4.1	Módulo de E/S Simple (<i>Simple I/O Module</i>)	70
1.6.4.2	Módulo de E/S con Latches (<i>Latched I/O Module</i>)	70
1.6.4.3	Módulo de E/S con Registros (<i>Registered I/O Module</i>)	71
1.7	Conclusión	72
Capítulo 2	MAX+plus II	73
2.1	Entorno del MAX+plus II	73
2.1.1	Introducción	73
2.1.2	Flujo de diseño	74
2.1.3	Gestión de diseños	74
2.1.4	Aplicaciones	75
2.1.5	Librerías de funciones lógicas	76
2.1.6	Proyectos en el entorno MAX+plus II	77
2.2	Tutorial MAX+plus II	78
2.2.1	Introducción	78
2.2.2	Iniciando la aplicación MAX+plus II	78
2.2.3	Herramientas de ayuda	79
2.2.4	Creación del archivo de diseño gráfico	82
2.2.4.1	Creación de un archivo nuevo	82
2.2.4.2	Especificación del nombre del proyecto	84
2.2.4.3	Dibujo del esquema del circuito	84
2.2.5	Creación del archivo de diseño de texto	92
2.2.5.1	Creación de un archivo nuevo y especificación del nombre del proyecto	92
2.2.5.2	Activación de colores de sintaxis	93
2.2.5.3	Entrada del nombre del diseño, entradas y salidas	93
2.2.5.4	Declaración de un nodo	95
2.2.5.5	Entrada de la Ecuación Booleana	96
2.2.5.6	Chequeo de errores de sintaxis y creación de un símbolo predeterminado	97
2.2.6	Creación del archivo de diseño gráfico Top-Level	97
2.2.7	Compilación del proyecto	99
2.2.7.1	Apertura de la ventana del compilador	100
2.2.7.2	Selección de la familia de dispositivos	101
2.2.7.3	Encendido del Comando de Recompilación Inteligente (<i>Smart Recompile Command</i>)	102
2.2.7.4	Encendido de la Utilidad Doctor de Diseño (<i>Design Doctor Utility</i>)	102
2.2.7.5	Selección Estilo de Síntesis de Proyecto Lógico Global (<i>Global Project Logic Synthesis Style</i>)	103
2.2.7.6	Encendido del Timing SNF Extractor	104
2.2.7.7	Ejecución del Compilador	104
2.2.8	Verificación de la jerarquía del proyecto	105
2.2.9	Verificación del Ajuste en el Floorplan Editor	106
2.2.9.1	Apertura de la ventana del Editor Floorplan	106
2.2.9.2	Comando Back-Annotate Project y Edición de Asignaciones	107
2.2.9.3	Recompilación del Proyecto	109
2.2.9.4	Vista del dispositivo	110
2.2.10	Edición del archivo Simulator Channel file	110
2.2.10.1	Creación del archivo Simulator Channel File	111
2.2.10.2	Reorganización del orden de los nodos	112
2.2.10.3	Edición de las formas de onda de los nodos de entrada	113
2.2.11	Simulación del Proyecto	114
2.2.12	Análisis de las Salidas de la Simulación	115
2.2.13	Ejemplos utilizando macro y megafunciones	116
2.2.13.1	Contador utilizando macrofunciones	116
2.2.13.2	Contador utilizando megafunciones LPM	119
2.2.13.3	Conclusiones	126

Capítulo 3	Medidor de Frecuencia y Período y Adquisidor autónomo de datos en lenguaje AHDL (<i>Altera Hardware Description Language</i>)	127
3.1	Introducción	127
3.2	Frecuencímetro	128
3.2.1	Arquitectura básica	128
3.2.2	Implementación en FPGA	130
3.2.2.1	Bloque generador de la base de tiempos	132
3.2.2.2	Bloque combinatorio	137
3.2.2.3	Contadores sincrónicos	138
3.2.2.4	Bloque de latches	139
3.2.2.5	Bloque multiplexor	140
3.2.2.6	Bloque decodificador BCD 7 segmentos	141
3.2.2.7	Bloque decodificador	141
3.2.2.8	Bloque de ubicación del punto dp.....	142
3.2.2.9	Bloque combinatorio + contadores + latches + multiplexor + decodificador BCD 7 segmentos + decodificador + ubicación del punto dp.....	144
3.2.3	Frecuencímetro en AHDL	149
3.3	Medidor de períodos	153
3.3.1	Arquitectura básica	153
3.3.2	Implementación en FPGA	154
3.3.2.1	Bloque generador de pulsos	156
3.3.2.2	Bloque divisor	159
3.3.2.3	Bloque combinatorio + contadores + latches + multiplexor + decodificador BCD 7 segmentos + decodificador	159
3.3.2.4	Bloque de ubicación del punto dp.....	160
3.3.2.5	Bloque combinatorio + contadores + latches + multiplexor + decodificador BCD 7 segmentos + decodificador + ubicación del punto dp.....	162
3.3.3	Medidor de períodos en AHDL	164
3.4	Medidor de frecuencia y período en AHDL	174
3.4.1	Implementación del Medidor de frecuencia y período.....	174
3.4.2	Verificación del rango de frecuencias y períodos a medir.....	186
3.4.2.1	Rango de frecuencias	187
3.4.2.2	Rango de períodos	187
3.4.3	Calculo de errores	188
3.4.3.1	Error de N° de pulsos contados	189
3.4.4	Implementación del prescaler en AHDL	190
3.4.5	Comunicación del medidor de Frecuencias y períodos con PC	197
3.4.5.1	Bits del puerto paralelo para comunicación	197
3.4.5.2	Programa en AHDL.....	198
3.5	Adquisidor de datos autónomo	204
3.5.1	Arquitectura	204
3.5.2	Implementación de la memoria FIFO en AHDL	206
3.5.3	Simulación del Adquisidor	212
Capítulo 4	Implementación	215
4.1	Introducción.....	215
4.2	Dispositivo EPF10K10LC84-3 y plaqueta Upx10K10	215
4.2.1	Dispositivo EPC10K10LC84-3.....	215
4.2.2	Paqueta experimental Upx10K10.....	216
4.3	Programa inicial en AHDL del proyecto Medidor de frecuencias y períodos + Adquisidor de datos autónomo.....	219
4.3.1	Pines comunes a ambos subproyectos.....	219
4.3.2	Modificaciones al Medidor de frecuencias y períodos.....	220
4.3.2.1	Pin de selección de base	220
4.3.2.2	Circuito anti-rebote para el pulsador de selección de base.....	220
4.3.3	Programa en AHDL	222
4.4	Diseño del Hardware del proyecto	228
4.4.1	Diseño general de las plaquetas	228
4.4.2	Asignación de los pines	230

4.4.3	Diseño de la plaqueta de comunicación con el puerto paralelo de la PC.....	234
4.4.3.1	Asignación de pines al puerto paralelo.....	235
4.4.3.2	Esquemático de la plaqueta 'FPGA con Puerto Paralelo de PC'.....	236
4.4.3.3	PCB de la plaqueta 'FPGA con Puerto Paralelo de PC'.....	237
4.4.3.4	Plaqueta 'FPGA con Puerto Paralelo de PC' final.....	238
4.4.4	Diseño de la plaqueta del Medidor de frecuencias y períodos.....	239
4.4.4.1	Esquemático de la plaqueta Frecuencímetro.....	240
4.4.4.2	PCB de la plaqueta Frecuencímetro.....	240
4.4.4.3	Plaqueta Frecuencímetro final.....	243
4.4.5	Diseño de la plaqueta del Adquisidor autónomo de datos.....	243
4.4.5.1	Esquemático de la plaqueta Adquisidor.....	244
4.4.5.2	PCB de la plaqueta Adquisidor.....	245
4.4.5.3	Plaqueta Adquisidor final.....	247
4.5	Diseño del Software del proyecto.....	247
4.5.1	Programa para el manejo del Medidor de Frecuencias y Períodos.....	247
4.5.2	Programa para el manejo del Adquisidor autónomo de datos.....	254
4.6	Armado y verificación del proyecto.....	261
4.6.1	Compilación del proyecto y programación de la FPGA a través del software MAX+plus II.....	261
4.6.1.1	Compilación del proyecto.....	261
4.6.1.2	Asignación de los pines a la FPGA.....	261
4.6.1.3	Programación de la FPGA.....	264
4.6.2	Armado y verificación del sistema Medidor de Frecuencias y Períodos.....	266
4.6.2.1	Armado del sistema Medidor de Frecuencia y Período.....	266
4.6.2.2	Cálculo de errores del sistema Medidor de frecuencia y período.....	267
4.6.2.3	Verificación del sistema Medidor de Frecuencias y períodos.....	268
4.6.3	Armado y verificación del sistema Adquisidor autónomo de datos.....	269
4.6.3.1	Armado del sistema Adquisidor autónomo de datos.....	269
4.6.3.2	Verificación del sistema Adquisidor autónomo de datos.....	270
4.6.3.3	Modificación de la plaqueta 'FPGA con puerto paralelo'.....	270
4.6.3.4	Optimización del sistema Adquisidor autónomo de datos.....	271
4.7	Programa final en AHDL del proyecto Medidor de frecuencias y períodos + Adquisidor de datos autónomo.....	274
A.1	Decodificadores.....	281
A.2	Demultiplexers.....	282
A.2.1	Resolución de ecuaciones lógicas mediante el uso de demultiplexers.....	283
B.1	Multiplexers.....	285
B.1.1	Multiplexers digitales.....	285
B.1.2	Generación de funciones booleanas a partir de multiplexers.....	287
C.1	Generación de funciones booleanas con ROMs.....	289
	Conclusiones.....	291
	Bibliografía.....	292

Capítulo 1 Lógica Programable

1.1 Introducción

Es válido realizar una clasificación de las formas de implementación de lógica de la siguiente manera:

1. Lógica convencional:

- Compuertas.
- Multiplexores
- Demultiplexores
- MUX + DEMUX.

2. Lógica programable por hardware

- ROM:
 - ROM
 - PROM
 - EPROM
 - EEPROM
 - Flash
- PLD:
 - SPLD:
 - PLA
 - PAL
 - GAL
 - CPLD
 - FPGA
- ASIC
- MPGA

3. Lógica programable por software.

- Microprocesador
- Microcontrolador
- DSP

El primer grupo corresponde a las aplicaciones convencionales, en donde son utilizados elementos básicos como compuertas, *flip-flops*, etc; los cuales derivan en los llamados SSI (*Small-Scale-Integration*) y MSI (*Medium-Scale-Integration*) así como también la utilización de MUXs y DEMUXs con los LSI (*Large-Scale-Integration*).

Pero estos circuitos presentan complicaciones, tal es el caso de un pequeño cambio en la función a implementar lo cual implica en la mayoría de los casos rediseñar el circuito. Otro problema es la testeabilidad del mismo cuando éste es complejo, como así también la gran variedad de stock de circuitos impresos, integrados, etc que se necesita para llevar a cabo el mismo; y el tamaño final de dicho circuito lógico.

De aquí surge la necesidad de integrar una enorme cantidad de transistores y compuertas en un único chip y con esto llegar a soluciones de tipo universal, y por lo tanto los dispositivos lógicos programables.

En éste trabajo trataremos solo la lógica programable por hardware, haciendo hincapié en las familias CPLDs y FPGAs.

1.2 ROM

1.2.1 ROM

Como sabemos un multiplexer con n variables de entradas puede ser usado para generar funciones de orden m (m mayor que n), dividiendo la función original en 2^n subfunciones de $k=(m-n)$ variables cada una (véase Apéndice B). Por otra parte, también sabemos que un demultiplexer con k líneas de selección y 2^k líneas de salida permite generar las 2^n funciones mencionadas, usando para cada una de ellas una línea de OR/AND cableado (véase Apéndice A).

El empleo conjunto de éstas dos técnicas define lo que se conoce como memoria ROM (*Read Only Memory*).

Como vemos en la figura 1.1, en una ROM las variables de entrada (líneas de dirección, *address line*) seleccionan una dada fila (palabra), a través del demultiplexer también llamado matriz de intersección, y en base a los valores de la matriz de fusibles (matriz unión) de esa fila generan en las distintas columnas las funciones que ingresan al multiplexer, que en base a las variables restantes selecciona el valor correspondiente a la función para el mintermino elegido, el que aparece en la salida de la ROM llamada línea de datos (*data line*) (véase Apéndice C).

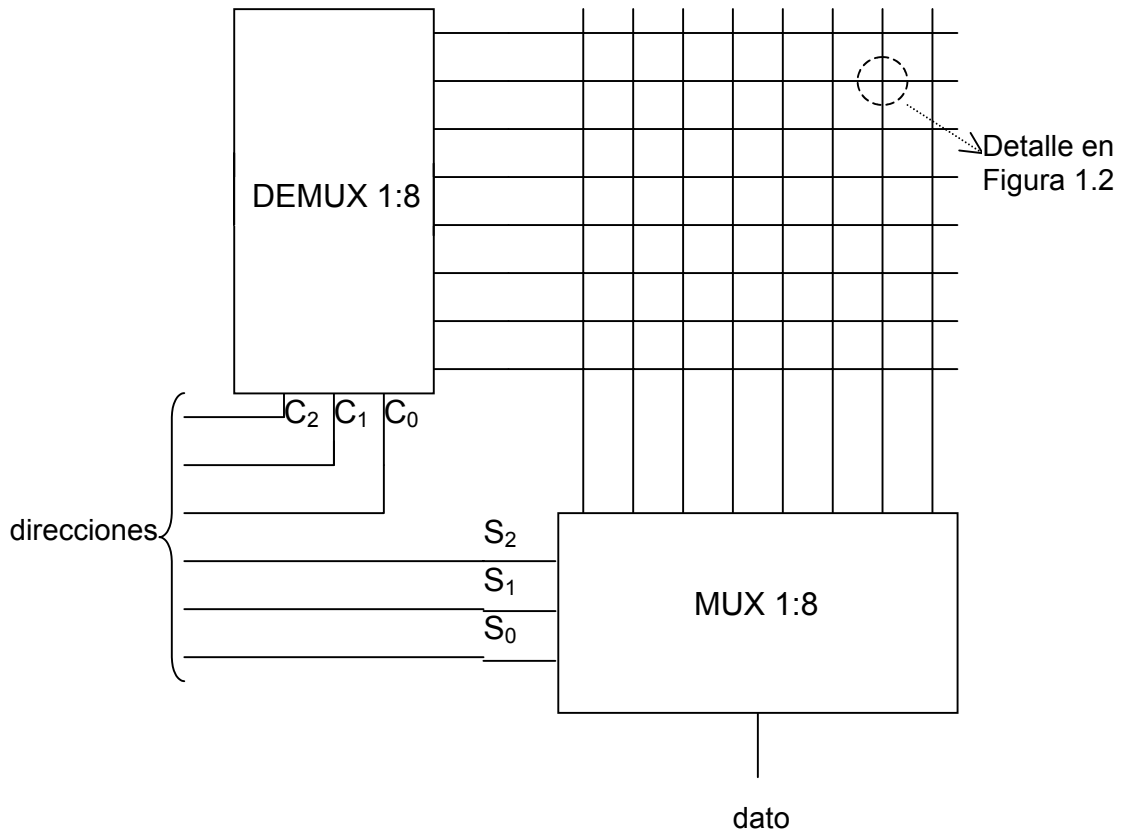


Figura 1.1: ROM

Por lo tanto, en una memoria con m líneas de dirección se podrá programar en forma independiente y luego leer al valor de 2^m posibles minterminos (o *bits*). Además de éste esquema, es habitual colocar en un mismo chip un único demultiplexer, varios multiplexers con líneas de control comunes, y varias matrices de fusibles, de modo tal de sintetizar simultáneamente varias funciones independientes de las mismas variables de entrada.

En una ROM, los datos quedan permanentemente almacenados. Esto se logra colocando o no diodos, mediante métodos fotográficos, al construir el circuito integrado. Otra posibilidad es construir el circuito con todos los diodos de la matriz; luego se aplica un voltaje adecuado de programación que rompe aquellos fusibles (figura 1.2) asociados a las filas requeridas en cada columna. El voltaje de programación se introduce por las líneas de salida.(véase Apéndice A, ítem A.2.1 ó Apéndice C).

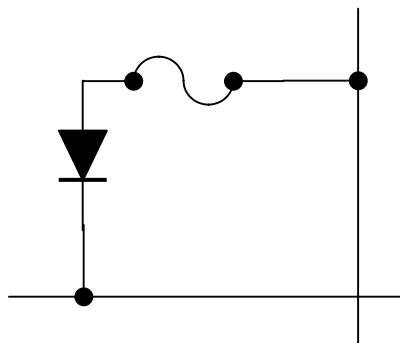


Figura 1.2: Detalle intersección de matriz

En general el empleo de ROMs permite resolver en forma económica cualquier tipo de problema lógico con un número apreciable de variables de entrada (típicamente hasta 12 o 14) y conviene emplearlas cuando se requieren muchas combinaciones de las entradas (por ejemplo, en cambiadores de código); pero presenta como principal desventaja su velocidad. Debido al uso del DEMUX, la lógica cableada, y el MUX tenemos al menos siete niveles lógicos que atravesar además de capacidades parásitas enormes debido al tamaño de las matrices de fusibles; por cada entrada adicional se duplica el tamaño de la ROM, y no puede emplearse minimización debida a condiciones superfluas.

1.2.2 PROM, EPROM, EEPROM, Flash

El esquema en que se puede programar, pero sólo una vez, a través de la ruptura de algunos fusibles del arreglo se clasificó como ROM. En las memorias PROM, puede colocarse varias veces información en la memoria. Se emplean transistores MOSFET con la base aislada, en lugar de diodos con fusibles. Mediante el efecto de avalancha se deposita carga en la base, quedando una baja impedancia. Para liberar la carga de la compuerta se aplica luz ultravioleta a través de una pequeña ventana que posee la PROM, dejándola con puros unos. Suele denominarse EPROM a las memorias de este tipo, donde la "E" simboliza "Erasable" (borrable); mientras que aquellas que son borrables eléctricamente EEPROM (*Electrically Erasable Programmable Read Only Memory*). Las Flash EPROM son similares a las EPROM con la diferencia en que el borrado debe ser total o por sectores.

Si vemos un esquema de la PROM con AND y OR cableado tal como lo muestra la figura 1.3, vemos que la matriz AND (o intersección) es fija generando todos los minterminos posibles (2^n) con las n variables de entrada ($n = 4$ para este caso y por lo tanto 16 minterminos), mientras que la matriz OR (o unión) es programable .

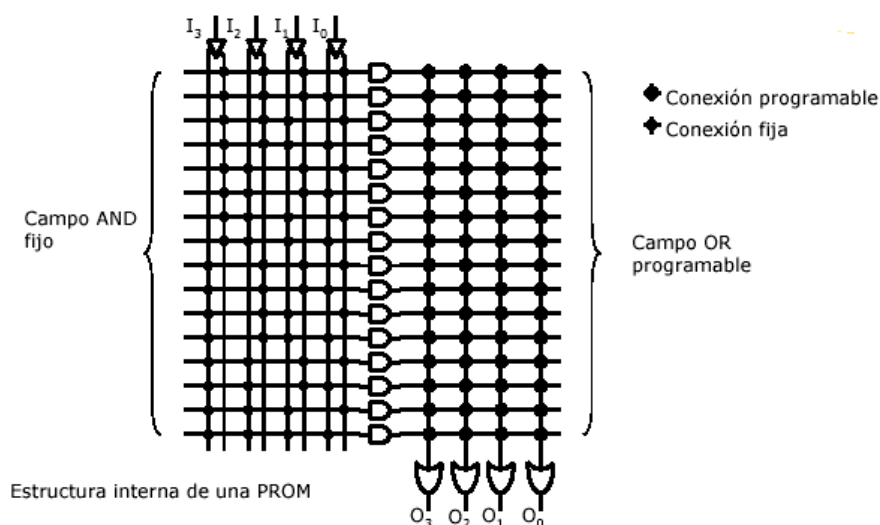


Figura 1.3: PROM

1.3 PLD

Un dispositivo lógico programable, o PLD (*Programmable Logic Device*), es un dispositivo cuyas características pueden ser modificadas y almacenadas mediante programación. Además los PLDs tienen una estructura circuital regular y flexible y pueden ser programados mediante una estructura de interruptores.

La figura 1.4 compara los PLDs de acuerdo a la cantidad de compuertas equivalentes.

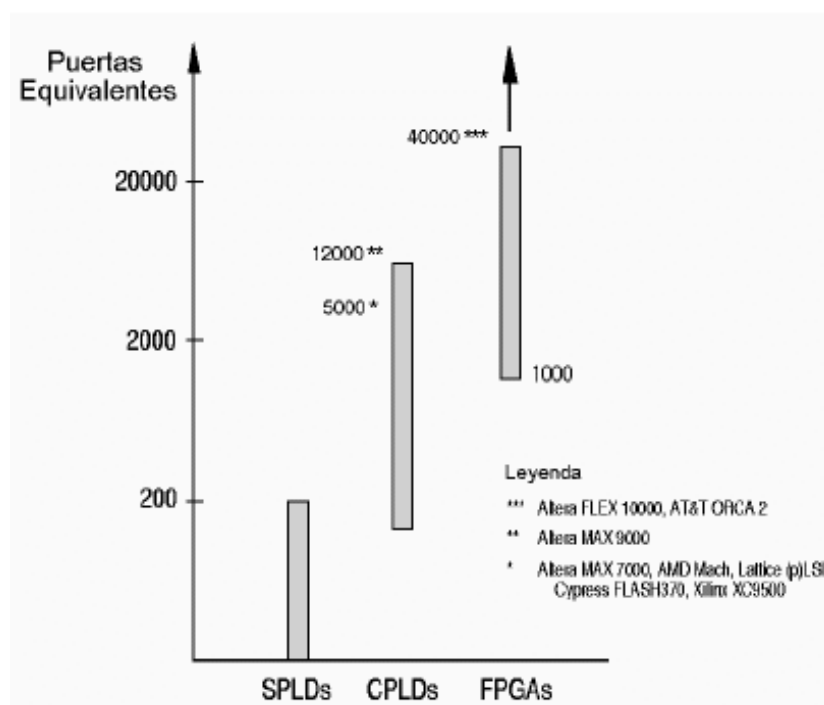


Figura 1.4: Clasificación de los PLDs

La fabricación de dispositivos de lógica programable se basa en los siguientes dos conceptos:

- **Funcionalidad completa**
La cual se fundamenta en el hecho de que cualquier función lógica se puede realizar mediante la suma de productos.
- **Celdas de funciones universales**
También denominadas generadores de funciones, son bloques lógicos configurados para procesar cualquier función lógica, similares en su funcionamiento a una memoria. En estas celdas se almacenan los datos de salida del circuito combinacional en vez de implementar físicamente la ecuación booleana.

1.3.1 SPLD (Simple PLD)

La mayoría de los PLDs (los SPLDs) están formados por matrices de conexiones: una matriz de compuertas AND, y una matriz de compuertas OR y algunos, además, con registros. Con estos recursos se implementan las funciones lógicas deseadas mediante un software especial y un programador. Las matrices pueden ser fijas o programables.

Una matriz de conexiones es una red de conductores distribuidos en filas y columnas con un fusible en cada punto de intersección, mediante el cual se seleccionan cuales entradas serán conectadas a las compuertas AND/OR.

1.3.1.1 PAL

Si analizamos estadísticamente la unión (o intersección en el caso de la 2^o forma canónica) de cuantos implicantes suele ser necesarios para sintetizar la gran mayoría de las funciones lógicas, veremos que dicho valor no supera los ocho implicantes primos esenciales. Por lo tanto si restringimos la matriz de intersección (matriz AND) a la generación de a lo sumo siete u ocho implicantes primos esenciales de las variables de entrada, es decir no mas de ocho compuertas AND por salida con sus entradas conectadas a las n variables con sus respectivas negaciones; y dimensionamos la matriz unión (matriz OR) solo para estos ocho implicantes reduciremos considerablemente el tamaño del chip, con lo cual obtendremos una disminución de las capacidades parásitas y por lo tanto un aumento en la velocidad de conmutación.

Éste tipo de circuito se conoce como circuito tipo **PAL** (*Programmable Array logic*), en el cual, como describimos anteriormente, la matriz de conexiones AND es programable mientras que la matriz OR es no programable

En la figura 1.5 vemos un ejemplo de una estructura interna de una PAL de 6 variables, 4 salidas con 4 implicantes primos por salida

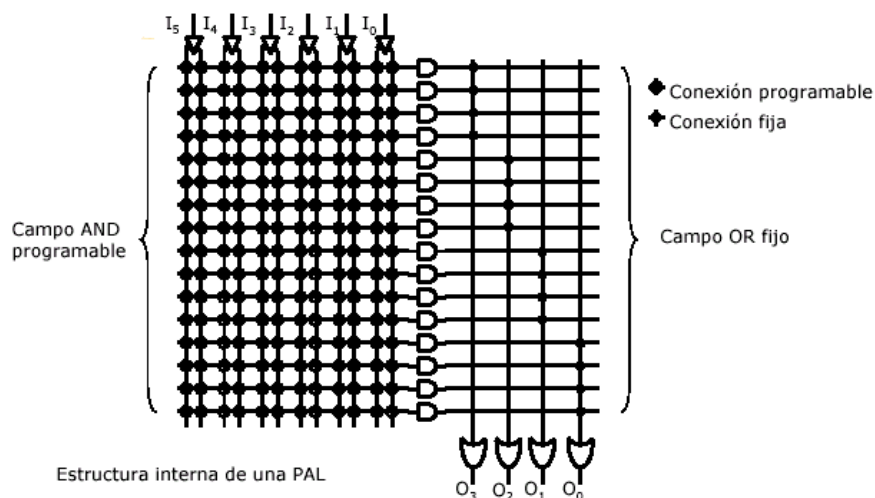


Figura 1.5: PAL

A continuación podemos ver en la Figura 1.6, la arquitectura de una de la PALs de primera generación, la 16L8 (16 entradas, L de Lógico y 8 salidas) de Monolithic Memories:

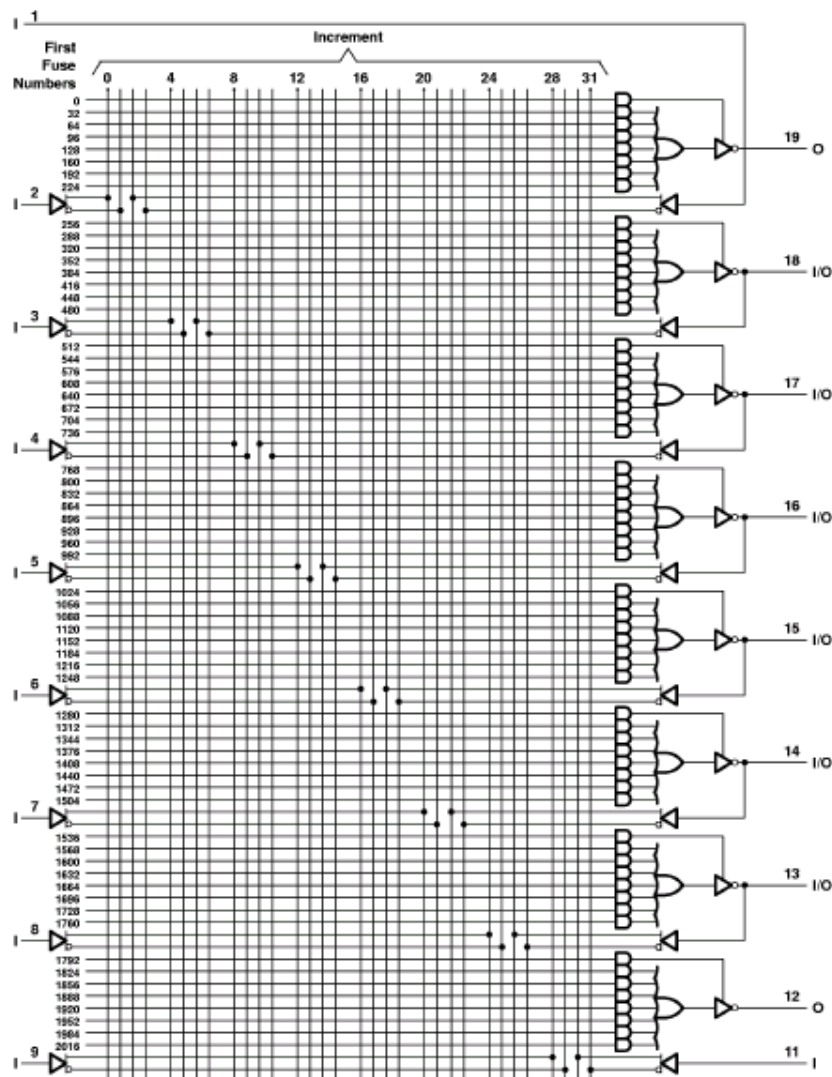


Figura 1.6: PAL 16L8

Si tomamos solo una de las ocho celdas (*macrocells*)

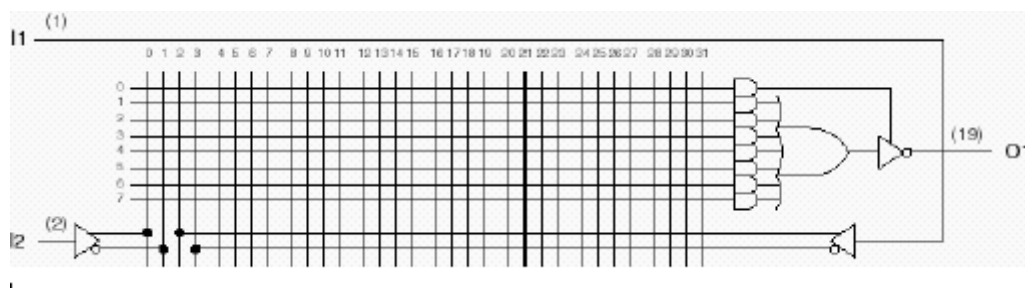


Figura 1.7: Macrocells de 16L8

Podemos ver en la figura 1.7 los siete implicantes nombrados anteriormente, con algunas otras características que detallaremos a continuación:

- cada variable de entrada es distribuida negada y sin negar en la matriz de intersección, mediante los literales (columnas).
- todos los literales cruzan líneas horizontales donde se pueden realizar el AND cableado de los literales que se elijan y con ello generar en cada fila el implicante primo que se desee.
- siete filas son unidas mediante una compuerta OR para realizar la suma de los implicantes
- la señal de salida es negada por cuestiones de velocidad, ya que es mas rápida una compuerta NOR que una OR, aunque existe la PAL 16H8, donde la compuerta OR sale sin negar.
- en algunas *macrocells*, el valor de la salida se realimenta negado y sin negar a todo el circuito, lo cual permite la conexión en cascada de lógica, para la síntesis de funciones mas complejas. Esto además facilita el diseño de circuitos realimentados para la elaboración de celdas de memorias.
- se pueden controlar las salidas individualmente a través de las compuertas TRI-STATE de cada salida mediante un implicante primo para cada una, y con esto poder seleccionar la salida como entrada.

Con lo descrito anteriormente, la PAL 16L8 consta de:

- un total de 16 entradas
- un total de 8 salidas con 7 ANDs por salida
- 10 entradas primarias
- 1 compuerta AND por salida para deshabilitar el TRI-STATE.
- 6 salidas disponible como entradas.

Una variación sobre los circuitos PALs básicos es la incorporación de elementos de memoria para que puedan generar maquinas secuenciales. Un típico ejemplo siguiendo con las PALs de Monolithic Memories es la 16R8 (16 entradas, R de Registro y 8 salidas) la cual incorpora registros de almacenamiento sensibles a la señal de reloj (*flip-flops edge triggered*) a la salida de la compuerta OR. Con esto puede ser empleado para realizar simultáneamente las funciones de almacenamiento de estados (bloque de memoria) y calculo del próximo estado (bloque combinatorio) que caracterizan una maquina secuencial tipo MOORE.

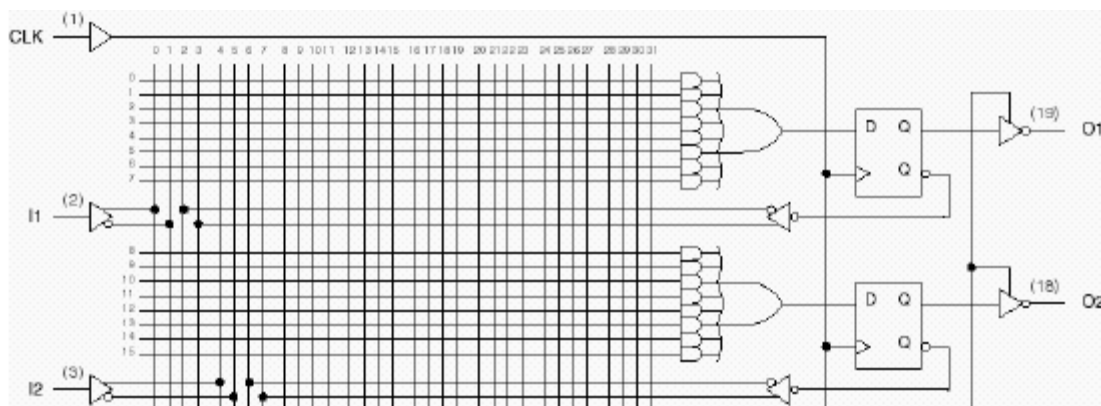


Figura 1.8: PAL 16R8

Podemos observar en la figura 1.8 que ya no tenemos el control individual sobre el TRI-STATE de cada salida, sino que tenemos un control general sobre todos, y el implicante primo anteriormente utilizado a tal fin, ahora se suma a los otros siete, permitiendo la suma de un total de ocho.

Los circuitos tipo PAL son no reprogramables ya que la síntesis de las ecuaciones lógicas se realiza mediante quema de fusibles en cada punto de intersección de los pines de entrada con las compuertas.

1.3.1.2 PLA

En una **PAL** se generan a lo sumo ocho implicantes primos esenciales de las variables de entrada a través de matriz de intersección (matriz AND) programable, los cuales entran a una matriz unión (matriz OR) fija.

En una **PROM**, todos los minterminos son generados por un decodificador fijo, luego los minterminos requeridos para producir la función, son combinados mediante otro arreglo programable con el papel de un OR.

Sin embargo, para ciertos diseños lógicos, sólo se usa una pequeña fracción de los minterminos generados por el decodificador.

La estructura arreglos lógicos programables (**PLA**) es una combinación entre los circuitos PAL y los PROM. Contiene un decodificador programable (arreglo lógico AND), y un arreglo lógico OR programable, es decir tanto la matriz de conexiones AND como la matriz OR son programables. De esta forma pueden implementarse funciones a partir de términos de mayor nivel que los minterminos; es decir, a partir de los implicantes primos. Son llamadas también *FPLA (Field Programmable Logic Array)* ya que son programadas por el usuario y no por el fabricante.

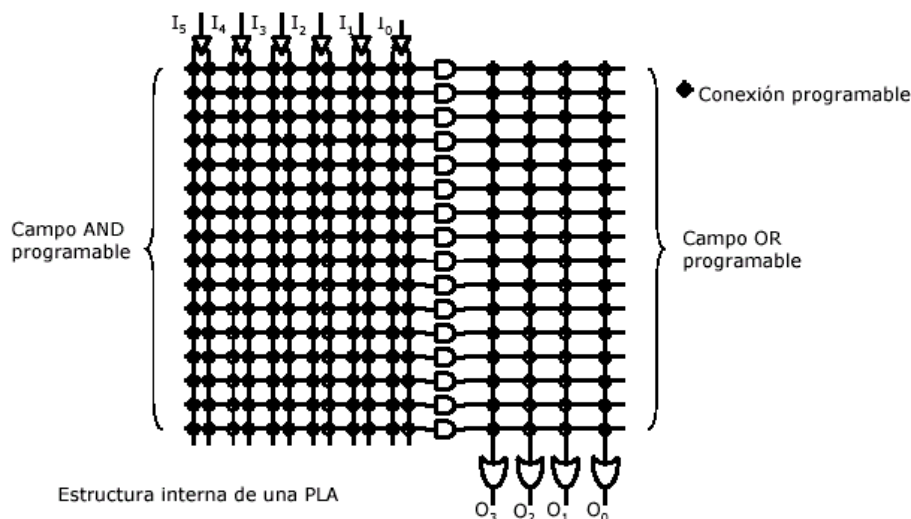


Figura 1.9: PLA

1.3.1.3 GAL

Las estructuras GAL son, básicamente estructuras CMOS PAL, son básicamente la misma idea que la PAL pero en vez de estar formada por una red de conductores ordenados en filas y columnas en las que en cada punto de intersección hay un fusible, el fusible se reemplaza por una celda CMOS eléctricamente borrable (EECMOS). Mediante la programación se activa o desactiva cada celda EECMOS y se puede aplicar cualquier combinación de variables de entrada, o sus complementos, a una compuerta AND para generar cualquier operación producto que se desee.

En la figura 1.10 vemos el diagrama en bloques de la GAL 22V10 (22 entradas, V de Versátil y 10 salidas)

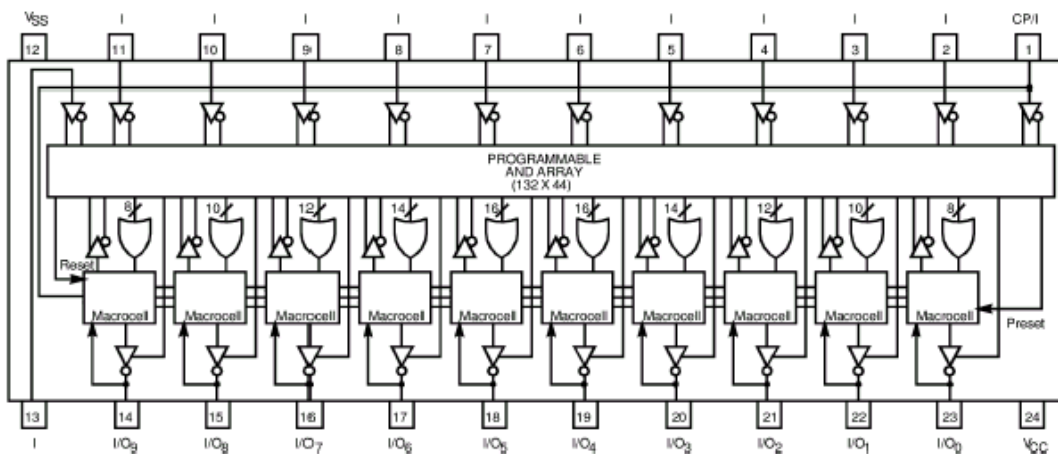


Figura 1.10: GAL 22V10

El multiplexer de la macrocelda de salida (figura 1.11) permite salidas en alto o en bajo tanto combinacional como con registro (salida versátil), tal como lo muestra la figura 1.12:

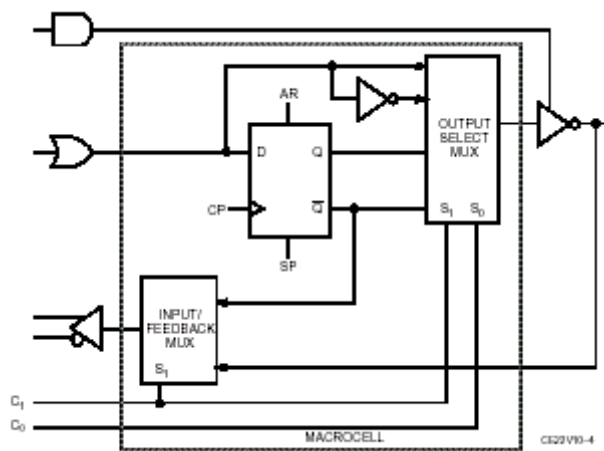


Figura 1.11: Macrocelda de GAL22V10

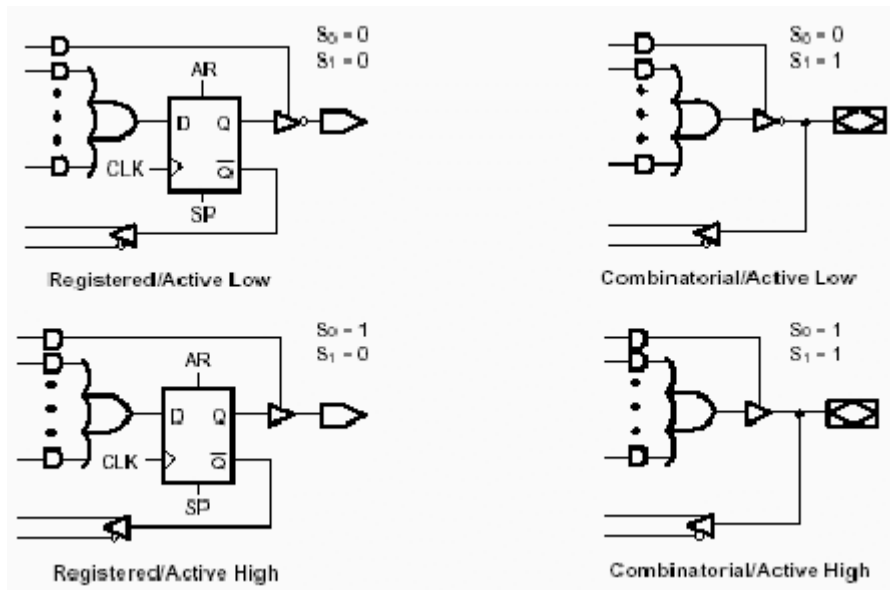


Figura 1.12: Posibles configuraciones de la macrocelda de salida

Otra innovación de la 22V10 es la distribución variable de términos producto en cada macrocelda. En las zonas superior e inferior de la estructura se asignan 8 términos producto por macrocelda, mientras que en el centro se asignan 16 términos producto por macrocelda. Disponemos además de 22 entradas y 10 salidas.

Una breve referencia a los tiempos de propagación: el siguiente diagrama de tiempos da cuenta de los parámetros temporales básicos

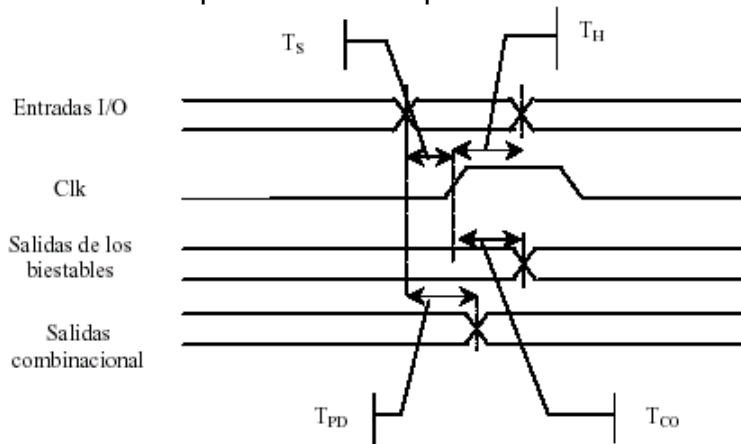


Figura 1.13: Tiempos de propagación en la 22V10

donde

Parámetro	Descripción	Min.	Max.
T_{PD}	Retardo de propagación		4 ns
T_S	Tiempo de "setup"	2.5 ns	
T_H	Tiempo de mantenimiento	0	
T_{CO}	Retardo reloj-salida		3.5 ns
T_{CO2}	Retardo reloj-salida a través de la matriz lógica		7 ns
T_{SCS}	Retardo de reloj		5.5ns

Tabla 1.1: Tiempos de propagación en la 22V10

1.3.2 CPLD (Complex PLD)

Un CPLD (*Complex Programmable Logic Device*) extiende el concepto de un PLD a un nivel de integración superior; esto es, se dispone de mayor número de puertas y de entradas/salidas en un circuito programable (con lo que se disminuye el tamaño del diseño, el consumo y el precio). En vez de hacer estos circuitos con mayor número de términos producto por macrocelda, o de mayor número de entradas/salidas, cada CPLD contiene bloques lógicos, cada uno de ellos similar a una estructura PAL o GAL. Estos bloques lógicos se comunican entre sí utilizando una matriz programable de interconexiones lo cual hace más eficiente el uso del silicio, conduciendo a un mejor desempeño y un menor costo. La figura 1.14 detalla un modelo genérico de una CPLD. Las tecnologías de programación son las mismas que las que utilizan las SPLDs, esto es, fusible, EPROM, EEPROM y Flash. A continuación se explican brevemente las principales características de la arquitectura de un CPLD.

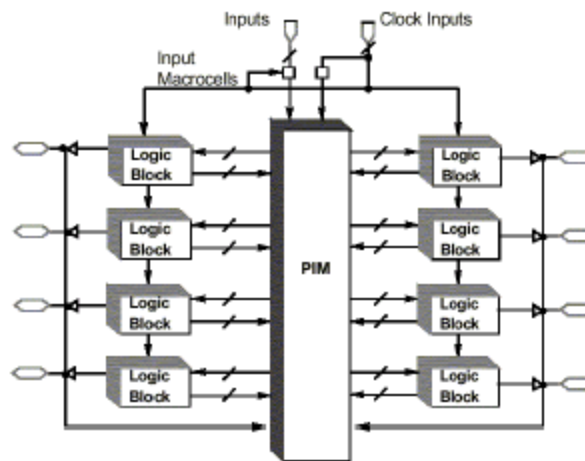


Figura 1.14: Arquitectura básica de un CPLD

1.3.2.1 Matriz de Interconexiones Programables

La Matriz de Interconexiones Programables (PIM, *Programmable Interconnect Matrix*) permiten unir los pines de entrada/salida a las entradas del bloque lógico, o las salidas del bloque lógico a las entradas de otro bloque lógico o inclusive a las entradas del mismo bloque; con una estructura de interconexión continua. La mayoría de los CPLDs usan una de dos configuraciones para esta matriz:

- interconexión mediante arreglo
- interconexión mediante multiplexores

La primera se basa en una matriz de filas y columnas con una celda programable de conexión en cada intersección. Al igual que en el GAL esta celda puede ser activada para conectar/desconectar la correspondiente fila y columna. Esta configuración permite una total interconexión entre las entradas y salidas del dispositivo o bloques lógicos. Sin embargo, estas ventajas

provocan que disminuya el desempeño del dispositivo además de aumentar el consumo de energía y el tamaño del componente.

En la interconexión mediante multiplexores, existe un multiplexor por cada entrada al bloque lógico. Las vías de interconexión programables son conectadas a las entradas de un número fijo de multiplexores por cada bloque lógico. Las líneas de selección de estos multiplexores son programadas para permitir que sea seleccionada únicamente una vía de la matriz de interconexión por cada multiplexor la cual se propagara a hacia el bloque lógico. Cabe mencionar que no todas las vías son conectadas a las entradas de cada multiplexor. La rutabilidad se incrementa usando multiplexores de mayor tamaño, permitiendo que cualquier combinación de señales de la matriz de interconexión pueda ser enlazada hacia cualquier bloque lógico. Sin embargo, el uso de grandes multiplexores incrementa el tamaño de dispositivo y reduce su desempeño.

1.3.2.2 Bloques Lógicos

Un bloque lógico es similar a un SPLD, cada uno posee un arreglo de compuertas AND y OR en forma de suma de productos, una configuración para la distribución de estas sumas de productos, y macroceldas. El tamaño del bloque lógico es una medida de la capacidad del CPLD, ya que de esto depende el tamaño de la función booleana que pueda ser implementada dentro del bloque. Los bloques lógicos usualmente tienen de 4 a 20 macroceldas.

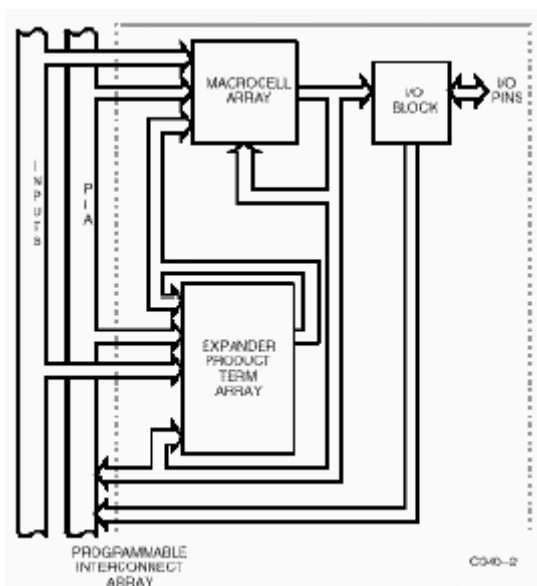


Figura 1.15: Estructura de un Bloque Lógico en dispositivos de las familias MAX340 y MAX5000

1.3.2.3 Distribución de Productos

Existen pequeñas diferencias en cuanto a las matrices de productos, esto dependerá del CPLD y del fabricante. Obviamente el tamaño de las sumas

sigue siendo el factor más importante para la implementación de funciones booleanas. Cada fabricante distribuye los productos de diferente forma. En la familia MAX de CPLDs que fue desarrollada por Cypress Semiconductor junto con Altera Corporation, siendo los primeros en sacar al mercado una familia de CPLDs. Altera la llamó MAX5000 y Cypress por su parte la clasificó como MAX340; la distribución de productos no es igual a la de un PLD. En un dispositivo como el 22V10 tenemos que la suma de productos es fija por cada macrocelda (8, 10, 12, 14 o 16), en la familia MAX se colocan 4 productos por macrocelda los cuales pueden ser compartidos con otras macroceldas. Cuando un producto puede ser únicamente utilizado por una macrocelda se le conoce como **termino - producto dirigido** (*product-term steering*), y cuando estos pueden ser utilizados por otras macroceldas se le llama **termino - producto compartido** (*product-term sharing*). Mediante estos productos compartidos se mejora la utilización del dispositivo, sin embargo, esto produce un retardo adicional al tener que retroalimentar un producto hacia otra macrocelda y con esto disminuye la velocidad de trabajo del circuito. La forma en que son distribuidos los productos repercute en la flexibilidad que proporciona el dispositivo para el diseñador. Además, que estos esquemas proporcionan también flexibilidad para los algoritmos del programa de síntesis que es el que finalmente selecciona la mejor forma en que deben ser distribuidas las funciones booleanas en el dispositivo.

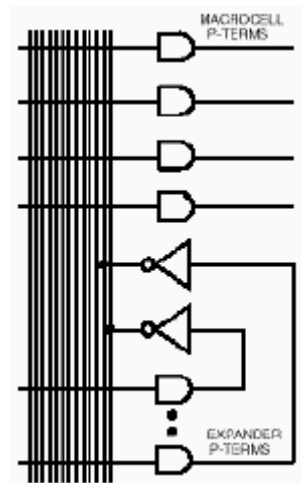


Figura 1.16: Distribución de Productos en dispositivos de las familias MAX340 y MAX5000

1.3.2.4 Macrocelda

Las macroceldas de un CPLD son similares a las de un PLD. Estas también están provistas con registros, control de polaridad, y *buffers* para salidas en alta impedancia. Por lo general un CPLD tiene macroceldas de entrada/salida, macroceldas de entrada y macroceldas internas u ocultas (*buried macrocells*), en tanto que un 22V10 tiene solamente macroceldas de entrada/salida. Una macrocelda interna es similar a una macrocelda de entrada/salida, sólo que esta no puede ser conectada directamente a un pin de salida. La salida de una macrocelda interna va directamente a la matriz de interconexión programable.

Por lo que es posible manejar ecuaciones y almacenar el valor de salida de estas internamente utilizando los registros de estas macroceldas. Las macroceldas de entrada, como la que se muestra en la siguiente figura 1.17, son utilizadas para proporcionar entradas adicionales para las funciones booleanas. En el diagrama se muestra la macrocelda de entrada de la familia FLASH 370. En general las macroceldas de entrada incrementan la eficiencia del dispositivo al ofrecer algunos registros adicionales con los que se pueden almacenar el valor del pin de entrada, lo cual puede ser útil al momento de obtener las funciones booleanas.

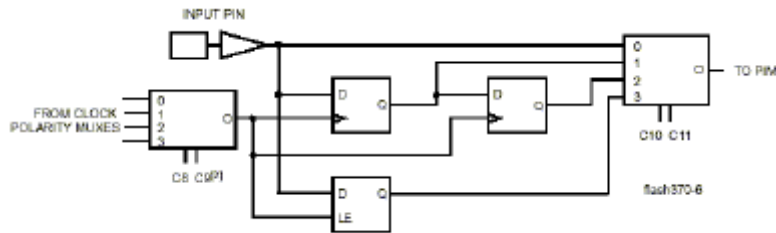


Figura 1.17: Macrocelda de entrada en dispositivos de la familia FLASH 370

En la figura 1.18 se muestra la estructura básica de las macroceldas de entrada/salida y macroceldas ocultas para dispositivos de la familia FLASH 370 de Cypress Semiconductors.

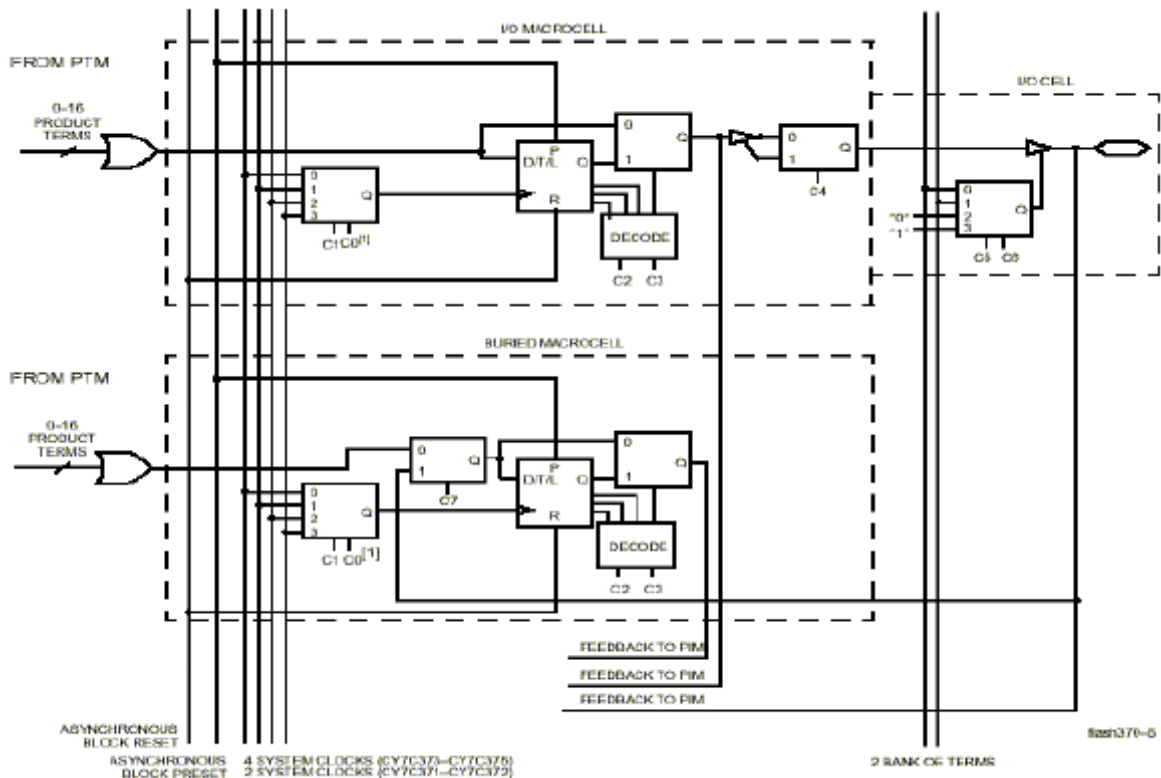


Figura 1.18: Macroceldas de entrada/salida y macroceldas ocultas en dispositivos de la familia FLASH 370

1.3.2.5 Celda de entrada/salida

En la figura 1.18 se puede apreciar una celda de entrada/salida, que bien podría considerarse parte del bloque lógico, pero no necesariamente tienen que estar a la salida de un bloque lógico. La función de una celda de entrada/salida es permitir el paso de una señal hacia el interior o hacia el exterior del dispositivo. Dependiendo del fabricante y de la arquitectura del CPLD estas celdas pueden ser consideradas o no consideradas parte del bloque lógico.

1.3.2.6 Tiempos de propagación

Las especificaciones en cuanto a los tiempos de propagación son parecidas que las descritas para las 22V10: Retardo de propagación, de setup, de reloj a salida, y de registro a registro (ver el diagrama de tiempos asociado a la GAL, figura 1.13). En cualquier caso los retardos son más sencillos de predecir que para estructuras del tipo FPGA

1.3.2.7 Otras características de las CPLD

Además de los recursos lógicos, mecanismos de ruteo, esquemas de distribución de términos productos y modelos de retardos temporales, existen otras características propias de los mismos.

Programabilidad en el sistema (*In System Programmability* o ISP): Es la capacidad de programar el dispositivo mientras se encuentra en la placa de circuito impreso en el que opera. De esta forma se reduce significativamente el costo de fabricación, ya que el dispositivo se manipula menos en fábrica, no es necesario llevar un inventario complejo, etc.

Re-programabilidad en el sistema (*In System Reprogrammability* o ISR): Es la capacidad de reprogramar el CPLD mientras se encuentra en la placa de circuito impreso. Se puede utilizar para el prototipo, cargar actualizaciones o incluso alterar la función del dispositivo mientras se encuentra en operación.

Boundary Scan: las especificaciones del grupo JTAG (*Joint Test Action Group*), que se han convertido en la norma IEEE 1149.1, definen un método para comprobar la funcionalidad de un dispositivo y las conexiones a otros Circuitos integrados. Desplazando datos a través de las celdas "boundary scan", se pueden comprobar las conexiones con otros dispositivos y se pueden aplicar vectores de test a la lógica interna.

JTAG es una metodología que se puede utilizar para comprobar (testear) y asegurar la calidad o para depurar código. Para limitar el número de vectores de test necesarios, JTAG también especifica un modo BIST (*built-in self-test*): Un dispositivo en este modo genera un conjunto de vectores de test pseudo-aleatorio como vectores de estímulo, compara las salidas con los resultados esperado e informa de los posibles errores.

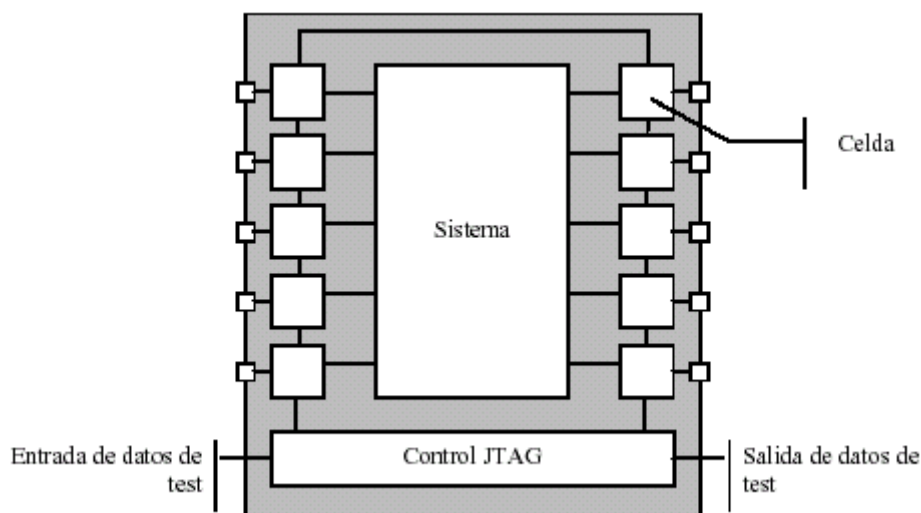


Figura 1.19: Dispositivo en la placa de circuito impreso

1.3.3 FPGAs

Los FPGA (*Field Programmable Gate Array*) son circuitos lógicos programables directamente por el usuario, lo cual requiere de herramientas de costo relativamente bajo, como lo son el *software* de desarrollo y el dispositivo grabador. La grabación o programación de uno de estos dispositivos se puede llevar a cabo en milisegundos.

Los FPGA son muy utilizados por fabricantes que producen tecnología a baja escala, como por ejemplo diseñadores de equipos de propósito específico, los cuales no pueden justificar la producción de ASICs por los bajos volúmenes de dispositivos que venden. Los FPGAs tienen una funcionalidad similar, a costos menores y con una velocidad ligeramente menor. También los FPGAs se utilizan como prototipos, los cuales se pueden depurar y permiten refinar el diseño. Con el *software* de diseño se puede simular en *hardware* antes de mandar a fabricar el ASIC correspondiente

1.3.3.1 Arquitectura general de un FPGA

Un FPGA consiste en arreglos de varios bloques programables (bloques lógicos) los cuales están interconectados entre sí y con celdas de entrada/salida mediante canales de conexión verticales y horizontales, tal como muestra la figura 1.20. En general, se puede decir que posee una estructura bastante regular, aunque el bloque lógico y la arquitectura de rutado varía de un fabricante a otro.

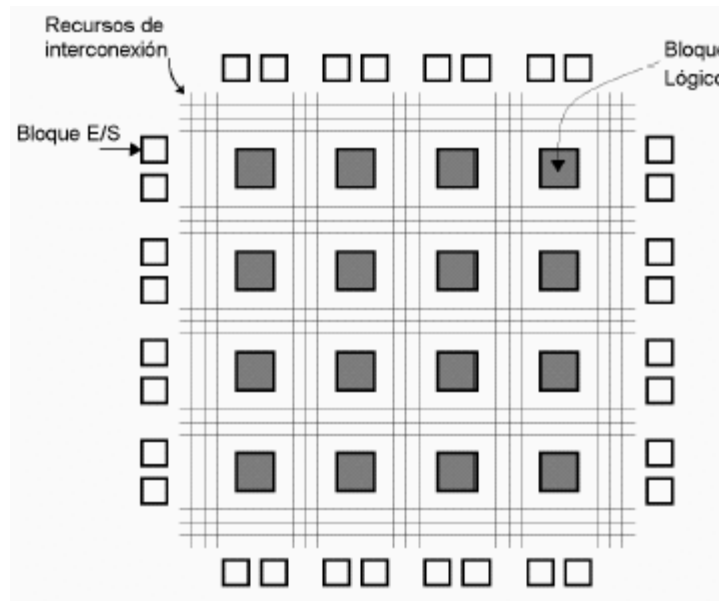


Figura 1.20: Arquitectura básica de un FPGA

La estructura de un FPGA, comparada con la de una CPLD, es mucho más regular, y se encuentra más orientada a diseños que manejan mayores transferencias de datos y registros, en tanto que las CPLD implementan más eficientemente diseños con una parte combinacional más intensa. La figura 1.21 muestra a primera vista la diferente estructura de ambos dispositivos.

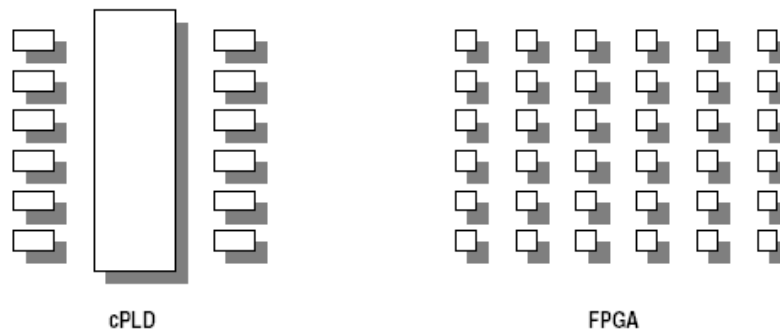


Figura 1.21: Esquema de bloques de la arquitectura interna de una CPLD y una FPGA

Como hemos visto, la arquitectura de una CPLD es una agrupación de PALs o GALs, interconectadas entre sí. Cada bloque lógico tiene su propia parte combinacional que permite realizar un gran número de funciones lógicas programables, más un biestable asociado al pin de entrada/salida en caso de encontrarse habilitado. La arquitectura de la FPGA cuenta también con un bloque lógico con una parte combinacional y una parte secuencial. La parte combinacional es mucho más simple que la de una de las PAL interna de una CPLD. La parte secuencial posee uno o dos biestables, que no están generalmente asociados a un pin de entrada salida, pues los bloques lógicos se distribuyen regularmente en todo el dispositivo.

1.3.3.2 Bloques Lógicos

El bloque lógico consta de una parte combinacional, que permite implementar funciones lógicas booleanas, más una parte secuencial que permite sincronizar la salida con una señal de reloj externa e implementar registros.

La parte combinacional varía de un fabricante a otro. A continuación, explicaremos dos de ellas, representativas porque poseen unas prestaciones opuestas.

- **Bloque lógico basado en LUT (*look-up table*):** Una LUT es un componente de células de memoria SRAM que almacena una tabla de verdad. Las direcciones de las células son las entradas de la función lógica que queremos implementar, y en cada celda de memoria se guardan el resultado para una de las combinaciones de las entradas. En una LUT de $n \times 1$ es posible implementar cualquier función lógica de n entradas. Veamos un ejemplo. Supongamos que queremos implementar la función lógica de tres entradas $f = X \cdot Y + Z$. La figura 1.22 indica cuál debe ser el contenido de una LUT de 8×1 .

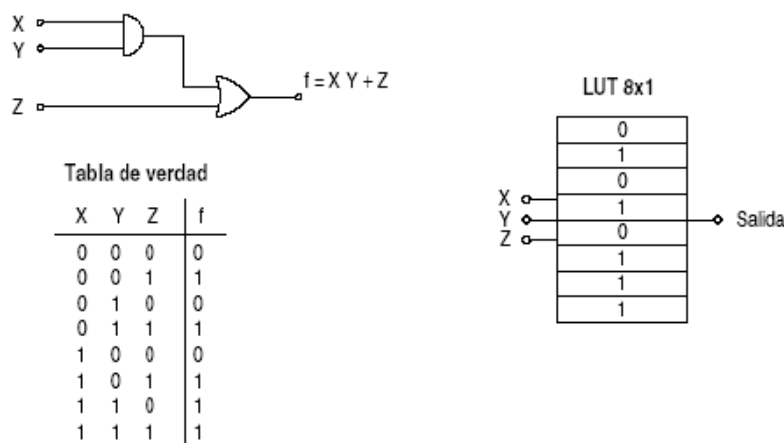


Figura 1.22: Ejemplo de implementación de una función lógica de tres entradas en una LUT de 8×1

- **Bloque lógico basado en multiplexores:** El bloque lógico basado en multiplexores, como el de la figura 1.23, se caracteriza porque requiere mucha menos lógica que el anterior basado en una LUT, y, en consecuencia, ocupa mucha menos área. De este modo, se pueden implementar mayor número de bloques lógicos en el mismo espacio, o, para el mismo número de bloques, disponer de más espacio para incrementar los recursos de rutado. Como contrapartida, no se puede implementar cualquier función lógica de n entradas, como ocurría con las LUTs. En caso necesario, esta función lógica hay que repartirla entre varios bloques lógicos.

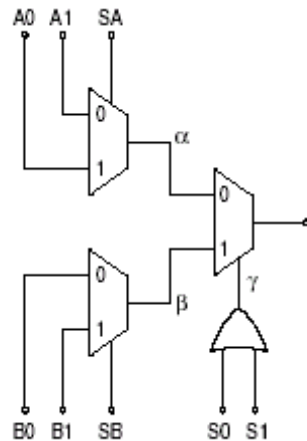


Figura 1.23: Bloque lógico basado en multiplexores

La salida f la podemos expresar en función de los valores intermedios α , β , γ , los cuales los podemos escribir en función de las entradas de la forma:

$$\alpha = \overline{SA} A_1 + SA A_0 \quad (1)$$

$$\beta = \overline{SB} B_0 + SB B_1 \quad (2)$$

$$\gamma = S_0 + S_1 \quad (3)$$

De donde resulta $f = \overline{\gamma} \alpha + \gamma \beta$. Evidentemente, cualquier función lógica que se desee implementar debe respetar las ecuaciones anteriormente escritas. Si se quiere realizar la misma función lógica del ejemplo anterior, $f = XY + Z$, será necesario elegir adecuadamente la asignación de las entradas para tener el resultado requerido. Si tomamos $A_0 = Y$, $A_1 = 0$ y $SA = X$ en la ecuación 1, resulta $\alpha = XY$. Tomando en la ecuación 2 $B_0 = 1$, $B_1 = 0$ y $SB = 0$, resulta que $\beta = 1$. Finalmente, sustituyendo $S_0 = Z$ y $S_1 = 0$ en la ecuación 3 se obtiene $\gamma = Z$. A partir de estos resultados, la salida $f = \overline{Z} XY + Z = XY + Z$. A través de otras asignaciones se puede llegar al mismo resultado. Lo que está claro es que no podemos efectuar cualquier función lógica de 8 entradas, pues tenemos las restricciones de las ecuaciones lógicas.

Como se dijo anteriormente, la estructura de los bloques programables y las formas en que estas pueden ser interconectadas, tanto salidas como entradas de los bloques, varía de acuerdo al fabricante. En general un bloque programable tiene menos funcionalidad que la combinación de sumas de productos y macroceldas de un CPLD, pero como cada FPGA tienen una gran cantidad de bloques programables es posible implementar grandes funciones utilizando varios bloques en cascada.

1.3.3.3 Interconexión entre bloques programables

Además de los bloques programables también es importante la tecnología utilizada para crear las conexiones entre los canales (tecnología de programación). Las más importantes son las siguientes:

1. Antifusible (*Antifuse*): Al igual que la tecnología PROM (memoria de solo lectura programable), un FPGA que utiliza este tipo de tecnología sólo se puede programar una sola vez, y utilizan algo similar a un fusible para las conexiones. Una vez que es programado ya no se puede recuperar. La diferencia entre un fusible y un antifusible es que el primero se desactiva deshabilitando la conexión, en cambio, para el segundo se produce una conexión cuando son programados, por lo que normalmente se encuentran abiertos. La desventaja obvia es que no son reutilizables, pero por otro lado disminuyen considerablemente el tamaño y costo de los dispositivos.
2. SRAM (*StaticRAM*): Estas guardan la configuración del circuito. Esto quiere decir que las SRAM son utilizadas como generadores de funciones y además son usadas para controlar multiplexores (que están incluidos en los FPGAs) y la interconexión entre bloques. En éstas el contenido se almacena mediante un proceso de configuración en el momento de encendido del circuito que contiene al FPGA. Ya que al ser SRAM, el contenido de la memoria se pierde cuando se deja de suministrar energía; la información binaria de las celdas SRAM generalmente se almacena en memorias seriales EEPROM conocidas como memorias de configuración o celdas de configuración. En el momento de encendido del circuito toda la información binaria es transferida a los bloques e interconexiones del FPGA mediante el proceso de configuración el cual es generalmente automático, dado que el propio FPGA contiene un circuito interno que se encarga de hacer toda la programación.
3. Flash: El avance experimentado en los últimos años en el diseño y prestaciones de las celdas de memoria Flash ha permitido su incorporación reciente al mundo de los dispositivos programables como tecnología de programación. Las FPGAs basadas en celdas Flash recogen las ventajas principales de las dos técnicas anteriores situándose en un punto intermedio. Su tamaño es bastante más reducido que el de una celda de SRAM, aunque sin llegar al tamaño reducido de un antifusible; son reprogramables, aunque la velocidad de programación es bastante más lenta que en el caso de una SRAM; y son no volátiles, por lo que no necesitan un dispositivo auxiliar para guardar la configuración interna, como en el caso de la SRAM

Un FPGA que tiene una gran cantidad de canales de interconexión tiende a tener pequeños bloques lógicos con muchas entradas y salidas en comparación con el número de compuertas que puede generar el bloque. Este bloque lógico se caracteriza por ser bastante sencillo, con poca lógica en su parte combinacional. Este es el caso del bloque lógico a base de multiplexores

del apartado anterior. Este tipo de FPGA generalmente utiliza tecnología antifusible. Un FPGA que tiene una estructura pequeña en canales de interconexión tiende a tener grandes bloques lógicos con pocas entradas y salidas en comparación con el número de compuertas que existe en el bloque. Este es el caso del bloque lógico basado en LUTs, que permiten implementar cualquier función lógica del mismo número de entradas. Este tipo de FPGA generalmente está fabricado con tecnología SRAM. Finalmente, las FPGAs basadas en celdas Flash suelen emplear un bloque lógico sencillo para incrementar los recursos de rutado, como ocurre con las FPGAs de antifusibles.

Una arquitectura con bloques pequeños permite utilizar todos los recursos del dispositivo, sin embargo tendremos que utilizar un gran número de estas para poder implementar funciones lógicas de varios términos, lo cual genera un tiempo de retardo por cada bloque implementado. Cuando el tamaño del bloque es grande sucede lo contrario, en este tipo de bloques es posible utilizar un gran número de compuertas por lo que podemos implementar funciones lógicas de varios términos con pocos bloques. El que el tamaño del bloque sea grande no afecta la frecuencia máxima de trabajo, ya que estamos hablando de que existe una gran cantidad de compuertas que pueden ser utilizadas en la función paralelamente, siendo el mismo tiempo de retardo para todas. Sin embargo, cuando las funciones son pequeñas en comparación con el tamaño del bloque, no es necesario utilizar todas las compuertas que soporta el bloque, por lo que este tipo de bloques no son precisamente las más indicadas para desempeñar pequeñas funciones.

1.3.3.4 Bloques entrada/salida

Al igual que en las CPLDs, la función de un bloque de entrada/salida es permitir el paso de una señal hacia dentro o hacia el exterior del dispositivo. Además debe contar con recursos tales como:

- Salidas configurables como TRI-STATE u *open-collector*.
- Entradas con posibilidad de *pull-up* o *pull-down* programables.
- Registros de salida.
- Registros de entrada.

En la Figura 1.24 tenemos como ejemplo un diagrama en bloques simplificado de un bloque de entrada/salida de la familia 4000 de las FPGA de Xilinx.

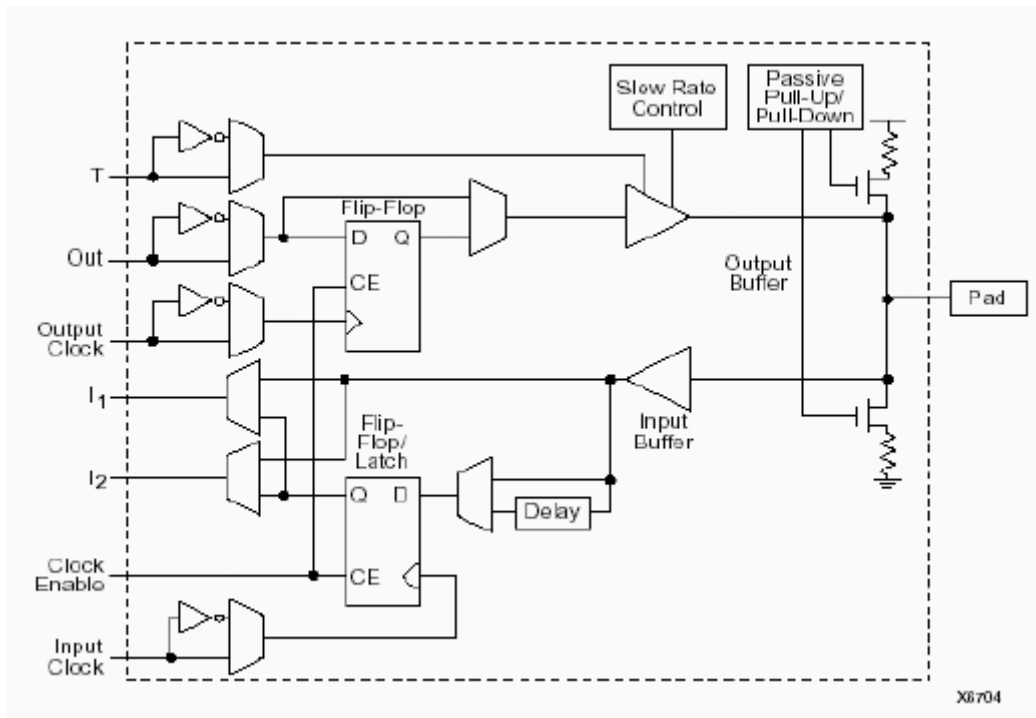


Figura 1.24: Bloque entrada/salida (IOB) de XC4000

1.3.3.5 Tiempos de propagación

El cálculo de los retardos incluidos en un diseño basado en FPGAs no es tan fácil de predecir como en el caso de CPLD. La propia estructura matricial de estos circuitos hace difícil saber cuántas celdas básicas, interconexiones programables o bloques de entradas/salida se utilizan para cada señal. Esto es, es necesario conocer cómo se ha realizado el procedimiento de *place and route* en el dispositivo para tener un conocimiento exacto de los retardos de propagación debidos a cada uno.

Debido a este factor, las herramientas de desarrollo para FPGAs suelen incluir un analizador de tiempos, que sirve para calcular tiempos de *set-up*, de *clock-to-output*, y sobre todo dar una estimación de la frecuencia máxima de operación.

1.4 Arquitectura de los Dispositivos FPGAs de Xilinx

En este punto se describirá la estructura interna básica que utiliza *Xilinx* para sus modelos más utilizados de FPGAs. Los dispositivos que se van a abarcar utilizan tecnología SRAM para su programación, y a esta memoria se le mencionará en adelante como celdas de configuración.

Los FPGAs de *Xilinx* constan de una cuadrícula de bloques lógicos programables (CLBs o bloques programables como se mencionó anteriormente). Tanto como los bloques lógicos como la estructura de interconexión se pueden programar cargando desde una fuente externa valores

a las celdas de configuración. En la figura 1.25 se puede apreciar la estructura general de un FPGA *Xilinx*.

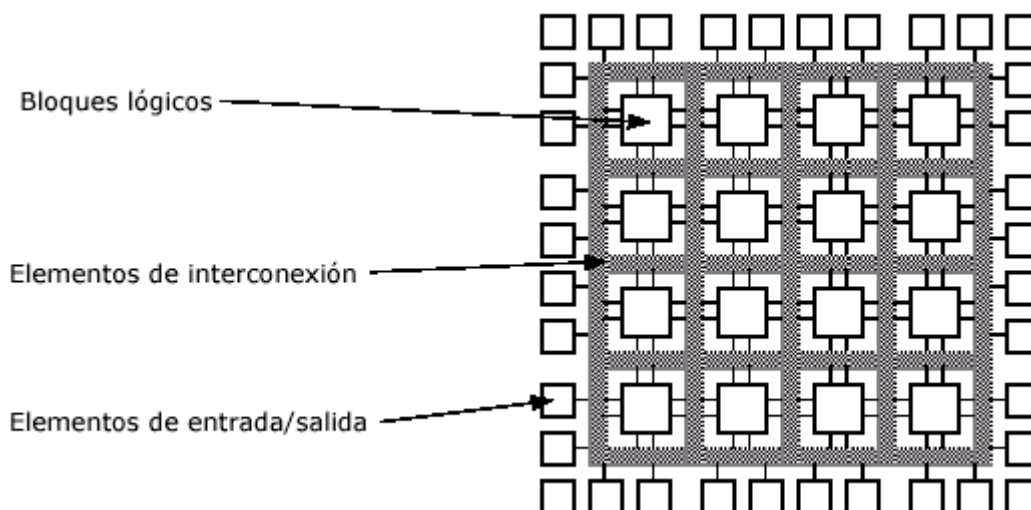


Figura 1.25: Arquitectura básica de una FPGA de Xilinx

Las celdas de configuración controlan la lógica y la interconexión que realizan la función de aplicación del FPGA. No hay un área de RAM separada en el chip, sino que las celdas de configuración están distribuidas en el chip. En los bordes, rodeando los bloques lógicos hay bloques de entrada/salida configurables (IOBs).

Los bloques básicos que componen un FPGA de *Xilinx* son:

1. CLBs (*Configurable Logic Block*)
2. IOBs (*Input Output Block*)
3. Estructura de interconexión

1.4.1 Bloques Lógicos (CLBs)

Como dijimos en la explicación sobre las FPGAs, el bloque lógico consta de una parte combinacional, que permite implementar funciones lógicas booleanas, más una parte secuencial que permite sincronizar la salida con una señal de reloj externa e implementar registros.

En el caso de Xilinx, la parte combinacional se realiza mediante tablas de *look-up*. (LUTs). Estas tablas se encuentran en las celdas de configuración (SRAM) que corresponde, por lo tanto, a la tabla de verdad de la implementación lógica. La cantidad de estas LUT dependerá del tipo de familia de dispositivos FPGA Xilinx utilizada.

Además se añaden componentes tales como *flip-flops*, multiplexores, *buffers* etc. Estos componentes están en posiciones fijas del dispositivo, y al igual que las LUTs, la cantidad depende de la familia utilizada.

1.4.2 Tecnología de programación

En el caso de las FPGAs de Xilinx, como mencionamos anteriormente, los elementos de programación se basan en células de memoria SRAM que controlan transistores de paso, puertas de transmisión o multiplexores. En la figura 1.26 se puede ver esquemáticamente cómo son. Dependiendo del tipo de conexión requerida se elegirá un modelo u otro.

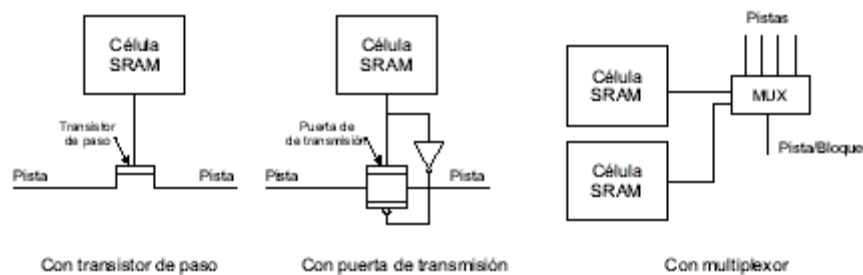


Figura 1.26: Tipo de conectores utilizados por Xilinx

Es importante destacar que si se utilizan células SRAM la configuración de la FPGA será válida únicamente mientras esté conectada la alimentación, pues es memoria volátil. En los sistemas finales está claro que hace falta algún mecanismo de almacenamiento no volátil que cargue las células de RAM. Esto se puede conseguir mediante PROMs, EPROMs, EEPROMs o disco.

1.4.3 Bloques de entrada/salida (IOB)

Estos bloques conectan el circuito con el exterior. Dichos bloques ofrecen salidas de tres estados y sirven para almacenar temporalmente las señales de entrada y salida.

1.4.4 Descripción de las principales familias

Hay múltiples familias lógicas dentro de Xilinx. Las primeras que surgieron son: XC2000, XC3000 y XC4000, correspondientes respectivamente a la primera, segunda y tercera generación de dispositivos, que se distinguen por el tipo de bloque lógico configurable (CLB) que contienen. En la actualidad existen también las familias de FPGA Spartan, SpartanII, SpartanXL, SpartanIIE, SpartanIII, Virtex, VirtexII y VirtexIIPro. La tabla 1.2 muestra la cantidad de CLBs que puede haber en cada FPGA de las algunas familias y ese mismo valor expresado en puertas equivalentes.

SERIE	Tipo CLB	Nº de CLBs	Puertas Equivalentes
XC2000	1 LUT, 1 FF	64-100	1200-1800
XC3000	1 LUT, 2 FF	64-484	1500-7500
XC4000XL	3 LUT, 2 FF	64-3136	1600-180000
Virtex II	4 LUT, 4 FF	64-10148	40000-800000
Spartan II	4 LUT, 4 FF	96-1176	15000-200000

Tabla 1.2: Algunas familias de FPGAs de Xilinx

1.4.4.1 Familias XC4000 y Spartan

La figura 1.27 es un esquema general de la FPGA XC4000 de Xilinx. Esta misma estructura es utilizada en la familia Spartan. Se trata de una FPGA de RAM estática, formada por una estructura regular de bloques lógicos (CLBs) y bloques de entrada/salida en la periferia (IOBs). En la familia Spartan II, cada bloque lógico (llamado *slice*) contiene el equivalente a dos CLBs de las series Spartan.

Todos estos bloques se unen unos con otros a través de las interconexiones programables que existen en los canales internos de rutado.

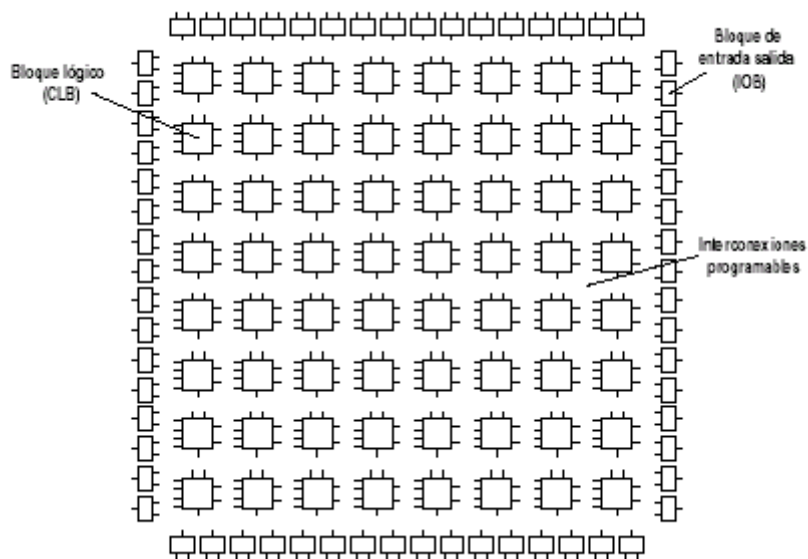


Figura 1.27: Esquema genérico de las FPGAs de Xilinx XC4000 y Spartan

El bloque lógico interno o CLB se ilustra mediante un diagrama simplificado en la figura 1.28. La parte combinacional es implementada a partir de tres generadores de funciones: F, G y H; los dos primeros son LUTs de cuatro entradas, y la H, de tres entradas. Cada uno de los generadores de funciones F y G pueden implementar independientemente cualquier función lógica de cuatro entradas, almacenando todos los posibles resultados en sus 16 celdas de memoria RAM estática. La LUT H tiene la posibilidad de trabajar con las salidas de las otras dos LUTs para implementar funciones de mayor número de entradas o directamente con las entradas C_x del CLB, según se programen los multiplexores internos. Es interesante observar el hecho de que la parte combinacional del CLB puede emplearse, no sólo para realizar funciones lógicas, sino para implementar una memoria RAM interna asincrónica, que podría ser incluso una memoria de doble puerto, o una memoria RAM sincrónica, usando los *flip-flops* de salida del CLB. La LUT H es más que una simple LUT de tres entradas y es optimizado para ser configurado como un pequeño elemento aritmético que implementar sumadores/restadores de acarreo paralelo. El CLB tiene un total de 13 entradas

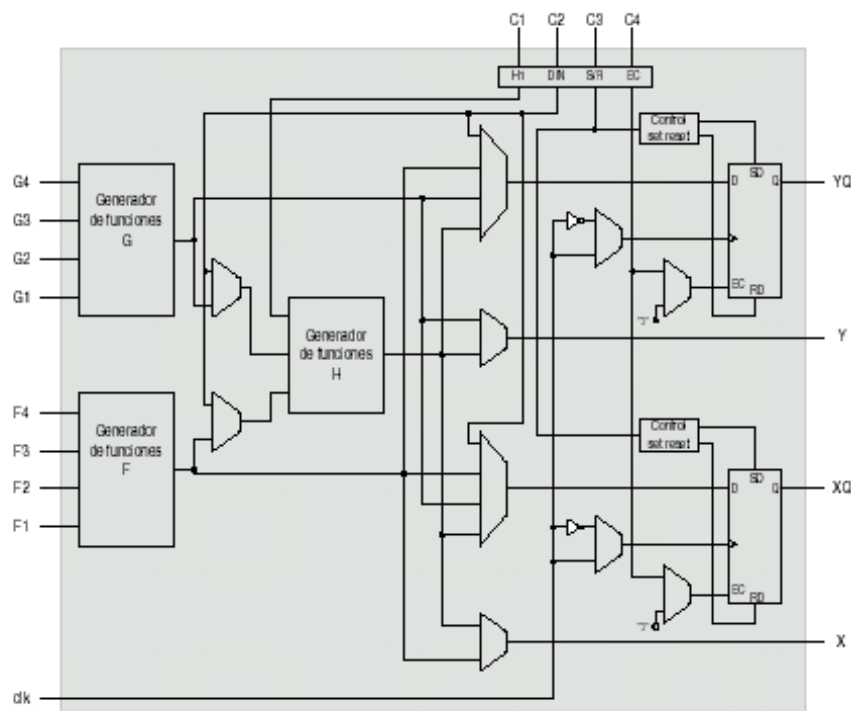


Figura 1.28: Diagrama de bloques simplificado del CLB de la FPGA XC4000 de Xilinx

En cuanto a la parte secuencial, consta de dos *flip-flops* o *latches*, según se configuren. Como *flip-flops*, la señal de reloj es una entrada del CLB, siendo programable la polaridad del reloj de cada *flip-flop*, a partir de los multiplexores internos encargados de ello. Como *latches*, ambos usan la señal de *clock enable* (EC), sin existir en este caso la posibilidad de invertirla dentro del CLB. Por último, ambos elementos de almacenamiento disponen de entradas de *set* y *reset* independientes.

Las salidas del CLB vienen gobernadas por un multiplexor que determina si ésta proviene de la salida de los registros o directamente de la parte combinacional. Es importante señalar que la programación de los multiplexores internos de configuración del CLB son totalmente transparentes al usuario, siendo el sistema de desarrollo quien, de una manera automática, se encarga de esta labor, tomando como punto de partida el diseño realiza por el usuario.

Los bloques lógicos de entrada y salida o IOBs proporcionan la interfase entre los pines externos y la lógica interna del integrado. Cada IOB controla un pin externo que puede ser configurado como entrada, salida o bidireccional. La figura 1.29 es un detalle de la estructura interna de una IOB.

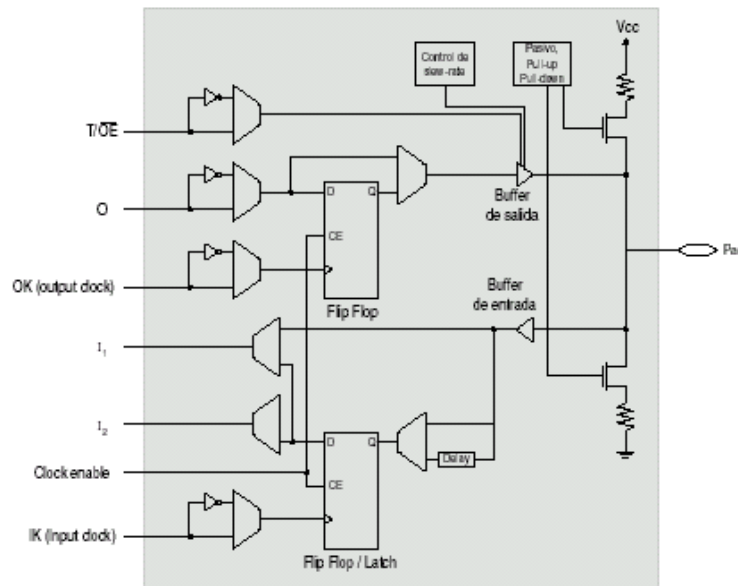


Figura 1.29: Diagrama de bloques simplificado de la IOB de la FPGA XC4000 de Xilinx

Las señales I_1 e I_2 son entradas a la lógica interna y pueden pasar por un *flip-flop* incluido en la propia IOB o pasar directamente desde el pin de entrada, según la programación interna de la IOB. La señal de salida O procedente de la lógica interna también tiene la posibilidad de pasar por un *flip-flop*. Los dos *flip-flops* internos de la IOB poseen señales de reloj independientes y carecen de *set* y *preset*, a diferencia de los *flip-flops* de la CLB. Finalmente, la señal T permite controlar el estado de alta impedancia del *buffer* de salida, necesario para implementar pines bidireccionales o conectar la salida a un bus.

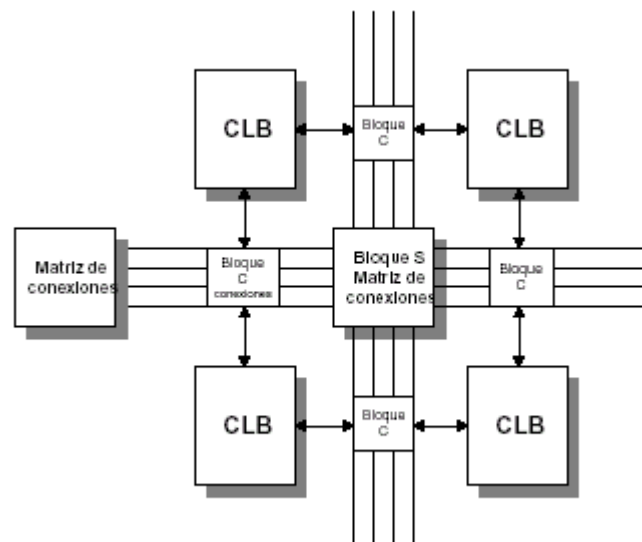


Figura 1.30: Conexiones internas en la FPGA XC4000 de Xilinx

Las interconexiones internas de una FPGA se pueden realizar de tres formas:

- *Conexiones directas*. Son las que tienen lugar entre CLB's adyacentes. Estas conexiones se indican mediante flechas en la figura 1.30.
- *Conexiones de propósito general*. Para realizar conexiones entre CLB's no adyacentes es necesario pasar por las matrices de interconexiones de la

figura 1.30. Su misión es conectar canales verticales y canales horizontales de rutado que permitan llegar al punto final de conexión. Cuanto más larga sea la línea de rutado, mayores serán los retrasos introducidos, tanto por la propia longitud de la línea, como por los elementos de conexión por los que va pasando.

- *Líneas largas.* En diseños complejos, siempre existen líneas que deben recorrer el integrado o que tienen longitudes grandes, como ocurre con las señales de reloj o los buses internos. Para evitar que estas líneas atraviesen muchos elementos de conexión, que introducen retrasos, se contempla la existencia de líneas largas, que recorren el integrado, y que normalmente se usan para llevar la señal de reloj o soportar buses internos.

1.4.4.2 Familia Virtex

La familia Virtex utiliza una arquitectura similar a la de la Spartan-IIe que se detallará más adelante.

La Virtex se compone de dos partes básicas, los CLBs y los IOBs. El arreglo se muestra en la figura 1.31. Los CLBs se encargan de proveer los elementos funcionales para construir la lógica, y los IOBs se encargan de la interfaz entre los pines del integrado y los CLBs.

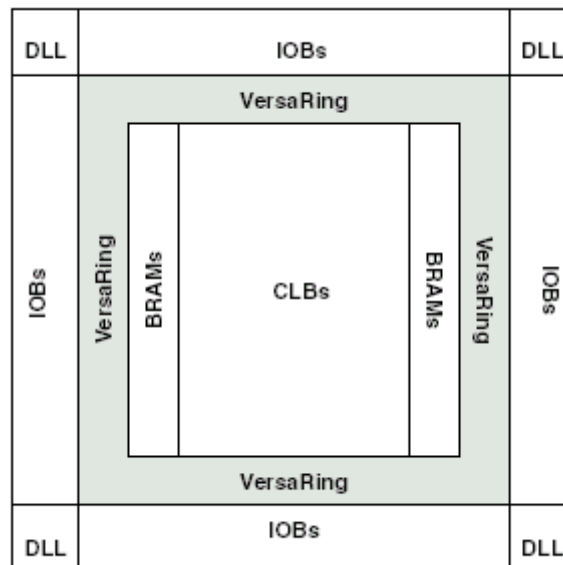


Figura 1.31: Arquitectura Virtex

Los CLBs están interconectados a través de una matriz de enrutamiento general o GRM (*General Routing Matrix*), la que se compone de un arreglo de interruptores locales ubicados en las intersecciones horizontal y vertical de los canales de enrutamiento.

Aparte del GRM, la arquitectura Virtex tiene los siguientes sistemas de enrutamiento:

- VersaBlock™: Corresponde a un tipo de bloque donde se anida cada CLB. Estos suministran recursos de enrutamiento locales para conectar un CLB con el GRM.

- VersaRing™: Suministra una interfaz adicional de enrutamiento alrededor de la periferia del dispositivo. Este sistema mejora el enrutamiento y facilita el asignamiento de pines

También se pueden distinguir los diferentes circuitos que se conectan al GRM:

- Bloques de memoria: Son bloques de memoria dedicados de 4096 bits de tamaño cada uno. Se encuentran dentro de los CLBs y sirven como RAM.
- DLLs (*Delay-Locked Loop*): Son dispositivos que controlan las señales de reloj y compensan el retardo en dichas señales.
- BUFTs (*3-State Buffers*): Buffers de tres estados. Están asociados a cada CLB, los cuales alimentan los canales horizontales dedicados dentro de la matriz de enrutamiento.

La estructura de los CLBs y de los IOBs son similares a los de la Spartan IIE, que como dijimos se explicarán detalladamente en el próximo punto.

La tabla 1.3 muestra los recursos que integran algunos de los miembros de la familia de dispositivos Virtex .

Dispositivo	Celdas Lógicas	Compuertas del Sistema	Matrices CLBs	bits de RAM distribuída	bits de bloques RAM
XCV50	1728	57906	16 x 24	24576	32K
XCV100	2700	108904	20 x 30	38400	40K
XCV150	3888	164674	24 x 36	55296	48K
XCV200	5292	236666	28 x 42	75264	56K
XCV300	6912	322970	32 x 48	98304	64K
XCV400	10800	468252	40 x 60	153600	80K
XCV600	15552	661111	48 x 72	221184	96K
XCV800	21168	888439	56 x 84	301056	112K
XCV1000	27648	1124022	64 x 96	393216	128K

Tabla 1.3: Dispositivos de la familia Virtex

1.4.4.3 Familia Spartan IIE

1.4.4.3.1 Introducción

Las FPGAs Xilinx Spartan™ son ideales para las aplicaciones de bajo costo y alto volumen y son designadas como reemplazos para arreglos de compuertas de lógica fija y para productos estándar de aplicación específica (ASSP), productos como sets de chips para interfase de bus. Hay cinco miembros de la familia: dispositivos Spartan-3 (1.2V), Spartan IIE (1.8V), Spartan II (2.5V), Spartan XL (3.3V) y Spartan (5V).

La familia FPGA Spartan IIE 1.8V provee a los usuarios con alta performance, recursos de la lógica abundantes, entre otras características mas. El séptimo

miembro de la familia ofrece densidades que van de 50.000 a 600.000 compuertas..

La Spartan II está fabricada en un proceso mixto de 0,18µm/0,22mm y seis capas de metal, utilizando una de las más avanzadas tecnologías de proceso existentes hoy en día.

Los dispositivos Spartan se caracterizan por tener una arquitectura flexible y regular que se compone de un arreglo de bloques lógicos configurables (*Configurable Logic Blocks* o CLBs), rodeados por bloques de entrada/salida programables (*programmable Input/Output Blocks* o IOBs). Hay cuatro *Delay-Locked Loops* (DLLs), uno a cada esquina del dispositivo. Se encuentran dos columnas de bloques RAM lados opuestos del dispositivo, entre los CLBs y las columnas de IOB que se puede utilizar para definir memorias RAM de puerto simple y de puerto doble (en forma de bloque y distribuída). El XC2S400E tiene cuatro columnas y el XC2S600E tiene seis columnas de bloques RAM.

Estos elementos funcionales se interconectan por una jerarquía poderosa de canales de la asignación de ruta versátiles.

El núcleo de la FPGA funciona a 1,8 voltios e incorpora una tecnología que la permite funcionar con interfaces de I/O de 3,3 voltios siendo tolerante a señales de 5 voltios.

El desempeño de estos dispositivos hace que puedan alcanzar tasas de 200MHz tanto en el interior como en los bloques de Entrada/Salida.

1.4.4.3.2 Arquitectura de los dispositivos Spartan IIE

Como se mencionó anteriormente, el dispositivo Spartan-IIE está compuesto de cinco elementos distinguibles configurables:

- IOBs (*Input/Output Blocks*): proporcionan la interfaz entre el paquete pines y la lógica interna.
- CLBs (*Configurable Logic Blocks*): proporcionan los elementos funcionales para construir más lógica.
- Bloques de RAM dedicados de 4096 bits cada uno.
- Clock DLLs (*Delay-Locked Loops*): para la compensación de retraso de distribución de reloj y control de dominio de reloj.
- Estructura de interconexión multi-nivel versátil.

Como puede verse en figura 1.32, los CLBs forman la estructura de lógica central con acceso fácil a todas las estructuras de soporte y ruteo. Los IOBs se localizan alrededor de toda la lógica y de los elementos de memoria para un ruteo de señales fácil y rápido dentro y fuera del chip.

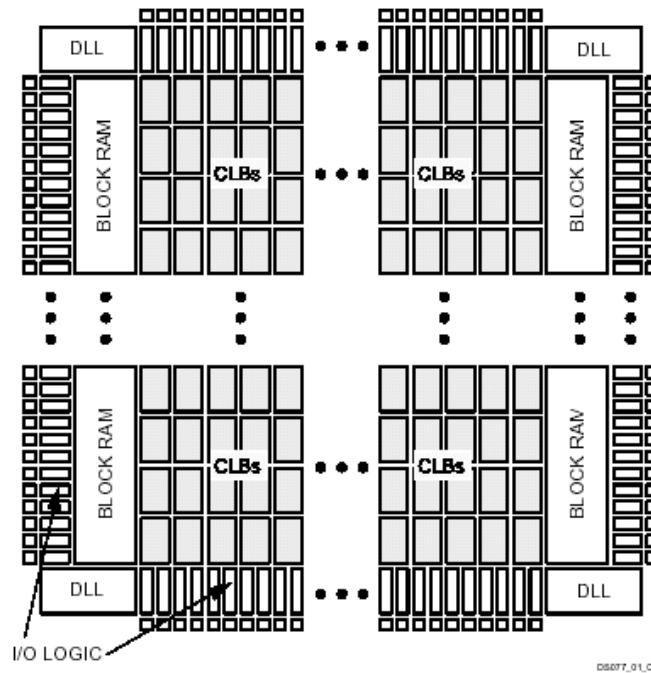


Figura 1.32: Diagrama en Bloques de una Básica familia FPGA Spartan-II

Los valores guardados en las células de memoria estáticas controlan todos los elementos lógicos configurables y recursos de interconexión. Estos valores son cargados en las células de memoria en *power-up*, y pueden ser recargados si necesario para cambiar la función del dispositivo.

Cada uno de estos elementos se discutirá en detalle en las secciones siguientes.

La tabla 1.4 muestra los recursos que integran algunos de los miembros de la familia de dispositivos Spartan IIE.

Dispositivo	Celdas Lógicas	Rango Típico de Compuertas del Sistema (Lógica y RAM)	CLBs	bits de RAM distribuída	bits de bloques RAM
XC2S50E	1728	23000 – 50000	384	24576	32K
XC2S100E	2700	37000 – 100000	600	38400	40K
XC2S150E	3888	52000 – 150000	864	55296	48K
XC2S200E	5292	71000 – 200000	1176	75264	56K
XC2S300E	6912	93000 – 300000	1536	98304	64K
XC2S400E	10800	145000 - 400000	2400	153600	160K
XC2S600E	15552	210000 - 600000	3456	221184	288K

Tabla 1.4: Dispositivos de la familia Spartan IIE

1.4.4.3 Bloque de Entrada/Salida

Las entradas y salidas soportan muchos tipos de estándar de señalización, como TTL, CMOS, PCI(3.3V), AGP, etc, que se diferencian en los niveles de voltaje que manejan.

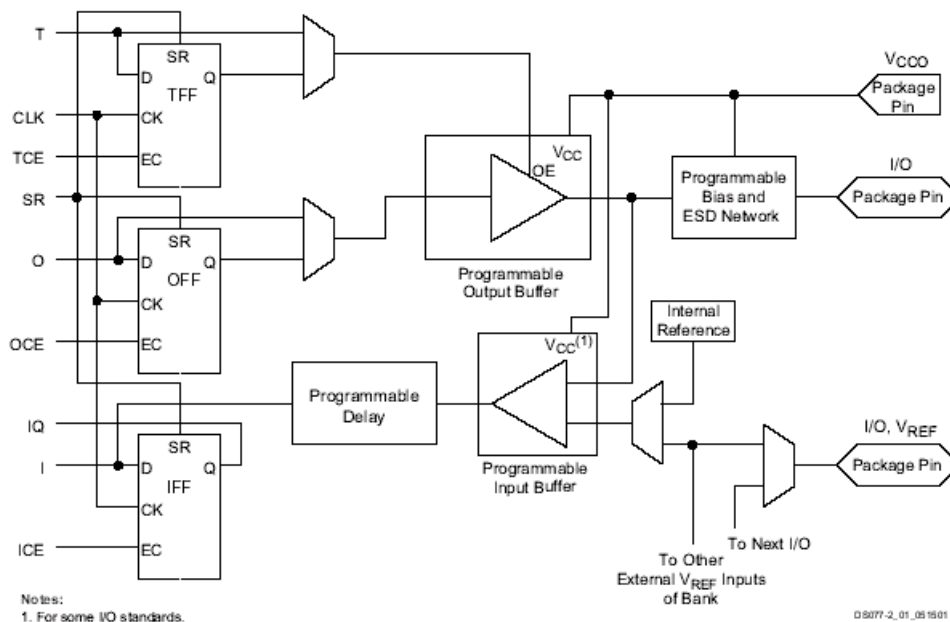


Figura 1.33: Spartan IIE IOB.

En la figura 1.33 se muestra el contenido de un IOB. Se pueden observar tres elementos de almacenamiento, los que pueden ser utilizados como *flip-flop D* o como *latches*. Cada *flip-flop* tiene una señal de reloj compartida, con señales independientes de habilitación para el reloj (*Clock Enable*) y una señal compartida *Set/Reset* (SR). Para cada registro, esta señal puede configurarse independientemente como un *Set* sincrónico, un *Reset* sincrónico, un *Preset* asincrónico, o un *Clear* asincrónico.

Un rasgo no mostrado en el diagrama de bloques, pero controlado por el software, es el control de polaridad. Los *buffer* de entrada y salida y todos las señales de control IOB tienen controles de polaridad independientes.

Resistencias *pull-up* y *pull-down* opcionales y un circuito guardián-débil (*weak-keeper*) optativo son adjuntados a cada bloque I/O. Antes de la configuración se fuerzan todas las salidas no involucradas en la configuración a su estado de alta-impedancia. Las resistencias *pull-down* y los circuitos guardián-débil son inactivos, pero las entradas pueden convertirse en *pull-up* opcionalmente. La activación de la resistencias *pull-up* antes de la configuración es controlada en una base global por los pines de modo de configuración. Si las resistencias *pull-up* no son activadas, todos los pines flotarán. Por consiguiente, resistencias *pull-up* o *pull-down* externas debes ser proporcionadas en los pines requeridos para un nivel de lógica bien definida antes de la configuración.

1.4.4.3.3.1 Entradas

Cada entrada puede conectarse, a través de un *buffer*, directamente a la lógica interna del Spartan o a la entrada de un *flip-flop* (opcional). Otro componente opcional es un elemento de retardo (*Programmable Delay*, figura 1.33), que elimina los problemas de tiempo en la entrada, haciendo coincidir el retardo interno que existe en las líneas de reloj del FPGA, y de esta manera minimizar la diferencia en tiempo entre la entrada y las señales internas.

Cada *buffer* de entrada puede ser configurado para soportar cualquier estándar de señalización soportado por el dispositivo. En algunos casos, el *buffer* de entrada utiliza un voltaje umbral proporcionado por el usuario, V_{REF} . La necesidad de proporcionar V_{REF} impone restricciones en cuales normas pueden usarse en proximidades cercanas de el uno al otro.

También se dispone de resistencias tipo *pull-up* y *pull-down*, para hacer más flexible aún el tipo de entrada. Por ejemplo, se puede configurar una entrada como *pull-up* para que pueda ser conectada a una salida de colector abierto.

1.4.4.3.3.2 Salidas

Las salidas están precedidas por un *buffer* de 3 estados. La entrada de este *buffer* puede venir desde un *flip-flop* de salida o directamente desde la lógica interna, y a su vez, la entrada de control de este *buffer* puede venir directamente desde la lógica interna o desde un *flip-flop*. Al igual que las entradas, las salidas pueden ser configuradas individualmente para soportar los diferentes estándares de señalización. Cada *buffer* de salida tiene una capacidad de 24 [mA] de corriente, con un control del *slew rate* el cual, a costa de hacer un poco mas lento la respuesta temporal del bloque de I/O, puede disminuir el efecto de oscilaciones que se pueden generar en la salida debido al cambio de un nivel lógico a otro

En la mayoría de los estándares de señalización, el nivel alto de tensión de salida depende de una referencia de tensión llamada V_{CC0} , que debe ser suministrado en forma externa por el usuario. Más información sobre esto se cubre en el próximo punto.

1.4.4.3.3.3 Bancos de Entrada/Salida

Algunos de los estándares de entrada y salida requieren de una tensión externa V_{CC0} y/o una tensión de referencia V_{REF} . Estas tensiones están conectadas a las entradas y salidas por grupos, llamados bancos. Es decir, si aplicamos un cierto estándar a una entrada o salida, ésta va a incluir también a las otras entradas o salidas que pertenecen al mismo banco. Por lo tanto existen ciertas restricciones al utilizar estos estándares.

Ocho bancos de entrada/salida resultan de subdividir cada periferia del dispositivo en dos bancos. Cada banco tiene varios pines de V_{CC0} , los cuales deben ser conectados al mismo nivel de tensión. El valor del nivel de tensión está especificado por el estándar utilizado. Dentro de un banco, los estándares pueden ser combinados sólo si utilizan el mismo valor para V_{CC0} . En el caso de las entradas, se requiere de una tensión umbral V_{REF} (suministrado externamente). Este valor se debe introducir en ciertos pines del dispositivo que son automáticamente asignados para este propósito. Dependiendo del caso, aproximadamente uno de seis pines de un banco es utilizado para este rol. Los pines de V_{REF} dentro de un banco están interconectados internamente, lo que implica el uso de solo un nivel de V_{REF} por banco, sin embargo pueden ser mezclados con pines del mismo banco que no requieren de esta tensión umbral.

El número y lugar de los pines dedicados a V_{CC0} y V_{REF} dependen del tipo de FPGA Spartan IIE que se utilice.

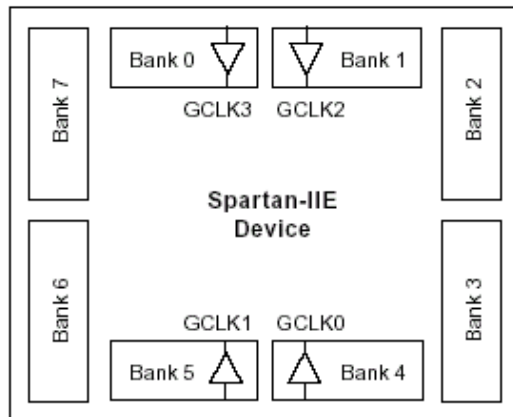


Figura 1.34: Bancos de Entrada/Salida.

1.4.4.3.4 Bloque Lógico Configurable (CLBs)

La estructura básica de un CLB es la celda lógica o LC (*Logic Cell*). Un LC incluye un generador de funciones (LUT) de 4 entradas, lógica de acarreo (*Carry Logic*), y un elemento de almacenaje. Cada CLB contiene cuatro LCs, organizados en dos trozos o *Slices* similares, tal como muestra la figura 1.35. Además de los cuatro LCs, un CLB contiene también una lógica adicional que hace posible la combinación de los generadores de función para que éstos puedan aceptar cinco o seis entradas. Por lo tanto, cuando se quiere estimar el número de compuertas de un dispositivo dado, cada CLB soporta hasta 4.5 LCs.

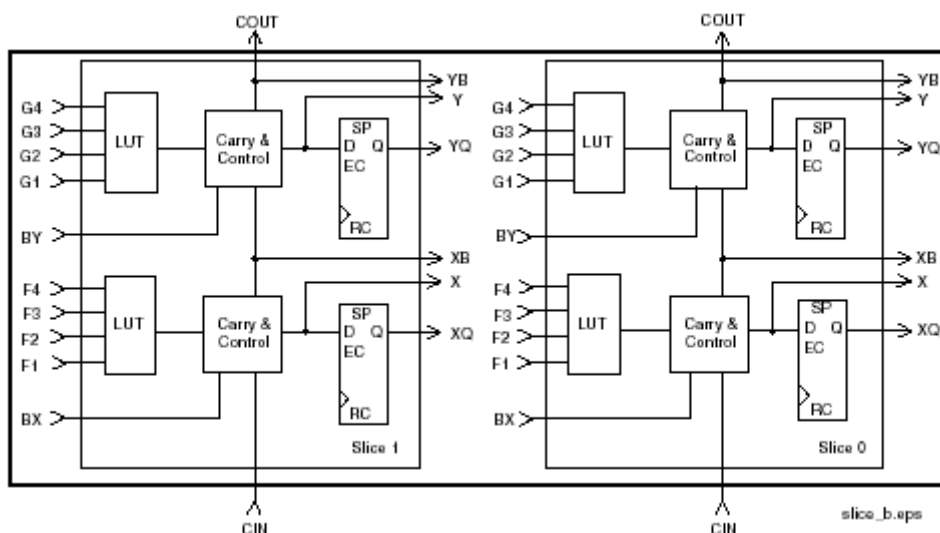


Figura 1.35: 2 – Slice dentro en un CLB.

Cada uno de los elementos que componen el CLB se describe en los siguientes puntos.

1.4.4.3.4.1 Tabla de Look up (*Look up Table* – LUT)

Los generadores de funciones están implementados como tablas de look up de 4 entradas. Además de trabajar como generadores de función, cada LUT puede funcionar como una RAM sincrónica de 16 x 1-bit. También dos LUTs dentro un *Slice* pueden ser combinadas para trabajar como una RAM sincrónica de 16 x 2-bits, de 32 x 1-bit o de 16 x 1-bit utilizando puerto dual.

A parte de lo anterior, una LUT puede utilizarse como un registro de desplazamiento de 16-bits, que es ideal para capturar datos a altas velocidades. Esta característica es muy utilizada en aplicaciones de procesamiento de señales (DSP).

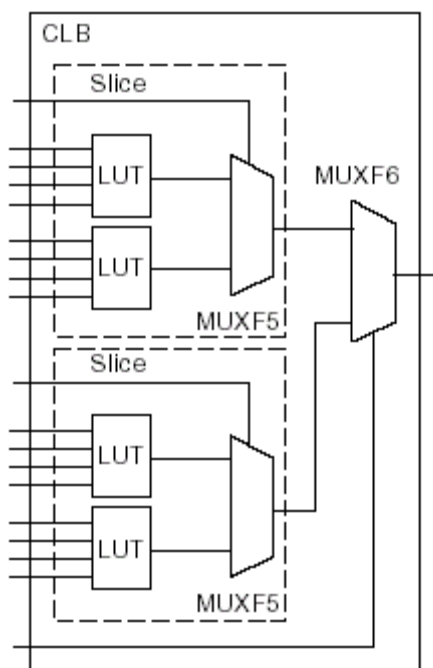
1.4.4.3.4.2 Elementos de Almacenamiento

Los elementos de almacenamiento de un *Slice* pueden ser configurados como un *flip-flop D* o como un *latch*. Las entradas (entradas D, en la figura 1.35) pueden ser alimentadas por el generador de funciones o directamente de las entradas del *Slice*.

Además, aparte de las señales de reloj y de habilitación del reloj (*Clock Enable*), cada *Slice* tiene señales sincrónicas de *Set* y *Reset* (SR y BY). SR fuerza a un elemento de almacenamiento a su estado de inicialización especificado en la configuración. Por otro lado, BY fuerza a dicho elemento de almacenamiento al estado opuesto. Alternativamente, ambas señales pueden configurarse para que trabajen en forma asincrónica. Todas las señales de control son invertibles en forma independiente, y son compartidas por los dos *flip-flops* dentro del *Slice*.

1.4.4.3.4.3 Lógica Adicional

Cada *Slice* contiene un multiplexor (llamado multiplexor F5), tal como se ve en figura 1.36, que se utiliza para combinar las salidas de los generadores de funciones. Gracias a esta combinación se puede implementar un generador de funciones de 5 entradas, un multiplexor 4:1, o un seleccionador de funciones de hasta nueve entradas. De forma similar, otro multiplexor (llamado multiplexor F6) combina las salidas de los cuatro generados de función dentro de un CLB seleccionando una de las dos salidas del multiplexor F5. Esto permite la implementación de un generador de funciones de 6 entradas, un multiplexor 8:1, o un selector de funciones de hasta 19 entradas. Por lo tanto, la lógica adicional descrita hace que la funcionalidad de un CLB sea mucho más flexible y potente.



DB077-2_05-111501

Figura 1.36: Multiplexores F5 y F6

1.4.4.3.4.4 Lógica Aritmética

Un acarreador lógico (*carry logic*) dedicado provee una rápida capacidad aritmética para aplicaciones que requieren un procesamiento de datos veloz. Cada CLB contiene dos cadenas de acarreadores (una cadena por *Slice*). La resolución de las cadenas de acarreadores es de dos bits por CLB.

La aritmética lógica incluye además una compuerta XOR, la que permite la implementación de un sumador de 1-bit dentro de un LC. Además, se dispone de una compuerta AND dedicada que cumple la función de mejorar la eficiencia en la implementación de multiplicadores.

Adicionalmente, los canales utilizados por los acarreadores pueden ser utilizados por los generadores de función para la implementación de funciones lógicas amplias.

1.4.4.3.4.5 BUFTs

Cada CLB contiene dos *buffers* de 3 estados (BUFTs) utilizados para alimentar los buses internos del chip. Cada BUFT tiene un pin de control y de entrada independiente.

1.4.4.3.5 Bloques de RAM (*BlockRAM*)

Las FPGAs Spartan-IIe incorporan varios bloques grandes de memoria RAM, llamadas por el fabricante *BlockRAM*. Estos bloques ofrecen un mejor complemento que los que ofrece la RAM distribuida como las Tablas de *Look-*

Up (LUTs) que proporcionan estructuras de memoria superficiales implementadas en CLBs.

Los bloques de memoria *Block RAM* están organizados en columnas. La mayoría de los dispositivos Spartan-IIe contienen dos de estas columnas, una a lo largo de cada borde vertical, donde estas columnas se extienden por todo el chip. La XC2S400E tiene cuatro columnas de bloques de RAM y el XC2S600E tiene seis columnas de bloques de RAM. El tamaño de cada bloque de memoria corresponde a cuatro veces el tamaño de un CLB, por lo tanto, un dispositivo Spartan-IIe con 16 CLBs contendrá cuatro bloques de memoria por columna, y un total de ocho bloques.

Cada celda *BlockRAM* es totalmente sincrónica, de doble puerto (es decir, son dos memorias en una e independientes), y de capacidad de 4096-bit.

Se dispone además de un sistema de enrutamiento dedicado que provee una eficiente comunicación entre ambos CLBs y otros bloques *BlockRAM*.

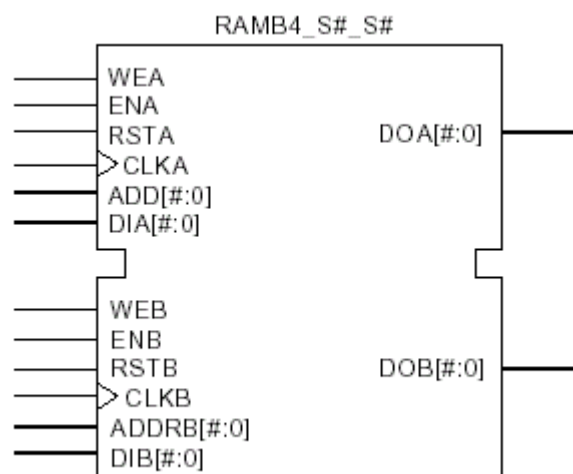


Figura 1.37: BlockRAM de doble puerto

1.4.4.3.6 Enrutamiento Programable

Corresponde a la ruta de mayor retardo que limita la velocidad en cualquier diseño. Es por esto que la arquitectura de enrutamiento trabaja en conjunto con el *software* dedicado a la localización y enrutamiento (*place-and-route software*), de tal manera de obtener el mínimo de retardo que se pueda producir en un diseño. Este *software* corresponde a un algoritmo que busca las mejores rutas según la implementación y el tipo de dispositivo.

Los componentes de la matriz de enrutamiento programable se describen a continuación.

1.4.4.3.6.1 Enrutamiento Local

Los recursos de enrutamiento local, tal como se muestra en la figura 1.38, suministran los siguientes tres tipos de conexiones:

- Interconexiones a lo largo de los LUTs, *flip-flops* y la GRM (Matriz General de Enrutamiento).

- Rutas internas retroalimentadas en los CLB, las cuales hacen posible una conexión de alta velocidad entre LUTs dentro de un mismo CLB, encadenadas entre ellas con un mínimo de retardo asociado.
- Rutas directas que hacen conexión entre CLBs horizontalmente adyacentes, a una alta velocidad, eliminando el retardo producido si se conectaran mediante la GRM

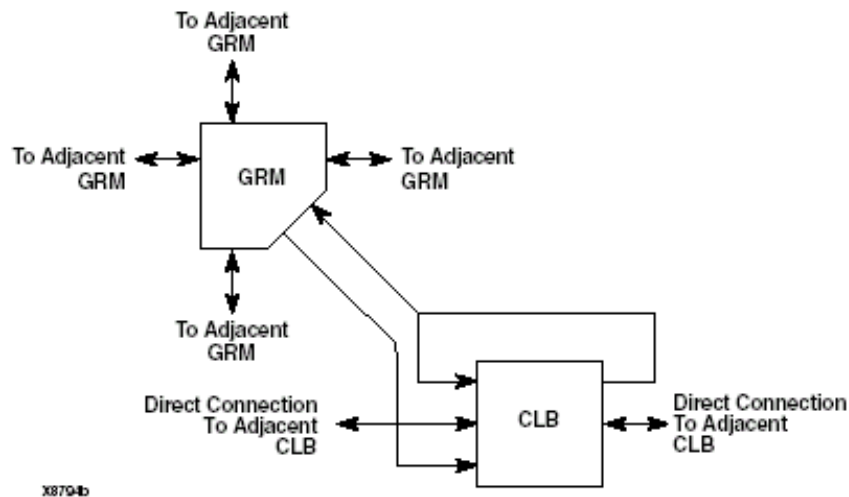


Figura 1.38: Enrutamiento Local.

1.4.4.3.6.2 Enrutamiento de Propósito General

Es en este sistema de enrutamiento por donde viajan la mayoría de las señales, y por consiguiente, la mayoría de los recursos de interconexión están asociados a este tipo de enrutamiento. Los recursos de enrutamiento general están localizados en los canales horizontales y verticales, los que a su vez están asociados a las filas y columnas de los CLBs. Los recursos de enrutamiento son los siguientes:

- Adyacente a cada CLB se encuentra un GRM. El GRM es la matriz de interruptores por la cual los recursos de enrutamiento horizontal y vertical se conectan, y también es la responsable de que los CLBs tengan acceso al sistema de enrutamiento de propósito general.
- 24 líneas que rutean las señales provenientes de la GRM a otras GRMs en cada una de las cuatro direcciones.
- 96 x 6 líneas provistas de *buffers* encaminan las señales de una GRM a otras GRMs ubicadas a seis bloques de distancia de la primera, en cada una de las cuatro direcciones. Estas líneas están diseñadas para interconectar GRMs que se encuentran lejos, donde la distancia es de seis bloques. Sin embargo, también pueden conectar GRMs que se encuentran a 3 bloques de distancia.
- Se dispone de 12 líneas de longitud larga, conectadas a *buffers*, bidireccionales, utilizadas para distribuir señales a través del dispositivo en forma rápida y eficiente.

Estas líneas cubren el dispositivo en forma vertical y horizontal.

1.4.4.3.6.3 Enrutamiento de Entrada/Salida

Este sistema de enrutamiento provee de recursos adicionales alrededor de la periferia para la conexión entre los bloques de IOBs y los CLBs. Este sistema de enrutamiento adicional llamado VersaRing™, facilita el intercambio (*pin-swapping*) y localización de los pines (*pin-locking*), lo que es muy útil en el rediseño, ya que se pueden adaptar las entradas y salidas cuando el dispositivo está inserto en una placa.

1.4.4.3.6.4 Enrutamiento Dedicado

Algunas señales son críticas en un diseño, por lo que necesitan de recursos dedicados para maximizar su desempeño. En la arquitectura Spartan IIE se dispone de dos tipos de enrutamiento dedicado para satisfacer dos clases de señales.

- Recursos de enrutamiento horizontal con buses de tres estados. Cuatro líneas de buses (que se pueden separar) se disponen por columna de CLBs, permitiendo múltiples buses dentro de una columna, tal como muestra la figura 1.39.
- Dos redes dedicadas por CLB propagan señales de tipo acarreo (*carry signals*) verticalmente a CLBs adyacentes.

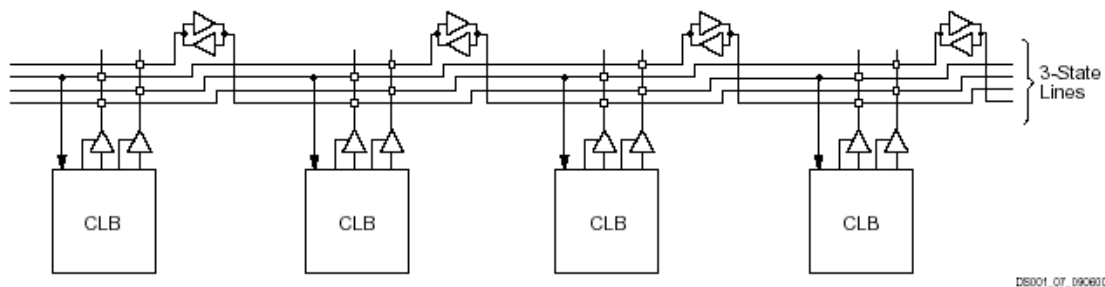


Figura 1.39: Conexiones de BUFT a las Líneas del Bus Horizontales Dedicadas

1.4.4.3.6.5 Enrutamiento Global

Los recursos del enrutamiento global distribuyen las señales de reloj y otras señales con gran nivel de alimentación de salida (*very high fanout*) a través del dispositivo. Los dispositivos Spartan IIE contienen dos hileras de recursos globales de enrutamiento, llamados recursos globales de enrutamiento primario y secundario:

- Los recursos de enrutamiento global primario se componen de cuatro redes globales con pines de entrada dedicados, que están diseñadas para distribuir con un gran *fanout* las señales de reloj con un mínimo de *skew* (*Skew* se refiere a la distorsión que afecta a los pulsos cuando viajan a través de las líneas. Esta distorsión produce que los extremos del pulso tiendan a ser más oblicuos). Cada red global puede alimentar todos los pines de reloj de los bloques CLB, IOB, y RAM. La red primaria

global puede ser alimentada sólo por los *buffers* globales, de los cuales existen cuatro, uno por cada red.

- Los recursos local secundarios consisten en una médula de 24 líneas, 12 recorren la parte de arriba del chip, y 12 recorren la parte baja del chip. Los recursos locales secundarios son más flexibles de los globales primarios, ya que no están restringidos sólo a señales de reloj.

1.4.4.3.7 Distribución de Reloj

La arquitectura Spartan IIE posee una distribución de reloj de alta velocidad y bajo *skew*, a través de los recursos globales primarios descritos anteriormente. Una red de distribución de reloj típica se muestra en la figura 1.40. Se disponen de cuatro *buffers* globales, dos arriba y al centro, y dos abajo y al centro del dispositivo. Estos alimentan las cuatro redes primarias globales que pueden conectarse a cualquier pin de reloj. Existen cuatro pines dedicados a señales de reloj, cada uno adyacente a cada *buffer* global. La entrada a los *buffers* globales es seleccionada desde estos pines o desde señales del enrutamiento de propósito general.

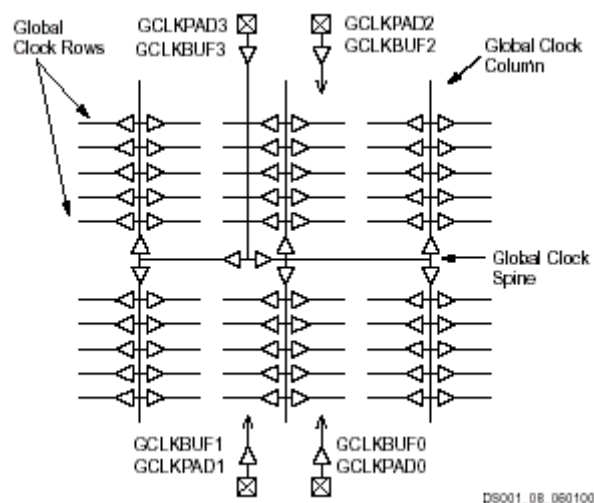


Figura 1.40: Red de Distribución Global de Reloj.

1.4.4.3.7.1 Delay-Locked Loop (DLL)

Un DLL realiza la misma tarea que los tradicionales PLLs (*Phase Lock Loops*), pero de una forma más robusta y menos susceptible a interferencias de ruido. Cada DLL de la familia Spartan II E está asociado a una entrada del *buffer* de reloj, es completamente digital y elimina el *skew* entre el pin de entrada y los pines internos de reloj a través del dispositivo. Cada DLL puede también alimentar dos redes globales de reloj. El DLL supervisa el reloj de la entrada y el reloj distribuido, y automáticamente ajusta un elemento de retraso de reloj (figura 1.41). Se introduce el retraso adicional de tal forma que los cantos del reloj alcanzan los *flip-flops* interiores exactamente un período del reloj después de que llegan a la entrada.

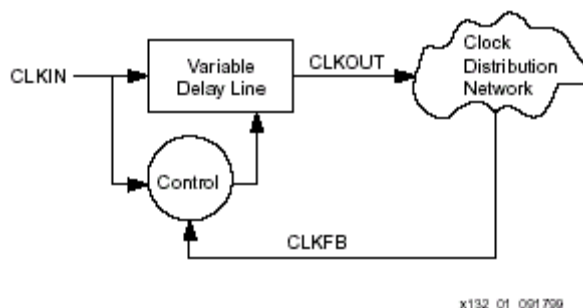


Figura 1.41: Diagrama en bloques de un DLL.

El DLL funciona con un sistema de retroalimentación que elimina eficientemente el retardo en la distribución de la señal de reloj asegurando que los cantos del reloj que llegan al interior del dispositivo estén en sincronismo con los cantos de reloj que llegan a la entrada.

Además de eliminar el retardo en las señales de reloj, el DLL tiene un sistema de control que permite múltiples dominios de reloj. Se disponen de cuatro fases en cuadratura respecto al reloj de entrada, que pueden multiplicar por dos, o dividir por 1.5, 2, 2.5, 3, 4, 5, 8, o 16 dicha señal de reloj.

En general el DLL de la familia Spartan II E permite realizar las funciones enumeradas a continuación:

- Desplazamiento de fase.
- Espejo de relojes.
- Multiplicar.
- Dividir.
- Sincronización con relojes externos.

Todos los dispositivos de la familia Spartan IIE poseen cuatro DLLs.

1.5 Arquitectura de los Dispositivos de Altera

Inicialmente debemos diferenciar la clasificación en CPLDs y FPGAs utilizadas por Altera con la descrita en nuestro trabajo en puntos anteriores.

Nosotros describimos la arquitectura de una CPLD como una agrupación de PALs o GALs, interconectadas entre sí, donde cada bloque lógico tiene una parte combinacional compuesta por matrices de compuertas AND y OR, más un registro asociado al pin de entrada/salida. En cambio la arquitectura de la FPGA la describimos también como un bloque lógico con una parte combinacional y una parte secuencial, en el cual la parte combinacional es mucho más simple que la de una de las SPLD interna de una CPLD; ya que puede ser basado en LUTs o en multiplexores.

Altera, por su parte, diferencia las CPLDs y las FPGAs por diferentes estructuras de interconexión. La estructura de interconexión segmentada es utilizada por las FPGAs y utilizan líneas múltiples de longitud variable unidas por transistores de paso o antifusibles para conectar las celdas lógicas. En contraste la estructura de interconexión continua es utilizada por las CPLDs

para proveer conexiones de celda lógica a celda lógica que lleva finalmente a velocidades más altas comparable con las FPGAs. De esta forma Altera ofrece las siguientes familias de CPLDs:

- APEX 20K
- ACEX 1K
- FLEX 10K
- FLEX 8000
- FLEX 6000
- MAX 9000
- MAX 7000
- MAX 5000
- Classic

con características como muestra la tabla 1.5:

Familia	Estructura del Bloque Lógico	Estructura de Interconexión	Tecnología de programación
Stratix	LUT	Contínua	SRAM
Cyclone	LUT	Contínua	SRAM
APEX 20K	LUT y termino producto	Contínua	SRAM
ACEX 1K	LUT	Contínua	SRAM
FLEX 10K	LUT	Contínua	SRAM
FLEX 8000	LUT	Contínua	SRAM
FLEX 6000	LUT	Contínua	SRAM
MAX 9000	termino producto	Contínua	EEPROM
MAX 7000	termino producto	Contínua	EEPROM
MAX 5000	termino producto	Contínua	EPROM
Classic	termino producto	Contínua	EPROM

Tabla 1.5: Arquitectura de dispositivos Altera

De acuerdo a nuestra clasificación tomaremos a los dispositivos APEX, ACEX y FLEX como FPGAs mientras que los MAX como CPLDs y los Classic como SPLDs .

Cave aclarar que también, como vemos en la tabla 1.5 existen dispositivos nuevos clasificados como FPGAs por Altera, los cuales son Cyclone y Stratix.

1.5.1 Familia FLEX 10K

1.5.1.1 Introducción

Los dispositivos de la familia de dispositivos *FLEX 10K* de Altera integran elementos de memoria SRAM dispuestos en una matriz reconfigurable de elementos lógicos (*FLEX: Flexible Logic Element matrix*). Los dispositivos

FLEX 10K, con hasta 250000 puertas, proporcionan la densidad, velocidad y prestaciones necesarias para integrar sistemas completos en un único dispositivo. Los dispositivos de 2.5V *FLEX 10KE*, fabricados en una tecnología de 0.22 μ m, son entre un 20% y un 30% más rápidos que los de la familia de 0.3 μ m *FLEX10 KA* que operan a 3.3V. De forma similar, los dispositivos *FLEX 10KA* son, en promedio, entre un 20% y un 30% más rápidos que los de los dispositivos *FLEX 10K* que operan a 5V y se fabrican en una tecnología de 0.42 μ m.

La arquitectura de los dispositivos *FLEX 10K* se inspira en el rápido crecimiento que han experimentado en la industria los dispositivos programables basados en matrices de puertas. A su vez, estos dispositivos disponen de zonas designadas para la integración de elementos de memoria que se pueden utilizar para construir funciones de mayor complejidad.

1.5.1.2 Arquitectura de los dispositivos FLEX 10K

Cada dispositivo de la familia *FLEX 10K* está integrado por bloques de memoria y una matriz de elementos lógicos. Los bloques de memoria son conocidos como EABs (*Embedded Array Blocks*) y pueden utilizarse para definir pequeñas memorias o funciones lógicas especiales. Cuando se utiliza como un elemento de memoria, cada EAB proporciona 2048 bits y se puede utilizar para definir memorias RAM, ROM, RAM de doble puerto o funciones *first-in first-out* (FIFO). Si un EAB se utiliza para definir funciones lógicas complejas tales como multiplicadores, controladores, máquinas de estados o funciones para DSP, contribuye con entre 100 y 600 puertas. Los EABs se pueden utilizar de forma independiente o también pueden utilizarse en modo combinado para realizar funciones más complejas o elementos de memoria de mayor capacidad.

La matriz de elementos lógicos (*Logic Array*) está constituida por bloques lógicos conocidos como LABs (*Logic Array Blocks*). Cada LAB agrupa 8 elementos lógicos o LEs (*Logic Elements*) e interconexiones locales. Un LE incluye una tabla de look-up (LUT, *Look Up Table*) de 4 entradas, un *flip-flop* programable y lógica para la rápida generación y propagación de acarreo (*carry*) y la conexión en cascada (*cascade*). Los ocho LEs de un LAB se pueden utilizar para definir pequeñas funciones lógicas como contadores y sumadores de hasta 8 bits; además, se pueden agrupar varios LABs para definir bloques lógicos mayores. Cada LAB representa aproximadamente 96 puertas lógicas. Las conexiones entre elementos de memoria y elementos lógicos se realizan mediante el llamado *FastTrack Interconnect*, una serie de rápidos canales de fila y columna continuos que recorren todo el ancho y el alto del dispositivo.

Cada pin de Entrada/Salida se alimenta por un elemento de Entrada/Salida (IOE, *Input/Output element*) localizado al final de cada fila y columna del *FastTrack Interconnect*. Cada IOE contiene un *buffer* de Entrada/Salida bidireccional y un *flip-flop* que pueden usarse tanto como registro de la salida o entrada para alimentar la entrada, la salida, o las señales bidireccionales.

La Figura 1.42 muestra un diagrama del bloque de la arquitectura de la *FLEX 10K*. Cada grupo de LEs se combina en un LAB; los LAB's son colocados en las filas y columnas. Cada fila también contiene un solo EAB. Los LAB's y EABs

son interconectados por el FastTrack Interconnect. Los IOEs son localizados al final de cada fila y columna del FastTrack Interconnect.

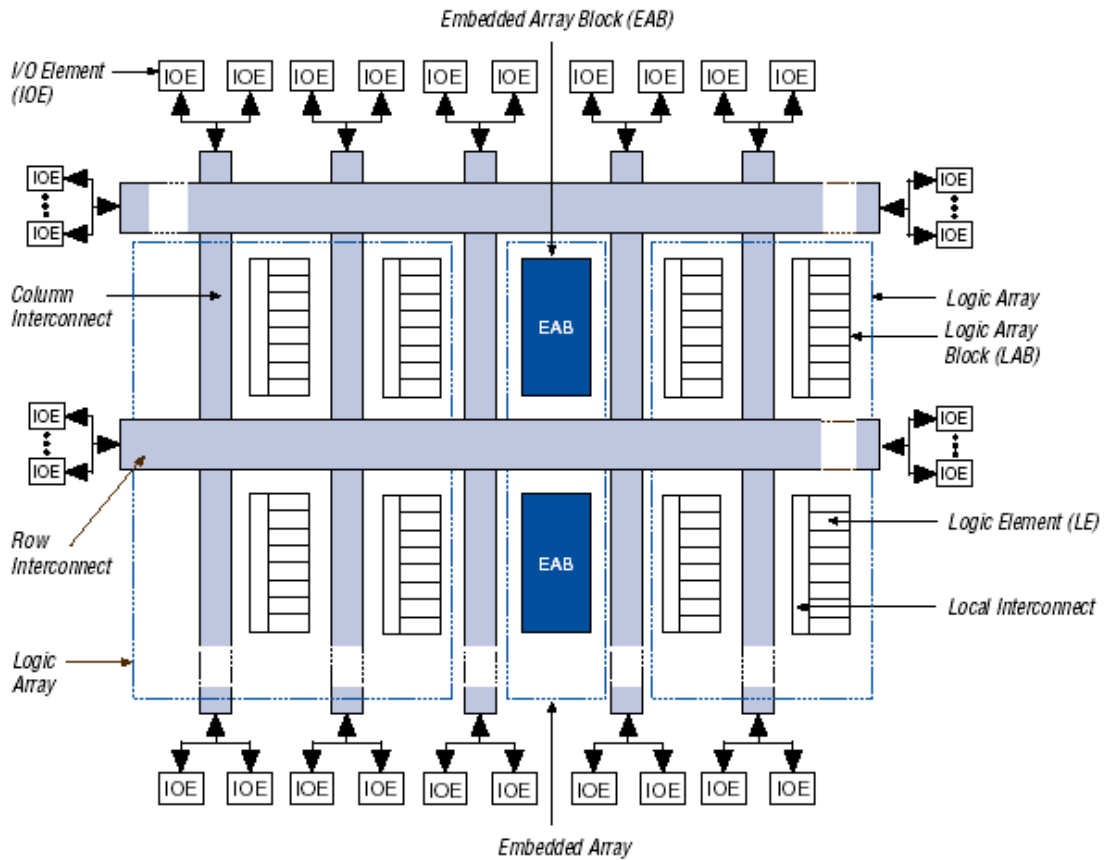


Figura 1.42: Arquitectura general de un dispositivo FLEX 10K

La tabla 1.6 muestra los recursos que integran algunos de los miembros de la familia de dispositivos FLEX 10K.

Dispositivo	LEs	EABs	Bits de memoria	Entradas Salidas
EPF10K10	576	3	6K	150
EPF10K20	1152	6	12K	189
EPF10K30	1728	6	12K	246
EPF10K40	2304	8	16K	189
EPF10K50	2880	10	20K	310
EPF10K70	3744	9	18K	358
EPF10K100	4992	12	24K	406
EPF10K130	6656	16	32K	470
EPF10K250	12160	20	40K	470

Tabla 1.6: Dispositivos de la familia FLEX 10K

Los dispositivos FLEX 10K proporcionan seis entradas especializadas que manejan los flip-flops que controlan las entradas para asegurar la distribución eficaz de alta velocidad, baja distorsión (skew) de las señales de control. Estas señales usan canales dedicados al ruteo que proporcionan retrasos más cortos y más bajas distorsiones que el FastTrack Interconnect. Cuatro de las entradas especializadas manejan cuatro señales globales. Estas cuatro señales globales

también pueden ser manejadas por la lógica interior, proporcionando una solución ideal para el divisor del reloj o una señal asincrónica internamente generada que limpia muchos registros en el dispositivo.

1.5.1.3 Embedded Array Block (bloques de arreglos integrados)

La figura 1.43 muestra un esquema interno del elemento de memoria. Cada EAB es una memoria RAM con registros en los puertos de E/S que se puede utilizar de diversas formas. Si se desea construir una función lógica compleja, el EAB se configura con un patrón de sólo lectura y se utiliza como una LUT. De esta manera, la función lógica deseada se almacena en la tabla y no es necesario calcularla para cada valor de entrada.

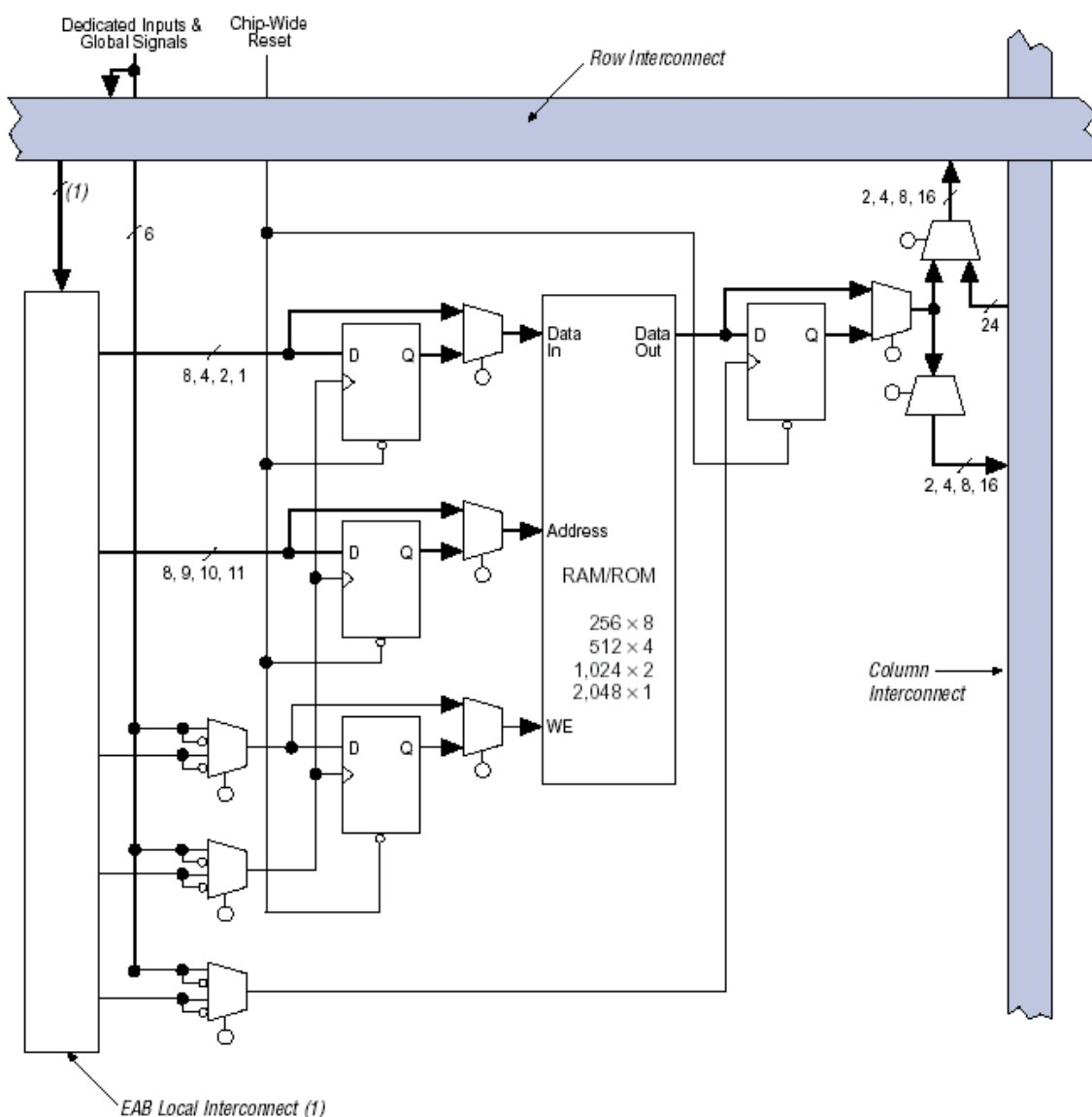


Figura 1.43: Embedded Array Block (EAB)

Las funciones lógicas de elevada complejidad así construidas pueden tener menor retardo que si se construyen haciendo uso de elementos lógicos.

Cuando el EAB se utiliza como una memoria RAM, se puede configurar como un bloque de tamaño 256 x 8, 512 x 4, 1024 x 2 o 2048 x 1. Se pueden crear bloques de memoria mayores combinando varios bloques de memoria. Por ejemplo, dos bloques de memoria RAM de 256 x 8 bits se pueden combinar para crear un solo bloque de 256 x 16; del mismo modo, dos bloques de 512 x 4 permiten crear un bloque de 512 x 8 y así sucesivamente.

Los EABs proveen opciones flexibles para manejar y controlar las señales del reloj. Pueden usarse diferentes relojes para las entradas y salidas del EAB. Pueden insertarse registros independientemente en la entrada de datos, salida del EAB, o las entradas de dirección y WE (*Write Enable* de la RAM). Las señales globales y las interconexiones locales (*Local Interconnect*) del EAB pueden manejar la señal WE. Las señales globales, pines dedicados a reloj, e interconexiones locales del EAB puede manejar las señales de reloj del EAB. Debido a que los LEs manejan las interconexiones locales del EAB, los LEs pueden controlar la señal WE o las señales de reloj del EAB.

Cada EAB se alimenta por una interconexión fila (*row interconnect*) y puede desembocar en interconexiones fila y columna. Cada salida del EAB puede conducir a dos canales de fila y a dos canales de columna; el canal de fila sin usar puede ser manejado por otros LEs. Este rasgo aumenta los recursos de la asignación de ruta disponible para las salidas de EAB.

1.5.1.4 Logic Array Block (bloques de arreglos lógicos)

Cada bloque lógico LAB está formado por ocho LEs, las cadenas de acarreo y conexión en cascada asociadas a estos LEs, señales de control, e interconexiones locales.

Cada LAB tiene cuatro señales de control de inversión programable que se pueden utilizar en los ocho elementos lógicos (LEs). Dos de estas señales se pueden utilizar como señales de reloj mientras las otras dos se pueden utilizar como señales de *preset* y *clear* de los registros. Las señales de reloj del LAB se pueden controlar por medio de los pines de entrada de reloj, señales globales, señales de E/S (I/O) o señales generadas internamente y conectadas por medio de la interconexión local de los LABs.

Las señales de control de *preset* y *clear* del LAB pueden ser manejadas por las señales globales, señales I/O, o señales interiores vía interconexión local del LAB. Las señales de control globales se usan típicamente para señales del reloj global, *clear*, o *preset* porque ellas proporcionan control sincrónico con muy baja distorsión a través del dispositivo. Si se requiere lógica en una señal de control, puede ser generada en uno o más LEs en cualquier LAB y manejada en la interconexión local de LAB designado. Además, las señales de control globales pueden generarse de las salidas de LE.

La figura 1.44. muestra el diagrama interno de un LAB.

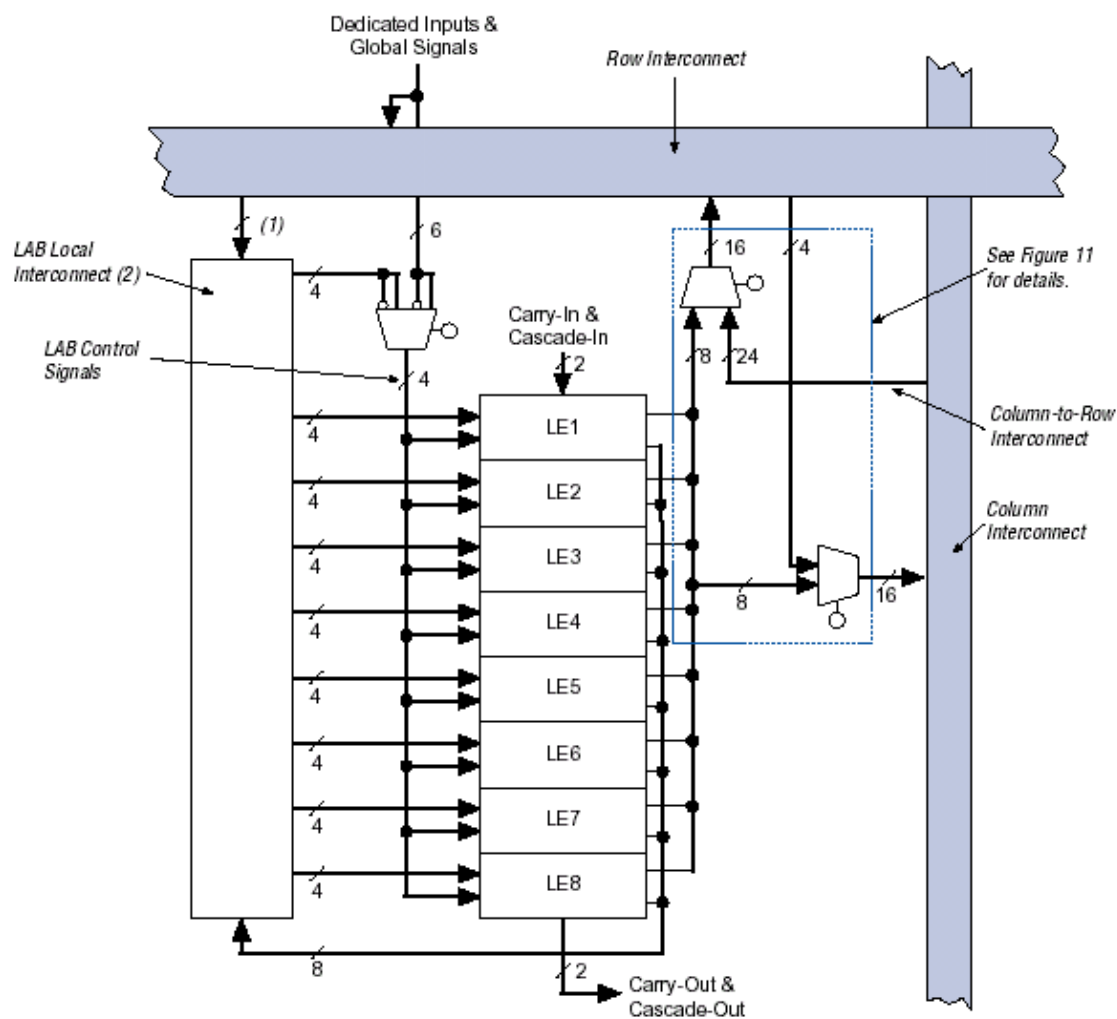


Figura 1.44: Logic Array Block (LAB)

1.5.1.4.1 Logic Element (Elemento Lógico)

Un LE es la entidad lógica básica de un dispositivo de la familia *FLEX10K*. Cada LE se compone de una LUT de cuatro entradas, la cual es un generador de funciones que permite computar cualquier función de cuatro variables rápidamente. A su vez, cada LE incluye un *flip-flop* programable con una entrada de habilitación sincrónica *enable*, lógica para la rápida propagación de acarreo entre LEs en un mismo LAB y lógica para la construcción de cadenas de conexión en cascada. La salida de cada LE se puede conectar local o globalmente en el dispositivo con otros elementos lógicos o de memoria. La figura 1.45 muestra el diagrama interno de un LE.

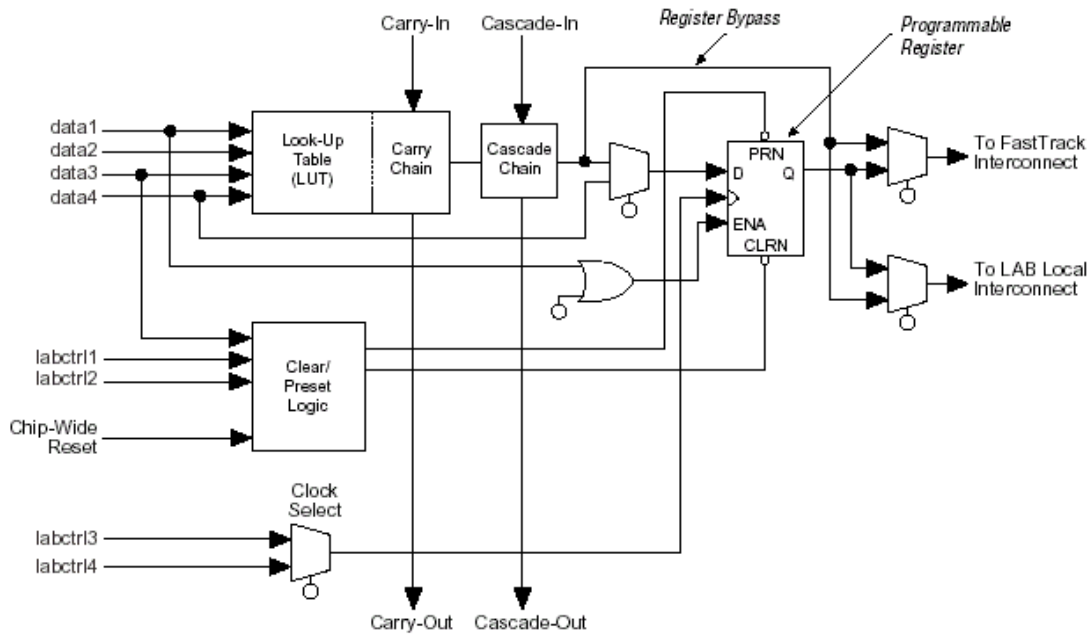


Figura 1.45: Logic Element (LE)

El *flip-flop* del LE se puede configurar para que opere como un biestable D, T, JK o SR. Las señales de control *clock*, *clear* y *preset* en el *flip-flop* pueden manejarse por las señales globales, pines I/O de uso general, o cualquier lógica interna. Para las funciones combinatorias, el *flip-flop* es "bypaseado" (esquivado) y la salida del LUT maneja la salida del LE.

Cada LE tiene dos conexiones de salida, una con la interconexión local y otra con la interconexión global que se encuentran organizadas por filas y columnas (FastTrack Interconnect). Las dos salidas se pueden controlar de forma independiente. Por ejemplo, la LUT puede ser una salida mientras que el registro puede ser otra. Esta posibilidad, llamada embalaje del registro (*register packing*), puede mejorar la utilización de LE porque pueden usarse el registro y la LUT para las funciones no relacionadas.

1.5.1.4.1.1 Cadenas de acarreo

Las cadenas de acarreo realizan la función de la rápida propagación de acarreo (2ns) entre LEs. La cadena de acarreo genera el acarreo de entrada del bit de orden superior a partir del acarreo de salida del bit que le precede y las entradas a la tabla.

Estas cadenas de acarreo permiten a los dispositivos *FLEX10K* realizar eficientemente sumadores, contadores y comparadores de gran velocidad y de longitud arbitraria. Las cadenas de acarreo de más de ocho LEs se implementan de forma automática uniendo LABs. Para una mejor distribución de la cadena de acarreo en el dispositivo, se evita la utilización de dos LABs consecutivos en una misma fila. Al mismo tiempo, no se generan cadenas de acarreo conectando dos LABs separados por el EAB situado en el centro de la fila.

1.5.1.4.1.2 Cadenas de conexión en cascada

Las cadenas de conexión en cascada se utilizan en los dispositivos *FLEX10K* para realizar funciones lógicas de un amplio número de variables. De esta manera, se utilizan tablas adyacentes para calcular porciones de la función en paralelo y la cadena de conexión en cascada conecta en serie las salidas intermedias. La cadena de conexión en cascada puede utilizar una puerta AND o una puerta OR para conectar las salidas de dos LEs adyacentes. Cada LE puede operar sobre cuatro entradas de la función que se calcula y el retardo de la conexión en cascada es de tan solo 0.7ns por LE. Se pueden implementar cadenas de conexión en cascada de más de ocho LEs uniendo varios LABs.

Para una mejor distribución en el dispositivo, se evita la utilización de dos LABs consecutivos en una misma fila. Al mismo tiempo no se generan cadenas de conexión en cascada conectando dos LABs separados por el EAB situado en el centro de la fila.

1.5.1.4.1.3 Modos de operación

Un elemento lógico puede configurarse de acuerdo con alguno de los cuatro modos de operación siguientes:

- modo normal (*normal mode*).
- modo aritmético (*arithmetic mode*).
- modo contador ascendente/descendente (*up/down counter mode*).
- modo contador con puesta a cero (*clearable counter mode*).

La figura 1.46 muestra el esquema interno del elemento lógico para cada tipo de operación.

Cada uno de estos modos usa de forma diferente los recursos que proporciona cada LE. En cada modo, las siete señales de entrada (las cuatro señales de entrada, la realimentación del registro y las de acarreo y conexión en cascada) se dirigen a distintos destinos para implementar la función lógica deseada. Las entradas de reloj, *preset* y *clear* de cada LE se utilizan para el control del registro.

- **Modo normal:**

Este modo está especialmente orientado para la generación de funciones lógicas y por tanto, se puede aprovechar la cadena de conexión en cascada. En el modo normal, las cuatro entradas de datos y la entrada de acarreo son entradas para la LUT. El compilador se encarga de seleccionar una de las señales *carry-in* o *data3* como una de las entradas de la tabla. La salida de la LUT se puede combinar con la señal *cascade-in* para generar una cadena de conexión en cascada a través de la señal *cascade-out*. Tanto la salida de la tabla como la del registro se pueden conectar con la interconexión local o global al mismo tiempo.

La LUT y el registro se pueden utilizar de forma independiente. Para ello el LE tiene dos salidas, una conectada localmente y la otra globalmente. Existen diversas formas de utilizar la LUT y el registro de forma separada. La señal *data4* puede ser utilizada como entrada del registro y así permite que la tabla

calcule una función que es independiente del valor registrado. La LUT puede también calcular una función de tres entradas al mismo tiempo que cuatro señales independientes se almacenan en el registro. Por último, la LUT puede calcular una función de cuatro entradas y una de estas entradas se puede utilizar para controlar el registro.

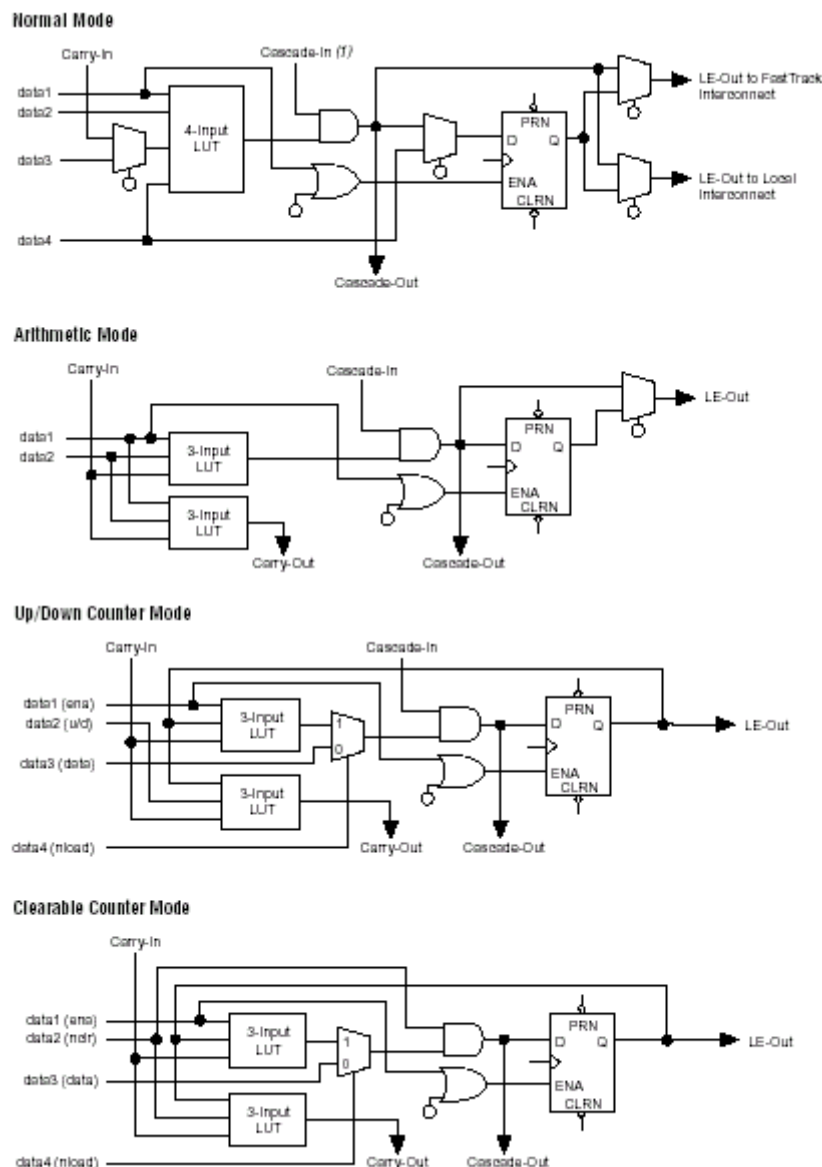


Figura 1.46: Modos de operación de un LE.

- **Modo aritmético:**

El modo aritmético ofrece dos LUTs de tres entradas y es ideal para la implementación de sumadores, acumuladores y comparadores. Una LUT calcula una función de tres entradas y la otra genera una salida de acarreo. Tal y como se muestra en la figura 1.46, la primera LUT usa la señal *carry-in* y dos entradas de datos para generar una salida combinacional o registrada. La segunda LUT utiliza las mismas entradas y genera el acarreo de salida *carry-out* y, por tanto, crea una cadena de acarreo. El modo aritmético también soporta el uso simultáneo de la cadena de conexión en cascada.

- **Modo contador ascendente/descendente:**

El modo de contador ascendente/descendente ofrece control de habilitación, incremento/decremento sincrónico y carga de datos. Las señales de control se toman de la interconexión local, de la entrada de acarreo *carry-in* y de la realimentación del registro. Este modo utiliza dos LUTs de tres entradas, una genera el dato del contador y la otra genera un bit de acarreo. Un multiplexor 2:1 permite la carga sincrónica del contador. También se pueden cargar datos de forma asincrónica y sin utilizar los recursos de la LUT utilizado las señales de control *clear* y *preset*.

- **Modo contador con puesta a cero:**

El modo de contador con puesta a cero es similar al modo de contador ascendente/descendente. La diferencia está en que éste soporta puesta a cero sincrónica en lugar de control del incremento ascendente o descendente. La función de puesta a cero se canaliza a través de la entrada *carry-in*. En este modo se utilizan dos LUTs de tres entradas, una genera los datos del contador y la otra genera el bit de acarreo. La carga sincrónica del contador se consigue mediante un multiplexor de 2:1.

1.5.1.5 FastTrack Interconnect (pista rápida de interconexión)

Las conexiones entre elementos lógicos (LEs) y pines de entrada/salida del dispositivo *FLEX 10K* se realiza por medio del FastTrack Interconnect, el cual es una serie de canales continuos de enrutamiento dispuestos en forma horizontal y vertical y extendiéndose sobre todo el dispositivo. Esta estructura de enrutamiento global es muy eficiente incluso en diseños complejos.

Con este esquema de enrutamiento cada fila de LABs dispone de un canal de interconexión horizontal. Este canal permite la conexión tanto con otros LABs del dispositivo como con pines de entrada/salida. La figura 1.47 detalla la organización de estas conexiones y la forma en que se comunican con los LEs del LAB. A cada fila se puede conectar un LE u otro de entre tres canales columna. Una de estas cuatro señales se conecta a través de un multiplexor de 4:1 con dos canales fila específicos. Estos multiplexores permiten conectar los canales columna con canales filas incluso cuando los ocho LEs de un LAB se encuentran conectados con la fila. Del mismo modo, cada columna de LABs se conecta con una interconexión columna. La interconexión columna puede manejar entonces pines de *I/O* u otra interconexión de fila para dirigir las señales a otros LABs en el dispositivo. Una señal de la interconexión columna, que tanto puede ser la salida de un LE o una entrada de un pin de *I/O*, debe ser ruteada a la interconexión fila antes de que pueda entrar un LAB o EAB. Cada canal de fila que es manejado por un IOE o EAB puede manejar un canal de columna específico.

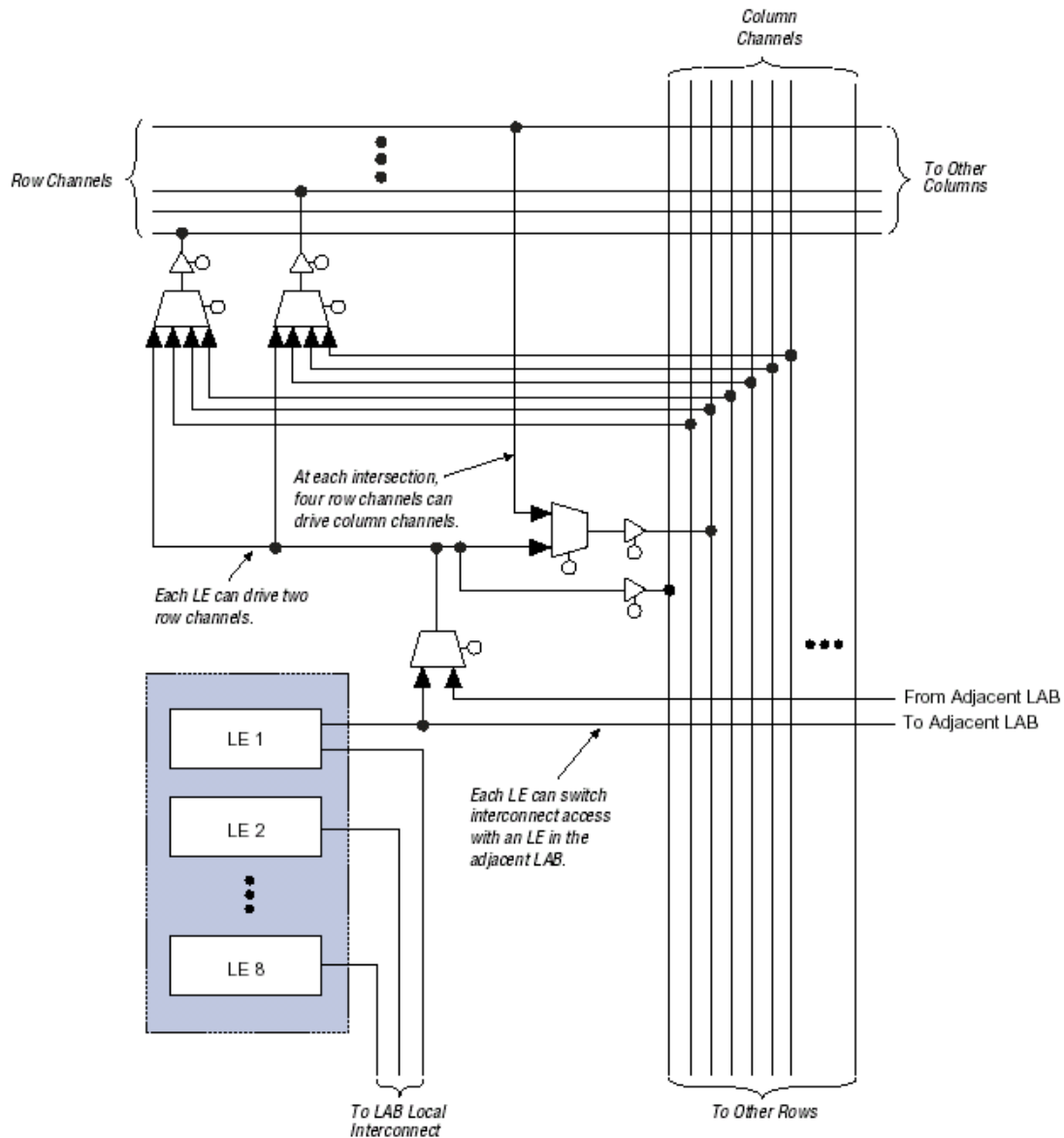


Figura 1.47: Interconexiones de los dispositivos FLEX10K

El acceso a canales de fila y columna puede ser conmutado entre LEs en los pares adyacentes de LABs. Por ejemplo, un LE en un LAB puede manejar los canales de fila y columna normalmente manejado por un LE particular en el LAB adyacente en la misma fila, y viceversa. Esta flexibilidad de ruteo permite usar los recursos de la asignación de ruta más eficazmente.

Para mejorar el ruteo, la interconexión fila está comprendida por una combinación de canales de longitud completa y longitud media. Los canales de longitud completa conectan a todos los LABs; los canales de longitud media conectan a los LABs por la mitad de la fila. El EAB puede ser manejado por los canales de longitud media en la mitad izquierda de la fila y por los canales de longitud completa. El EAB desemboca en los canales de longitud completa. Pueden conectarse dos LABs vecinos usando un canal de media fila, por lo tanto ahorrar la otra la mitad del canal para la otra la mitad de la fila.

Además de los pines de entrada/salida, los dispositivos *FLEX10K* disponen de seis pines de entrada específicos que propagan señales a través del dispositivo con muy bajo retardo. Estas seis entradas se pueden utilizar normalmente o

como señales de reloj, *clear*, *preset* y habilitación de salida periférica y de reloj. Estas señales están disponibles como señales de control para todos los LABs e IOEs en el dispositivo.

Las entradas especializadas también pueden usarse como entradas de datos de uso general porque pueden alimentar la interconexión local LAB en el dispositivo. Sin embargo, el uso de entradas especializadas como entradas de datos puede introducir retraso adicional en la red de señal de control.

La figura 1.48 muestra la interconexión de LABs adyacentes y EABs con interconexiones fila, columna, y local, así como las cadenas de *carry* y cascada asociadas. Cada LAB se etiqueta según su ubicación: una letra representa la fila y un número representa la columna. Por ejemplo, LAB B3 está en la fila B, columna 3.

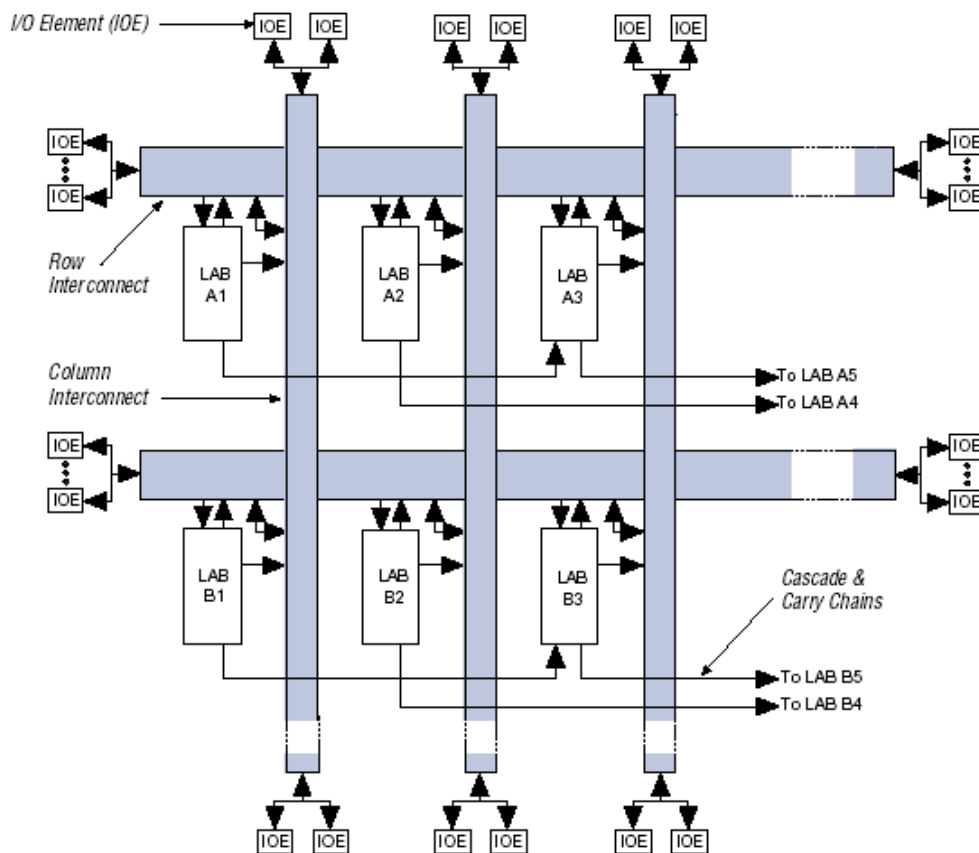


Figura 1.48: Recursos de Interconexión.

1.5.1.6 I/O Element

Un elemento de entrada/salida (IOE, *I/O Element*) integra un *buffer* bidireccional de entrada/salida y un registro de gran velocidad que se puede utilizar como un registro de entrada o salida de datos. En algunas ocasiones, la utilización de un LE para un registro de entrada puede producir tiempo de establecimiento (*setup*) más rápidos que usando un registro de IOE. IOEs pueden ser usados como entrada, salida, o pines bidireccionales. Para implementación de registro *I/O* bidireccional, el registro de salida debe estar en el IOE y, el registro de habilitación de entrada y salida de datos deben ser registros LE puestos adyacente al pin bidireccional. El compilador usa la opción

de inversión programable para invertir señales de la interconexión de fila y columna automáticamente donde sea apropiado. La figura 1.49 muestra los registros de I/O bidireccionales.

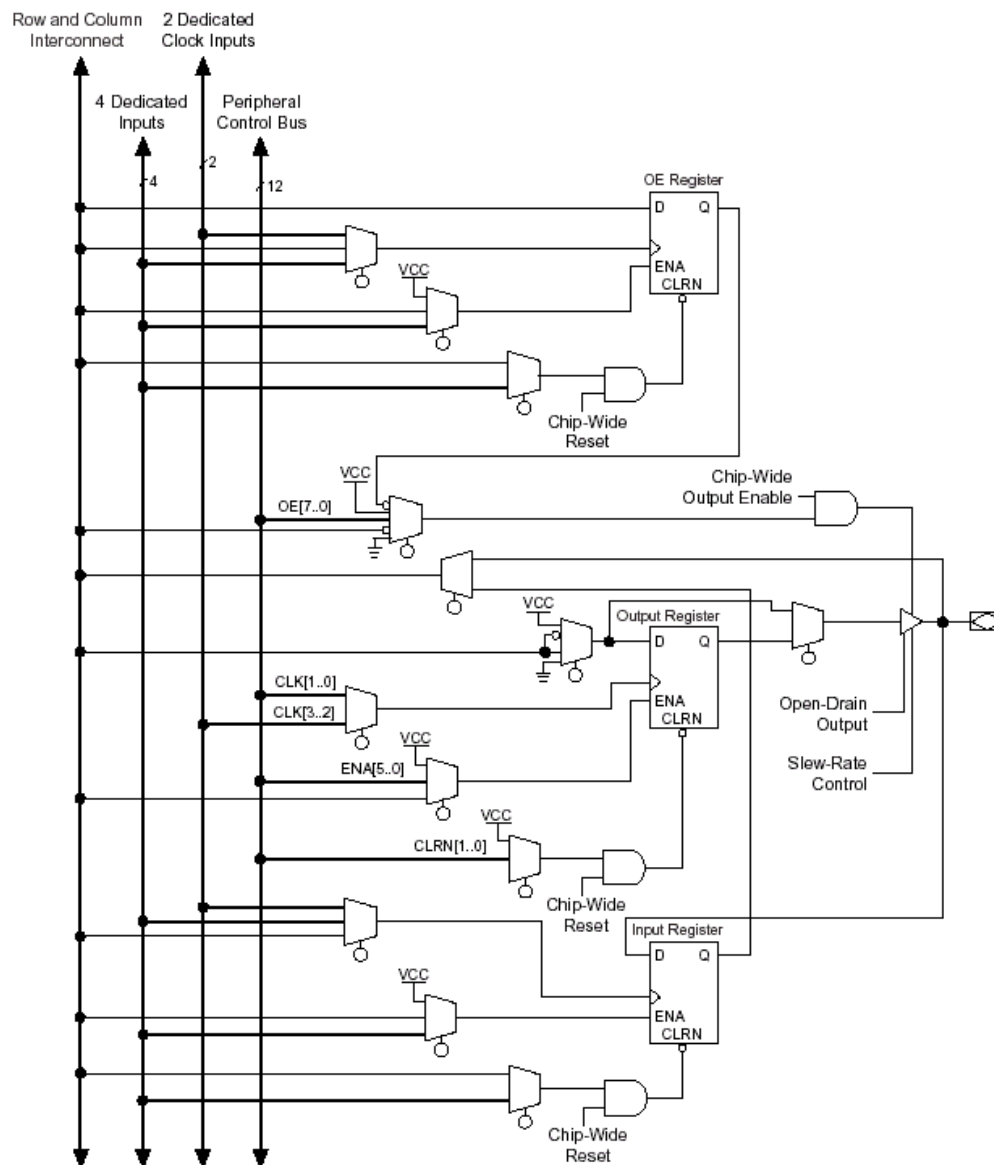


Figura 1.49: Registros de I/O bidireccionales.

Cada IOE selecciona los controles de reloj, *clear*, habilitación de reloj, y habilitación de salida de una red de señales de control de I/O llamado el bus de control periférico (*the peripheral control bus*). El bus de control periférico usa a los *drivers* de gran velocidad para minimizar la distorsión de señales a través de los dispositivos; proporciona hasta 12 señales del control periféricas como que pueden ser asignadas de la siguiente forma:

- Hasta ocho señales de habilitación de salida.
- Hasta seis señales de habilitación de reloj.
- Hasta dos señales de reloj.
- Hasta dos señales de *clear*.

Si son requeridas más de seis señales de habilitación de reloj u ocho de habilitación de salida, cada IOE en el dispositivo puede ser controlado por señales de habilitación de reloj y la de habilitación de salida manejadas por LEs específicos. Además de las dos señales de reloj disponibles en el bus de control periférico, cada IOE puede usar uno de dos pines de reloj especializados. Cada señal de control periférica puede ser manejada por cualquiera de los pines de entrada especializados o el primer LE de cada LAB en una fila particular. Además, un LE en una fila diferente puede manejar una interconexión columna, lo cual produce que una interconexión fila maneje la señal del control periférica.

1.5.2 LA FAMILIA APEX 20K

La familia de dispositivos *APEX 20K* de Altera representa una evolución de la arquitectura de los dispositivos *FLEX 10K*. Esta nueva familia incorpora nuevas características, mayor densidad y mejores prestaciones en velocidad. Los dispositivos *APEX 20K*, con hasta 1.500.000 puertas, se fabrican en tecnologías de 0.22 μ m, 0.18 μ m y 0.15 μ m.

Los dispositivos *APEX 20K* incorporan lógica basada en LUT, lógica basada en término producto y memoria, en un único dispositivo. Las interconexiones de señales dentro del dispositivo *APEX 20K* son proporcionadas por el FastTrack Interconnect.

La matriz de elementos lógicos agrupa los elementos lógicos en LABs de 10 LEs. Los bloques de memoria, ahora llamados ESBs (*Embedded System Blocks*), incorporan nuevas funcionalidades como por ejemplo la capacidad de definir memorias direccionable por el contenido (CAM, *Content Addressable Memory*). Cada ESB proporciona 2048 bits de memoria y se puede configurar como un elemento de memoria de 128 x 16, 256 x 8, 512 x 4, 1024 x 2 o 2048 x 1. Del mismo modo que para un dispositivo de la familia *FLEX 10K*, se pueden definir bloques de memoria mayores agrupando dos o más ESBs. El tamaño de estos ESBs es programable. La figura 1.50 muestra un diagrama en bloques de la Arquitectura de los dispositivos *APEX 20K*.

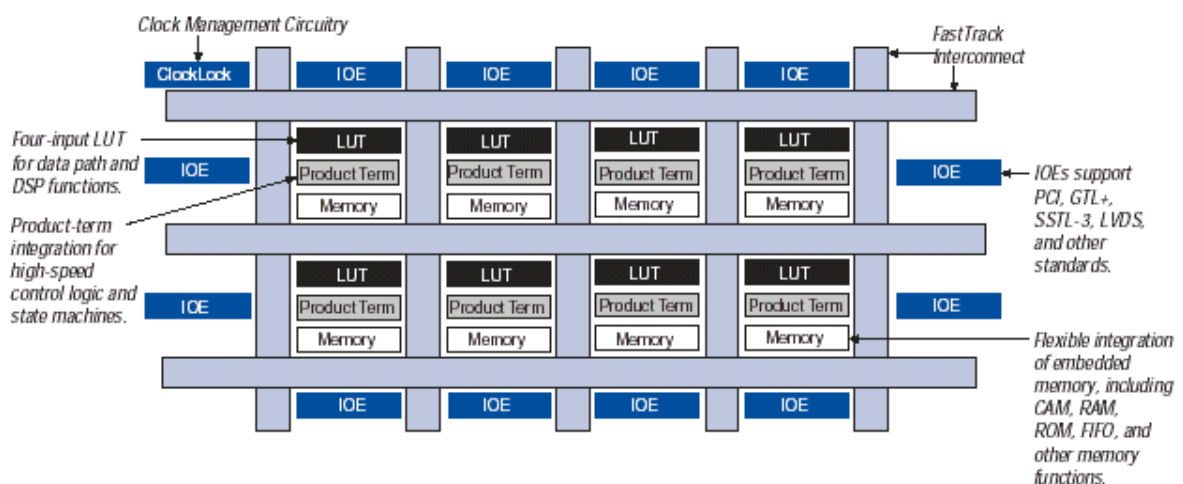


Figura 1.50: Arquitectura general de un dispositivo APEX 20K

La tabla 1.7 muestra algunos miembros de la familia *APEX 20K* y sus principales características.

Dispositivo	LEs	ESBs	Bits de memoria	Entradas Salidas
EP20K60E	2260	16	32768	504
EP20K100(E)	4160	26	53248	252
EP20K160E	6400	40	81920	316
EP20K200(E)	8320	52	106496	382
EP20K300E	11520	72	147456	408
EP20K400(E)	16640	104	212992	502
EP20K600E	24320	152	311296	624
EP20K1000E	38400	160	327680	708
EP20K1500E	51840	216	442368	808

Tabla 1.7: Dispositivos de la familia APEX 20K

1.5.3 LA FAMILIA ACEX 1K

La arquitectura de los dispositivos *ACEX 1K* es similar a la de los dispositivos *FLEX 10K*. Están integrados por bloques de memoria llamados EABs y una matriz de elementos lógicos contituída por bloques lógicos (LABs). Cada LAB agrupa 8 elementos lógicos (LEs) e interconexiones locales. Un LE incluye una tabla de look-up de 4 entradas, un *flip-flop* programable y lógica para la rápida generación y propagación de acarreo y la conexión en cascada. Las conexiones entre elementos de memoria y elementos lógicos se realizan mediante el llamado *FastTrack Interconnect*.

Cada pin de Entrada/Salida se alimenta por un elemento de Entrada/Salida (IOE) localizado al final de cada fila y columna del *FastTrack Interconnect*. Cada IOE contiene un *buffer* de Entrada/Salida bidireccional y un *flip-flop* que pueden usarse tanto como registro de la salida o entrada para alimentar la entrada, la salida, o las señales bidireccionales.

La tabla 1.8 muestra los recursos que integran algunos de los miembros de la familia de dispositivos *ACEX 1K*.

Dispositivo	LEs	EABs	Bits de memoria	Entradas Salidas
EP1K10	576	3	12288	136
EP1K30	1728	6	24576	171
EP1K50	2880	10	40960	249
EP1K100	4992	12	49152	333

Tabla 1.8: Dispositivos de la familia ACEX 1K

La principal diferencia con los dispositivos *FLEX 10K*, es que en los dispositivos *ACEX 1K* el EAB también puede usarse para aplicaciones de memoria bidireccional de doble puerto, donde dos puertos leen o escriben simultáneamente o sea se permite el acceso simultáneo de dos procesadores al mismo bloque de memoria. Para llevar a cabo este tipo de memoria, se usan dos EABs para apoyar dos lecturas o escrituras simultáneas.

Alternativamente, un reloj y habilitación de reloj pueden usarse para controlar el registro de entrada del EAB, mientras un reloj diferente y habilitación de reloj controlan el registro de salida (vea figura 1.51).

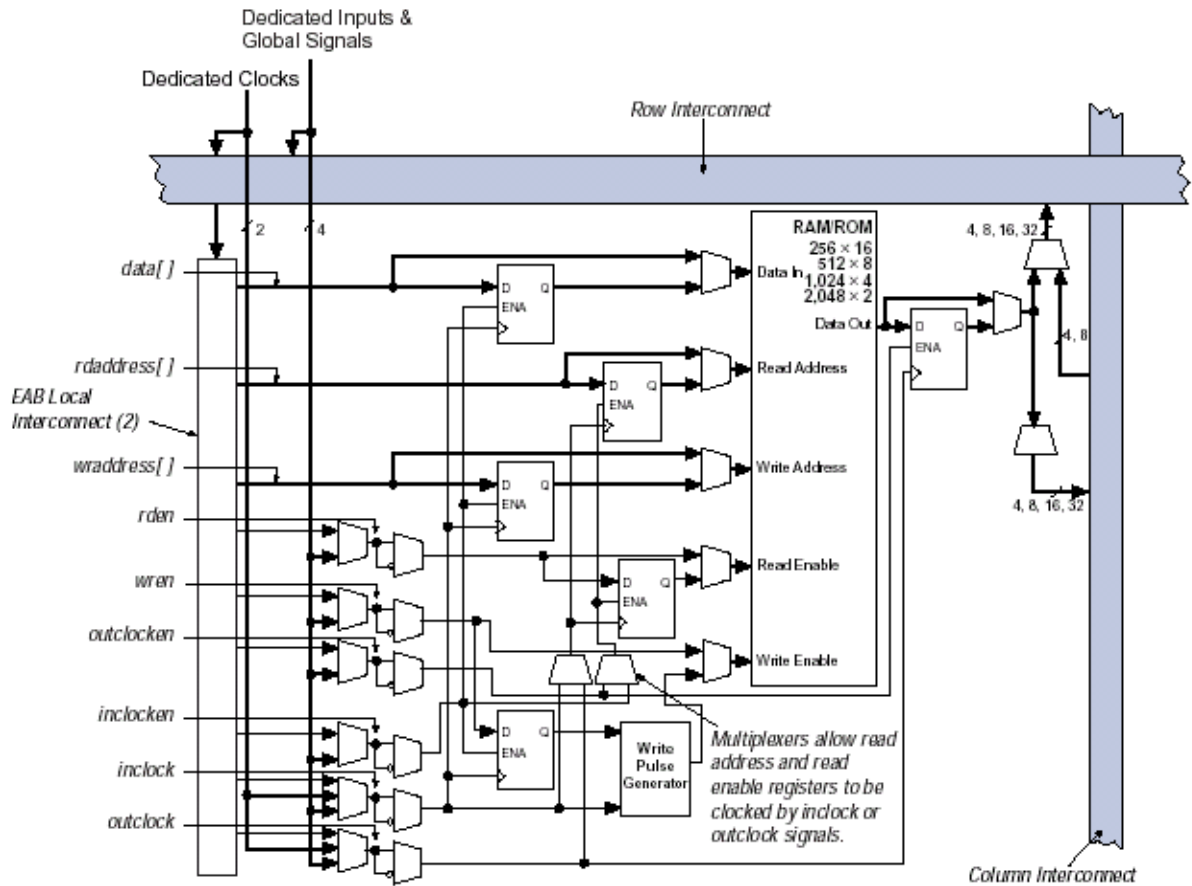


Figura 1.51: Dispositivos ACEX 1K en modo RAM de doble puerto

El EAB puede usar megafunciones de Altera para implementar aplicaciones de RAM de doble puerto donde ambos puertos pueden leer o escribir, como se muestra en figura 1.52.

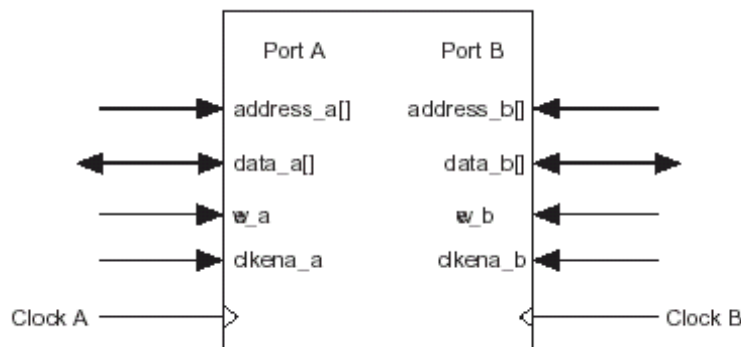


Figura 1.52: EAB del ACEX 1K en modo RAM de doble puerto

El EAB del ACEX 1K también puede usarse en un modo del puerto simple tal como muestra la figura 1.53.

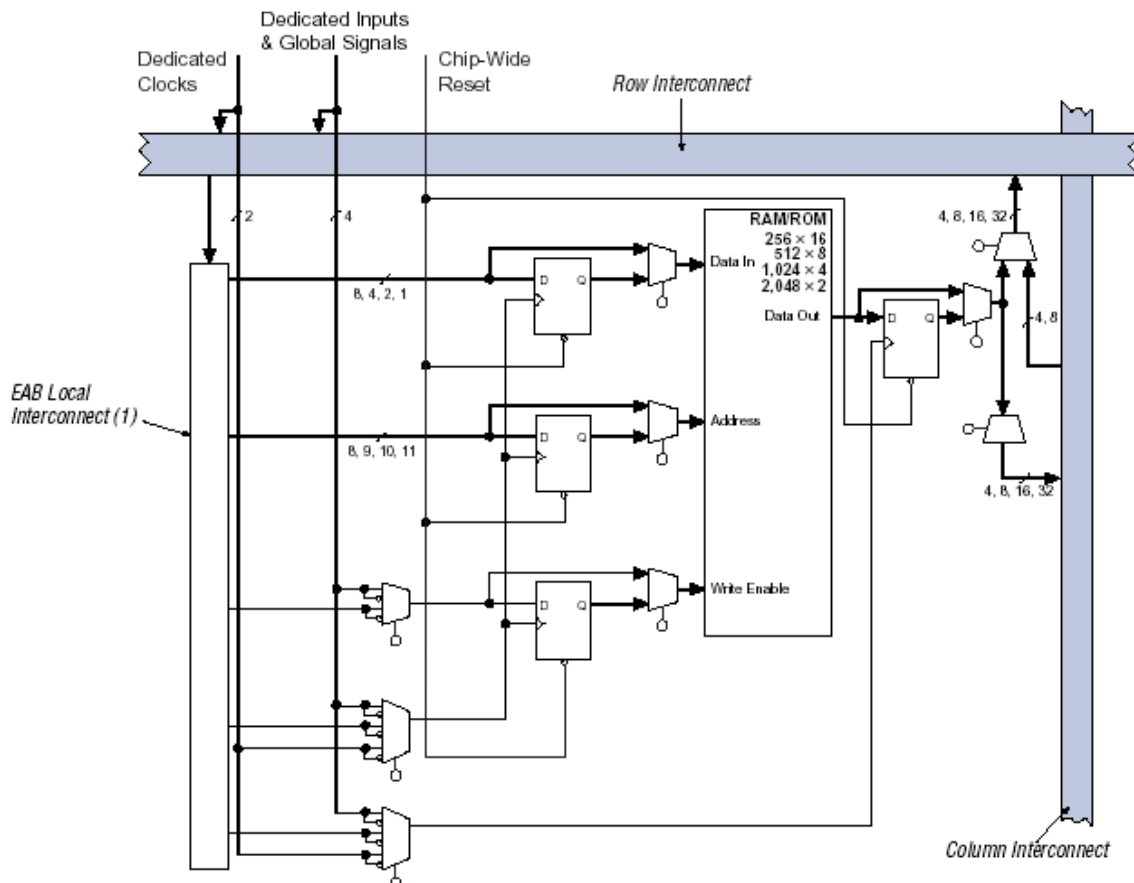


Figura 1.53: Dispositivos ACEX 1K en modo RAM de puerto simple

1.5.4 Familia Cyclone

Las FPGAs *Cyclone* 1.5 V, utilizan tecnologías de $0.13\mu\text{m}$ con densidades de hasta 20,060 elementos lógicos (LEs, *Logic Elements*) y hasta 288 Kbits de RAM.

Los dispositivos *Cyclone* contienen una arquitectura basada en fila y columna de dos dimensiones para implementar lógica. Las interconexiones columna y fila de distintas velocidades proporcionan señales de interconexión entre los LABs y los bloques integrados de memoria.

El arreglo lógico está constituido por LABs, con 10 LEs en cada LAB. Un LE es una unidad pequeña de lógica que proporciona una aplicación eficaz de funciones lógicas. Los LABs se agrupan en filas y columnas a través del dispositivo. Los dispositivos *Cyclone* tienen entre 2910 a 20060 LEs.

Los bloques M4K RAM son los verdaderos bloques de memoria de doble puerto con 4K bits de memoria más la paridad (4608 bits). Estos bloques proporcionan memorias de doble puerto especializado, doble puerto simple, o de un puerto de hasta 36 bits de ancho llegando a los 200 MHz. Estos bloques se agrupan en columnas a través del dispositivo entre ciertos LABs. Los dispositivos *Cyclone* ofrecen entre 60 y 288 Kbits de RAM integrada.

Cada pin E/S del dispositivo *Cyclone* es alimentado por un elemento de E/S (IOE) ubicado en los finales de las filas y columnas del LAB, alrededor de la periferia del dispositivo. Cada IOE contiene un *buffer* de E/S bidireccional y tres

registros para registrar señales de entrada, de salida, y señales de habilitación de salida.

Los dispositivos *Cyclone* proporcionan una red de reloj global y hasta dos PLLs. La red de reloj global consiste en ocho líneas de relojes globales que recorren el dispositivo entero. La red de reloj global puede proveer a relojes a todos los recursos dentro del dispositivo, como IOEs, LEs, y bloques de memoria. Las líneas de relojes globales también pueden usarse para señales del control. La figura 1.54 muestra un diagrama arquitectura de los dispositivos EP1C12 de la familia *Cyclone*.

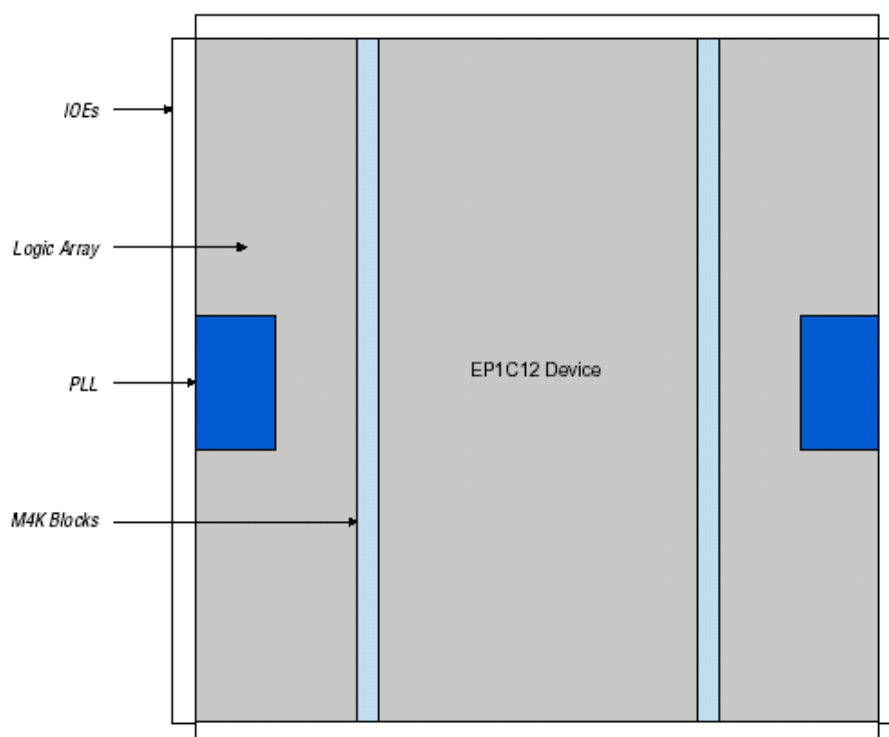


Figura 1.54: Arquitectura de un dispositivo EP12C12

La tabla 1.9 muestra los recursos que integran algunos de los miembros de la familia de dispositivos *Cyclone*.

Dispositivo	LEs	Bloques M4K RAM(128x36)bits	Bits de memoria	PLLs	Entradas Salidas
EP1C3	2910	13	59904	1	104
EP1C4	4000	17	78336	2	301
EP1C6	5380	20	92160	2	185
EP1C12	12060	52	239616	2	249
EP1C20	20060	64	294912	2	301

Tabla 1.9: Dispositivos de la familia Cyclone

1.5.5 Familia Stratix

Las FPGAs *Stratix 1.5 V*, utilizan tecnologías 0.13 μ m con densidades de hasta 79,040 elementos lógicos (LEs) y hasta 7.5 Mbits de RAM. Los dispositivos *Stratix* ofrecen hasta 22 bloques de Procesamiento de Señales Digitales (*Digital Signal Processing blocks*) con hasta 176 multiplicadores integrados de 9 \times 9 bits, optimizados para aplicaciones DSP que permiten implementar eficientemente filtros y multiplicadores de alta performance. Los dispositivos *Stratix* soportan varias normas de E/S y también ofrecen, con su estructura del reloj jerárquica, una solución completa de manejo de reloj llegando a los 420MHz y hasta 12 PLLs.

Al igual que los *Cyclone*, los dispositivos *Stratix* contienen una arquitectura basada en fila y columna de dos dimensiones para implementar lógica. Las interconexiones columna y fila de velocidades y longitudes variantes proporcionan señales de interconexión entre LABs, estructuras de bloque de memoria, y bloques DSP (*Digital Signal Processing*, Procesamiento de Señales Digitales).

El arreglo lógico está constituido por LABs, con 10 LEs en cada LAB. Como sabemos un LE es una unidad pequeña de lógica que proporciona una aplicación eficaz de funciones lógicas. Los LABs se agrupan en filas y columnas a través del dispositivo.

Los bloques M512 RAM son simples bloques de memoria de doble puerto con 512 bits de memoria más la paridad (576 bits). Estos bloques proporcionan memorias de doble puerto simple o de un puerto de hasta 18 bits de ancho llegando a los 318 MHz. Estos bloques se agrupan en columnas a través del dispositivo entre ciertos LABs.

Los bloques M4K RAM son verdaderos bloques de memoria de doble puerto con 4Kbits de memoria más la paridad (4608 bits). Estos bloques proporcionan memorias de doble puerto especializado, doble puerto simple, o de un puerto de hasta 36 bits de ancho llegando a los 291 MHz. Estos bloques se agrupan en columnas a través del dispositivo entre ciertos LABs.

Los bloques M-RAM son verdaderos bloques de memoria de doble puerto con 512Kbits de memoria más la paridad (589824 bits). Estos bloques proporcionan memorias de doble puerto especializado, doble puerto simple, o de un puerto de hasta 144 bits de ancho llegando a los 269 MHz. Algunos bloques M-RAM se localizan individualmente o en pares dentro del arreglo lógico del dispositivo.

Los bloques de Procesamiento de Señales Digitales (DSP blocks) pueden implementar ocho multiplicadores de 9 \times 9 bits de precisión completa, cuatro multiplicadores de 18 \times 18 bits de precisión completa, o un multiplicador de 36 \times 36 bits de precisión completa con características de adición y sustracción. Estos bloques también contienen registros de cambios de entrada de 18 bits para aplicaciones del procesamiento de señales digitales, incluyendo filtros de respuesta a impulsos infinitos (IIR, *Infinite Impulse Response*). Los bloques DSP se agrupan en dos columnas en cada dispositivo.

Cada pin E/S del dispositivo *Stratix* es alimentado por un elemento de E/S (IOE) ubicado en los finales de las filas y columnas del LAB, alrededor de la periferia del dispositivo. Cada IOE contiene un *buffer* de E/S bidireccional y seis registros para registrar señales de entrada, de salida, y señales de habilitación de salida. Cuando son usados con relojes especializados, estos registros proporcionan una excepcional performance y sustento de la interfaz con

dispositivos de memoria externos como DDR SDRAM, FCRAM, ZBT, y QDR SRAM.

La Figura 1.55 muestra un diagrama arquitectura de los dispositivos *Stratix*.

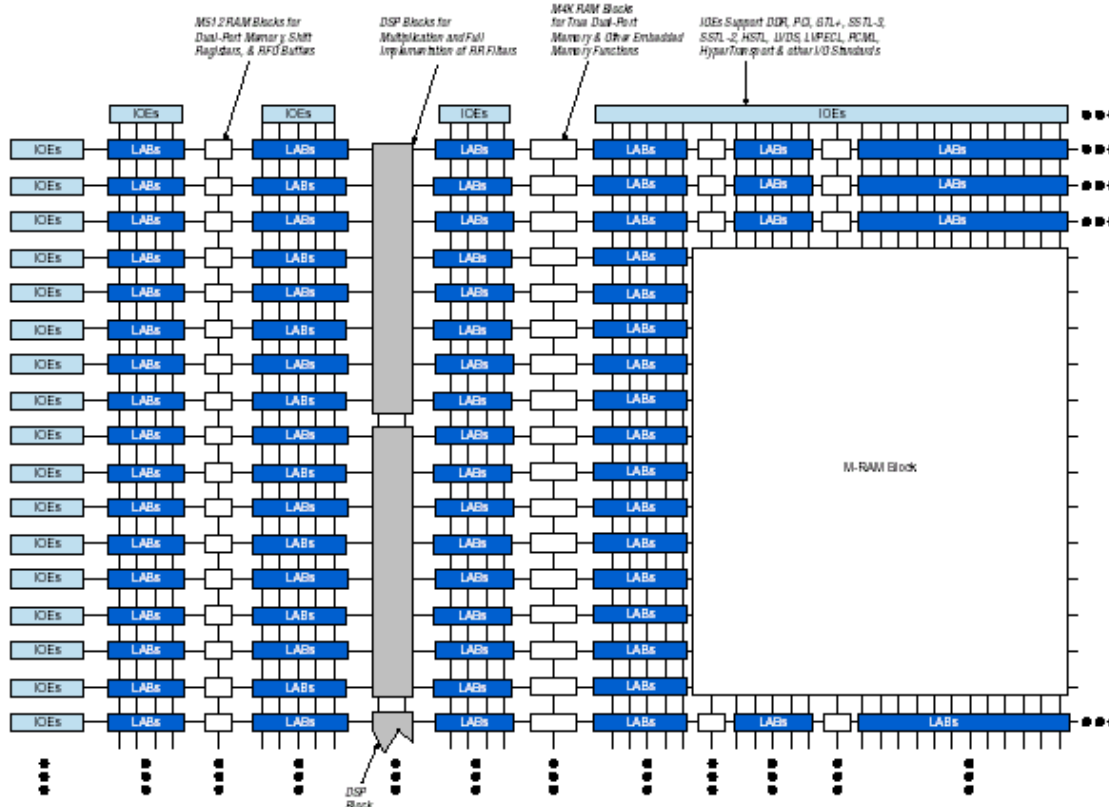


Figura 1.55: Diagrama en bloques de un dispositivo Stratix

La tabla 1.10 muestra los recursos que integran algunos de los miembros de la familia de dispositivos *Stratix*.

Dispositivo	LEs	Bloques M512.RAM (32x18)bits	Bloques M4K RAM (128x36)bits	Bloques M-RAM (4Kx144)bits	bits de memoria	bloques DSP	PLLs	Entradas Salidas
EP1S10	10570	94	60	1	920448	6	6	426
EP1S20	18460	194	82	2	1669248	10	6	586
EP1S25	25660	224	138	2	1944576	10	6	706
EP1S30	32470	295	171	4	3317184	12	10	726
EP1S40	41250	384	183	4	3423744	14	12	822
EP1S60	57120	574	292	6	5215104	18	12	1022
EP1S80	79040	767	364	9	7427520	22	12	1238

Tabla 1.10: Dispositivos de la familia Stratix

1.6 Arquitectura de los Dispositivos FPGAs de Actel

La arquitectura fundamental de una FPGA de Actel es muy similar a un arreglo de compuertas convencional. La esencia del dispositivo consiste en módulos lógicos simples usados para implementar compuertas de lógica y elementos de almacenamiento. Éstos módulos lógicos son interconectados por una gran cantidad de pistas de ruteado segmentadas (*Segmented Routing Tracks*). A diferencia de los arreglos de compuertas, las longitudes del segmento son predefinidas y pueden ser conectadas con elementos de conmutación (*switching*) de baja-impedancia para crear la longitud precisa de ruteado requerida por la señal de interconexión. Rodeando el centro de la lógica esta la interfaz a los bloques de entrada/salida de los dispositivos. Esta interfaz consiste en módulos de E/S que traducen e interconectan las señales lógicas del centro del dispositivo a los pines de salida del FPGA. Un diagrama en bloques de una FPGA Actel genérica se da en la figura 1.56.

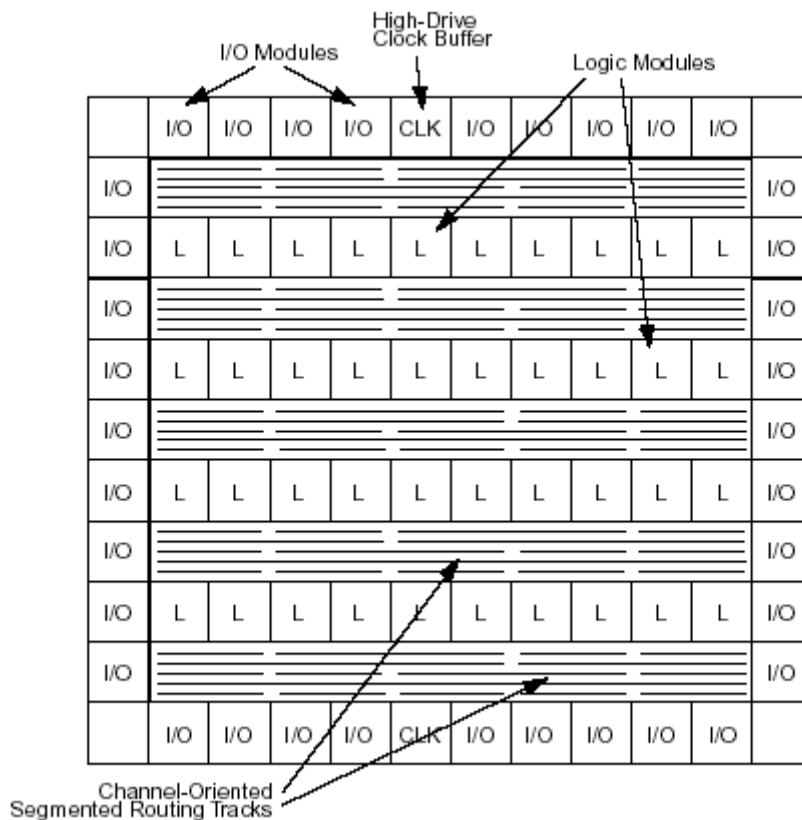


Figura 1.56: Arquitectura básica de una FPGA Actel

Los elementos principales de la arquitectura FPGA de Actel son los módulos de E/S, los recursos de interconexión, los recursos de reloj, y los módulos de lógica. Cada familia FPGA de Actel tiene una mezcla ligeramente diferente de estos recursos, perfeccionados para diferentes costos, performance, y requisitos de densidad. La tabla 1.11 muestra las capacidades de cada familia FPGA de Actel. Cada capacidad es definida en las secciones siguientes.

Capability	ACT 1	ACT 2/1200XL	3200DX	ACT 3
Core Module	Simple Logic Module	Combinatorial and Sequential Modules	Combinatorial and Sequential Modules Wide Decode Modules Embedded Dual-Port SRAM	Combinatorial and Enhanced Sequential Modules
Interconnect	Channeled	Channeled	Channeled	Channeled
Clocking Resources	Routed Clock (1)	Routed Clocks (2)	Routed Clocks (2) Quad Clocks (4)	Routed Clocks (2) Dedicated Array Clock Dedicated I/O Clock
I/O Module	Simple I/O Module	Latched I/O Module	Latched I/O Module	Registered I/O Module

Tabla 1.11: Características de la arquitecturas de las familias de Actel

1.6.1 Módulos lógicos

El módulo lógico óptimo debe proveer al usuario con la mezcla correcta de performance, eficiencia, y facilidad de diseño exigida para llevar a cabo la aplicación. El módulo lógico óptimo debe equilibrar estos intercambios cuidadosamente para asegurar que las muchas metas contradictorias del diseñador sean logrables.

1.6.1.1 Módulos Lógicos Simples

El primer módulo lógico Actel era el Módulo Lógico Simple (*Simple Logic Module*) usado en la familia ACT 1, el cual se muestra en la figura 1.57. Es un módulo lógico basado en multiplexer (*multiplexer-based logic module*). Las funciones lógicas son implementadas interconectando señales de las pistas de ruteado (*routing tracks*) a las entradas de datos y líneas de selección de los multiplexores. Las entradas también pueden fijarse a un '1' o '0' lógico si es requerido, ya que estas señales siempre están disponibles en el canal de ruteado (*routing channel*).

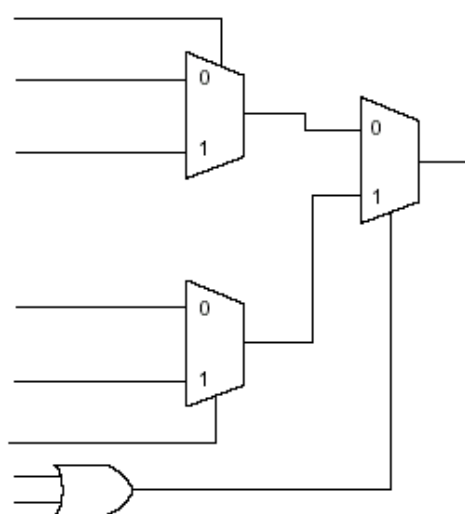


Figura 1.57: Módulo Lógico Simple

Un número sorprendente de funciones lógicas útiles puede ser implementadas con este módulo. Claramente, el multiplexar es muy eficiente, pero las funciones lógicas aleatorias y secuenciales también son eficientes. Estas opciones le proporcionan al diseñador una mezcla excelente de capacidades de lógica, requerida para aplicaciones que exigen una variedad de funciones lógicas. La figura 1.58 muestra un ejemplo de una función lógica implementada con el Módulo Lógico Simple de Actel. Notar que pueden ser implementados *latches* en un solo módulo lógico por *bit* y que los registros requieren dos módulos lógicos por *bit*. El módulo lógico ACT 1 es así sumamente flexible cubriendo una gama amplia de mezclas de lógica combinatorial y secuencial.

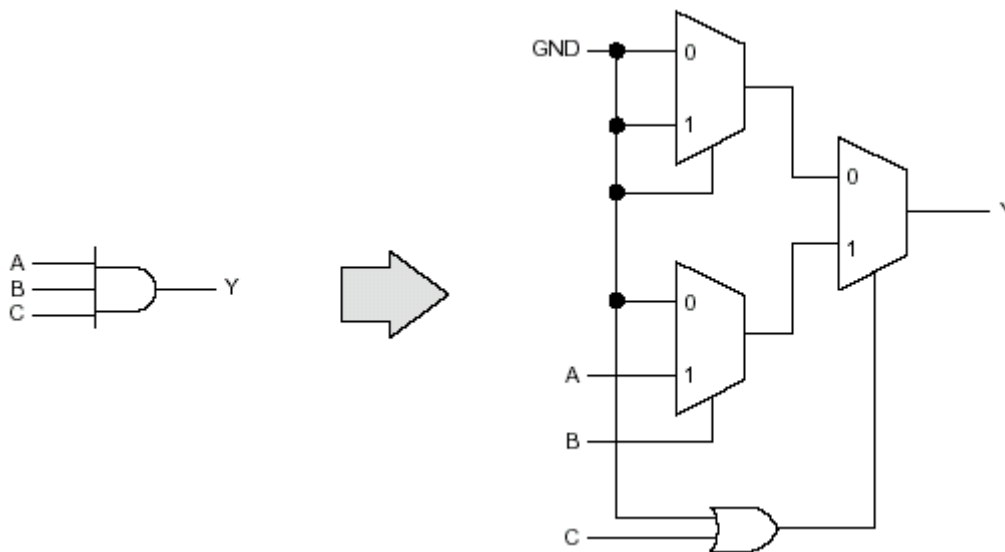


Figura 1.58: Ejemplo de una función lógica implementada con el Módulo Lógico Simple ACT 1

1.6.1.2 Módulos Lógicos Combinacionales

Se realizaron algunas mejoras al Módulo Lógico Simple cuando fue desarrollada la segunda generación de la familia ACT 2. El Módulo Lógico Simple fue reemplazado por dos módulos lógicos diferentes, uno para implementar la lógica combinatorial, (el Módulo Lógico Combinacional, *the Combinatorial Logic Module*) y uno para implementar elementos de almacenamiento (el Módulo Lógico Secuencial, *the Sequential Logic Module*). El Módulo Lógico Combinacional, mostrado en el diagrama en la figura 1.59, es similar al Módulo Lógico Simple, pero fue colocada una compuerta lógica adicional en el primer-nivel del multiplexor. La compuerta agregada mejora la aplicación de algunas funciones combinatoriales. También, las líneas del primer-nivel del multiplexor en el Módulo Lógico Simple fueron combinadas en el Módulo Lógico Combinacional. En el Módulo Lógico Simple, las líneas de selección del multiplexor separadas fueron usadas para implementar eficientemente *latches* y registros. Esto no fue requerido en el Módulo Lógico Combinacional debido a la suma del Módulo Lógico Secuencial. La figura 1.60 muestra un ejemplo de una función lógica implementada con el Módulo Lógico Combinacional.

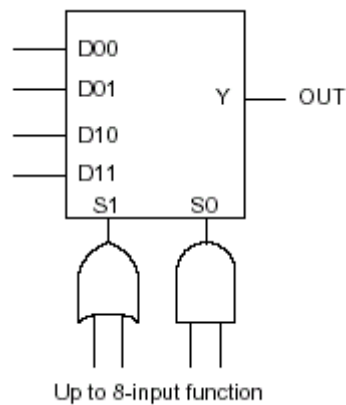


Figura 1.59: Módulo Lógico Combinacional

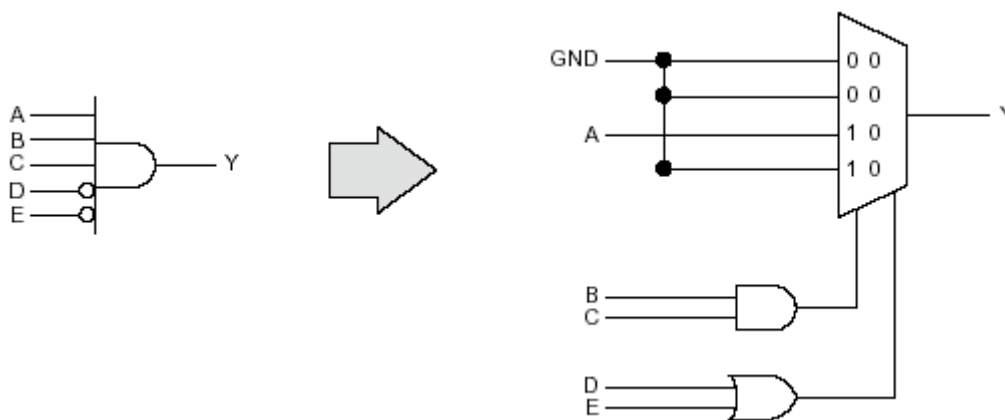
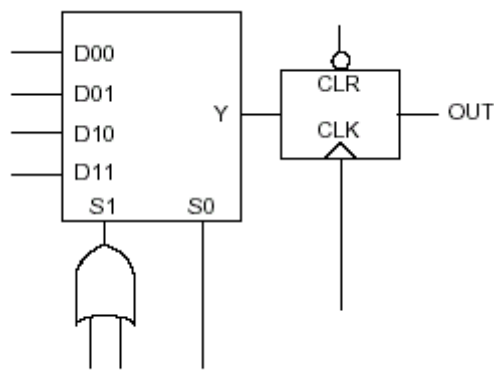


Figura 1.60: Ejemplo de una función lógica implementada con el Módulo Lógico Combinacional.

1.6.1.3 Módulos Lógicos Secuenciales

El Módulo Lógico Secuencial, mostrado en la figura 1.61, tiene una lógica combinacional con un elemento del almacenamiento especializado en la salida del módulo lógico. El elemento de almacenamiento puede ser un registro o un *latch* (también puede ser "bypaseado" (esquivado) para que el módulo lógico pueda ser usado como un Módulo Lógico Combinacional). La entrada de reloj puede ser seleccionada para ser tanto activa en alto como activa en bajo. Falta una de las compuertas lógicas en la sección de lógica combinacional, haciéndolo ligeramente diferente del Módulo Lógico Combinacional. La exclusión de esta sola compuerta lógica permite a la señal de *reset*, que es compartida con la sección de lógica combinacional, estar disponible al elemento de almacenamiento sin aumentar el número de entradas del módulo totales requeridos. Si el elemento de almacenamiento es *bypaseado*, la señal *reset* es usada para implementar la entrada del módulo combinacional requerida. En la Serie Integrador, los módulos secuenciales y combinacionales se entrelazan, resultando una mezcla 50-50 de módulos lógicos. Esto ha sido determinado para hacer una mezcla óptima para una ancha variedad de diseños y resultados en una utilización excelente.



Up to 7-input function plus D-type flip-flop with clear

Figura 1.61: Módulo Lógico Secuencial

1.6.1.4 Módulo Lógico de Decodificación Amplia (*Wide Decode Logic Module*)

Cada miembro de la familia 3200DX tiene varios módulos de la lógica especiales optimizados para implementar funciones lógicas combinacionales de entrada amplia (*wide-input combinatorial logic functions*) directamente manejando los bloques de salida del dispositivo. El *Wide Decode Logic Module* consiste en una compuerta AND de siete entrada con habilitación de inversión de salida. La salida de este módulo "bypasea" (esquiva) la red de ruteado normal y conecta directamente a un *buffer* de salida particular. Este rasgo minimiza el retraso de la salida del módulo al bloque del dispositivo y es ideal para implementar funciones de decodificación amplia implementadas típicamente en los dispositivos PLDs pequeños. La salida del *Wide Decode Logic Module* también está disponible a los módulos lógicos de centro, para que pueda usarse junto con otras funciones lógicas interiores al dispositivo.

1.6.1.5 Integrado SRAM de doble puerto (*Embedded Dual-Port SRAM*)

Algunos miembros de la familia 3200DX incluyen bloques especializados SRAM de doble puerto de gran velocidad. Los bloques SRAM son colocados en bloques del 256 bits, que pueden ser configurados como 32 x 8 o 64 x 4. Los bloques SRAM pueden ser puestos en cascada para formar bloques de memoria profundos o extensos. El SRAM es de doble puerto, con direcciones de lectura y escritura separadas, un puerto de entrada de datos separado (para escritura), y un puerto de salida de datos separado (para lectura). Las lecturas y escrituras son controladas por habilitaciones de lectura y escritura con reloj separadas, facilitando los tiempos requeridos para usar el SRAM. La estructura de puerto-dual es ideal para implementar FIFO, *buffers* de ráfagas, y registros internos de almacenamiento para estados, control, o datos constantes. Los dispositivos 3200DX tienen desde 8 bloques SRAM (en el A32100DX) a 16 bloques SRAM (en el A32400DX).

1.6.1.6 Módulo Lógico Secuencial Mejorado (*Enhanced Sequential Logic Module*)

El Módulo Lógico Secuencial Mejorado usado en la familia ACT 3 es una sutileza del Módulo Lógico Secuencial y se muestra en la figura 1.62. La entrada *reset* del registro en la sección secuencial no es compartido con la función de lógica combinacional, entonces el total de la lógica combinacional está disponible en el Módulo Lógico Combinacional para ser usado delante del registro.

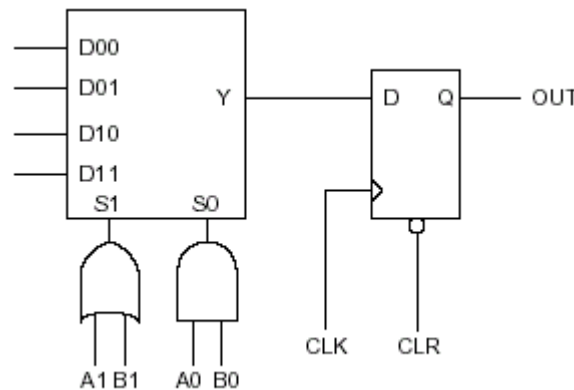


Figura 1.62: Módulo Lógico Secuencial Mejorado

1.6.2 Interconexión por canal (*Channeled Interconnect*)

Todos los dispositivos de Actel usan una arquitectura interconexión por canal para hacer conexiones entre los módulos lógicos interiores y pines de E/S del dispositivo. Esto hace que estén disponibles una cantidad grande de recursos de ruteado y normalmente asegura que las señales tienen la longitud de pista disponible que ellos requieren. Además, pueden unirse las pistas juntas para construir las pistas más largas, cuando se requiere, programando un fusible de interconexión. Las salidas del módulo lógico abarcan cuatro canales (dos sobre y dos debajo) y pueden ser conectadas a cualquier pista. Esto significa que la mayoría de las señales exigen sólo dos fusibles para conectar cualquier salida de módulo lógico a cualquier entrada del mismo. Hay bastantes recursos de ruteado disponible en los dispositivos de Actel entonces colocar y dirigir (*place and route*) es una tarea automática. Ningún ruteado a mano se requiere.

1.6.3 Recursos de reloj

Los dispositivos de Actel tienen una gama amplia de flexibilidad de reloj. La entrada de reloj de cada elemento secuencial puede ser conectada a interconexiones regulares dentro del canal, así como a recursos de reloj optimizados. Las interconexiones regulares ofrecen la mayor flexibilidad, permitiendo miles de potenciales relojes separados. Cada dispositivo de Actel tiene también recursos de reloj dedicados en el interior del chip para mejorar la performance del reloj y para simplificar el diseño de señales secuenciales. Los

recursos de reloj también pueden ser usados, en la mayoría de los casos, como señales globales *high-drive* tales como *reset*, habilitación de salida, o señales de selección. Cada familia FPGA es ligeramente diferente en la manera que implementa funciones de reloj.

1.6.3.1 Relojes ruteados (*Routed Clocks*)

Todas las familias FPGAs de Actel tiene uno o dos *buffers* especiales que proporcionan señales de baja distorsión y *high-drive*; y eso puede ser usado para manejar cualquier señal que requiera estas características. Estos relojes ruteados se distribuyen para cada canal de ruteado y están disponible para cada módulo lógico. Esto permite usar una señal del reloj ruteado por ambos Módulos Lógicos (Secuencial y Combinacional), ofreciendo máxima flexibilidad con ligeramente más baja performance que los relojes especializados.

1.6.3.2 Arreglo de reloj especializado (*Dedicated Array Clock*)

La familiar ACT 3 tiene un recurso de reloj adicional que consiste en un *buffer* de reloj especializado de gran velocidad perfeccionado para manejar los módulos secuenciales en el arreglo central. Este *buffer* de reloj puede ser manejado desde un pin externo o desde una señal interior. El arreglo de reloj especializado es optimizado para manejar módulos secuenciales y no puede manejar elementos de almacenamiento construidos por los módulos combinacionales.

1.6.3.3 Reloj Entrada/Salida Especializado (*Dedicated I/O Clock*)

La familia ACT 3 tiene otro recurso de reloj que consiste en un *buffer* de reloj especializado de gran velocidad perfeccionado para manejar los módulos secuenciales en los módulos de E/S. El reloj de E/S especializado es optimizado para manejar los módulos de E/S y no puede manejar los elementos de almacenamiento en el arreglo. Si todos los elementos de almacenamiento necesitan ser manejados por un reloj común, pueden ser conectados juntos externamente el reloj del arreglo y reloj de E/S.

1.6.3.4 Relojes de Cuadrantes (*Quad Clocks*)

La familia 3200DX tiene un recurso de reloj adicional que consiste en cuatro *buffers high drive* especiales llamados relojes de cuadrante. Cada *buffer* proporciona una señal *high drive* que abarca un cuarto del dispositivo (un cuadrante). Estos *buffers* pueden ser usados para relojes locales rápidos (quizás para cambiadores de escala o contadores), para selección de mux, o para habilitación de E/S. Desde que éstos son cuadrantes orientados, sólo un único reloj de cuadrante puede usarse por el cuadrante. *Quad clocks* pueden ser conectados juntos internamente para abarcar hasta una mitad del

dispositivo. Adicionalmente, los *quad clocks* pueden ser alimentados por las señales internas así como por pines externos. Así ellos pueden ser usados como redes de alto *fan-out* manejadas internamente.

1.6.4 Módulos E/S

Cada familia FPGA de Actel tiene un módulo de E/S ligeramente diferente. El Módulo de E/S Simple (*Simple I/O Module*), encontrado en la familia ACT 1, es optimizado para bajo costo, y el Módulo de E/S con Latches (*Latched I/O Module*), encontrado en la Serie Integrador, ofrece un equilibrio entre velocidad y costo (valor). El Módulo de E/S con Registros (*Registered I/O Module*) en la familia ACT 3 es optimizado para alta velocidad en aplicaciones sincrónicas.

1.6.4.1 Módulo de E/S Simple (*Simple I/O Module*)

El Módulo de E/S Simple (figura 1.63) usado en la familia ACT 1 fue el primer módulo de E/S desarrollado por Actel y es un simple *buffer* de E/S con interconexiones al arreglo de lógica. Todas las señales de entrada, salida, y de control de TRI-STATE están disponibles para el arreglo. Las salidas son compatibles con TTL y CMOS y el sumidero o la fuente de 10 mA a niveles de TTL.

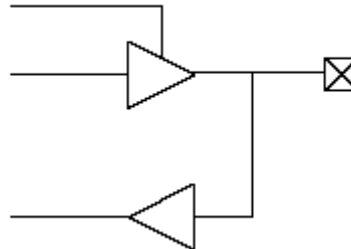


Figura 1.63: Módulo de E/S Simple

1.6.4.2 Módulo de E/S con Latches (*Latched I/O Module*)

El Módulo de E/S con Latches, mostrado en la figura 1.64, se usa en la Serie Integrador y se complica ligeramente más que el Módulo de E/S Simple. El *Latched I/O Module* contiene *latches* de entrada y salida que pueden ser usados como a tal o pueden combinarse con *latches* internos para construir registros de entrada o salida. Las salidas son compatibles con TTL y CMOS y el sumidero o la fuente de 10 mA a niveles de TTL.

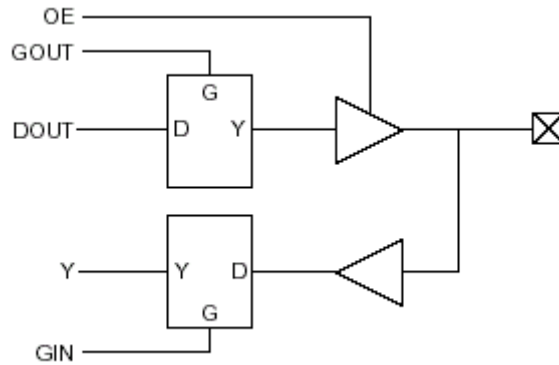


Figura 1.64: Módulo de E/S con Latches

1.6.4.3 Módulo de E/S con Registros (*Registered I/O Module*)

El Módulo de E/S Registrado, usado en la familia de dispositivos ACT 3, es perfeccionado para velocidad y funcionalidad en los diseños de sistema sincrónicos. Contiene registros completos tanto en los caminos de entrada como de salida, tal como muestra la figura 1.65. Los datos pueden ser guardados en el registro de salida (bajo el control del ODE, señal de habilitación de datos de salida, *Output Data Enable signal*), o para "bypasear" el registro si el *bit* de control OTB es puesto en bajo. Tanto el registro de salida como el registro de entrada puede ser limpiados (*cleared*) o prefijados (*preset*) vía la señal global IOPCL, y los dos son sincronizados vía la señal global IOCLK. La salida del registro de salida puede ser seleccionada como una entrada al arreglo (en la señal de Y). Esto permite a las máquinas de estados, por ejemplo, ser construídas correctamente en el módulo de E/S para los requisitos del reloj a salida rápidos.

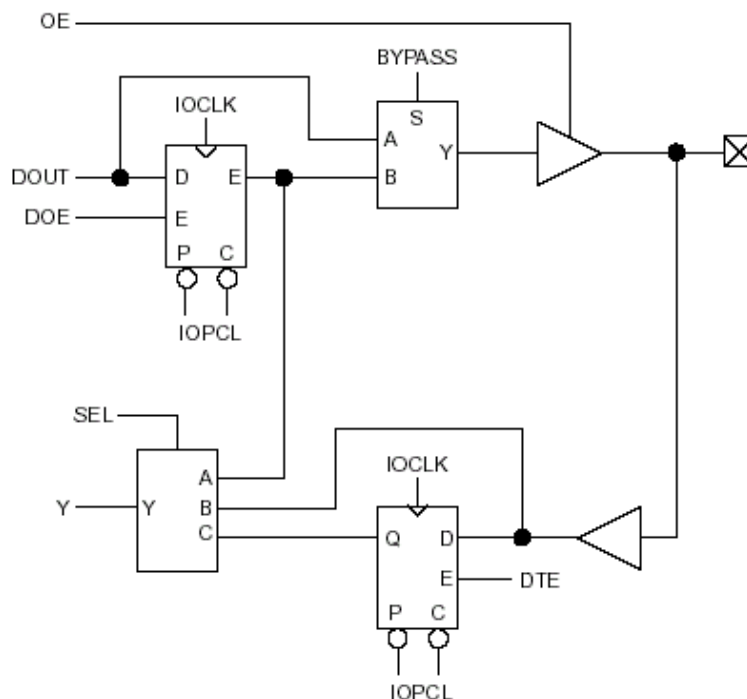


Figura 1.65: Módulo de E/S con Registros

1.7 Conclusión

En conclusión los circuitos lógicos programables tienen un gran campo de aplicación en la implementación de circuitos lógicos de todo tipo.

Los SPLDs son dispositivos más pequeños que se aplican a circuitos lógicos simples y de bajo costo.

Los CPLDs por su parte implementan más eficientemente diseños con parte combinatorial abundante a un mediano costo.

En cambio, las FPGAs son utilizadas para diseños que manejan mayor transferencia de datos y registros; pudiéndose implementar dentro de ellas memorias del tipo RAM, ROM, FIFO, etc, a través de sus pequeños bloques de memoria.

La elección de la familia a utilizar dentro de un SPLD, CPLD o FPGA; dependerá de la cantidad de lógica combinatorial (compuertas), la cantidad de lógica secuencial (celdas lógicas y macroceldas) y, en caso de FPGAs, la cantidad de memoria a implementar.

También un factor a tener en cuenta son los tiempos de propagación internos, como así también la cantidad de pines del dispositivo.

Capítulo 2 MAX+plus II

2.1 Entorno del MAX+plus II

2.1.1 Introducción

El software **MAX+plus II** es uno de los ambientes de desarrollo que posee **Altera** para la realización de diseños con las distintas familias de dispositivos programables que comercializa. Se trata de un entorno particularmente flexible y eficiente, con un nivel de integración de las herramientas muy elevado, capaz de soportar distintas metodologías de diseño y de adecuarse a la realización de sistemas digitales de distinta complejidad.

Las herramientas que integran el entorno pueden clasificarse, de acuerdo con sus funciones, en los siguientes grupos:

1. **Herramientas de especificación de diseños:** Una captura de esquemas, un editor de texto y dos editores auxiliares.
2. **Herramientas para la verificación de diseños:** Una herramienta gráfica para la especificación de los vectores de test de los diseños y la visualización de resultados, un simulador lógico y un simulador y analizador de tiempos.
3. **Herramientas para el procesamiento de los diseños:** El compilador y el procesador de mensajes.
4. **Herramientas de programación de dispositivos:** El programador y el hardware de programación asociado.

Existe otro ambiente de desarrollo de **Altera** llamado **Quartus** el cual es utilizado para la programación de la nuevas familias de dispositivos de Altera como la Cyclone y la Stratix, pero éste programa solo puede ser utilizado en PCs de muy buenas prestaciones.

2.1.2 Flujo de diseño

El nombre que reciben los diseños en el entorno **MAX+plus II** es el de proyectos (*projects*). El proceso de un diseño comprende la serie de operaciones que se detallan a continuación:

1. Crear un nuevo archivo de diseño o una jerarquía de múltiple archivos de diseño en cualquier combinación editores del diseño del **MAX+plus II** (Editores Gráficos, de Texto y de Forma de onda.).
2. Especificar el nombre del archivo de diseño *top-level* como nombre del proyecto.
3. Asignar una familia de dispositivos para el proyecto (se puede permitir al Compilador que seleccione el dispositivo dentro de la familia).
4. Abrir el Compilador y ejecutarlo para compilar el proyecto. También; si es necesario, se puede abrir el módulo Timing SNF Extractor para crear un archivo netlist para simulación y análisis de tiempos.
5. Si el proyecto se compiló con éxito, se puede opcionalmente realizar una simulación y un análisis de tiempos.
 - Para correr el análisis de tiempos, abrir el Analizador de tiempos (*Timing Analyzer*), seleccionar un modo de análisis, y presionar el botón **Start**.
 - Para correr la simulación, se debe primero crear un vector de entradas en un Simulator Channel File (.scf) en el Editor de Forma de onda (*Waveform Editor*) o en un Vector File (.vec) en el Editor de Texto (*Text Editor*). Luego abrir el Simulador y presionar el botón **Start**
6. Abrir el Programador del **MAX+plus II** e insertar el dispositivo en la placa para programar.
7. Elegir el botón **program** para programar dispositivos basados en EPROM o EEPROM o el botón **configure** para configurar dispositivos basados en SRAM.

2.1.3 Gestión de diseños

Las tareas descritas anteriormente requieren la cooperación de dos o más herramientas del entorno; la ejecución de una simulación se realiza sobre un modelo lógico generado por el compilador al que se aplican vectores de test especificados en el editor de forma de onda. El entorno **MAX+plus II** gestiona automáticamente la interacción entre las herramientas a través del concepto de **proyecto**. Un proyecto está almacenado en un directorio del ordenador, comprende los archivos de diseño y un conjunto de archivos auxiliares. Las herramientas del entorno (exceptuando los editores) procesan siempre los archivos correspondientes al proyecto de trabajo actual. El entorno **MAX+plus II** proporciona una serie de utilidades para la gestión de los proyectos con el fin de facilitar el desarrollo de las tareas que componen el ciclo de diseño en el entorno **MAX+plus II**.

2.1.4 Aplicaciones

El software **MAX+plus II** consiste en 11 programas de aplicación y el Gerente MAX+plus II (*MAX+plus II Manager*). A continuación se describen los programas de aplicación:

- **Hierarchy Display** : Despliega la jerarquía actual de archivos como un árbol jerárquico con ramas que representan los sub-diseños. Se puede decir de una ojeada qué archivos están actualmente abiertos y también se pueden directamente abrir o cerrar uno o más archivos en un árbol jerárquico.
- **Graphic Editor** : Permite entrar en un diseño lógico esquemático en un verdadero ambiente del tipo lo-que-ves-es-lo-qué-consigues. Mientras las primitivas, megafunciones y macrofunciones proporcionadas por Altera sirven como sus ladrillos básicos, también puede usar símbolos personalizados.
- **Text Editor** : Permite crear y editar textos basados en archivos de diseño de lógica escritos en AHDL, VHDL, y Verilog HDL. Con el editor del texto, se puede también crear, ver, y editar otros archivos de ASCII usados con aplicaciones de **MAX+plus II**.
- **Waveform Editor** : Sirve de dos formas: como una herramienta de entrada de diseño y como una herramienta para entrar vectores de test y ver los resultados de la simulación .
- **Floorplan Editor** : Permite asignar lógica al pin físico del dispositivo y recursos de celdas lógicas en un ambiente gráfico. Se pueden editar las colocaciones de pines en una vista del dispositivo y asignar señales a celdas lógica individuales en una vista del *Logic Array Block* (LAB) más detallada. Se puede ver también el resultado de la última recopilación.
- **Compiler** : Procesa proyectos lógicos designados para las familias de dispositivos de Altera (por ejemplo MAX 5000, MAX 7000, FLEX 10K). El Compilador realiza la mayoría de las tareas automáticamente. Sin embargo, se pueden personalizar todos o parte del proceso de la recopilación.
- **Simulator** : permite testear el funcionamiento lógico y los tiempos interiores del circuito de lógica. Están disponibles la simulación funcional, simulación de tiempo, y simulación del multi-dispositivos vinculados.
- **Timing Analyzer** : Analiza la performance del circuito lógico después de que haya sido sintetizado y perfeccionado por el Compilador.
- **Programmer** : Permite programar, configurar, verificar, examinar, y testear los dispositivos de Altera.
- **Message Processor** : Despliega mensajes de error, advertencia e información del estado del proyecto y permite localizar la fuente del mensaje automáticamente.

2.1.5 Librerías de funciones lógicas

Altera provee librerías de funciones lógicas (primitivas, megafunciones y macrofunciones) incluyendo funciones que están optimizadas para una arquitectura de una familia de dispositivos particular. Estas librerías se detallan a continuación:

- **Primitivas:** son bloques funcionales básicos como *buffers*, *flip-flops*, *latches*, entradas (*inputs*), salidas (*outputs*), compuertas lógicas (AND, OR, XOR, etc), etc; usados para diseñar circuitos. Pueden ser usados en archivos de diseño gráficos, y en archivos de texto en los lenguajes AHDL, VHDL y Verilog HDL.

En archivos de diseño HDL, las Primitivas son un subconjunto de los símbolos de primitivas usados en los archivos de diseño gráfico. Otras funciones primitivas pueden ser representadas por operadores lógicos, puertos y varias instrucciones.

En esquemas del Editor Gráfico, se puede crear un arreglo de Primitivas, en el cual una única Primitiva conectada a una o más llamadas líneas de bus (*bus lines*) representa una serie de Primitivas idénticas. Durante el procesamiento del proyecto, el Compilador automáticamente traduce el arreglo de Primitivas en el número correcto de Primitivas individuales.

- **Megafunciones:** son bloques complejos o de alto nivel de construcción que pueden ser usados junto con primitivas y otra mega o macrofunción para crear diseños de lógica.

Algunas megafunciones, incluyendo funciones de la Librería de Módulos Parametrizados (LPM, *Library of Parameterized Modules*), son inherentemente parametrizadas a medida, comportamiento e implementación de silicio. La escalabilidad de las LPM y otras funciones parametrizadas pueden simplificar considerablemente el diseño. Las megafunciones pueden ser libremente utilizadas en archivo de diseño gráfico y en todos los archivos de texto de lenguaje HDL. Cuando el Compilador analiza la lógica completa del circuito, automáticamente usa cualquier lógica disponible de megafunción específica de la familia de dispositivos y remueve todas las compuertas y *flip-flops* que no son utilizados, para garantizar una eficiencia óptima del diseño.

- **Macrofunciones:** son bloques de alto nivel de construcción que simulan los componentes de lógica estándar (ej: serie 74) que pueden ser usados junto con primitivas y otra mega o macrofunción para crear diseños de lógica. Pueden ser libremente utilizadas en archivo de diseño gráfico y en todos los archivos de texto de lenguaje HDL. Cuando el Compilador analiza la lógica completa del circuito, automáticamente usa cualquier lógica disponible de macrofunción específica de la familia de dispositivos y remueve todas las compuertas y *flip-flops* que no son utilizados, para garantizar una eficiencia óptima del diseño. Todas las entradas de macrofunciones también tiene niveles predeterminados de señales de entrada de modo que los pines no utilizados pueden ser dejados sin conectar.

Altera recomienda utilizar megafunciones LPM en lugar de las macrofunciones equivalentes. Los LPM y otras funciones parametrizadas son más fáciles de usar, escalables, y son implementadas más eficientemente en silicio.

2.1.6 Proyectos en el entorno MAX+plus II

La primera tarea que hay que ejecutar para realizar un diseño en el entorno **MAX+plus II** es la definición del proyecto de trabajo actual, el cual servirá de referencia a las herramientas del entorno para buscar los archivos que deben procesar y/o conocer el directorio donde se debe almacenar el resultado del procesamiento.

Un proyecto está constituido por un conjunto de archivos almacenados en un directorio.

Los archivos pueden clasificarse en dos categorías: **archivos de diseño** (*design files*) y **archivos auxiliares** (*ancillary files*).

Un archivo de diseño es un archivo gráfico o de texto, creado con un editor del entorno **MAX+plus II** o, también, con otro esquemático estándar, editor de texto o generador de **netlist** en formato EDIF, XNF o VHDL. Estos archivos contienen la especificación lógica de los diseños que pueden ser procesados por el compilador para generar modelos de simulación y/o archivos de programación de dispositivos (archivos que sirven para que un chip fabricado por Altera, implemente el circuito que se ha diseñado). El compilador puede procesar automáticamente distintos tipos de archivos que se identifican por una determinada extensión.

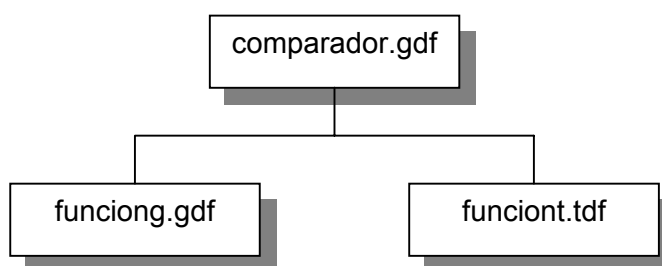
Los archivos auxiliares son los que están asociados a un proyecto **MAX+plus II** sin formar parte del diseño, pudiendo ser, en algunos casos, editables por el usuario; entre ellos se encuentran todos los archivos de salida de las aplicaciones del entorno (exceptuando, por supuesto, los archivos de diseño). Hay archivos auxiliares que son generados automáticamente por las aplicaciones, mientras que otros sólo se crean bajo requerimiento del usuario.

2.2 Tutorial MAX+plus II

2.2.1 Introducción

Para aprender a utilizar el **MAX+plus II**, realizaremos un ejemplo guiado muy sencillo. Crearemos un proyecto llamado **comparador** comprendido por un archivo de diseño gráfico (**.gdf**) de alto-nivel (*top-level*), **comparador.gdf**, que incorpora dos archivos de diseño de bajo-nivel (*lower-level*): un archivo de diseño gráfico, **funciong.gdf**, y un archivo de diseño de texto, **funciont.tdf**. Cada archivo de diseño de bajo-nivel implementa de la función $Y = A_1 * A_2 + /B$, a través del Editor Gráfico y del Editor de Texto mediante lenguaje AHDL respectivamente. Luego el archivo *top-level* conectará ambas salidas de los dos archivos anteriores a una compuerta XOR con lo cual la salida de dicha compuerta, coincidente con una de las salidas de **comparador.gdf**, comparará las salidas **funciong.gdf** y **funciont.tdf**, siendo '0' cuando coinciden y '1' cuando no.

Compilaremos el proyecto asignándole un dispositivo EPM7064LC44 de la familia MAX 7000, lo simularemos y verificaremos las salidas mediante el Editor de Formas de onda



2.2.2 Iniciando la aplicación MAX+plus II

En este punto iniciaremos **MAX+plus II** para comenzar con la creación del proyecto.

Iniciamos la aplicación **MAX+plus II** en Windows, con lo cual se abre la ventana del entorno, que, a partir de aquí, denominaremos **MAX+plus II Manager** o ventana principal, tal como muestra la figura 2.1.

La ventana principal dispone de un conjunto de menús, algunos de los cuales también están disponibles al activar las aplicaciones del entorno, y de una serie de iconos que dan acceso a distintas herramientas y utilidades. Además contiene una barra de estado, situada en la parte inferior de la ventana, que muestra el comando que ejecuta la opción de un menú o el icono sobre el que esté situado el cursor del mouse. Los once programas de aplicación del entorno **MAX+plus II** son accesibles por medio del menú **MAX+plus II** y algunos de ellos, además, a través de los iconos de la ventana principal. Cuando se ejecuta una aplicación se abre una ventana de trabajo que puede minimizarse o mantenerse abierta junto con otras, pudiendo activarse la que se desee pulsando el mouse sobre la ventana correspondiente. Además, algunas

aplicaciones (las que realizan algún tipo de procesamiento) pueden ejecutarse al mismo tiempo que se está trabajando con los editores.

El aspecto de la ventana principal puede configurarse por medio de una opción, **Preferences**, del menú **Options**.

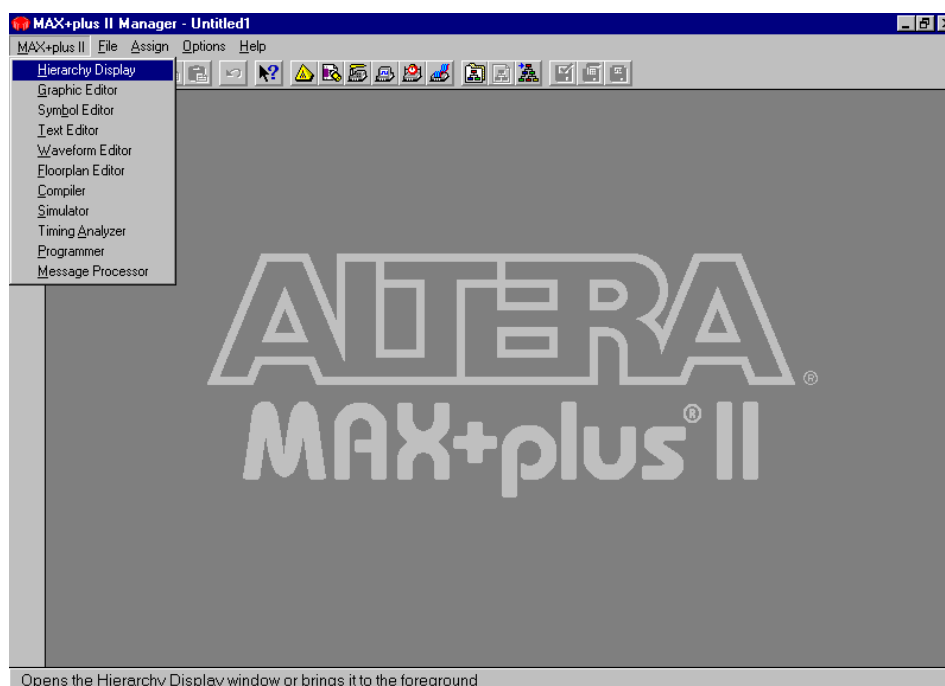


Figura 2.1: Manager

2.2.3 Herramientas de ayuda

Antes de continuar con el desarrollo del diseño, vamos a presentar una utilidad del entorno **MAX+plus II** que puede servir de apoyo al desarrollo del tutorial: la herramienta de ayuda. La ayuda del entorno **MAX+plus II** proporciona toda la información relevante acerca de las herramientas básicas, de las primitivas, megafunciones y macrofunciones de las librerías del entorno, y de la implementación de los lenguajes de especificación hardware (lenguajes que permiten describir circuitos lógicos). Además, ofrece información de todos los dispositivos y accesorios hardware de Altera. Puede resultar muy útil para ampliar el conocimiento sobre las herramientas y procedimientos que se utilizan durante un diseño.

Los temas de la herramienta de Ayuda contienen palabras resaltadas que facilitan el enlace entre temas o el acceso a información complementaria, de acuerdo con la siguiente convención:

- a. Los textos resaltados en color verde y con subrayado continuo son la entrada a otros temas de ayuda. Para saltar al tema apuntado, basta con pulsar el botón izquierdo del mouse sobre el propio texto.
- b. Los textos resaltados en color verde y con subrayado discontinuo dan acceso a una ventana de explicación del término correspondiente. Para leer la información se pulsa el botón izquierdo del mouse sobre el texto; al volver a pulsar, desaparece la ventana.

- c. Los textos en color azul funcionan de manera similar a los subrayados con trazos discontinuos pero dan acceso a ejemplos o ilustraciones.
- d. En algunas ilustraciones también es posible acceder a temas de ayuda; para ello hay que situar el cursor del mouse sobre ellas, buscar los puntos en que cambia de forma el cursor, y apretar el botón izquierdo del mouse.

El menú de ayuda siempre está presente en la ventana principal del entorno. Los submenús de ayuda son los siguientes:

- **Search for Help on:** Activa una herramienta de búsqueda de temas de ayuda relacionados con un texto introducido por el usuario. Suele utilizarse como último recurso cuando han fallado otros métodos para acceder a la información.
- **MAX+plus II Table of Contents:** Muestra los temas principales que contiene la ayuda del entorno.
- **Ayuda de la herramienta activa:** Es el único submenú que cambia dependiendo de la aplicación activa. Da acceso a los temas relacionados con dicha aplicación. Dispone de varios submenús para facilitar aún más la búsqueda de información.
- **AHDL, VHDL y Verilog:** Temas de ayuda relacionados con los lenguajes de especificación hardware.
- **Megafunctions/LPM Old-Style, Macrofunctions y Primitives:** Temas de ayuda sobre los componentes y módulos de las librerías predefinidas del entorno.
- **Devices & Adapters:** Información sobre los dispositivos y el hardware de programación de Altera.
- **Messages:** Listado de todos los avisos del procesador de mensajes (una herramienta que aparece automáticamente cuando se produce algún error).
- **Glossary:** Listado y definición de todos los términos manejados en el entorno.
- **READ.ME:** Información de última hora sobre la versión del software. Incluye relaciones de problemas de funcionamiento y erratas existentes en los manuales.
- **New Features in This Release:** Informa de las innovaciones introducidas en la versión actual del software respecto a las últimas versiones.
- **How to Use Help y How to Use MAX+plus II Help:** Informan de los procedimientos para utilizar la ayuda.
- **About MAX+plus II:** Muestra los **copyrights** de los productos de Altera.

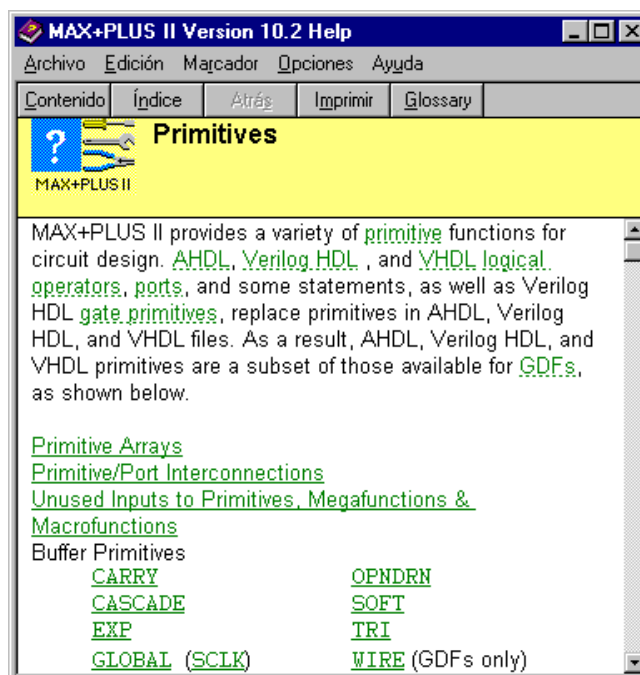


Figura 2.2: Ventana de ayuda sobre primitivas

A continuación vamos a buscar ayuda, a modo de ejemplo, sobre uno de los componentes necesarios para la realización del circuito en el editor gráfico.

Activamos la opción del menú **Help**→**Primitives**, con lo cual se abre la ventana de ayuda, tal como muestra la figura 2.2, que informa sobre el contenido de la librería de primitivas y que lista sus componentes

Si situamos el cursor del mouse sobre el texto **AND**, por ejemplo, y pulsamos el botón izquierdo, accedemos a la ayuda sobre la compuerta AND, como se ve en la figura 2.3.

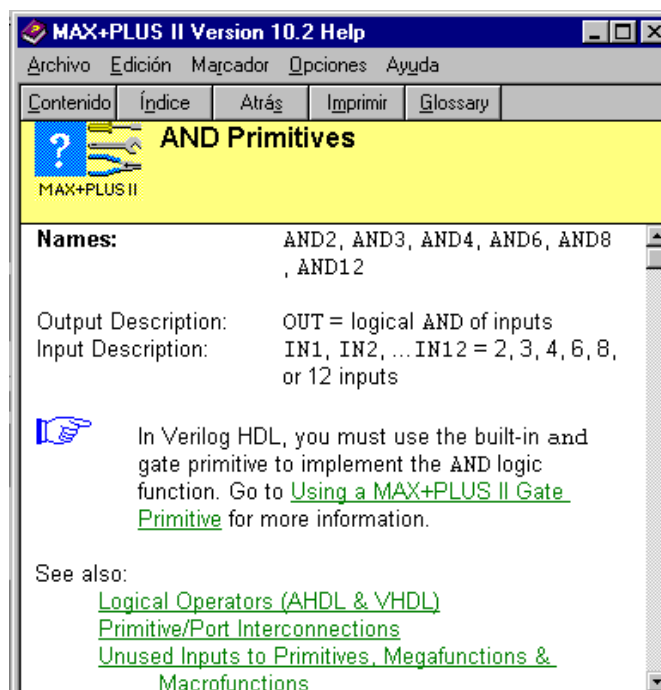
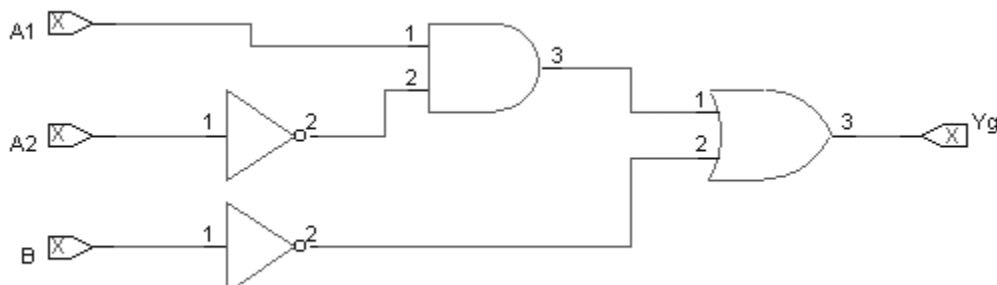


Figura 2.3: Ventana de ayuda sobre primitiva AND

2.2.4 Creación del archivo de diseño gráfico

En este paso debemos crear un nuevo archivo gráfico llamado **funciong.gdf** que contenga un circuito que represente la función $Yg = A_1 * A_2 + /B$.

El esquema del circuito puede ser de la forma:



2.2.4.1 Creación de un archivo nuevo

Para crear el archivo:

1. Seleccionamos del menú de la ventana principal **File**→**New** con el botón izquierdo del mouse.
2. En la ventana que se activa (figura 2.4), indicamos que se desea crear un archivo gráfico seleccionando la opción **Graphic Editor File**, junto con la extensión por defecto, **.gdf (graphic design file)**.

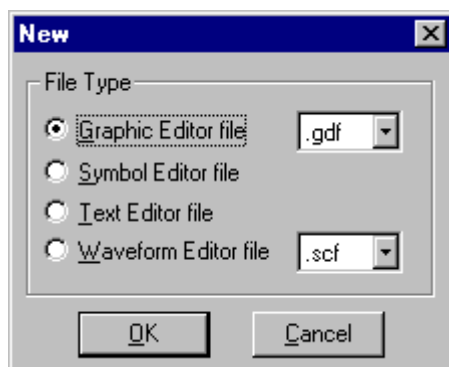


Figura 2.4: Ventana de archivo nuevo

3. Pulsamos el botón **OK** y aparece la ventana del editor de archivos gráficos que muestra la figura 2.5

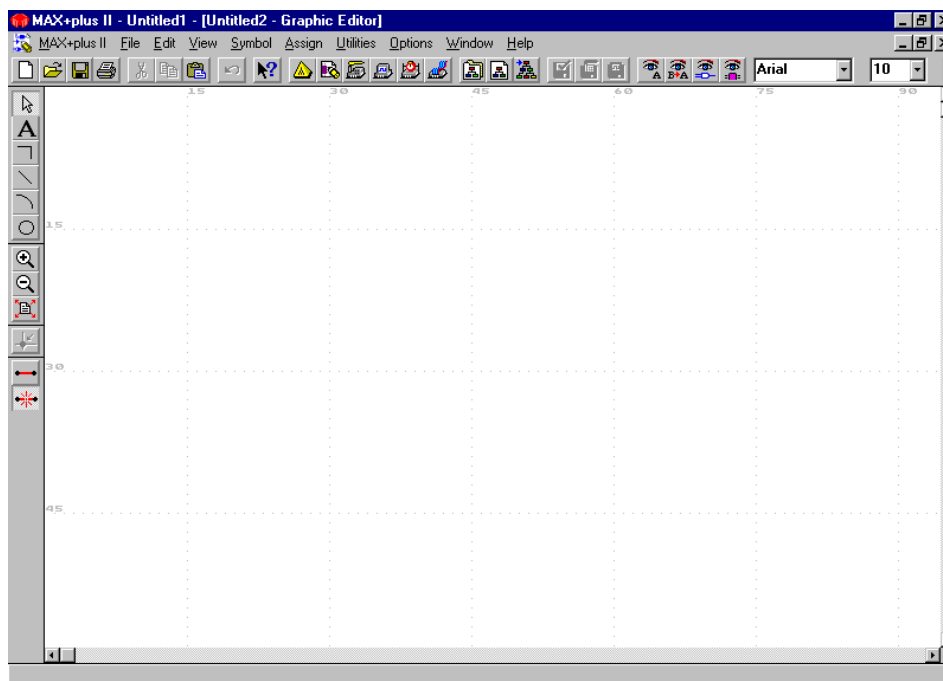


Figura 2.5: Ventana del Editor de archivos gráficos

4. Para guardar el archivo, elegimos del menú **File**→**Save As**. La ventana de dialogo del **Save As** se abre como muestra la figura 2.6.

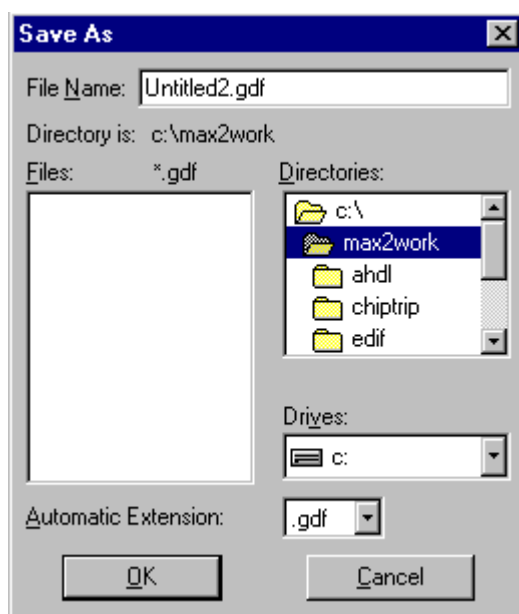


Figura 2.6: Ventana de Save As

5. Tipeamos `comparador\funciong.gdf` en el cuadro *File Name*. Con esto creamos un nuevo directorio llamado **comparador**, y dentro de él, el archivo gráfico **funciong.gdf**.
6. Pulsamos el botón **OK** para guardar el archivo **funciong.gdf**
7. Al no existir el directorio, el programa pedirá la conformidad para crearlo; la cual aceptamos.

2.2.4.2 Especificación del nombre del proyecto

En **MAX+plus II**, debemos especificar un archivo de diseño como nuestro proyecto actual antes de compilarlo o simularlo. El software procesa un proyecto a la vez y debemos estar seguros que todos los archivos de diseño en un proyecto aparecen en la jerarquía del proyecto. Debemos siempre crear un subdirectorio separado (**comparador** en nuestro caso) por cada nuevo proyecto. Cuando entramos el nombre del proyecto, debemos especificar el nombre del subdirectorio en el cual grabaremos dicho proyecto.

Para especificar el nombre del proyecto:

1. Elegimos del menú **File**→**Project**→**Name**. La ventana de dialogo del Project Name se abre como muestra la figura 2.7:

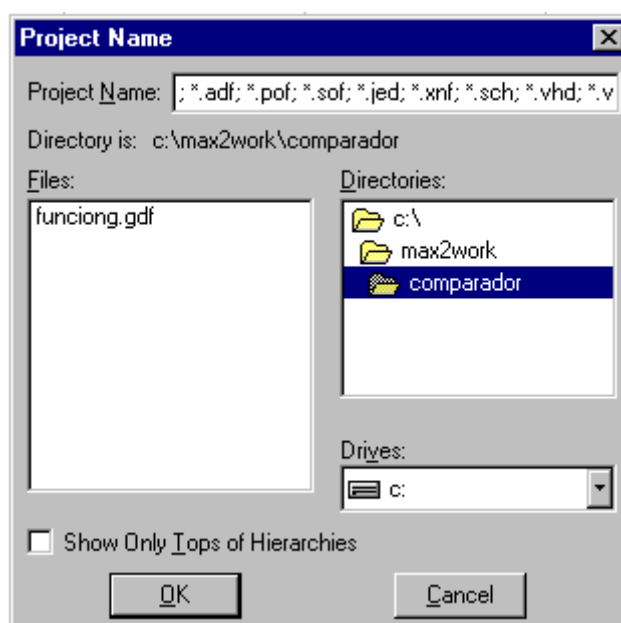


Figura 2.7: Ventana de Project Name

2. Si es necesario, desactivamos la opción *Show Only Tops of Hierarchies*.
3. Seleccionamos **funciong.gdf** en el cuadro *Files*.
4. Pulsamos OK.

Como una alternativa de usar el comando **Project Name**, podemos simplemente seleccionar del menú **File**→**Project**→**Set Project to Current File**, cuando **funciong.gdf** está abierto en la ventana activa del Editor Gráfico.

2.2.4.3 Dibujo del esquema del circuito

En este punto vamos a dibujar el esquema del circuito del ejemplo utilizando algunas de las utilidades de Editor Gráfico.

Las utilidades de edición del Editor Gráfico pueden ser activadas mediante cinco procedimientos:

1. Desde los menús de la ventana principal, que son distintos de los que aparecían hasta el momento, ya que se adaptan siempre a las necesidades de la herramienta activa. Por este procedimiento se pueden ejecutar todas las utilidades.
2. Mediante los iconos situados en la parte izquierda de la ventana; de este modo se pueden ejecutar las operaciones de uso más frecuente.
3. Mediante los menús de *pop-up* que se despliegan al activar el botón derecho del mouse en el área de dibujo; el contenido de estos menús cambia según el mouse se pulse sobre una zona vacía o sobre un objeto seleccionado.
4. Mediante el teclado del ordenador. La combinación de teclas correspondiente a cada función se indica en los menús de la ventana principal, junto al nombre de la misma; por ejemplo, la función de copia de objetos (**C**opy), puede invocarse con la pulsación simultánea de las teclas **Ctrl** y **C** (abreviadamente **Ctrl + C**).
5. Mediante acciones del mouse. Algunas funciones se realizan, de manera exclusiva o alternativa, activando el mouse en el contexto adecuado; por ejemplo, una doble pulsación del botón izquierdo del mouse en un área libre de la hoja de esquemas, permite seleccionar el símbolo de un módulo de librería que se desee incluir en el esquema.

Las utilidades de elaboración del esquema del circuito se pueden clasificar, de acuerdo a la funcionalidad que proporcionan, en los siguientes grupos:

1. **Utilidades de edición:** Tales como copiar, borrar y rotar objetos; se encuentran agrupadas en el menú **Edit**
2. **Utilidades para controlar la visualización del esquema:** Son funciones que permiten determinar el tamaño del área de la hoja de esquemas que se visualiza en la ventana (funciones de zoom), qué parte del esquema se visualiza, así como los colores empleados en la visualización de los objetos (opciones de configuración). Las funciones de zoom se encuentran en el menú **View**, las de configuración en el menú **Options**, ambos en la ventana principal, y la función de pantalla (en el Editor Gráfico se suele llamar *pan* a la función que permite mover el área visible de la hoja de dibujo) se controla con las barras de desplazamiento de ventanas de Windows.
3. **Utilidades para el emplazamiento de objetos en el esquema:** Permiten elegir los módulos y primitivas que se emplazan en la hoja de dibujo, la gestión de las librerías de usuarios, la actualización de símbolos y la creación de hilos y etiquetas de conexión, así como la definición del estilo de líneas y texto empleados. Se encuentran agrupadas en los menús **Symbol** y **Options** de la ventana principal, excepto el dibujo de hilos de conexión y definición de etiquetas que se realizan mediante acciones del mouse.

Además de las funciones anteriores, se dispone de otras utilidades entre las que destacan las siguientes:

1. **Utilidades para la gestión de archivos:** Tales como salvar el archivo activo, abrir archivos, imprimir, etc. Se agrupan en el menú **File**.
2. **Utilidades para el “enlace” de la captura con otras herramientas del entorno:** Desde el menú **File** se pueden ordenar operaciones que requieren la ejecución de otras herramientas (tomando como referencia, siempre, el archivo de diseño correspondiente al proyecto de trabajo); por ejemplo, puede ordenarse la compilación y simulación del esquema (cuando ya exista un archivo con la definición de los estímulos de test), o la generación automática de un símbolo.

Además, están accesibles los contenidos de los menús **MAX+plus II** y **Help**.

2.2.4.3.1 Entrada de los símbolos de funciones lógicas

Para entrar los símbolos:

1. Hacemos un doble-clic, con el botón izquierdo del mouse, sobre una zona libre de la hoja de dibujo.
2. Se abre la ventana de selección de símbolo, en ella tipeamos `and2` (AND de 2 entradas) en el cuadro *Symbol Name* (figura 2.8).

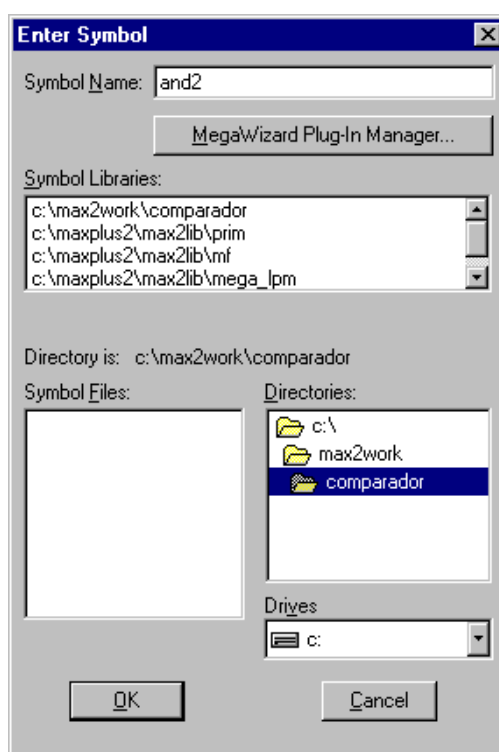


Figura 2.8: Ventana de Selección de Símbolo

3. Pulsamos el botón OK

Observamos que aparece el componente sobre la hoja de dibujo (figura 2.9). En el símbolo pueden distinguirse los siguientes elementos:

- El área de selección, dibujada con trazo discontinuo, limita la zona ocupada por el símbolo. Al pulsar el botón izquierdo del mouse dentro de este área se selecciona la instancia y se resalta, en un determinado color, el perímetro del símbolo.
- Un dibujo y un nombre que permiten identificar el componente.
- Un número de instancia, situado en la parte inferior izquierda, que se genera automáticamente al emplazar el símbolo, que es único, en cada esquema, y sirve para identificar cada símbolo del circuito. El número de instancia se va incrementado a medida que se van incluyendo nuevos símbolos en el dibujo.

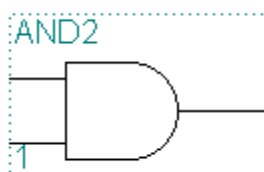


Figura 2.9: Símbolo AND

- Repetimos el procedimiento anterior para colocar una compuerta OR de 2 entradas (OR2) y un negador (NOT)(figura 2.10).

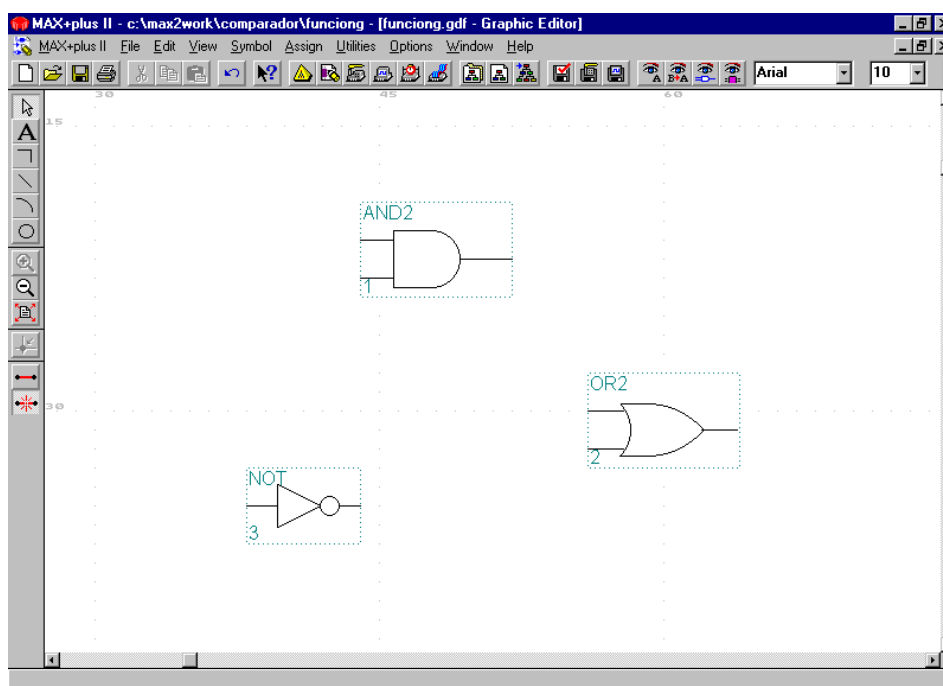




Figura 2.10: Ventana del Editor Gráfico

Como necesitamos dos negadores vamos a hacer una copia de el que ya tenemos

- Hacemos un clic con el botón izquierdo del mouse sobre el negador NOT.
- Pulsamos las teclas **Ctrl+C (Copy)**.
- Hacemos un clic con el botón izquierdo de mouse en una zona libre de la hoja.
- Pulsamos las teclas **Ctrl+V (Paste)**.

Para trabajar más cómodamente, se pueden llevar los símbolos al centro de la hoja de dibujo.

9. Para visualizar la hoja completa, pulsamos el botón izquierdo del mouse sobre el icono , en la barra de herramientas situada en la parte izquierda de la ventana.
10. Arrastramos el cursor del mouse, con el botón izquierdo pulsado, hasta definir un rectángulo que comprenda en su interior todos los símbolos. Una vez definido el rectángulo, soltamos el botón izquierdo del mouse.
11. Pulsamos el botón izquierdo del mouse dentro del área del rectángulo y, sin soltarlo, lo movemos hasta el centro de la hoja. Una vez desplazados los símbolos soltamos el botón del mouse.
12. Para recuperar una visualización adecuada de los símbolos, pulsamos, tantas veces como sea necesario, el icono  de la barra de herramientas.

A continuación, debemos colocar los símbolos en la posición adecuada para la realización del circuito, aunque esto, normalmente, suele hacerse en el momento de la colocación del símbolo (de modo que, si ya están colocados, ignoramos las siguientes instrucciones).

13. Pulsamos el botón izquierdo del mouse sobre el símbolo que deseamos mover, para seleccionarlo.
14. A continuación, situamos el mouse sobre el símbolo y lo arrastramos a la posición deseada manteniendo el botón izquierdo de mouse apretado.
15. Repetimos las operaciones anteriores hasta obtener una colocación parecida a la de la figura 2.11.

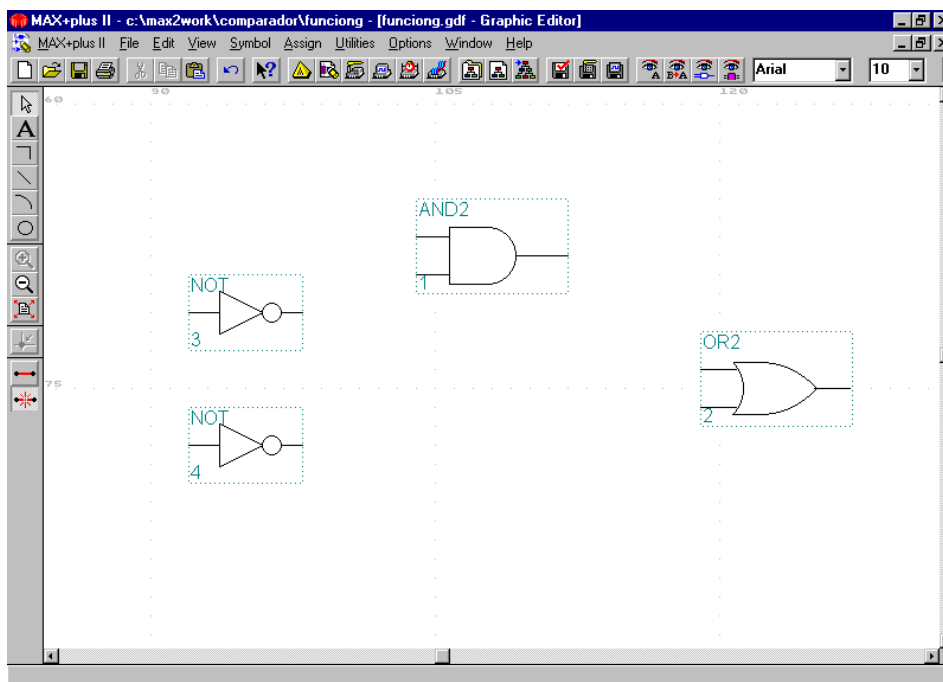


Figura 2.11: Ventana del Editor Gráfico

2.2.4.3.2 Entrada de los pines de entrada y salida

Para introducir en el dibujo los pines de entrada (`INPUT`) y de salida (`OUTPUT`):

1. Hacemos doble-clic con el botón izquierdo del mouse en un espacio en blanco en la izquierda de la hoja de dibujo para abrir el cuadro de diálogo **Enter Symbol**, tipeamos `input` en el cuadro *Symbol Name* y pulsamos **OK**. El pin de entrada aparecerá en el dibujo.
2. Presionamos la tecla **Ctrl.** y el botón izquierdo del mouse en el símbolo `INPUT`. Con **Ctrl.** y el botón izquierdo del mouse presionados arrastramos el mouse hacia abajo para crear una copia del símbolo y colocarla debajo del original. (éste símbolo es copiado pero no es colocado en el portapapeles).
3. Repetimos el 2º paso para crear la tercer entrada.
4. Repetimos el 1º paso pero tipeando `output` en el cuadro *Symbol Name*, para crear la salida.

En la figura 2.12 se ve como queda el circuito:

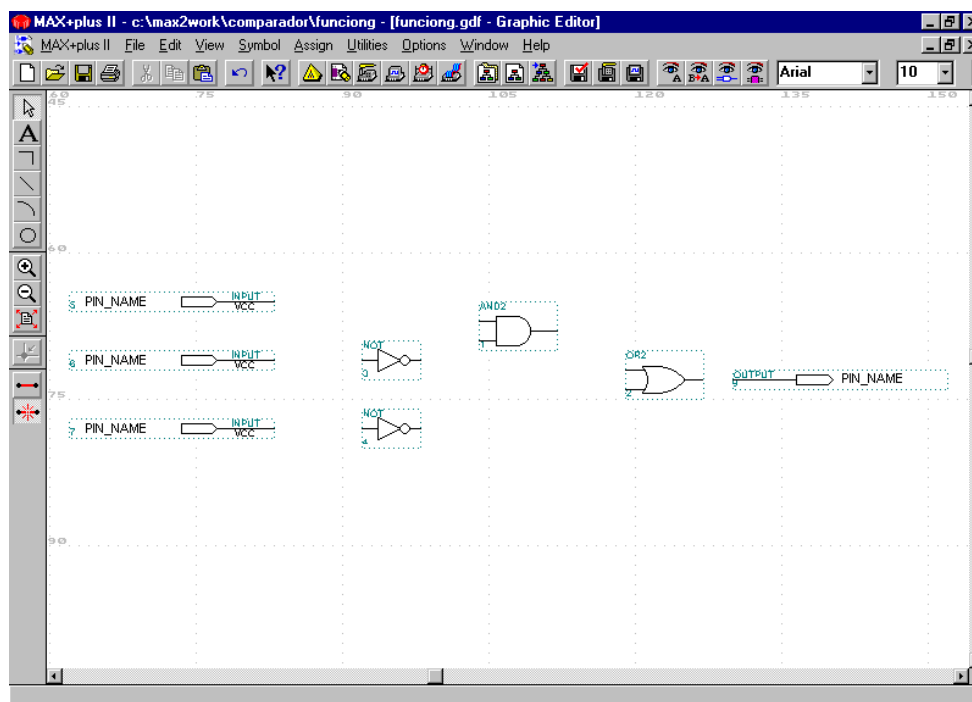


Figura 2.12: Ventana del Editor Gráfico

2.2.4.3.3 Nombramiento de los pines

Ahora debemos colocar los nombres a los pines

1. Hacemos un doble-clic, con el botón izquierdo de mouse, en el campo **PIN_NAME** del símbolo del pin 5.
2. Con el teclado, escribimos el nombre `A1`, y pulsamos la tecla **Enter** (↵) luego de editar el nombre del pin, el próximo nombre del pin (`PIN_NAME`) debajo es automáticamente seleccionado para editarlo.

- Colocamos los nombres a los otros dos pines de entrada y al de salida de forma que queden como muestra la figura 2.13.

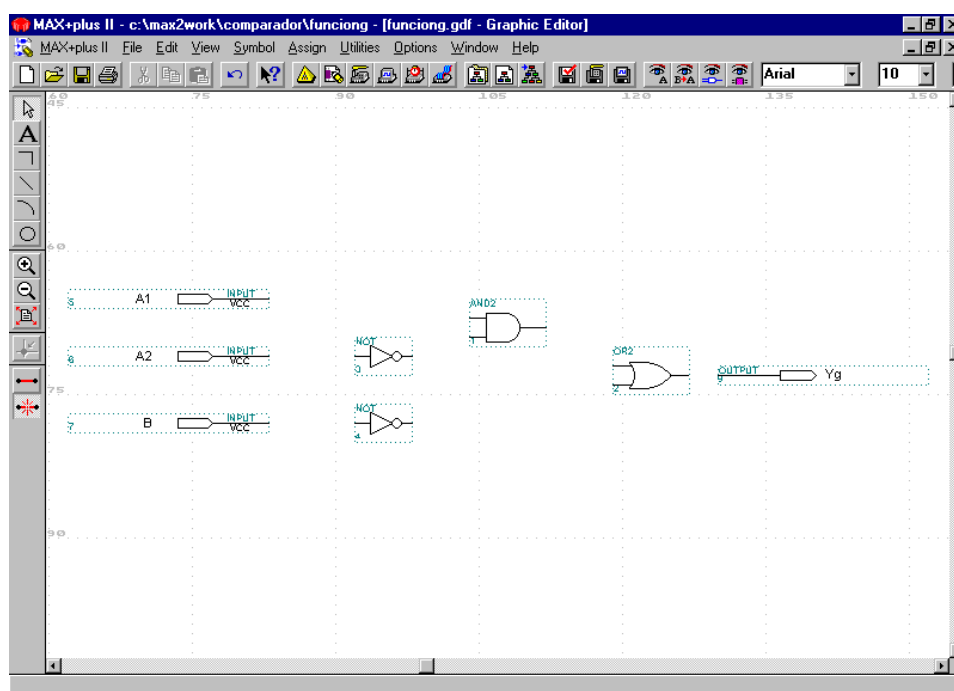




Figura 2.13: Ventana del Editor Gráfico

2.2.4.3.4 Conexión de los símbolos

Seguidamente se debemos realizar las conexiones del circuito.

- Seleccionamos el estilo de línea eligiendo del menú **Option**→**Line Style**→.La ubicada en la parte superior es la recomendada para unir nodos.
- Pulsamos el icono  de la barra de herramientas. El cursor del mouse toma forma de cruz.
- Realizamos las conexiones necesarias para obtener el circuito de la figura 2.14, pulsando el botón izquierdo del mouse en el inicio de cada conexión, y soltándolo en el punto al que se desea conectar. Si en algún caso nos equivocamos, seleccionamos la línea mal dibujada haciendo un clic sobre ella con el botón izquierdo del mouse y presionamos la tecla **Supr.**
- Finalmente cuando ya dibujamos todas las líneas, pulsamos el icono  de la barra de herramientas para volver al modo de funcionamiento normal del mouse.

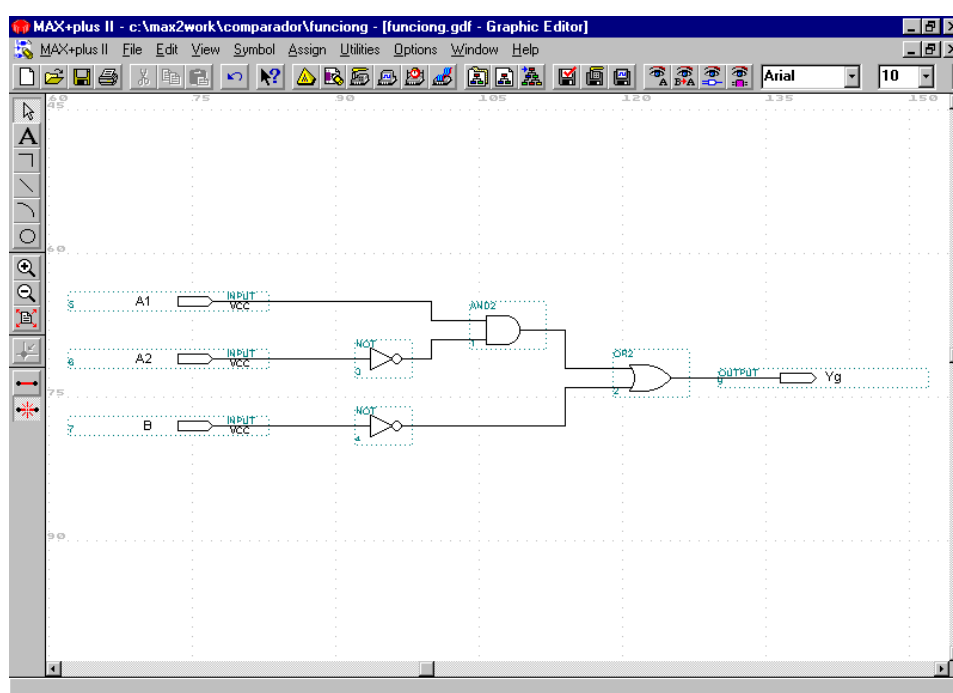


Figura 2.14: Ventana del Editor Gráfico

2.2.4.3.5 Salvado del archivo y chequeo de errores básicos

Para asegurar que se ha entrado la lógica correctamente, se puede salvar el archivo y chequear errores simple.

Para salvar el archivo y chequear errores:

1. Seleccionamos del menú **File**→**Project**→**Save & Check**. El archivo es salvado y se abre la ventana del Compilador del **MAX+plus II**; el Módulo Compiler Netlist Extractor chequea los errores del archivo, actualiza el Hierarchy Display y despliega un mensaje indicando el número de errores y advertencias.
2. Si el comando **Project Save & Check** se lleva a cabo satisfactoriamente y no hay errores ni advertencias, (figura 2.15) elegimos el botón **Aceptar** para cerrar la ventana de mensaje.
3. Cerramos la ventana del compilador utilizando los controles de Windows.

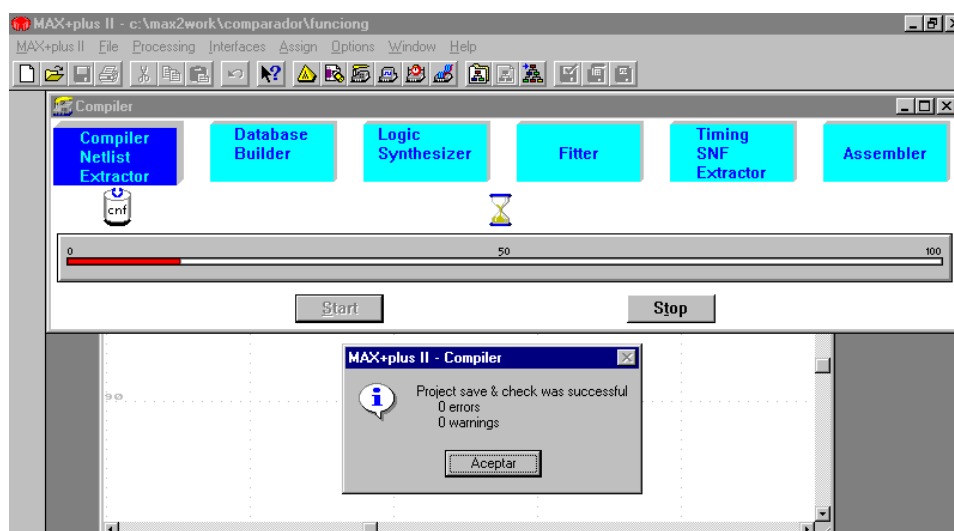


Figura 2.15: Ventana del Compilador

2.2.4.3.6 Creación de un símbolo predeterminado

Vamos a crear un archivo de símbolo (.sym) que representa el archivo actual

- Seleccionamos del menú **File**→**Create Default Symbol**. Si el Símbolo ya existe, **MAX+plus II** preguntará si sobrescribiremos el existente, lo cual aceptaremos.

Luego cerramos el archivo de gráficos con los controles de Windows.

2.2.5 Creación del archivo de diseño de texto

En este paso debemos crear un nuevo archivo de texto llamado **funciont.tdf** que represente la función $Y_t = A_1 * A_2 + /B$ escrito en lenguaje AHDL (*Altera Hardware Description Language*). Para esto vamos a describir en el lenguaje AHDL el circuito que utilizamos en el Editor Gráfico de modo que al compilar el archivo de texto utilice la misma lógica que el Gráfico.

Cabe destacar que el lenguaje AHDL es un lenguaje de descripción de hardware exclusivo de Altera. También existen los lenguaje estándar como el VHDL y Verilog HDL los cuales tienen un mayor nivel de abstracción.

2.2.5.1 Creación de un archivo nuevo y especificación del nombre del proyecto

Para crear el archivo y especificar el nombre del proyecto:

1. Seleccionamos del menú de la ventana principal **File**→**New** con el botón izquierdo del mouse.
2. En la ventana que se activa (figura 2.4), indicamos que se desea crear un archivo gráfico seleccionando la opción **Text Editor File**.

3. Pulsamos el botón **OK** y aparece la ventana del editor de archivos de textos.
4. Elegimos del menú **File**→**Save As**. Tipeamos `funciont.gdf` en el cuadro *File Name* asegurándonos de que estemos en el directorio **comparador**. Si no lo estamos lo seleccionamos del cuadro *Directories*.
5. Seleccionamos del menú **File**→**Project**→**Set Project to Current File**, con lo cual cambiará el nombre del proyecto a **funciont**.

2.2.5.2 Activación de colores de sintaxis

Para realizar secciones de nuestro archivo de texto mas visible podemos utilizar el comando el coloreo de sintaxis (*Syntax Coloring*) del Editor de Texto, con lo cual veremos las distintas partes de nuestro archivo de texto en colores diferentes.

Para activar los colores de sintaxis:

- Elegimos del menú **Options**→**Syntax Coloring**. Cuando el comando se activa, aparece una marca (✓) al lado del nombre del comando en el menú.

2.2.5.3 Entrada del nombre del diseño, entradas y salidas

Para entrar el nombre del diseño, las entradas y las salidas:

1. Elegimos del menú **Templates**→**AHDL Templates**, con lo cual se abre la ventana de dialogo de Plantillas AHDL (*AHDL Templates*).

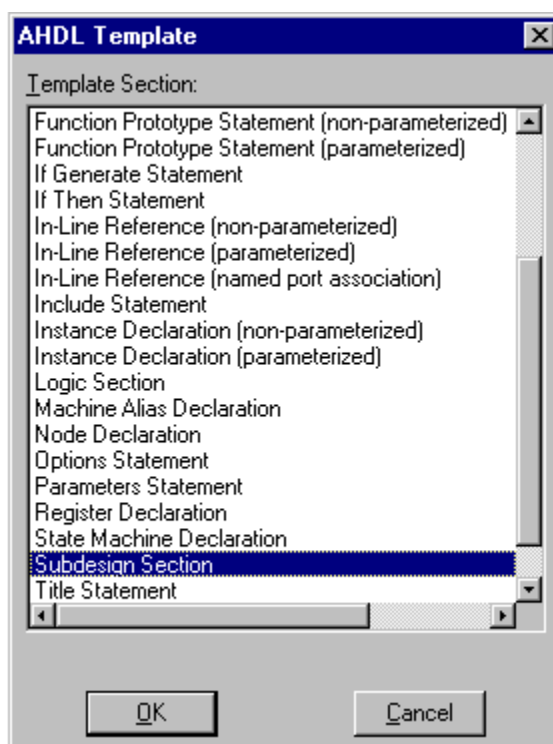


Figura 2.16: Ventana de Plantillas AHDL

2. Seleccionamos *Subdesign Section* en el cuadro *Template Section*, tal como muestra la figura 2.16.
3. Pulsamos el botón **OK**, con lo cual aparece una plantilla para la sección *Sudesign* en el punto de inserción tal como se ve en la figura 2.17.

```

SUBDESIGN __design_name
{
    __input_name, __input_name      : INPUT = __constant_value;
    __output_name, __output_name    : OUTPUT;
    __bidir_name, __bidir_name      : BIDIR;

    __state_machine_name           : MACHINE INPUT;
    __state_machine_name           : MACHINE OUTPUT;
}

```

Figura 2.17: Ventana de Editor de Texto

4. Hacemos doble-clic con el botón izquierdo del mouse en la variable `__design_name` y tipeamos `funciont`.
5. Para añadir los nombres de las entradas, hacemos doble clic con el botón izquierdo del mouse en la primer variable `__input_name` y tipeamos `a1`, en la segunda variable `__input_name` tipeamos `a2` y agregamos una tercera entrada tipeando `a` continuación de la segunda `,` `b`. Borramos la variable `__constant_value` y el signo igual (`=`) que la precede.
6. Para añadir los nombres de las salidas, hacemos doble-clic con el botón izquierdo del mouse en la primer variable `__output_name` y tipeamos `yt`. Borramos la segunda variable `__output_name` y la coma (`,`) que le precede.
7. Borramos las líneas que contienen las palabras claves `BIDIR`, `MACHINE INPUT`, y `MACHINE OUTPUTS`.
8. Agregamos los espacios y tabulaciones para mejorar la legibilidad, de modo que quede como muestra la figura 2.18.

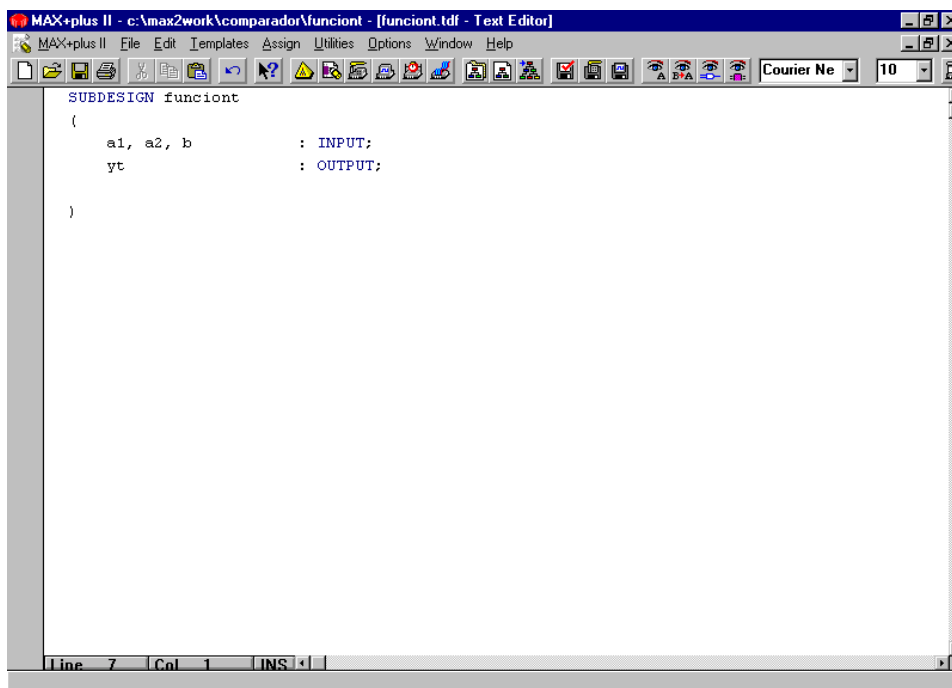


Figura 2.18: Ventana de Editor de Texto

2.2.5.4 Declaración de un nodo

Como dijimos anteriormente vamos a describir en el lenguaje AHDL el circuito que utilizamos en el Editor Gráfico, con lo cual a la salida de la compuerta OR tenemos un nodo, es decir un valor que utilizaremos luego como entrada de la compuerta AND.

Para declarar el nodo:

1. En una nueva línea luego de la sección *Subdesign*, tipeamos la palabra clave `VARIABLE` y presionamos **Enter** (↵).
2. Elegimos del menú **Templates**→**AHDL Templates**.
3. Seleccionamos *Node Declaration* en el cuadro *Template Section*.
4. Pulsamos el botón **OK**, con lo cual aparece una plantilla para la declaración de nodo.
5. Hacemos doble-clic con el botón izquierdo del mouse en la primer variable `_node_name` y tipeamos `nodo`.
6. Borramos la línea que contiene la palabra clave `TRI_STATE_NODE`, como muestra la figura 2.19.

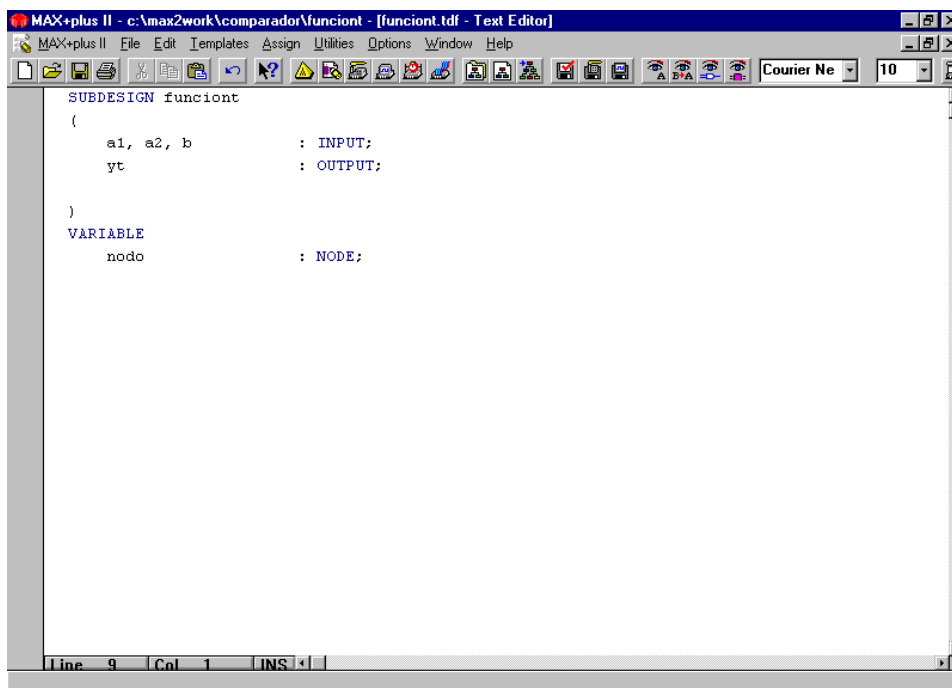


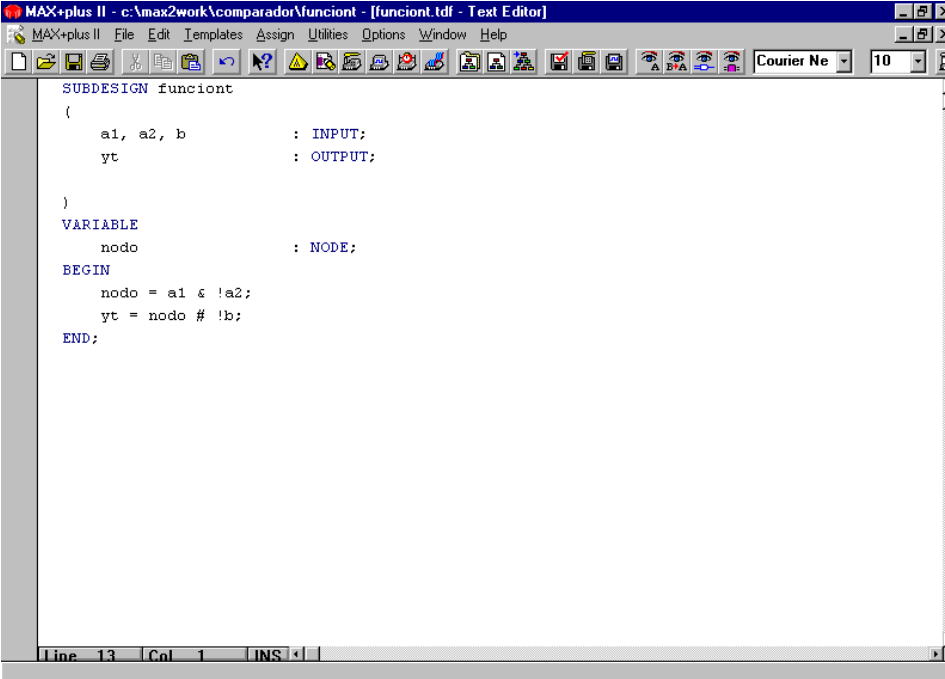
Figura 2.19: Ventana de Editor de Texto

2.2.5.5 Entrada de la Ecuación Booleana

Las ecuaciones booleanas son introducidas en la sección Lógica (*Logic Section*), que está delimitada por las palabras claves `BEGIN` y `END`.

1. Ubicamos el cursor luego de `NODE`; y presionamos **Enter** (↵).
2. Tipeamos `BEGIN` y presionamos **Enter** (↵).
3. Presionamos la tecla **Tabs** (⇧), tipeamos `nodo = a1 & !a2;` y presionamos **Enter** (↵).
4. Presionamos **Tabs** (⇧), tipeamos `yt = nodo # !b;` y presionamos **Enter** (↵).
5. Tipeamos `END`.

La ventana del Editor de Texto queda como muestra la figura 2.20.



```
SUBDESIGN funciont
(
    a1, a2, b      : INPUT;
    yt             : OUTPUT;
)
VARIABLE
    nodo          : NODE;
BEGIN
    nodo = a1 & !a2;
    yt = nodo # !b;
END;
```

Figura 2.20: Ventana de Editor de Texto

2.2.5.6 Chequeo de errores de sintaxis y creación de un símbolo predeterminado

Para asegurar que se ha entrado el lenguaje correctamente, se puede salvar el archivo y chequear errores en la sintaxis. Luego vamos a crear un símbolo predeterminado, el cual lo usaremos en el archivo *top-level*.

1. Seleccionamos del menú **File**→**Project**→**Save & Check**. El archivo es salvado y se abre la ventana del Compilador del **MAX+plus II**; el Modulo Compiler Netlist Extractor chequea los errores del archivo, actualiza el Hierarchy Display y despliega un mensaje indicando el número de errores y advertencias, de la misma forma que lo hizo para el archivo del Editor Gráfico.
2. Seleccionamos del menú **File**→**Create Defaul Symbol** para crear el Archivo de símbolo **funciont.sym**.
3. Cerramos la ventana del compilador utilizando los controles de Windows.

Luego cerramos el archivo de Texto con los controles de Windows.

2.2.6 Creación del archivo de diseño gráfico Top-Level

En esta parte vamos a usar el Editor Gráfico del **MAX+plus II** para crear el archivo de diseño *top-level* para el proyecto **comparador**. El archivo **comparador.gdf** incorpora los símbolos que representan los dos archivos de

bajo nivel, **funciong.gdf** y **funciont.tdf** creados en los pasos anteriores; además de una compuerta XOR.

Para crear **comparador.gdf**:

1. Creamos un nuevo archivo de diseño gráfico y lo guardamos como **comparador.gdf** en el directorio **comparador**.
2. Especificamos **comparador** como nombre del proyecto (Seleccionamos del menú **File**→**Project**→**Set Project to Current File**).
3. Entramos los símbolos para el esquemático:
 - a. Haciendo doble-clic, con el botón izquierdo del mouse, sobre una zona libre de la hoja de dibujo o seleccionando el comando del menú **Symbol**→**Enter Symbol** para abrir la ventana de selección de símbolo; entramos los símbolos que representan los archivos de diseño de bajo nivel **funciong** y **funciont**.
 - b. Entramos una primitiva XOR.
 - c. Entramos tres pines de entrada (**input**) y tres de salida (**output**).
4. Llamamos los pines de la forma:

nombres de pines de entrada	nombres de pines de salida
A1	ygraf
A2	outcomp
B	ytext

5. Conectamos los símbolos de modo que queden como muestra la figura 2.21.

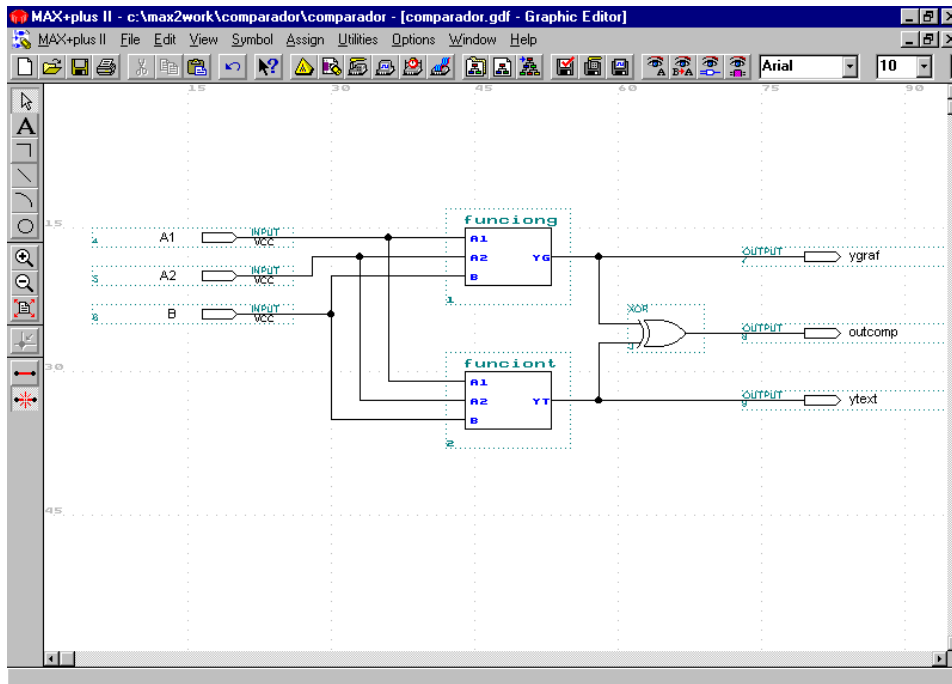


Figura 2.21: Ventana de Editor Gráfico

Para no tener tantos cruces de líneas en las entradas de los símbolos podemos conectarlos asignando nombres a los nodos sin conectar de modo de conectarlos por sus nombres:

- Extendemos un poco la longitud de la línea de entrada del pin A1. Hacemos un clic con el botón izquierdo del mouse sobre el bus y tipeamos a1.
- Repetimos para los demás nodos sin conectar de modo que quede de la forma como muestra la figura 2.22.

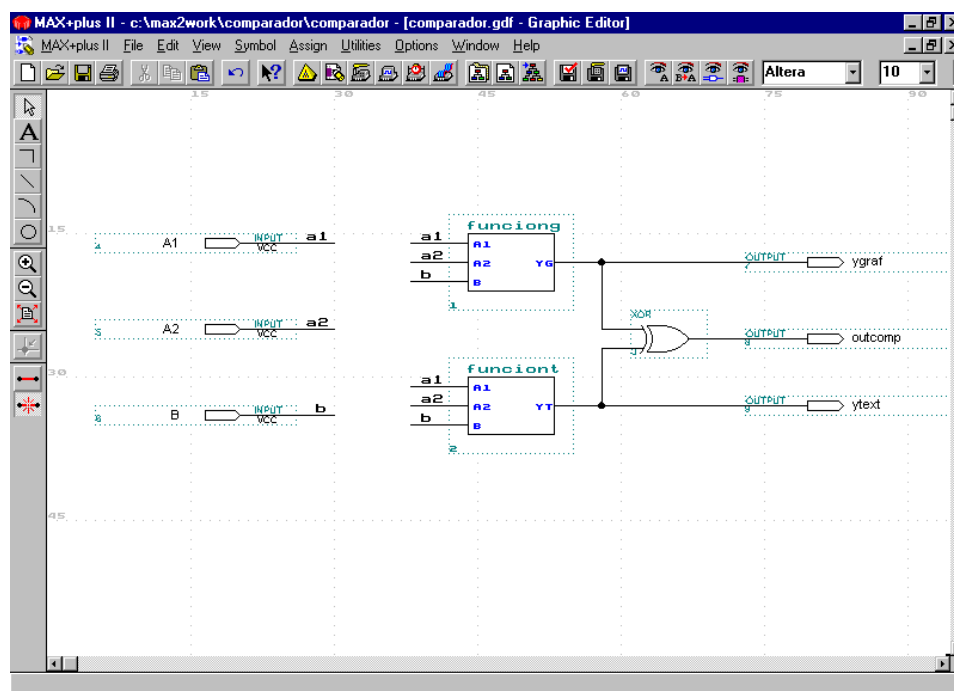



Figura 2.22: Ventana de Editor Gráfico

- Seleccionamos del menú **File**→**Save** o directamente pulsamos el icono  de la barra de herramientas, para guardar el archivo.

2.2.7 Compilación del proyecto

El compilador del entorno **MAX+plus II** procesa automáticamente los archivos del proyecto de trabajo con el fin de completar la operación que el usuario haya ordenado. La ejecución del compilador está dividida en una serie de módulos que aparecen representados por medio de cajas en la ventana del compilador. La información de entrada del compilador es la contenida en los archivos de especificación que forman la jerarquía de diseño y en algunos de los ficheros auxiliares (los de símbolos y el de asignaciones).

Los distintos módulos del compilador realizan las siguientes tareas:

- **Compiler Netlist Extractor:** analiza las reglas de diseño de los archivos (gráficos o textuales) que contienen la descripción del circuito (es el único

módulo del compilador que se ejecuta en el chequeo de reglas de diseño) y extrae la información que define las conexiones jerárquicas entre los módulos del diseño, creando un mapa con la organización del proyecto.

- **Database Builder:** procesa la información generada por el primero para crear una base de datos plana, en la que se disuelve la jerarquía del proyecto con el fin de facilitar el trabajo de los siguientes módulos.
- **Logic Synthesizer:** El Compilador aplica un rango de técnicas para aumentar la eficiencia del proyecto y minimizar el uso de recurso de dispositivo. La utilidad de Doctor de Diseño (*Design Doctor*) optativa verifica la fiabilidad de lógica de proyecto antes y después de la síntesis de la lógica.
- **Partitioner:** Si un proyecto es demasiado grande para encajar en un solo dispositivo, el Compilador lo divide entre los múltiples dispositivos de la misma familia de dispositivos, tanto automáticamente o según especificaciones.
- **Fitter:** El Ajustador (*Fitter*) genera un Archivo de Informe personalizable que detalla el uso de recursos y describe cómo el proyecto se llevará a cabo en uno o más dispositivos.
- **Timing SNF Extractor:** crea un archivo SNF que contiene los datos cronometrando de tiempos provenientes de simulación de tiempos y análisis de tiempos.
- **Functional SNF Extractor:** crea el Archivo Netlist del Simulador funcional (*functional Simulator Netlist File*)(.snf) requerido para la simulación funcional. El SNF funcional no contiene información de tiempos, con lo cual se genera más rápidamente que el Timing SNF.
- **Assembler:** genera uno o más archivos para la programación del dispositivo.

La ejecución del compilador puede ordenarse desde las herramientas de especificación de diseños o desde la ventana principal del entorno. Cuando el compilador está ejecutándose, aparece un reloj de arena y se resaltan las cajas que representan los módulos a medida que avanza el procesamiento. Debajo de las cajas pueden aparecer iconos que representan los archivos generados por el módulo correspondiente. Además, si durante la ejecución del compilador se genera algún mensaje de aviso o error, se abre la ventana del procesador de mensajes para comunicarlo.

2.2.7.1 Apertura de la ventana del compilador

Para abrir la ventana del compilador:

- Seleccionamos del menú **MAX+plus II**→**Compiler**, y se abre la ventana del compilador tal como muestra la figura 2.23.

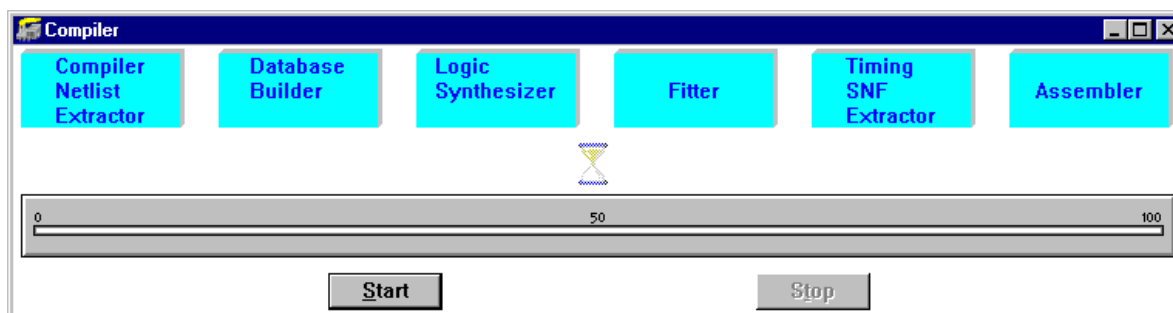


Figura 2.23: Ventana de Compilador

Los módulos e icono que aparecen dependen de cómo haya sido estructurado **MAX+plus II** antes de éste tutorial.

2.2.7.2 Selección de la familia de dispositivos

Podemos seleccionar para nuestro proyecto cualquier familia de dispositivos soportada por **MAX+plus II**. Además podemos permitirle al Compilador escoger automáticamente el dispositivo más apropiado dentro de una familia particular.

Para especificar la familia de dispositivos

1. Seleccionamos de menú **Assign**→**Device**. Se abre el cuadro de dialogo **Device** que muestra la figura 2.24.

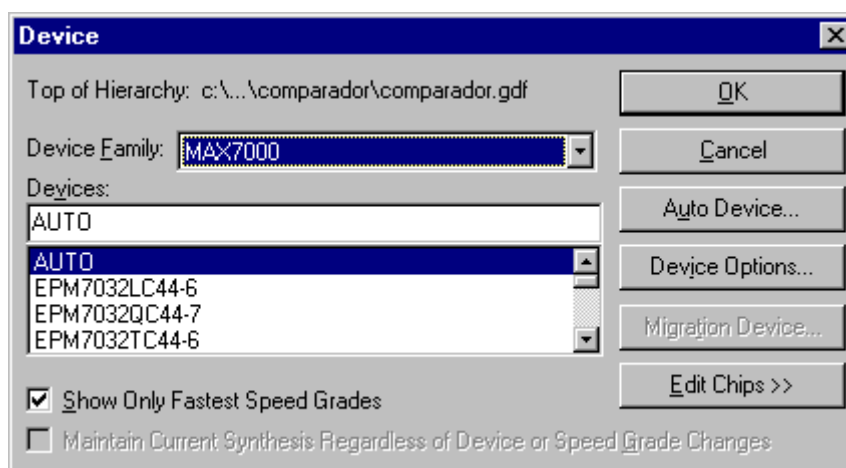


Figura 2.24: Ventana de dispositivos

2. Si la familia MAX 7000 no está todavía seleccionada, seleccionamos **MAX7000** en el cuadro *Device Family*.
3. Seleccionamos **EPM7064LC44-7** en el cuadro *Device*.
4. Presionamos **OK**.

2.2.7.3 Encendido del Comando de Recompilación Inteligente (Smart Recompile Command)

Cuando la recompilación “inteligente” es encendida, el Compilador guarda información de la base de datos extra del proyecto actual para el uso en las compilaciones subsiguientes. Durante la compilación “inteligente”, el Compilador puede determinar qué módulos no se necesitan al recompilar el proyecto, y los saltará durante la recompilación, por consiguiente reduciendo tiempo de recompilación.

Para encender el Comando de Recompilación Inteligente:

- Seleccionamos de menú **Processing**→**Smart Recompile**.

2.2.7.4 Encendido de la Utilidad Doctor de Diseño (*Design Doctor Utility*)

Durante la compilación, la utilidad opcional Doctor de Diseño, chequea todos los archivos de diseño del proyecto a causa de lógica que puede causar problemas de confiabilidad en un dispositivo programado.

Para encender la utilidad Doctor de Diseño y especificar el conjunto de reglas de diseño para el análisis:

1. Seleccionamos de menú **Processing**→**Design Doctor**. Aparecerá en la ventana del Compilador, un icono del Doctor de Diseño debajo del módulo *Logic Synthesizer*, tal como muestra la figura 2.25.

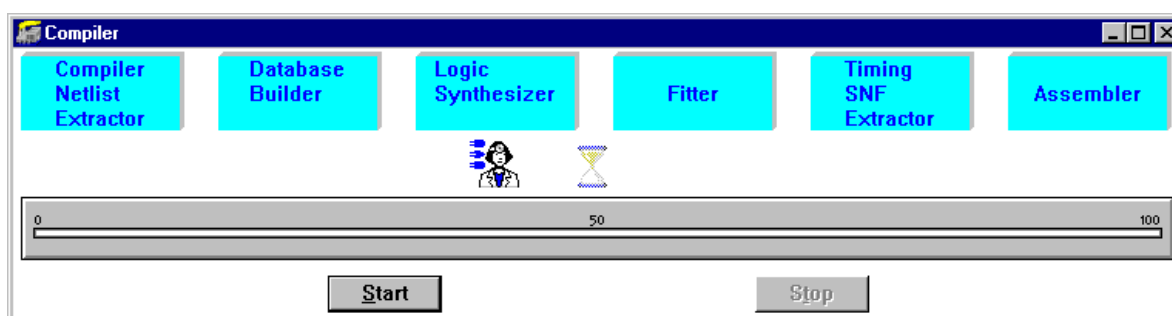


Figura 2.25: Ventana del Compilador

2. Seleccionamos de menú **Processing**→**Design Doctor Settings**. Se abre el cuadro de diálogo de Design Doctor Settings (figura 2.26), seleccionamos si es necesario *EPLD Rules* y presionamos **OK**.



Figura 2.26: Cuadro Design Doctor Settings

2.2.7.5 Selección Estilo de Síntesis de Proyecto Lógico Global (Global Project Logic Synthesis Style)

Podemos seleccionar un estilo de síntesis de lógica para el proyecto que guía el módulo de Sintetizador de Lógica del Compilador durante la compilación. Los dos estilos principales son "minimización de recurso de silicio" y "minimización de retardo". El estilo de síntesis de lógica predefinido para un nuevo proyecto es "Normal". Los valores de opción de lógica en este estilo perfeccionan la lógica del proyecto para el uso de recurso de silicio mínimo.

Para seleccionar un estilo de síntesis de lógica para el proyecto.

1. Seleccionamos de menú **Assign**→**Global Project Logic Synthesis**. Se abre el cuadro de diálogo de Global Protect Logic Synthesis (figura 2.27).

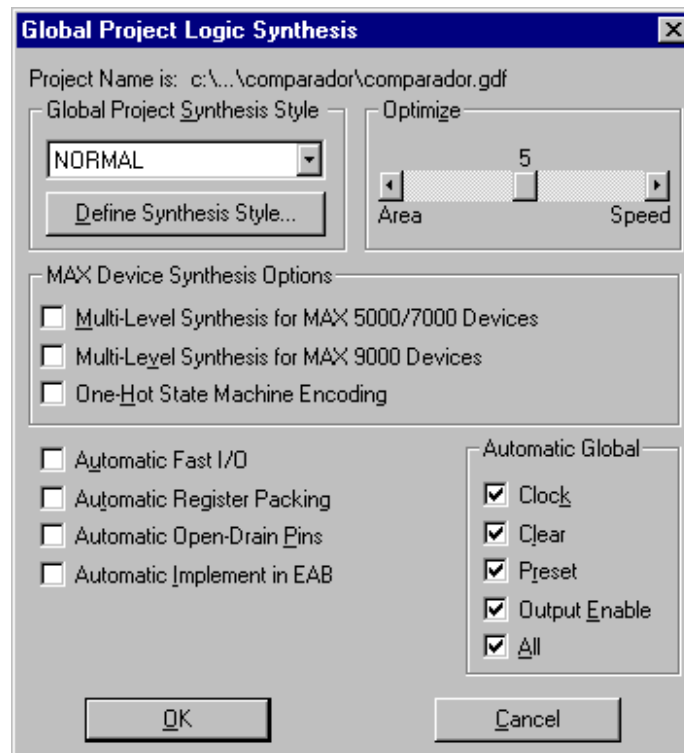


Figura 2.27: Cuadro Global Protect Logic Synthesis

2. Si es necesario, seleccionamos *Normal* en el cuadro *Global Protect Synthesis Style* y presionamos **OK**.

2.2.7.6 Encendido del Timing SNF Extractor

El Compilador puede crear un archivo Simulator Netlist File (**.snf**) que contiene la información de la lógica y los tiempos usados por el Simulador y el analizador de tiempos del **MAX+plus II**.

En una simulación de tiempo, el Simulador del **MAX+plus II** testea el proyecto después de que ha sido sintetizado y optimizado totalmente. La simulación de tiempo es realizada con una resolución de 0.1ns.

Para encender el módulo Timing SNF Extractor (si es necesario):

- Seleccionamos de menú **Processing**→**Timing SNF Extractor**. Cuando el comando está activo aparece en la ventana del compilador como muestra la figura 2.25.

2.2.7.7 Ejecución del Compilador

Para compilar el proyecto:

1. Pulsamos el botón **Start**. Mientras el Compilador procesa el proyecto **compilador**, cualquier información, error, o advertencia aparece en la ventana Message Processor, que se abre automáticamente.
2. Cuando la compilación finaliza, aparecen iconos debajo de los módulos representando los archivos de salidas generados por el Compilador (figura 2.28). Haciendo clic con el botón izquierdo del mouse sobre dichos iconos podemos abrir el archivo correspondiente.
3. Luego utilizamos los controles de ventana de Windows para cerrar las ventanas del compilador

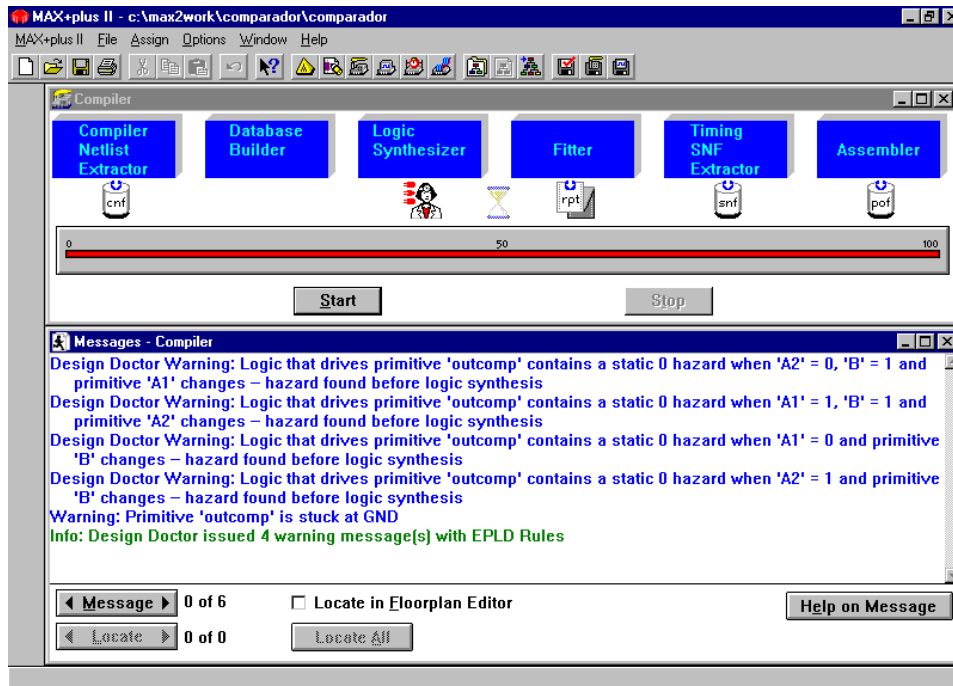


Figura 2.28: Ventanas del Compilador y sus mensajes

Cabe destacar que el Simulador del **MAX+plus II** soporta la simulación funcional para testear el funcionamiento lógico de un proyecto antes de que se sintetice, permitiéndole por lo tanto al diseñador identificar rápidamente y corregir los errores lógicos. En el modo de simulación funcional, los niveles de lógica de salida cambian al mismo tiempo como los vectores de la entrada es decir, no aparece ningún retardo de propagación. Se puede encender la simulación funcional escogiendo del menú **Processing**→**Funcional SNF Extractor**.

2.2.8 Verificación de la jerarquía del proyecto

En éste punto vamos a ver la jerarquía del proyecto **comparador** en la ventana Hierarchy Display.

Para ver la jerarquía de **comparador**:

- Seleccionamos de menú **MAX+plus II**→**Hierarchy Display**, con lo cual se abre la ventana Hierarchy Display que contiene el árbol de jerarquía del proyecto tal como muestra la figura 2.29.

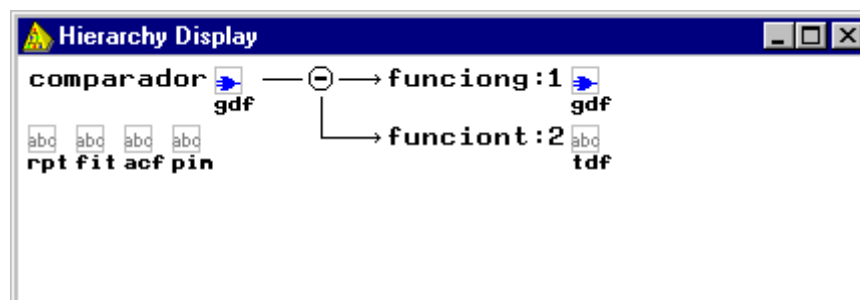


Figura 2.29: Ventana Hierarchy Display

Para abrir los archivos que se encuentran en el árbol representados por iconos, basta con hacer un doble-clic con el botón izquierdo del mouse sobre el icono correspondiente

2.2.9 Verificación del Ajuste en el Floorplan Editor

En ésta parte vamos a utilizar la ventana del Editor Floorplan para ver los resultados de partición y adaptación del Compilador así como la asignación de pines por medio del usuario. También veremos la colocación de lógica del Compilador y editaremos nuestra asignación de pines.

2.2.9.1 Apertura de la ventana del Editor Floorplan

El Editor Floorplan provee dos pantallas : la Vista del Dispositivo (*Device View*) y la Vista del LAB (*LAB View*). La Vista del Dispositivo muestra todos los pines en un dispositivo y sus funciones. La Vista de LAB muestra el interior del dispositivo, incluyendo todos los LABs (*Logic Array Blocks*); las celdas lógicas individuales dentro de cada LAB; y celdas de E/S , celdas integradas y EABs (*Embedded Array Blocks*) si están disponibles en el dispositivo designado. (El dispositivo EPM7064 usado en el proyecto no incluye celdas de E/S, celdas integradas y EABs.) La Vista del LAB también visualiza la ubicación de pines de modo de ver la relación entres pines y recursos de lógica en el interior del dispositivo.

Para ver el proyecto **comparador** en la ventana del Editor Floorplan:

1. Seleccionamos del menú **MAX+plus II**→**Floorplan Editor**, y se abre la ventana del Floorplan Editor mostrando la ventana que fue usada por ultima vez para examinar el dispositivo seleccionado para el proyecto.

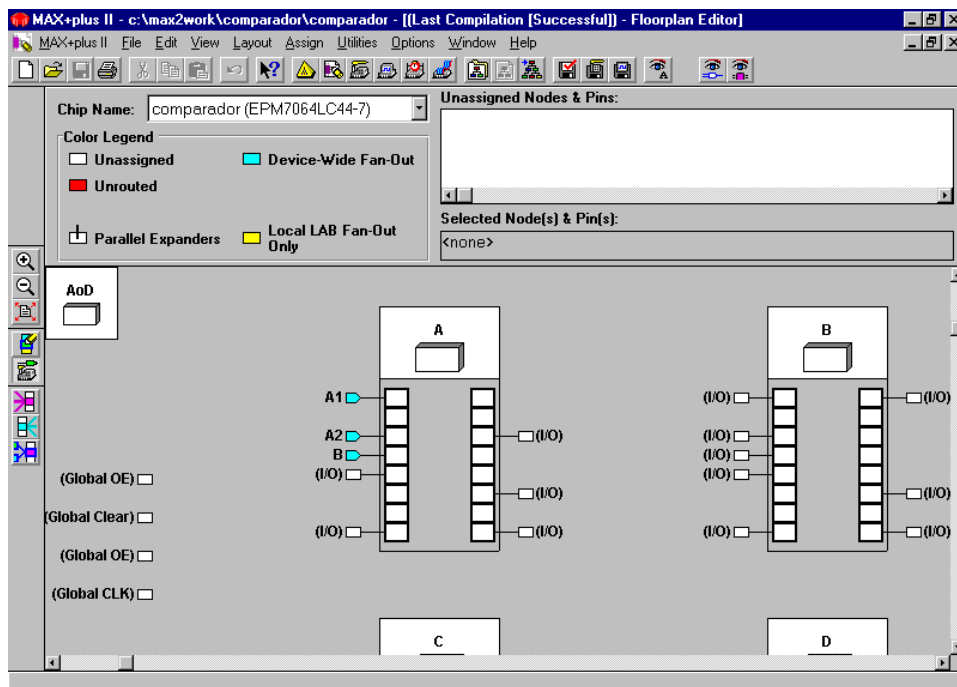


Figura 2.30: Ventana del Floorplan (Vista de LAB)

- Si es necesario, seleccionamos los comandos del menú **Layout**→**LAB View** y **Layout**→**Last Compilation Floorplan**. El proyecto **comparador** se visualiza en la ventana del Floorplan Editor como muestra la figura 2.30, mostrando los cuatro LABs del dispositivo (A, B, C y D) y sus macroceldas.

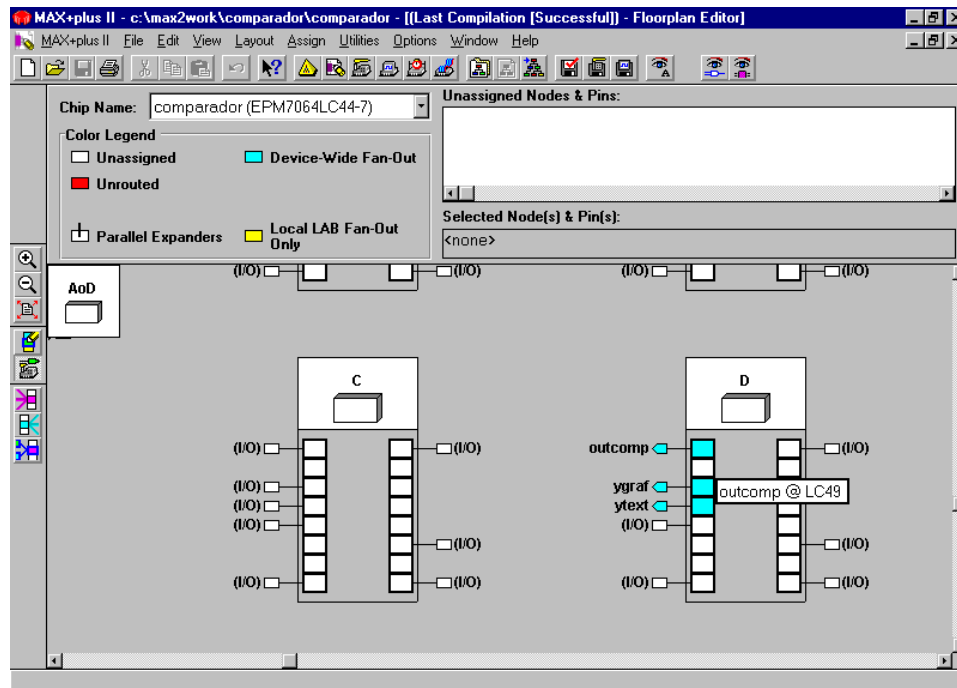


Figura 2.31: Ventana del Floorplan (Vista de LAB)

- El “texto globo” despliega información del elemento al cual está apuntando el puntero del mouse, como por ejemplo en la figura 2.31 situamos el puntero en la macrocelda (49) de la salida **outcomp** ubicada en el LAB D.

2.2.9.2 Comando Back-Annotate Project y Edición de Asignaciones

El Editor Floorplan nos permite ver y editar las asignaciones actuales, que se encuentran almacenadas en el archivo de Asignación y Configuración del proyecto (*Assignment & Configuration File*)(.acf).

Luego de haber compilado el proyecto, podemos editar las asignaciones del Compilador, que se encuentran almacenadas en el archivo de Ajuste del proyecto (*Fit File*)(.fit), ejecutando el comando *Back-Annotate Project* y luego eligiendo las asignaciones actuales del Floorplan.

- Elegimos del menú **Assign**→**Back-Annotate Project**, con lo cual se abre la ventana de dialogo del Back-Annotate Project, tal como muestra la figura 2.32.
- Encendemos la opción *Chip, Logic Cells, Pins & Device*

3. Pulsamos **OK**. MAX+plus II copia las asignaciones de pines, celdas lógicas, chip y dispositivo del archivo de Ajuste al archivo de Asignación y configuración.
4. Seleccionamos del menú **Layout**→**Current Assignment Floorplan**. La ventana del Editor Floorplan muestra las asignaciones actuales para el proyecto **comparador**.

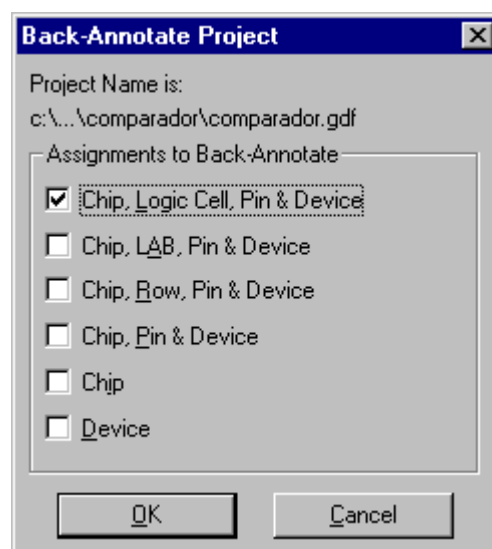


Figura 2.32: Cuadro Back-Annotate Project

Vamos ahora a modo de ejemplo reubicar algunos pines en el dispositivo. Vamos a reasignar los pines de salida a tres nuevas ubicaciones en el LAB B.

Para editar la asignación del pin `outcomp`:

1. Encendemos la opción, si es necesario, del menú **Options**→**Show Moved Nodes in Gray**, con lo cual aparece en el cuadro *Color Legend*, junto con los demás, el color gris que identificara a los pines que son cambiados de lugar.
2. Hacemos un clic con el botón izquierdo del mouse sobre el pin `outcomp` para seleccionarlo, notaremos que su borde toma el color azul y en el cuadro *Selected Node(S) & Pin(S)* aparece `outcomp @ 33(I/O)`.
3. Con el boton izquierdo del mouse presionado arrastramos el pin a un pin de E/S desocupado del LAB B (por ejemplo el 16). La Asignación aparece en color gris en su nueva ubicación tal como se ve en la figura 2.33.
4. Repetimos los tres pasos anteriores para pines `ygraf` e `ytext`, de modo que quede como en la figura 2.34.

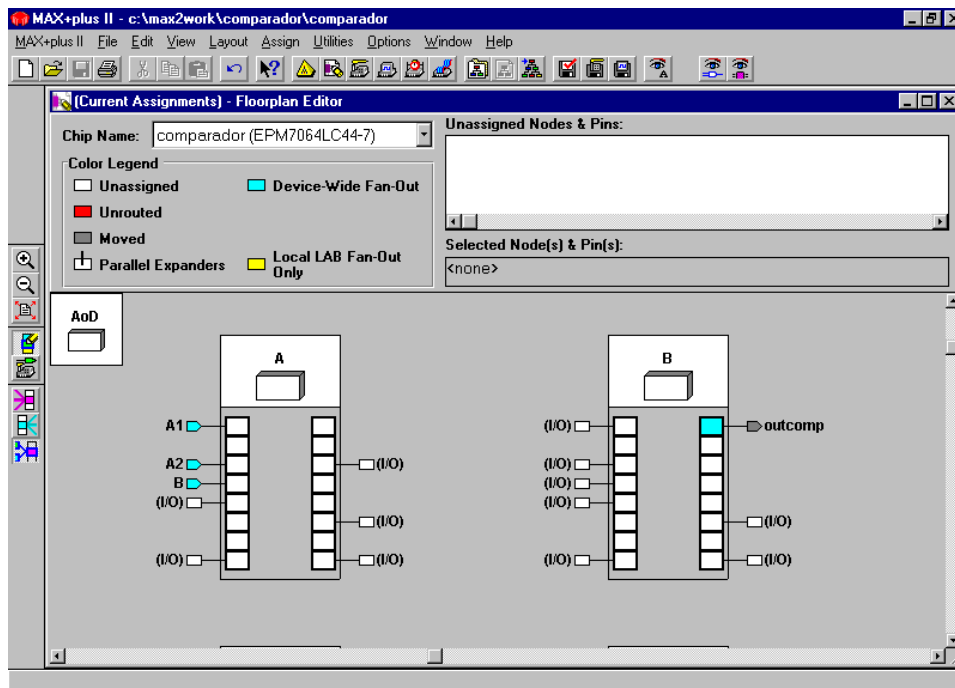


Figura 2.33: Ventana del Floorplan (Vista de LAB)

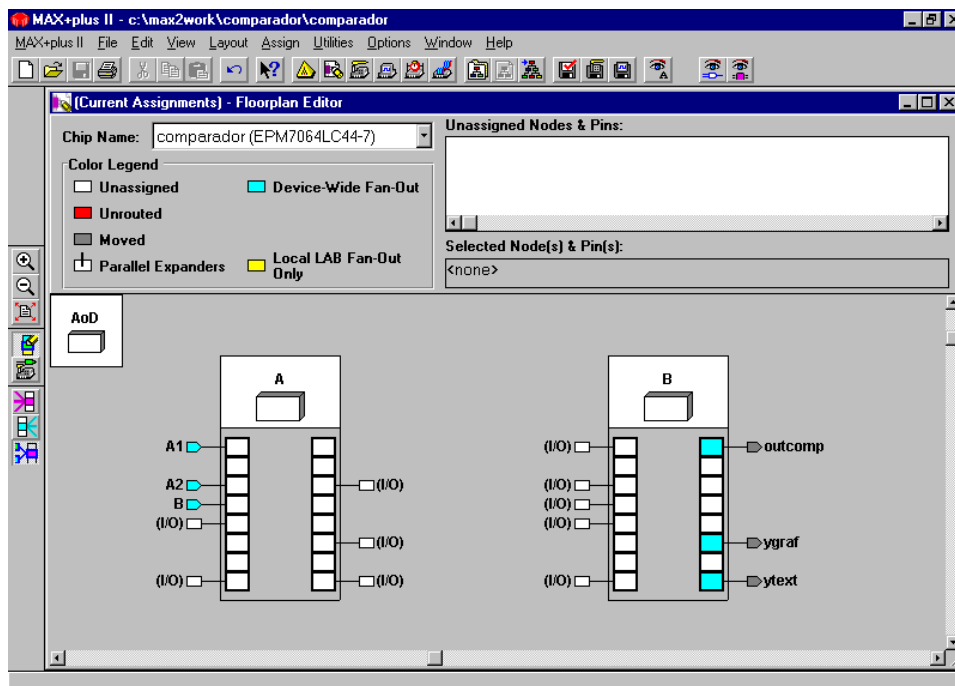


Figura 2.34: Ventana del Floorplan (Vista de LAB)

2.2.9.3 Recompilación del Proyecto

Una vez editada la asignación de los pines `outcomp`, `ygraf` e `ytext`; debemos recompilar el proyecto **comparador** para verificar si la nueva asignación es legal para el dispositivo EPM7064LC44.

1. Seleccionando del menú **MAX+plus II**→**Compiler**, abrimos la ventana del Compilador.
2. Pulsamos el botón **Start**. El Compilador comienza a procesar el proyecto, basado en la nueva asignación de pines.
3. Seleccionando del menú **MAX+plus II**→**Floorplan Editor** para retornar a la ventana del Editor Floorplan.
4. Elegimos del menú **Layout**→**Last Compilation Floorplan**. Los pines `outcomp`, `ygraf` e `ytext` aparecen ahora en los pines 16, 14 y 13 respectivamente.

2.2.9.4 Vista del dispositivo

Vamos a ver ahora la ubicación física de los pines en el dispositivo.

- Seleccionamos del menú **Layout**→**Device View**, con lo cual se abre la vista del dispositivo, tal como muestra la figura 2.35

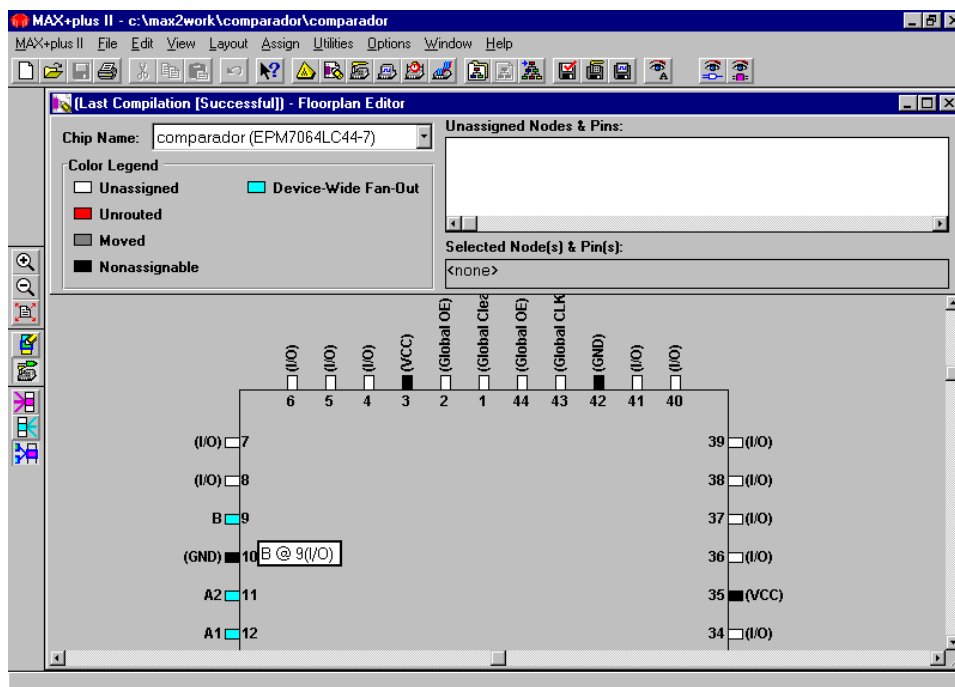


Figura 2.35: Ventana del Floorplan (Vista del Dispositivo)

2.2.10 Edición del archivo Simulator Channel file

El Simulador del **MAX+plus II** proporciona flexibilidad y control para planear proyectos de simples o multi-dispositivo. El Simulador usa un archivo de netlist de simulación binario que se genera durante la compilación para realizar simulaciones del proyecto funcionales, de tiempo, o simulación del multi-dispositivos vinculados. Se pueden definir los estímulos de entrada con un vector de lenguaje de entrada o se pueden dibujar directamente las formas de onda con el Editor de Formas de onda del **MAX+plus II**. Pueden verse los

resultados de la simulación en el Editor de Formas de onda y pueden ser impresos como los archivos de forma de onda.

En éste punto vamos a aprender como se usa el Editor de Formas de onda para crear y editar vectores de simulación de entrada para llevar a cabo una tarea específica.

2.2.10.1 Creación del archivo Simulator Channel File

Para crear el archivo SCF:

1. Seleccionamos del menú **File**→**New**, seleccionamos *Waveform Editor file*, la extensión *.scf* y pulsamos **OK** para crear un nuevo archivo.
2. Seleccionamos del menú **File**→**End Time** y tipeamos 420ns . El tiempo final determina cuando el Simulador va a dejar de aplicar los vectores de entrada durante la simulación.
3. Seleccionamos del menú **Option**→**Grid Size**, tipeamos 50ns y presionamos **OK**.
4. Seleccionamos del menú **Node**→**Enter Nodes from SNF**, se abre la ventana mostrada en la figura 2.36.

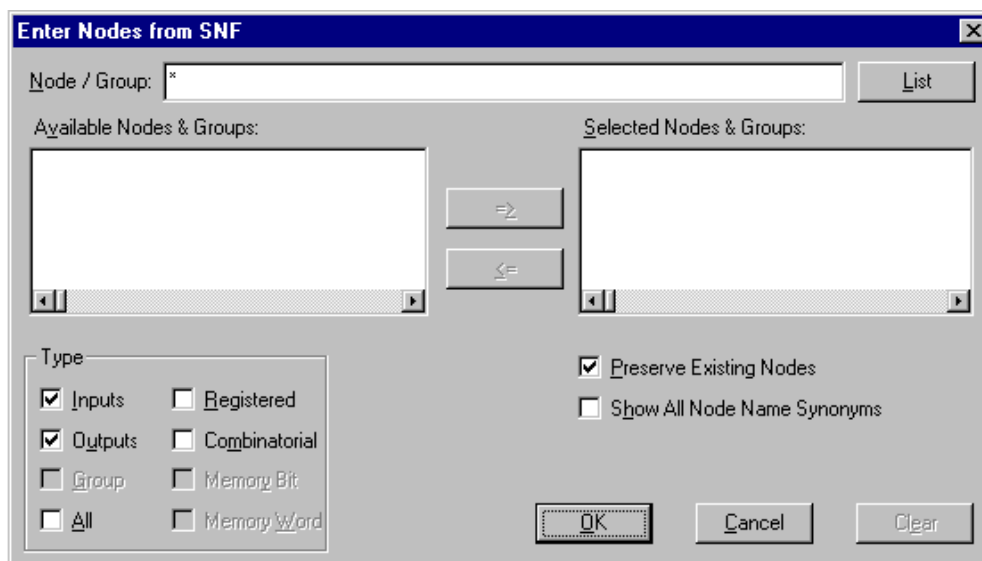



Figura 2.36: Ventana Enter Nodes from SNF

5. Elegimos **List** para listar los nodos de entrada (I) y salida (O) disponibles.
6. Apretamos el botón izquierdo del mouse en el nodo más alto del cuadro *Available Nodes & Groups* y arrastramos el mouse hacia abajo para resaltar todas las entradas y salidas.
7. Pulsamos el botón  para copiar los nodos seleccionados dentro del cuadro *Selected Nodes & Groups*.
8. Presionamos **OK** con lo cual los nodos indicados aparecen en la ventana de edición, tal como se ve en la figura 2.37.

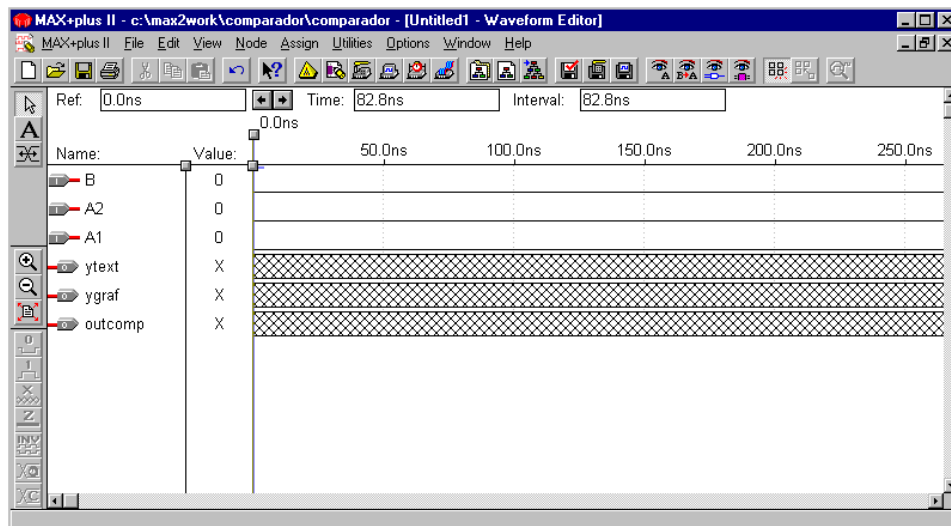


Figura 2.37: Ventana del Editor de Forma de onda

9. Seleccionamos del menú **File**→**Save As**. El nombre **comparador.scf** aparece automáticamente en el cuadro *File Name*.
10. Presionamos **OK** para salvar el archivo **comparador.scf**.

2.2.10.2 Reorganización del orden de los nodos

Para reorganizar la ubicación de los nodos en el Editor de Formas de onda:

1. Pulsando el botón izquierdo del mouse sobre el icono del nodo **A1** y, sin soltarlo, desplazamos el cursor del mouse hasta situarlo como el primero de la lista ; luego soltamos el botón.
2. Repetimos para situar el nodo **A2** debajo del **A1** y el nodo **ygraf** debajo del **B**. El aspecto de la pantalla queda como muestra la figura 2.38.

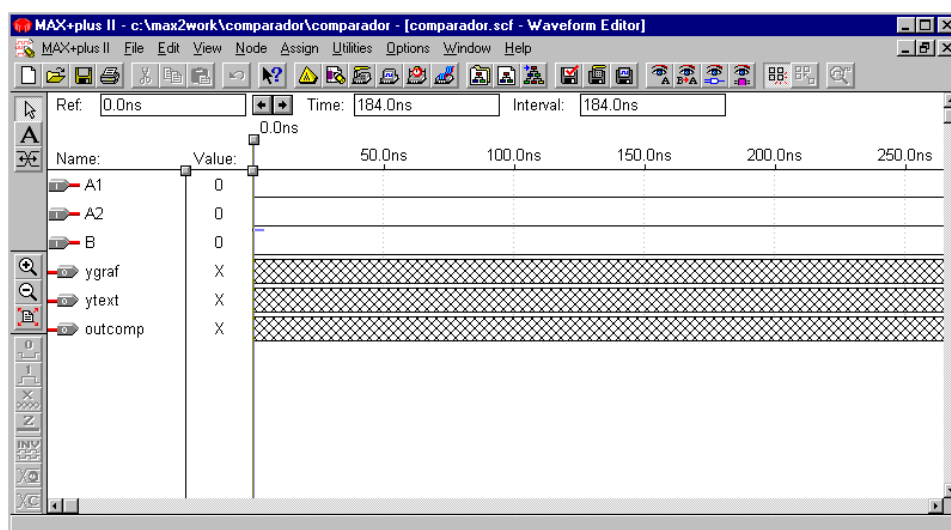




Figura 2.38: Ventana del Editor de Forma de onda

2.2.10.3 Edición de las formas de onda de los nodos de entrada

Debemos ahora editar las formas de onda de entrada para proveer los vectores de entrada para la simulación. Mediante éstas formas de onda debemos representar las ocho posibles combinaciones de las variables de entrada.

Para editar las formas de onda de onda

1. Activamos el icono , para visualizar todo el tiempo simulación.
2. Seleccionamos el nodo A1, situando el cursor del mouse encima del nombre del nodo y pulsando el botón izquierdo del mouse.
3. Activamos el icono . Aparecerá la ventana de la figura 2.39.

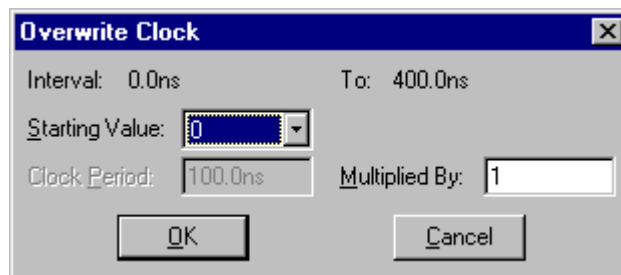


Figura 2.39: Ventana del Overwrite Clock

4. Aceptamos los valores por defecto y pulsamos el botón **OK**. Observamos que aparece dibujada una señal periódica en el archivo de edición.
5. Repetimos la operación para A1, pero escribiendo un 2 en la casilla *Multiplied By*. Observe los resultados para entender que el campo *Multiplied By* sirve para controlar la duración de los niveles lógicos.
6. Repetimos de nuevo para B, escribiendo en *Multiplied By* el valor 4, respectivamente. El archivo debe quedar con un aspecto como el de la figura 2.40.

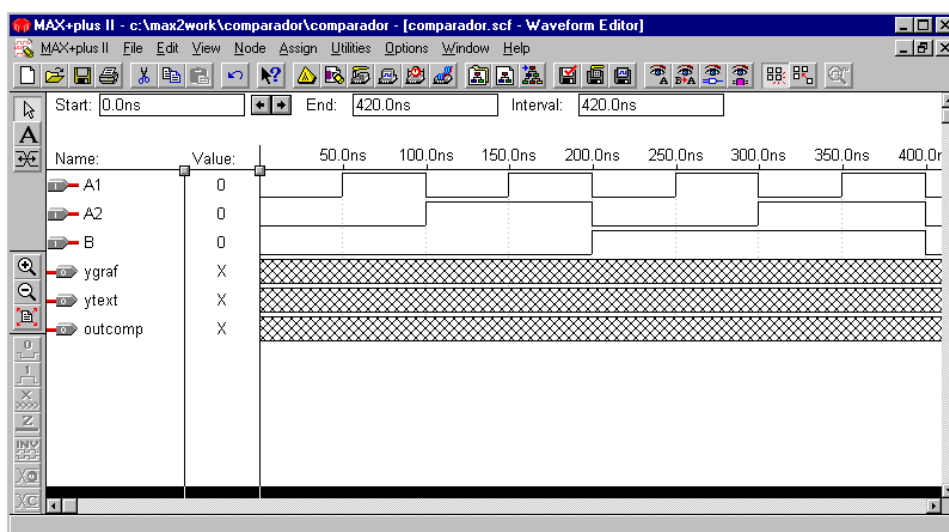




Figura 2.40: Ventana del Editor de Forma de onda

7. Luego guardamos el archivo seleccionando del menú **File**→**Save**, y lo cerramos con los comandos de Windows.

Notar que para este caso utilizamos formas de onda de entrada como si fueran relojes de distintas frecuencias, para poder lograr ordenadamente todas las combinaciones de variables de entrada posibles.

También podemos editar las formas de onda seleccionando los intervalos en que un dado nodo posee un nivel y asignándole dicho nivel mediante la

activación del icono  para nivel alto; o el icono  para nivel bajo.

2.2.11 Simulación del Proyecto

En éste punto vamos a usar el Simulador para simular el proyecto **comparador**, para eso usaremos el archivo Simulator Channel llamado **comparador.scf** creado en el punto 2.2.10.1.

1. Seleccionamos del menú **MAX+plus II**→**Simulator**, se abre la ventana del Simulador tal como muestra la figura 2.41.

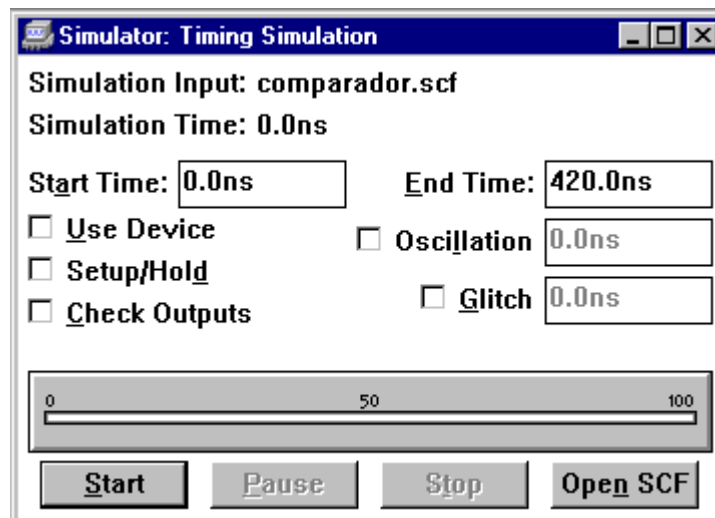


Figura 2.41: Ventana del Simulador

2. Presionamos el botón **Start**, para iniciar la simulación. Cuando ésta finalice aparecerá un cuadro de mensajes como el que muestra la figura 2.42.

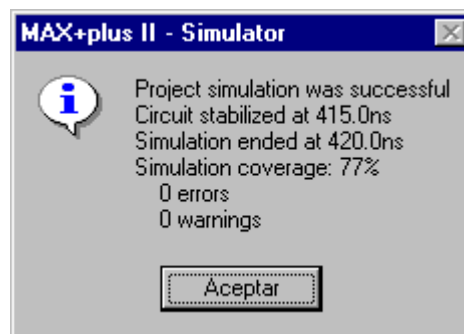


Figura 2.42: Cuadro de finalización de la simulación

3. Presionamos el botón **Aceptar**.

2.2.12 Análisis de las Salidas de la Simulación

Debemos ahora verificar que las salidas de nuestro proyecto responden correctamente o sea según la tabla de verdad que detallamos a continuación.

$$y_{graf} = y_{text} = Y_g = Y_t = A_1 \cdot \overline{A_2} + \overline{B}$$

$$outcomp = Y_g \oplus Y_t$$

B	A ₂	A ₁	Y _g ≡ y _{graf}	Y _t ≡ y _{text}	outcomp
0	0	0	1	1	0
0	0	1	1	1	0
0	1	0	1	1	0
0	1	1	1	1	0
1	0	0	0	0	0
1	0	1	1	1	0
1	1	0	0	0	0
1	1	1	0	0	0

Para visualizar los resultados de la simulación, con el Editor de Formas de onda (figura 2.43), pulsamos el botón **Open SCF** en la ventana del simulador.

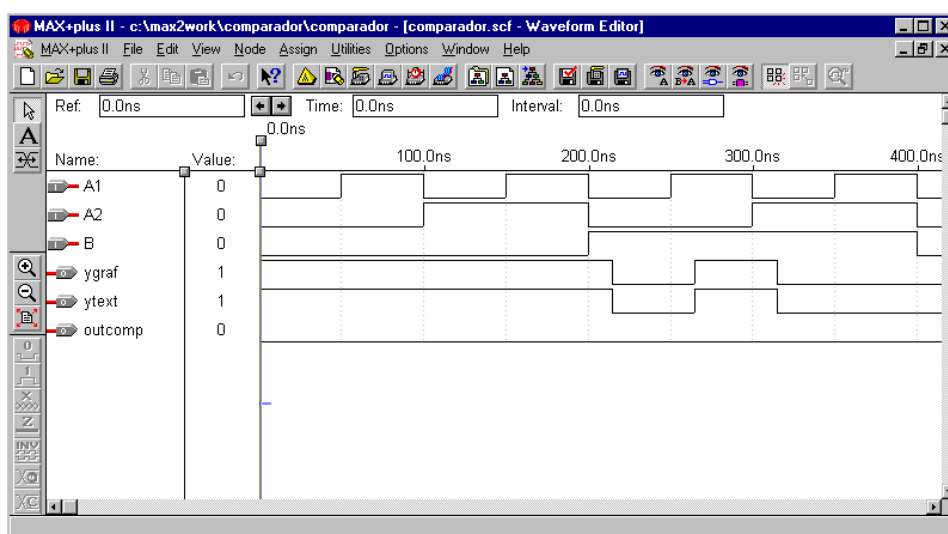



Figura 2.43: Ventana del Editor de Forma de onda

Como vemos se verifican los resultados de la simulación con los de la tabla de verdad, con un retardo debido a los niveles de lógica.

Podemos medir éste retardo moviendo el cursor de referencia a través de las flechas  ubicadas al lado del cuadro **Ref**. Presionamos la flecha derecha hasta ubicar el cursor de referencia en la primer transición de '1' a '0' de la salida ygraf (ó ytext), con lo cual el cuadro **Ref** indica 215.0ns (figura 2.44). Midiendo de la misma forma la transición correspondiente de la entrada indica 200.0ns; con lo cual el retardo es de **15.0ns**.

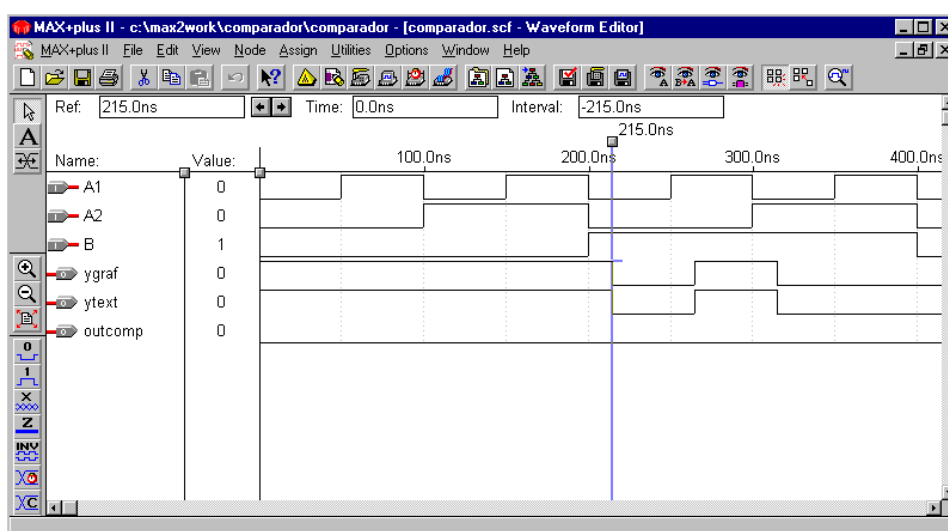


Figura 2.44: Ventana del Editor de Forma de onda

Finalmente cerramos el editor de formas de onda, la ventana del simulador y la ventana principal con los comandos de Windows.

2.2.13 Ejemplos utilizando macro y megafunciones

Vamos ahora a modo de ejemplo a implementar en un dispositivo EPM7032LC44 de la familia MAX7000, un contador binario ascendente de 5 bits, con entrada de clear asincrónico. Para esto generaremos dos proyectos:

- En el primero llamado **macrocont**, implementaremos el contador mediante dos contadores ascendentes normalizados 74LS161 de 4 bits cada uno por medio del editor gráfico.
- En el segundo llamado **megacont**, implementaremos el contador mediante la función `lpm_counter` de la Librería de Módulos Parametrizados (LPM) por medio tanto del editor gráfico como por medio del editor de texto.

Luego compararemos la cantidad de macroceldas utilizadas por cada uno mediante el archivo de Reporte (*Report File*)(.rpt). y por último simularemos para verificar el correcto funcionamiento de cada uno

2.2.13.1 Contador utilizando macrofunciones

Inicialmente creamos un nuevo archivo de diseño gráfico, lo guardamos con el nombre **macrocont.gdf** dentro de un directorio con el mismo nombre y especificamos el nombre del proyecto como **macrocont** de la misma forma que hicimos en los puntos 2.2.4.1 y 2.2.4.2.

Luego con los mismos pasos que en el punto 2.2.4.3, entramos al gráfico 2 contadores 74LS161 (`74161`), un negador (`not`), 2 pines de entrada (`input`), y 5 pines de salida (`output`). Conectamos los símbolos y nombramos los pines de modo que quede como muestra en la figura 2.45.

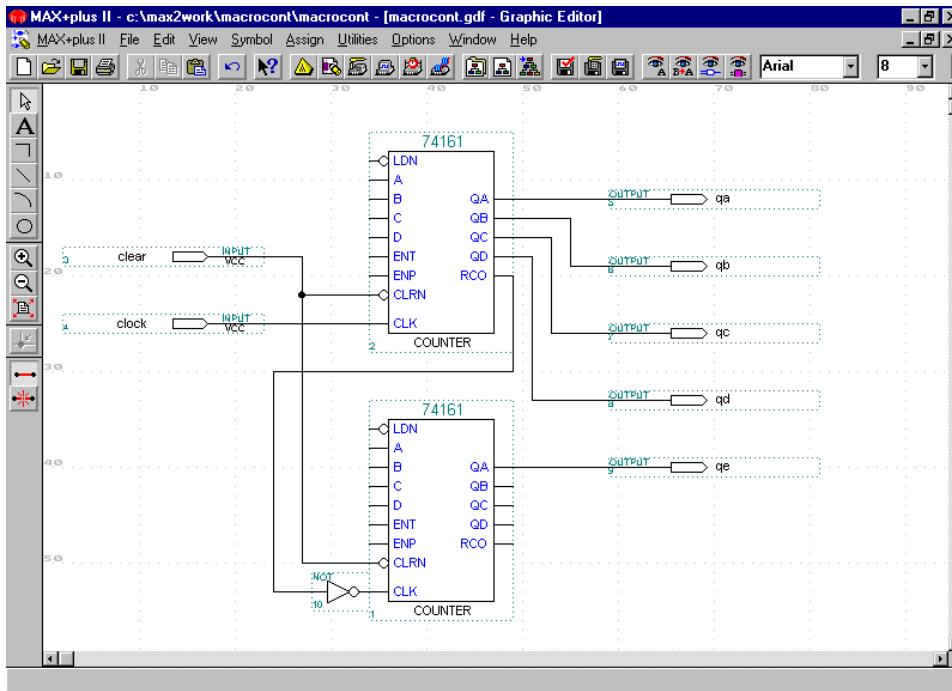


Figura 2.45: Ventana del Editor Gráfico

Para guardar el archivo y chequear los errores repetimos los pasos del punto **2.2.4.3.5**.

Como paso siguiente compilamos el proyecto **macrocont** asignándole el dispositivo EPM7032LC44 de la familia MAX7000. (punto **2.2.7**). Una vez compilado, en la ventana del Compilador abrimos el archivo de Reporte haciendo un doble clic con el botón izquierdo del mouse sobre el icono rpt que aparece debajo del módulo Fitter. Éste archivo nos muestra, como una parte de su contenido, una tabla con la cantidad de recursos utilizados:

Device-Specific Information: c:\max2work\macrocont\macrocont.rpt
macrocont

**** RESOURCE USAGE ****

Logic Array Block	Logic Cells	I/O Pins	Shareable Expanders	External Interconnect
A: LC1 - LC16	0/16(0%)	0/16(0%)	0/16(0%)	0/36(0%)
B: LC17 - LC32	5/16(31%)	5/16(31%)	1/16(6%)	4/36(11%)

Como podemos ver utiliza 5 macroceldas, es decir 5 *flip-flops*, uno por cada bit del contador.

Ahora vamos a crear un archivo *Simulator Channel File* (punto **2.2.10**) **macrocont.scf** de forma tal que quede como muestra la figura 2.46.

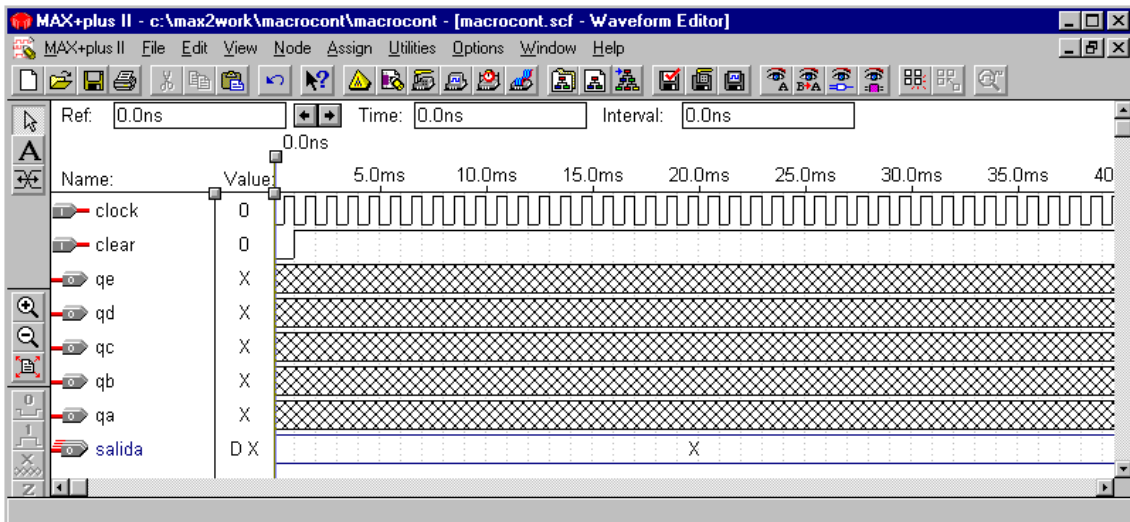


Figura 2.46: Ventana del Editor de Forma de onda

En donde el pin salida simplemente surge de agrupar los pines qe, qd, qc, qb, y qa; o sea que equivale a:

```

salida[4] = qe
salida[3] = qd
salida[2] = qc
salida[1] = qb
salida[0] = qa
    
```

Finalmente simulamos como el punto 2.2.11, y verificamos en el Editor de Formas de Onda (figura 2.47) que actúa como un contador binario ascendente de 5 bits.

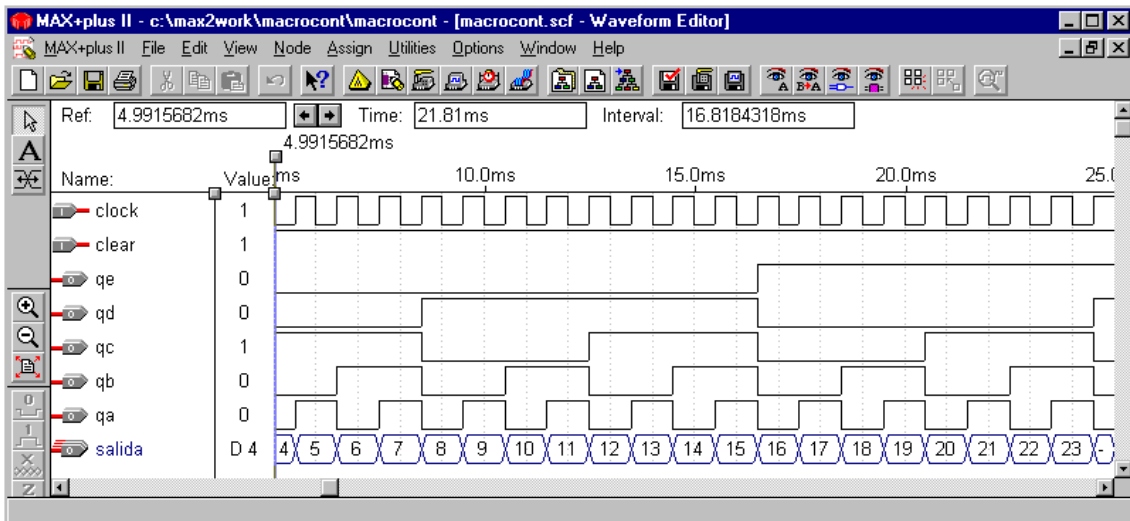
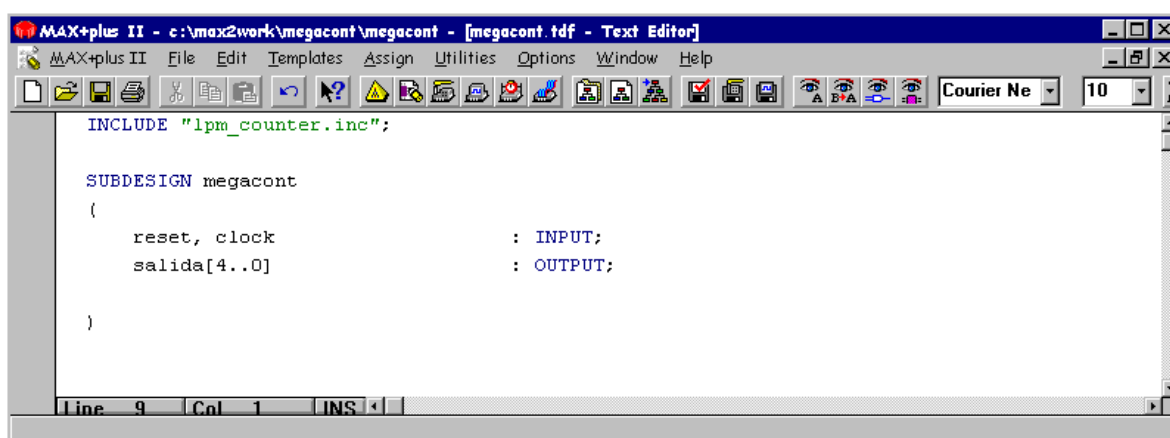


Figura 2.47: Ventana del Editor de Forma de onda

2.2.13.2 Contador utilizando megafunciones LPM

a) Utilizando el Editor de texto

De la misma forma que en el punto 2.2.5.1 creamos un nuevo archivo de texto, lo guardamos con el nombre **megacont.tdf** dentro de un directorio con el mismo nombre y especificamos el nombre del proyecto como **megacont**. Luego con los mismos pasos que en el punto 2.2.5.3, entramos la plantilla de AHDL *Include Statement* y haciendo doble-clic con el botón izquierdo del mouse el la variable `__include_filename`, tipeamos `lpm_counter`, para declarar la función LPM que vamos a utilizar. Luego entramos la plantilla *Subdesign Section* tipeamos el nombre del diseño (`megacont`), las dos entradas (`reset` y `clock`), y las cinco salidas (`salida[4..0]`) de modo que quede como muestra en la figura 2.48.



```

MAX+plus II - c:\max2work\megacont\megacont - [megacont.tdf - Text Editor]
MAX+plus II File Edit Templates Assign Utilities Options Window Help
INCLUDE "lpm_counter.inc";

SUBDESIGN megacont
(
  reset, clock           : INPUT;
  salida[4..0]           : OUTPUT;
)
Line 9 Col 1 INS
  
```

Figura 2.48: Ventana de Editor de Texto

Ahora debemos entrar el contador parametrizado, el cual cuenta con los parámetros mostrados en la tabla 2.1:

Parámetro	Tipo	requerido?	Descripción
LPM_WIDTH	Entero	Si	El número de bits del contador, o el ancho de los puertos <code>q[]</code> y <code>data[]</code> , si son usados.
LPM_DIRECTION	Cadena	No	Los valores son "UP", "DOWN", y "UNUSED". Si el parámetro LPM_DIRECTION es usado el puerto <code>updown</code> no puede ser conectado. Cuando el puerto <code>updown</code> no es conectado, el valor predeterminado para el parámetro LPM_DIRECTION es "UP".
LPM_MODULUS	Entero	No	El máximo conteo, más uno. El número de únicos estados en el ciclo del contador. Si el valor de carga es más grande que el parámetro LPM_MODULUS, la conducta del contador no se especifica.

LPM_AVALUE	Entero / Cadena	No	Valor constante que es cargado cuando <code>aset</code> es alto. Si el valor especificado es más grande que <code><modulus></code> , la conducta del contador es un nivel de lógica indefinido (X), donde <code><modulus></code> es <code>LPM_MODULUS</code> o $2^{\text{LPM_WIDTH}}$. El parámetro <code>LPM_AVALUE</code> es limitado a un máximo de 32 bits. Altera recomienda que usted especifique este valor como un número decimal para los diseños de AHDL.
LPM_SVALUE	Entero / Cadena	No	Valor constante que es cargado en el flanco ascendente de reloj cuando <code>sset</code> o <code>sconst</code> es alto. Debe ser usado si <code>sconst</code> es usado. Altera recomienda que usted especifique este valor como un número decimal para los diseños de AHDL.
LPM_INT	Cadena	No	Le permite especificar los parámetros específicos de Altera en los Archivos de Diseño de VHDL. El valor predeterminado es "UNUSED".
LPM_TYPE	Cadena	No	Identifica el nombre de la entidad LPM en los Archivos de Diseño de VHDL.
CARRY_CNT_EN	Cadena	No	Parámetro específico de Altera. Los valores son "SMART", "ON", "OFF", o "UNUSED". Habilita las funciones <code>lpm_counter</code> para propagar la señal <code>cnt_en</code> a través de la cadena de carry. En algunos casos, el valor del parámetro <code>CARRY_CNT_EN</code> puede tener un impacto ligero en la velocidad, por lo tanto usted deseará apagarlo. El valor predefinido es "SMART", el cual proporciona una mejor relación entre tamaño y velocidad.
LABWIDE_SCLR	Cadena	No	Parámetro específico de Altera. Los valores son "ON", "OFF", o "UNUSED". El valor predefinido es "ON". Permite desactivar la característica LAB-wide <code>sclr</code> encontrada en dispositivos FLEX 6000.

Tabla 2.1: Parámetros del contador

A continuación vamos a declarar el contador con los parámetros utilizados, dentro del Editor de texto. Para esto, en una nueva línea luego de la sección *Subdesign*, tipeamos la palabra clave `VARIABLE` y presionamos **Enter** (↵). Luego asignándole el nombre `contador` y una cantidad de 5 al parámetro

LPM_WIDTH (número de bits del contador), incluimos el contador de 5 bits dentro de nuestro proyecto, tal como lo muestra la figura 2.49:

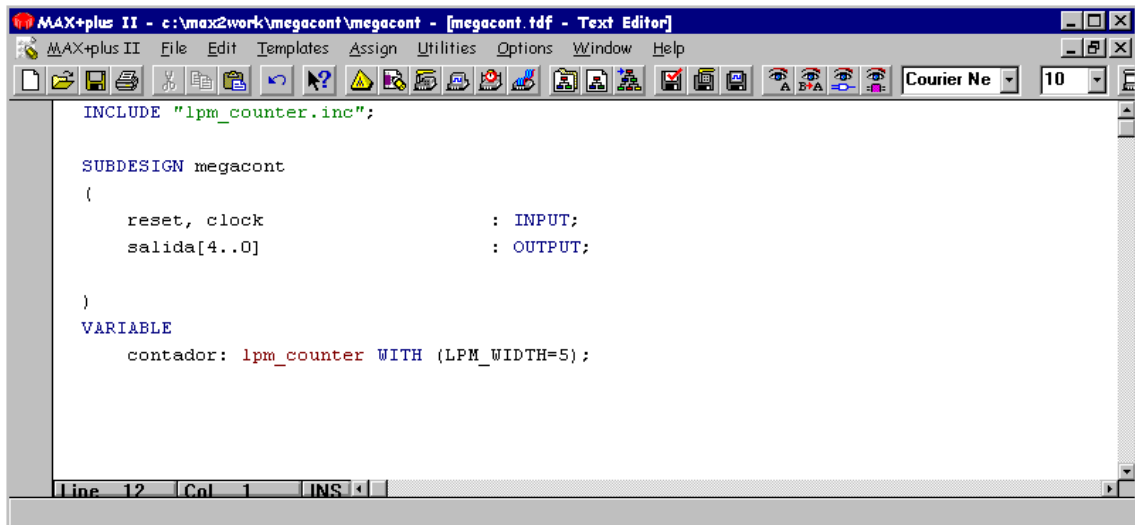


Figura 2.49: Ventana de Editor de Texto

Ahora debemos conectar las entradas y salidas de nuestro proyecto a los puertos de entrada y salida del contador parametrizado. Dichos puertos se enumeran respectivamente en las tablas 2.2 y 2.3:

Nombre del puerto	Requerido?	Descripción	Comentario
data[]	No	Entrada de datos paralela del contador	Puerto de Entrada de ancho LPM_WIDTH
clock	Si	Reloj de activación por flanco ascendente	
clk_en	No	Entrada de habilitación de reloj. Habilita todas las actividades sincrónicas	Predeterminado = 1 (habilitado)
cnt_en	No	Entrada de habilitación del contador. Desactiva el contador cuando está en bajo (0) sin afectar sload, sset, o sclr	Predeterminado = 1 (habilitado)
updown	No	Controla la dirección del contador. Alto (1) = contador ascendente. Bajo (0) = contador descendente.	Predeterminado = ascendente (1). Si es usado el parámetro LPM_DIRECTION el puerto updown no puede ser conectado. Si LPM_DIRECTION no es usado, el puerto updown es

cin	No	Entrada de carry del bit de menor orden	opcional Predeterminado = 1 (Vcc)
aclr	No	Entrada de Clear asincrónico	Predeterminado = 0 (deshabilitado). Si ambos aset y aclr son usados y afirmados aclr anula aset.
aset	No	Entrada de establecimiento asincrónico	Predeterminado = 0 (deshabilitado). Fija todas las salidas q[] a 1 o al valor especificado por LPM_AVALUE. Si ambos aset y aclr son usados y afirmados aclr anula aset.
aload	No	Entrada de carga asincrónica. Carga asincrónicamente el contador con el valor de la entrada data.	Predeterminado = 0 (deshabilitado). Si se utiliza aload, se debe conectar data[].
sclr	No	Entrada de Clear sincrónico. Limpia el contador en el próximo flanco activo del reloj	Predeterminado = 0 (deshabilitado). Si ambos sset y sclr son usados y afirmados sclr anula sset.
sset	No	Entrada de establecimiento sincrónico. Fija el contador en el próximo flanco activo del reloj	Predeterminado = 0 (deshabilitado). Fija todas las salidas q[] a 1 o al valor especificado por LPM_SVALUE. Si ambos sset y sclr son usados y afirmados sclr anula sset.
sload	No	Entrada de carga sincrónica. Carga el contador con data[] en el próximo flanco activo del reloj.	Predeterminado = 0 (deshabilitado). Si se utiliza sload, se debe conectar data[].

Tabla 2.2: Puertos de entrada

Nombre del puerto	Requerido?	Descripción	Comentario
Q[]	No	Salida de datos del contador	Puerto de Salida de ancho LPM_WIDTH. Cualquier q[] o al menos uno de los puertos eq[15..0] debe ser conectado
eq[15..0]	No	Salida descifrada del contador. Se activa en alto cuando el	Solo (AHDL). Cualquier q[] o al menos uno de los puertos eq[15..0] debe ser

cout	No	<p>contador alcanza el valor de conteo especificado.</p> <p>Carry de salida del bit mas significativo (MSB)</p>	<p>conectado. Hasta c puertos pueden ser usados ($0 \leq c \leq 15$). Solo los 16 valores mas bajos de conteo pueden ser descifrados. Cuando el valor de conteo es c, la salida eq_c se fija en alto (1). Por ejemplo cuando el conteo es 0, $eq_0 = 1$; cuando el conteo es 1, $eq_1 = 1$. Las salidas descifradas para valores de conteo de 16 o mayores requieren decodificación externa. Las salidas $eq[15..0]$ son asincrónicas para las salidas $q[]$.</p>
------	----	---	---

Tabla 2.3: Puertos de salida

Cabe recalcar que estas tablas se encuentran en el menú **Help**→**AHDL**, cliqueando en el vínculo Megafunctions/LPM y luego en `lpm_counter`.

Para conectar los puertos utilizados con los pines, lo hacemos de la misma forma que en el punto **2.2.5.5**, de modo que quede como se ve en la figura 2.50:

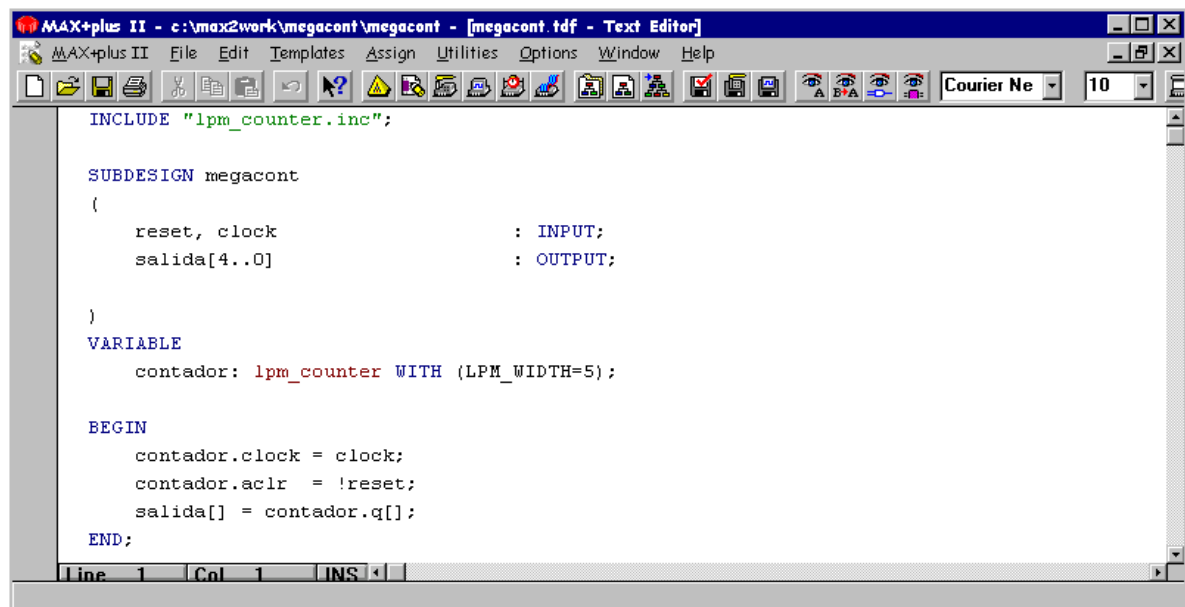


Figura 2.50: Ventana de Editor de Texto

b) Utilizando el Editor de diseño gráfico

Creamos un nuevo archivo de diseño gráfico, lo guardamos con el nombre **megacont.gdf** dentro de un directorio con el mismo nombre y especificamos el nombre del proyecto como **megacont**.

Luego con los mismos pasos que en el punto 2.2.4.3.1, entramos al gráfico 1 contador LPM (`lpm_counter`), un negador (`not`), 2 pines de entrada (`input`), y 1 pin de salida (`output`). Al entrar el contador se abrirá el cuadro de edición de los puertos y los parámetros del contador tal como muestra la figura 2.51.

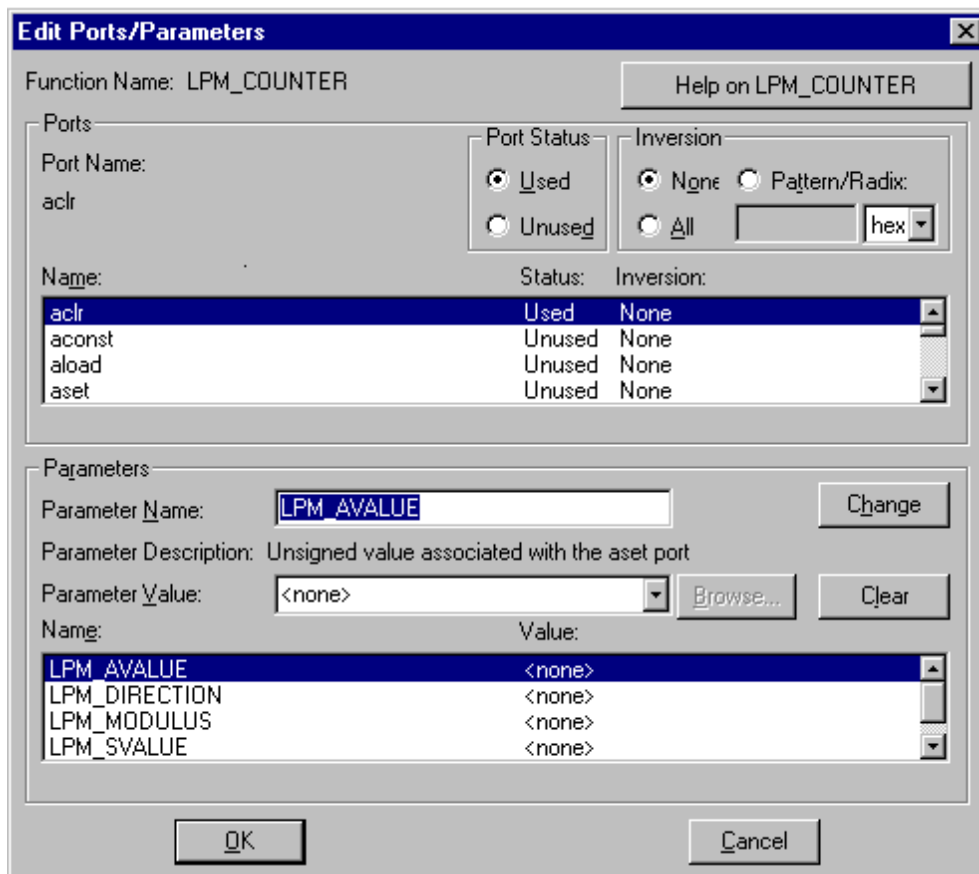


Figura 2.51: Cuadro Edit Ports/Parameters

Dentro del cuadro *Ports* tomamos como puertos usados (*Used*), los puertos `aclr` (clear asincrónico) y `clock`. Esto lo logramos seleccionando el puerto en el cuadro *Name*, y en el cuadro *Port Status* lo marcamos como usado (*Used*) o no usado (*Unused*).

Dentro del cuadro *Parameters* asignamos al parámetro `LPM_WIDTH` (*Parameter Name*) el valor 5 (*Parameter Value*). Éste parámetro indica el número de bits del contador.

Podemos ayudarnos pulsando el botón `Help on LPM_COUNTER` que se encuentra dentro del cuadro de la figura 2.51, con lo cual se abre la ayuda que nos indica entre otras cosas qué es y qué hace cada puerto y cada parámetro.

Luego conectamos los símbolos y nombramos los pines de modo que quede como muestra en la figura 2.52.

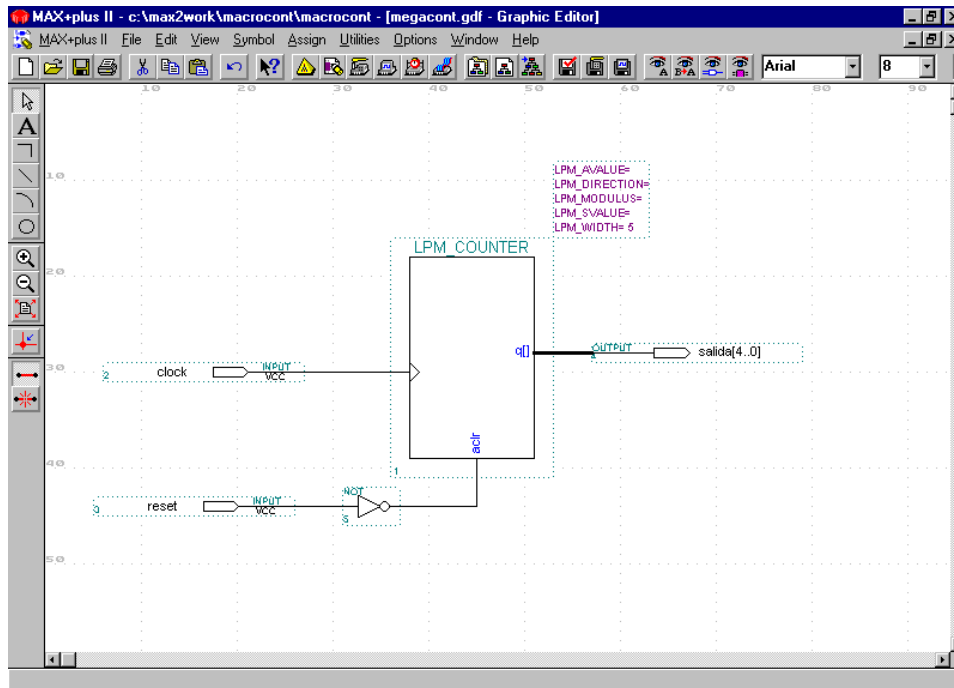


Figura 2.52: Ventana del Editor Gráfico

Ambos archivos **megacont** (.tdf y .gdf) se comportan de la misma forma, por lo tanto elegimos cualquiera de los dos, lo guardamos, chequeamos los errores y compilamos el proyecto **macrocont** asignándole el dispositivo EPM7032LC44 de la familia MAX7000.

Una vez compilado, en la ventana del Compilador abrimos el archivo de Reporte para revisar la cantidad de recursos utilizados:

Device-Specific Information: c:\max2work\megacont\megacont.rpt
megacont

**** RESOURCE USAGE ****

Logic Array Block	Logic Cells	I/O Pins	Shareable Expanders	External Interconnect
A: LC1 - LC16	0/16(0%)	0/16(0%)	0/16(0%)	0/36(0%)
B: LC17 - LC32	5/16(31%)	5/16(31%)	0/16(0%)	4/36(11%)

Como vemos utiliza la misma cantidad de celdas lógicas que en el caso anterior, una macrocelda por bit del contador.

A continuación creamos el archivo *Simulator Channel File* **megacont.scf** de forma tal que quede como muestra la figura 2.53.

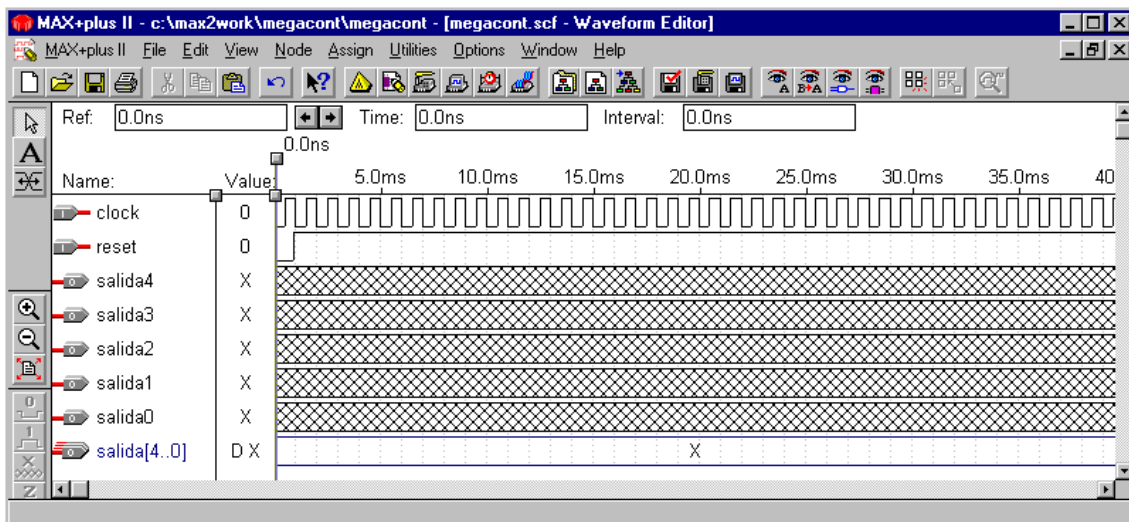


Figura 2.53: Ventana del Editor de Forma de onda

Finalmente simulamos y verificamos en el Editor de Formas de Onda (figura 2.54) que actúa como un contador binario ascendente de 5 bits.

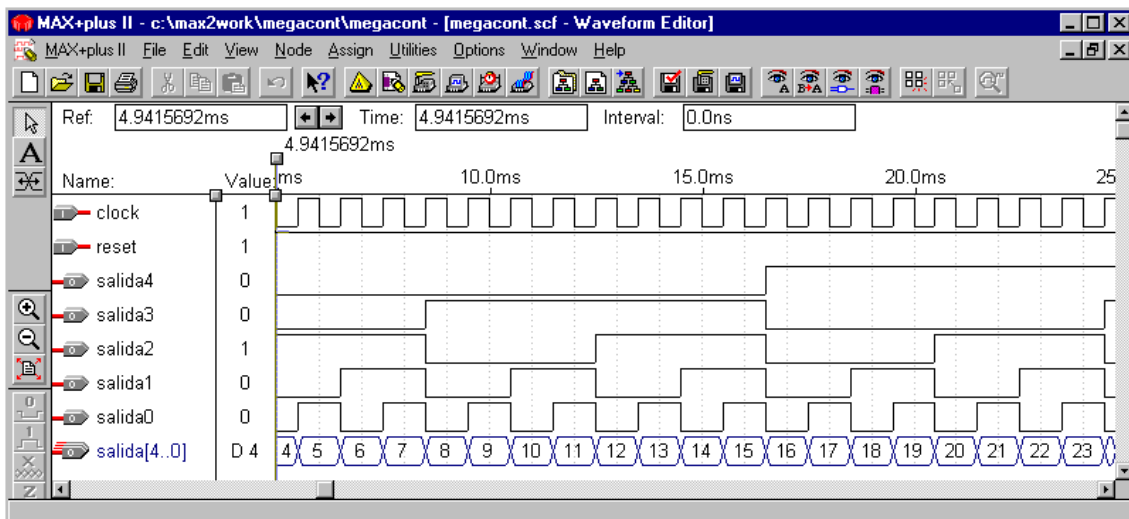


Figura 2.54: Ventana del Editor de Forma de onda

2.2.13.3 Conclusiones

Como vimos en los dos puntos anteriores con ambas funciones (mega y macro) pudimos implementar el contador con la misma cantidad de lógica del dispositivo. Esto se debe a que, como se explicó en el punto 2.1.5, al utilizar mega o macrofunciones, cuando el Compilador analiza la lógica completa del circuito, automáticamente usa cualquier lógica disponible de mega o macrofunción específica de la familia de dispositivos y remueve todas las compuertas y *flip-flops* que no son utilizados, para garantizar una eficiencia óptima del diseño.

La principal diferencia entre las macro y las megafunciones es que en el primer caso tenemos más limitaciones en cuanto a la lógica del bloque símbolo. Por ejemplo si hubiésemos querido que el contador sea descendente no nos habrían alcanzado los 2 contadores 74LS161, sino que deberíamos haber agregado mas lógica o bien utilizar otro tipo de contador (por ej 2 contadores 74LS191); mientras que con la función parametrizada LPM_COUNTER basta con agregar el puerto `updown` y excitarlo con un nivel bajo.

Capítulo 3 Medidor de Frecuencia y Período y Adquisidor autónomo de datos en lenguaje AHDL (*Altera Hardware Description Language*)

3.1 Introducción

En éste capítulo vamos a implementar en lenguaje AHDL, la lógica de los siguientes proyectos:

- *Medidor de frecuencias y períodos*: con las siguientes características
 - Entradas de medición compatibles con lógica TTL.
 - Rango en modo frecuencia: 1Hz a 100MHz.
 - Rango en modo período: 100ns a 10s.
 - Representación mediante 6 dígitos.
 - Comunicación con PC a través de puerto paralelo en modo SPP.
- *Adquisidor autónomo de datos*: con las siguientes características
 - Controlado desde PC a través de puerto paralelo en modo SPP.
 - Control directo del proceso de adquisición de datos del conversor analógico digital ADC0820.
 - Capacidad de almacenamiento de hasta 500 muestras.

En el próximo capítulo, ambos proyectos serán implementados conjuntamente en la placa experimental Upx10K10, la cual contiene la FPGA FLEX10K10 de Altera, por lo cual la lógica secuencial (implementada en LEs) no deberá superar los 576 *flip-flops*, y la memoria del Adquisidor deberá utilizar no más de 2Kbits, que es la memoria interna máxima que provee el dispositivo.

Debido a que el dispositivo FLEX10K (EPF10K10LC84-3) posee un retardo interno de 3ns, usaremos como oscilador externo de 10MHz (100ns).

3.2 Frecuencímetro

3.2.1 Arquitectura básica

Podemos definir a un frecuencímetro como un contador de eventos cíclico, esto es, cuenta una serie de sucesos (los ciclos de la frecuencia que estamos midiendo), los presenta en un display, vuelve a cero y comienza a contar nuevamente.

En la figura 3.1 podemos ver un diagrama en bloques elemental de un frecuencímetro digital como el que implementaremos:

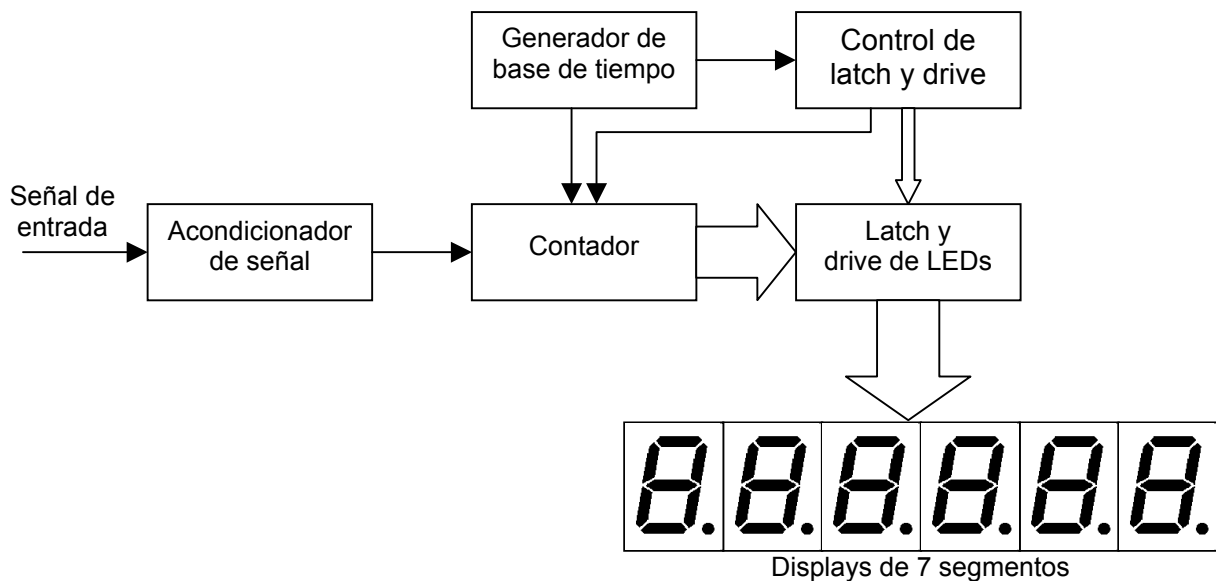


Figura 3.1: Diagrama en bloques del frecuencímetro digital

El primer paso de la señal de entrada es el acondicionador, que es el que adapta la señal analógica a una 'digital', con lo cual obtendremos una forma de onda cuadrada para que cada uno de sus ciclos pueda ser debidamente detectado por el contador.

Luego de tener la señal en condiciones, ésta pasa por el contador digital, el cual es gobernado por una base de tiempos generada por el bloque superior. Ésta base de tiempos permite fijar un tiempo de medición al contador, y luego:

$$f_{señal} = \frac{\text{N}^\circ \text{ de pulsos contados}}{\text{Tiempo de medición}}$$

Es decir que dentro de este tiempo de medición se activará el contador para contar los pulsos de la señal cuadrada de igual frecuencia que la de entrada, tal como se muestra en el diagrama de tiempos en la figura 3.2:

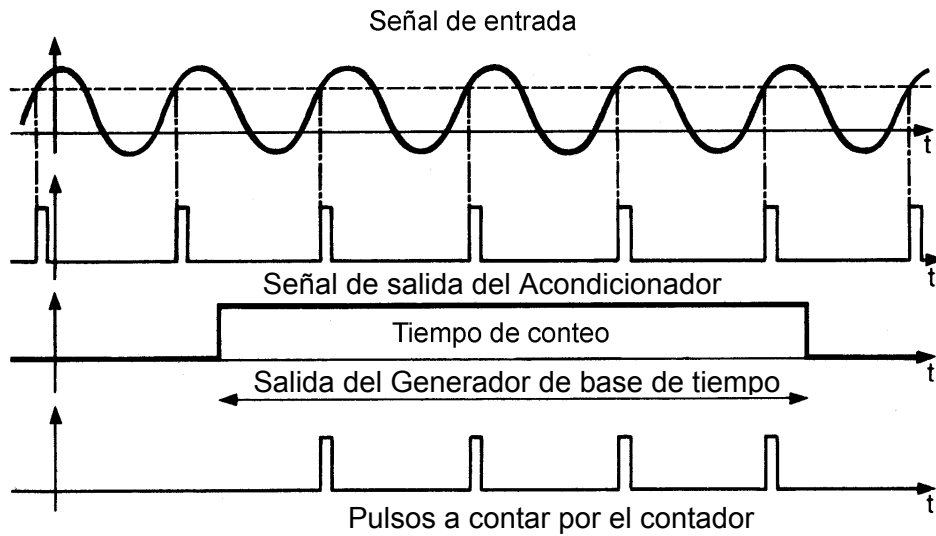


Figura 3.2: Diagrama de tiempos

Una vez que tengamos el valor tomado por el contador al finalizar el tiempo de conteo, deberá ser cargado en el latch o cerrojo y el contador puesto a cero, ya que el sistema no efectúa una única medida, sino que está continuamente midiendo para detectar distintas frecuencias de entrada y por lo tanto la señal de la base de tiempos es una señal periódica. De aquí aparece el bloque de control de latch el cual se encarga secuencialmente de que una vez finalizado el tiempo de medición, cargar el valor del contador en el latch y poner a cero dicho contador de modo que esté listo para el próximo tiempo de medición. El diagrama de tiempos de la figura 3.3 muestra las señales de base de tiempo (BT), carga de latch (Z) y puesta a cero del contador (W).

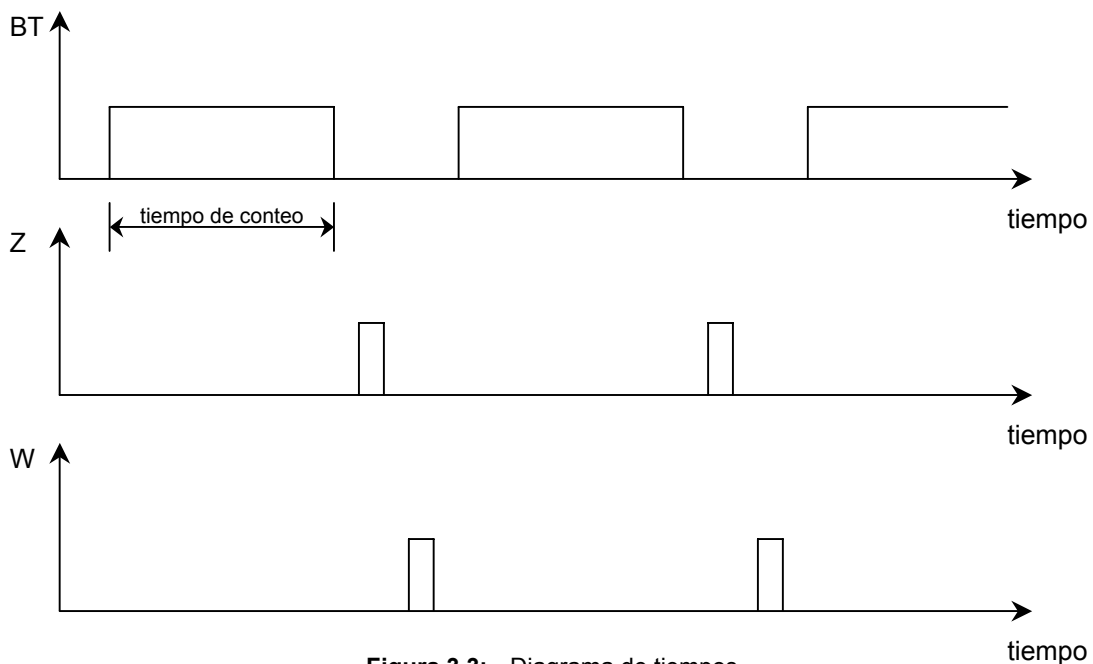


Figura 3.3: Diagrama de tiempos

Supongamos por un momento que no tenemos latch. En el display observaríamos los números ir moviéndose rápidamente aumentando hasta que termina el pulso de tiempo de conteo. Allí se quedarían quietos (y podríamos ver la frecuencia) hasta que llegue el pulso de reset o puesta a cero, con lo que veríamos los números irse a cero para, al habilitarse nuevamente el contador con el nivel alto de la base de tiempo, volver a verlos incrementándose rápidamente hasta la cifra final. Como se puede imaginar, esto es muy cansador para la vista y dependiendo del tiempo de medición fijado a veces imposible de ver. Es así que se intercala entre el contador digital y la presentación (los display de 7 segmentos) dicho dispositivo cerrojo o latch el cual permite mantener el valor del contador en el display durante el próximo ciclo de conteo e ir refrescándose periódicamente al finalizar el tiempo de medición. Si la frecuencia es siempre la misma no veremos entonces cambio alguno en el display.

Como afirmamos anteriormente, el valor del contador luego de ser cargado en el latch es representado para su visualización en el display. Para traducir este lenguaje binario del contador a un lenguaje decimal del display, se utiliza un decodificador BCD a 7 segmentos con lo cual podemos excitar el display de 7 segmentos para la correcta representación de la frecuencia medida.

3.2.2 Implementación en FPGA

Para nuestro caso en particular, el frecuencímetro en principio que implementaremos debe cumplir las siguientes especificaciones:

- Entada de medición compatible con TTL.
- Rango: 1Hz a 100MHz.

Como vemos mediremos frecuencias de señales TTL (0 - 5V) de forma de onda cuadrada con lo cual no necesitaremos el acondicionador de señal que incluimos en la arquitectura básica.

Para la visualización de la medición utilizaremos seis displays de siete segmentos, con lo cual tendremos un total de seis dígitos para representar la frecuencia medida.

El Generador de base de tiempo proveerá de 4 señales de bases de tiempos con distintos tiempos de medición (los cuales estarán dados por el tiempo en alto de la base) las cuales serán seleccionadas por líneas exteriores.

El grupo de contadores estará formado por seis contadores (uno por cada dígito) de cuatro bits cada uno, conectados sincrónicamente de modo que un contador corresponderá a las unidades, otro a las decenas, otro a las centenas y así hasta el contador correspondiente a los millares. Los valores de los contadores serán cargados en seis latches de 4 bits cada uno.

Para la visualización en los displays utilizaremos el método de multiplexación en el tiempo, es decir que las líneas de entrada a, b, c, d, e, f, g, y dp serán

comunes a los 6 displays mientras que no así las de selección de cada dígito. Para esto utilizaremos un cuádruple MUX 6:1 para multiplexar las salidas de los latches, cuyas líneas de selección serán comandadas por un contador ubicado en el bloque Generador de la base de tiempos de modo de obtener una secuencia repetitiva de 1 a 6. A su vez este contador estará conectado a un decodificador para manejar la selección de dígito a encender en ese momento.

Finalmente generaremos un decodificador BCD a siete segmentos para la correcta representación en decimal de la salida de los contadores. La figura 3.4 representa un diagrama en bloques de nuestro sistema:

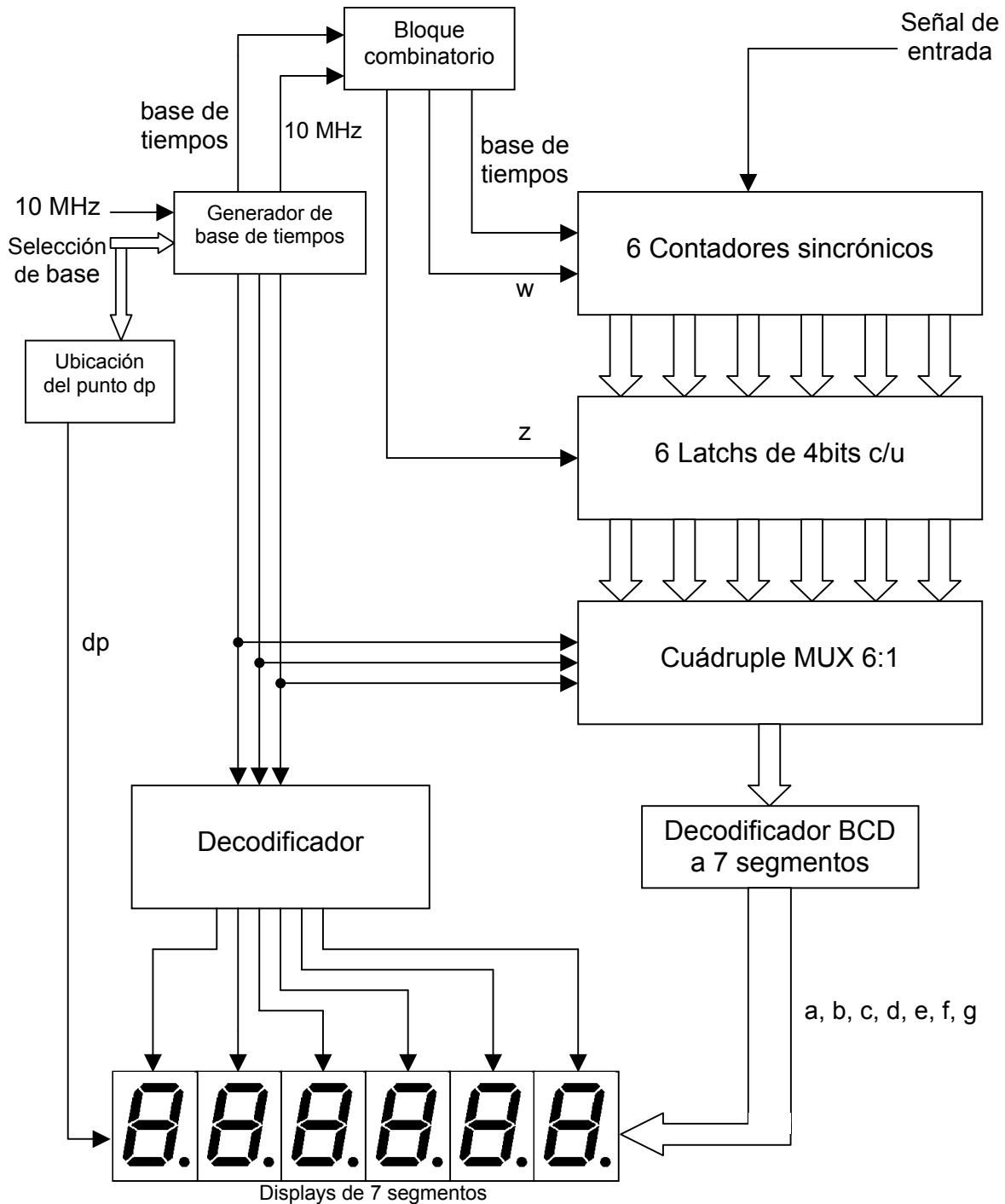


Figura 3.4: Diagrama en bloques del frecuencímetro en FPGA

Como vemos en el diagrama en bloques, debemos utilizar también un oscilador externo (de 10MHz para nuestro caso) para la generación de la base de tiempos.

3.2.2.1 Bloque generador de la base de tiempos.

El bloque generador de la base de tiempos debe generar, como dijimos anteriormente, cuatro bases de tiempos de acuerdo a las entradas externas de selección. Esta selección la podemos realizar mediante el ingreso de dos líneas (selec_base[1..0]) de modo que :

selec_base[1..0]	Duración del nivel alto de la base de tiempos
"00" = 0	10000 ms
"01" = 1	1000 ms
"10" = 2	100 ms
"11" = 3	10 ms

Tabla 3.1: Bases de tiempos

Por su parte el nivel bajo de la base de tiempos será de 1ms para todos los casos, permitiendo en este intervalo la carga de los latches y la puesta a cero de los contadores.

Para conseguir estas bases y partiendo de la frecuencia del oscilador externo (10MHz), diseñaremos un prescaler de cuatro etapas, donde cada una de ellas divide por 10, obteniendo así una división total por 10000 y por lo tanto una frecuencia de 1KHz (1ms de período).

Cada etapa del prescaler será un contador de 4 bits, el cual haremos que cuente hasta 10, y con una conexión sincrónica entre ellos para lograr las 4 etapas y con lo cual contará el total hasta 10000.

Vamos ahora a generar los contadores en lenguaje AHDL mediante la función parametrizada `lpm_counter`:

```
prescaler1: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
prescaler2: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
prescaler3: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
prescaler4: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
```

Como vemos, hemos generado cuatro contadores (`prescaler1`, `prescaler2`, `prescaler3` y `prescaler4`); donde cada uno es de 4 bits (`LPM_WIDTH=4`) y cuenta hasta 10 (`LPM_MODULUS=10`).

Posteriormente vamos a conectarlos en forma sincrónica. Para estos inicialmente conectaremos sus entradas de *clear* asincrónico a un pin *reset*, el cual debe haber sido configurado como pin de entrada (INPUT); luego el reloj de 10MHz (*reloj_xtal*) a las entradas de *clock* de los contadores y finalmente las entradas de habilitación *cnt_en* de los mismos a las salidas *eq[9]* del contador que lo precede. Las salida *eq[9]* se activa cuando el contador llega a 9.

```
prescaler1.aclr = reset;
prescaler2.aclr = reset;
prescaler3.aclr = reset;
prescaler4.aclr = reset;
```

```

prescaler1.clock = reloj_xtal;
prescaler2.clock = reloj_xtal;
prescaler3.clock = reloj_xtal;
prescaler4.clock = reloj_xtal;
prescaler2.cnt_en = prescaler1.eq[9];
prescaler3.cnt_en = prescaler2.eq[9] & prescaler1.eq[9];
prescaler4.cnt_en = prescaler3.eq[9] & prescaler2.eq[9] & prescaler1.eq[9];

```

Para obtener una señal de frecuencia de 1KHz, bastará con tomar la salida *eq[9]* del último contador de la cascada. Esta salida la conectaremos a un nodo que llamaremos *reloj_1ms*:

```
reloj_1ms = prescaler4.eq[9];
```

Para generar las distintas bases, definiremos un nuevo contador que cuente hasta 10000 (al que llamaremos *cuenta*), un flip flop tipo D (al que llamaremos *salida*), un nodo (*borrar*) y sentencias *CASE* e *IF* de modo que:

```

constant MAX_COUNT = 10000;
...
base_tiempos                : OUTPUT;
...
cuenta: lpm_counter WITH (LPM_WIDTH=ceil(log2(MAX_COUNT)));

salida                        : DFF;

borrar                        : NODE;
...
salida.clrn = !reset;
salida.clk = reloj_1ms;
cuenta.clock = reloj_1ms;
cuenta.sclr = borrar;
CASE selec_base[] IS
    WHEN B"00" =>
        IF cuenta.q[] < 10000 THEN
            salida.d = VCC;
            borrar = GND;
        ELSE
            borrar = VCC;
            salida.d = GND;
        END IF;
    WHEN B"01" =>
        IF cuenta.q[] < 1000 THEN
            salida.d = VCC;
            borrar = GND;
        ELSE
            salida.d = GND;
            borrar = VCC;
        END IF;
    WHEN B"10" =>
        IF cuenta.q[] < 100 THEN
            salida.d = VCC;
            borrar = GND;
        ELSE
            borrar = VCC;
            salida.d = GND;
        END IF;
    WHEN B"11" =>
        IF cuenta.q[] < 10 THEN
            salida.d = VCC;
            borrar = GND;
        ELSE
            borrar = VCC;
            salida.d = GND;
        END IF;
END CASE;

```

```
base_tiempos = salida.q;
```

En la primera sentencia declaramos una constante llamada `MAX_COUNT`, la cual adopta el valor 10000. En la segunda generamos un contador llamado `cuenta` el cual tiene una cantidad de bits dada por la sentencia:

```
nº bits = ceil(log2(MAX_COUNT));= ceil(log2 (10000))
```

la cual devuelve un valor entero tal que $2^{nº\text{bits}} \geq 10000$, para este caso `nº bits = 14`.

Luego de acuerdo a los valores de la entrada `selec_base[1..0]`, mediante al contador, el flip flop y las sentencias `CASE` e `IF` se generará una salida de distintos tiempos en alto para lograr las distintas bases mostradas en la tabla 3.1.

Nos queda ahora generar el contador que comanda las líneas de selección del cuádruple MUX 6:1. Para esto creamos un contador de 3 bits que cuente hasta 6, el cual lo llamamos `presc_sel_decodig`:

```
presc_sel_decodig: lpm_counter WITH ( LPM_WIDTH=3, LPM_MODULUS=6);
```

como dijimos anteriormente este contador comandará los tiempos de la multiplexación de los dígitos a mostrar en el display, por lo tanto su frecuencia deberá ser lo suficientemente rápida como para visualmente ver todos los dígitos encendidos, pero no demasiado elevada para permitir que los leds de los dígitos se alcancen a encender. Adoptamos entonces una frecuencia de 1KHz, con lo cual conectamos el contador de la forma:

```
presc_sel_decodig.aclr = reset;
presc_sel_decodig.clock = reloj_xtal;
presc_sel_decodig.cnt_en = prescaler4.eq[9] & prescaler3.eq[9] &
                          prescaler2.eq[9] & prescaler1.eq[9];
```

```
selec_decodig[2..0] = presc_sel_decodig.q[];
```

Luego las salidas del contador (`selec_decodig[2..0]`) serán las que comandan el MUX y el decodificador.

Vamos ahora a generar un programa completo el AHDL (*GBdT.tdf*) que represente el bloque generador de base de tiempos, con todas las sentencias anteriores, cuyas entradas serán `reloj_xtal`, `reset` y `selec_base[1..0]`; y sus salidas `selec_decodig[2..0]`, `clock5M` y `base_tiempos`.

```
constant MAX_COUNT = 10000;

INCLUDE "lpm_counter.inc";

SUBDESIGN GBdT
(
    reloj_xtal, reset, selec_base[1..0]           : INPUT;
    selec_decodig[2..0], clock10M, base_tiempos  : OUTPUT;
)
)
```



```

VARIABLE
  cuenta: lpm_counter WITH (LPM_WIDTH=ceil(log2(MAX_COUNT)));
  presc_sel_decodig: lpm_counter WITH (LPM_WIDTH=3, LPM_MODULUS=6);
  prescaler1: lpm_counter WITH (LPM_WIDTH=4, LPM_MODULUS=10);
  prescaler2: lpm_counter WITH (LPM_WIDTH=4, LPM_MODULUS=10);
  prescaler3: lpm_counter WITH (LPM_WIDTH=4, LPM_MODULUS=10);
  prescaler4: lpm_counter WITH (LPM_WIDTH=4, LPM_MODULUS=10);

  salida                                     : DFF;

  borrar, reloj_lms                          : NODE;

BEGIN
  presc_sel_decodig.aclr = reset;
  prescaler1.aclr = reset;
  prescaler2.aclr = reset;
  prescaler3.aclr = reset;
  prescaler4.aclr = reset;
  presc_sel_decodig.clock = reloj_xtal;
  prescaler1.clock = reloj_xtal;
  prescaler2.clock = reloj_xtal;
  prescaler3.clock = reloj_xtal;
  prescaler4.clock = reloj_xtal;
  prescaler2.cnt_en = prescaler1.eq[9];
  prescaler3.cnt_en = prescaler2.eq[9] & prescaler1.eq[9];
  prescaler4.cnt_en = prescaler3.eq[9] & prescaler2.eq[9] &
    prescaler1.eq[9];
  presc_sel_decodig.cnt_en = prescaler4.eq[9] & prescaler3.eq[9] &
    prescaler2.eq[9] & prescaler1.eq[9];
  selec_decodig[2..0] = presc_sel_decodig.q[];
  reloj_lms = prescaler4.eq[9];
  clock10M = reloj_xtal;

  salida.clrn = !reset;
  salida.clk = reloj_lms;
  cuenta.clock = reloj_lms;
  cuenta.sclr = borrar;
  CASE selec_base[] IS
    WHEN B"00" =>
      IF cuenta.q[] < 10000 THEN
        salida.d = VCC;
        borrar = GND;
      ELSE
        borrar = VCC;
        salida.d = GND;
      END IF;
    WHEN B"01" =>
      IF cuenta.q[] < 1000 THEN
        salida.d = VCC;
        borrar = GND;
      ELSE
        salida.d = GND;
        borrar = VCC;
      END IF;
    WHEN B"10" =>
      IF cuenta.q[] < 100 THEN
        salida.d = VCC;
        borrar = GND;
      ELSE
        borrar = VCC;
        salida.d = GND;
      END IF;
    WHEN B"11" =>
      IF cuenta.q[] < 10 THEN
        salida.d = VCC;
        borrar = GND;
      ELSE

```

```

        borrar = VCC;
        salida.d = GND;
    END IF;
END CASE;

base_tiempos = salida.q;

END;
```

Realizamos ahora una simulación funcional del programa para verificar las salidas del mismo para los distintos valores de entrada. Tomaremos solo la generación de las bases de 100 y 10ms en estado alto debido al elevado tiempo de simulación que se necesitaría para simular las bases de 1000 y 10000ms en alto teniendo en cuenta que la señal de reloj externo es de 10MHz.

El resultado para la salida `base_tiempos`:

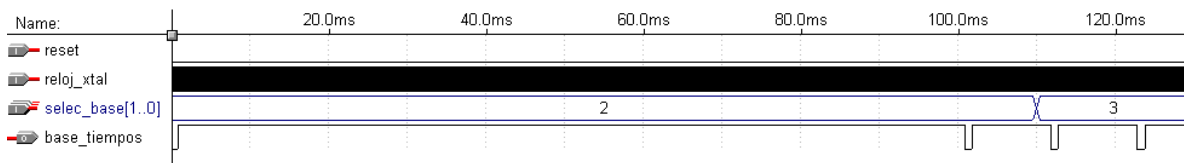


Figura 3.5: Simulación en el Editor de Formas de onda.

Como vemos para los valores 2 y 3 de la entrada `selec_base[1..0]`, la salida `base_tiempos` tiene un tiempo en alto de 100 y 10ms respectivamente tal cual lo requerido.

Vamos ahora a ver la salida `selec_decodig[2..0]` y el nodo `reloj_1ms`:

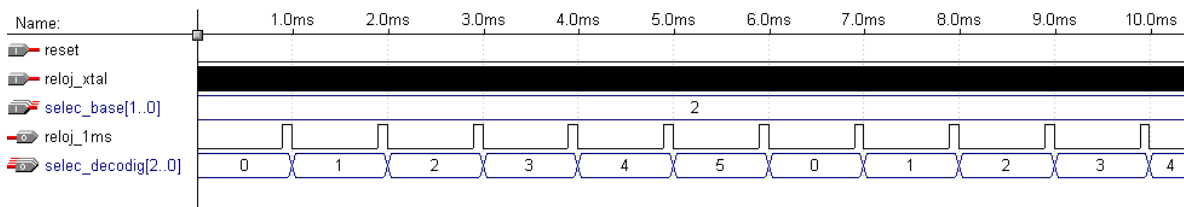


Figura 3.6: Simulación en el Editor de Formas de onda.

Podemos verificar el período de `reloj_1ms` así como también la secuencia de la salida del contador `presc_sel_decodig` (`selec_decodig[2..0]`) y su tiempo de duración en cada nivel.

Por último veremos la salida `clock10M` (cuya frecuencia es 10MHz) que nos servirá como entrada al bloque combinatorio que generará las señales de carga de latches y reseteo de contadores

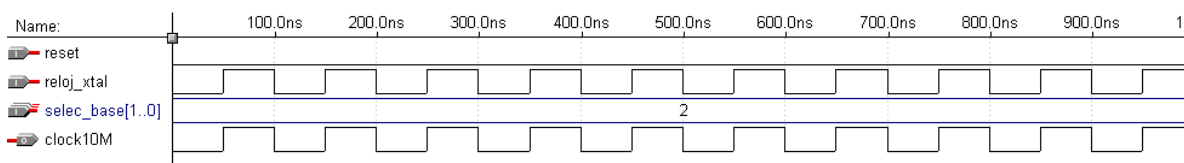


Figura 3.7: Simulación en el Editor de Formas de onda.

3.2.2.2 Bloque combinatorio.

El bloque combinatorio es el encargado de generar la señal de carga de latch (Z) y la señal de puesta a cero de los contadores (W).

Tal como vimos en la figura 3.3 la señal Z deberá ser un pulso luego de que la base de tiempos caiga a cero. Para construir esta señal utilizaremos un monoestable que construiremos a partir de una maquina de estados de Moore, cuya entrada será `base_tiempos`, su salida la señal `z` y su reloj `clock10M`; con el siguiente diagrama de estados:

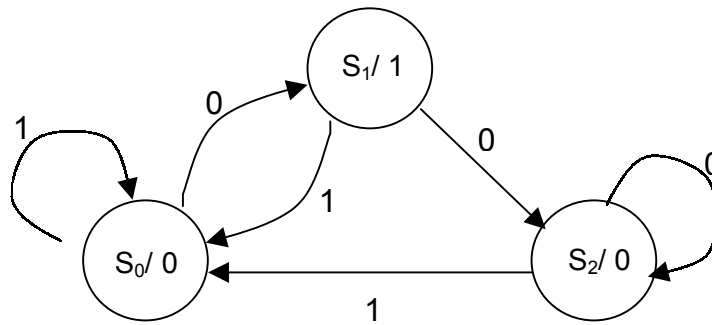


Figura 3.8: Diagrama de estados.

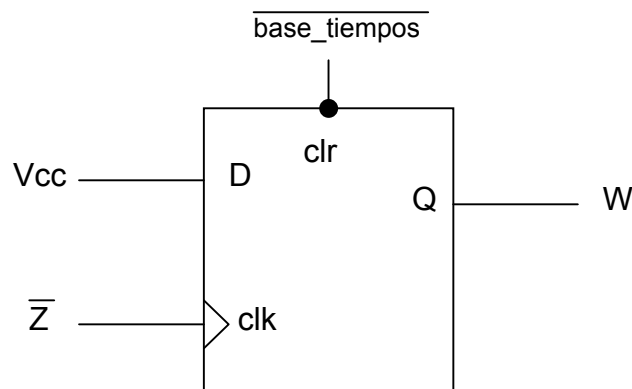
Ahora implementándolo en AHDL:

```

ss: MACHINE OF BITS (z) WITH STATES      (s0 = 0, s1 = 1, s2 = 0);
...
ss.clk = clock10M;
ss.reset = reset;
TABLE
%   estado   entrada   próximo   %
%   actual   actual   estado   %
ss,  base_tiempos =>  ss;
s0,  1        =>  s0;
s0,  0        =>  s1;
s1,  0        =>  s2;
s1,  1        =>  s0;
s2,  0        =>  s2;
s2,  1        =>  s0;
END TABLE;

```

Por su parte la señal W que se encarga de la puesta a cero de los contadores, deberá ser un pulso que se genere luego de la señal Z, y debe durar en alto, como máximo, hasta el flanco ascendente de la base de tiempos. Para conseguir esto implementaremos el siguiente circuito lógico:



donde el flip flop tipo D es disparado por flanco ascendente de reloj, con lo cual cuando la base esta en alto, el ff se resetea continuamente, cuando ésta baja para un flanco descendente de Z, copia la entrada en la salida (o sea Vcc) hasta resetearse (volver a cero) cuando la base vuelva al nivel alto.

El circuito implementado en lenguaje AHDL, llamando al flip flop `resetcont`, queda de la forma:

```
resetcont                                : DFF;
...
resetcont.clrn = !base_tiempos;
resetcont.d = VCC;
resetcont.clk = !z;
w = resetcont.q;
```

3.2.2.3 Contadores sincrónicos

Este bloque cuenta con seis contadores en conexión sincrónica los cuales se encargan de contar los pulsos de la señal a medir que entran en un tiempo de conteo (tiempo en el que la base de tiempos esta en nivel alto). Vamos a ver la representación de los mismos en lenguaje de descripción de hardware de Altera:

```
contador1: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
contador2: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
contador3: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
contador4: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
contador5: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
contador6: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);

sal_cont1[3..0], sal_cont2[3..0], sal_cont3[3..0]           : NODE;
sal_cont4[3..0], sal_cont5[3..0], sal_cont6[3..0]         : NODE;
...
contador1.clock = signal_in;
contador2.clock = signal_in;
contador3.clock = signal_in;
contador4.clock = signal_in;
contador5.clock = signal_in;
contador6.clock = signal_in;

contador1.cnt_en = base_tiempos;
contador2.cnt_en = base_tiempos & contador1.eq[9];
contador3.cnt_en = base_tiempos & contador1.eq[9] & contador2.eq[9];
contador4.cnt_en = base_tiempos & contador1.eq[9] & contador2.eq[9] &
  contador3.eq[9];
contador5.cnt_en = base_tiempos & contador1.eq[9] & contador2.eq[9] &
  contador3.eq[9] & contador4.eq[9];
contador6.cnt_en = base_tiempos & contador1.eq[9] & contador2.eq[9] &
  contador3.eq[9] & contador4.eq[9] & contador5.eq[9];

contador1.aclr = w;
contador2.aclr = w;
contador3.aclr = w;
contador4.aclr = w;
contador5.aclr = w;
contador6.aclr = w;

sal_cont1[3..0] = contador1.q[];
sal_cont2[3..0] = contador2.q[];
sal_cont3[3..0] = contador3.q[];
sal_cont4[3..0] = contador4.q[];
```

```
sal_cont5[3..0] = contador5.q[];
sal_cont6[3..0] = contador6.q[];
```

donde `signal_in` es la señal de entrada cuya frecuencia queremos medir.

Podemos diseñar también un caso general para una cantidad de dígitos dada, la cual en nuestro caso es de 6. Esto lo logramos mediante la sentencia

FOR...GENERATE:

```
constant N_DIGITOS = 6;                                % De modo que %
constant POT_K = ceil(log2(N_DIGITOS));                % 2^POT_K >= N_DIGITOS %
...
contador[N_DIGITOS..1]: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);

sal_cont[N_DIGITOS..1][3..0], nodo[N_DIGITOS..1]      : NODE;
...
FOR i IN 1 TO N_DIGITOS GENERATE
    contador[i].clock = signal_in;
    contador[i].aclr = w;
END GENERATE;

sal_cont[N_DIGITOS..1][3..0] = contador[N_DIGITOS..1].q[];

contador[1].cnt_en = base_tiempos;
nodo[1] = base_tiempos;

FOR n IN 2 TO N_DIGITOS GENERATE
    nodo[n] = nodo[n-1] & contador[n-1].eq[9];
    contador[n].cnt_en = nodo[n];
END GENERATE;
```

Si queremos representar la frecuencia en otra cantidad de dígitos distinta de 6, bastará con cambiar el valor de la constante `N_DIGITOS`.

3.2.2.4 Bloque de latches

El bloque de latches cuenta con seis latches de 4 bits cada uno, los cuales se encargan de mantener los valores de las salidas de los contadores, cuando finaliza un ciclo de conteo y hasta que finalice el próximo. Una forma de implementarlos en AHDL sería de la forma:

```
latch1: lpm_latch WITH (LPM_WIDTH=4);
latch2: lpm_latch WITH (LPM_WIDTH=4);
latch3: lpm_latch WITH (LPM_WIDTH=4);
latch4: lpm_latch WITH (LPM_WIDTH=4);
latch5: lpm_latch WITH (LPM_WIDTH=4);
latch6: lpm_latch WITH (LPM_WIDTH=4);

qlatch1[3..0], qlatch2[3..0], qlatch3[3..0]          : NODE;
qlatch4[3..0], qlatch5[3..0], qlatch6[3..0]          : NODE;
...
latch1.gate = z;
latch2.gate = z;
latch3.gate = z;
latch4.gate = z;
latch5.gate = z;
latch6.gate = z;

latch1.data[] = sal_cont1[];
latch2.data[] = sal_cont2[];
```

```

latch3.data[] = sal_cont3[];
latch4.data[] = sal_cont4[];
latch5.data[] = sal_cont5[];
latch6.data[] = sal_cont6[];

```

```

qlatch1[] = latch1.q[];
qlatch2[] = latch2.q[];
qlatch3[] = latch3.q[];
qlatch4[] = latch4.q[];
qlatch5[] = latch5.q[];
qlatch6[] = latch6.q[];

```

y para el caso general de una cantidad de dígitos dada:

```

constant N_DIGITOS = 6;                                % De modo que %
constant POT_K = ceil(log2(N_DIGITOS));                % 2^POT_K >= N_DIGITOS %
...
latches[N_DIGITOS..1]: lpm_latch WITH (LPM_WIDTH=4);

qlatch[N_DIGITOS..1][3..0]                            : NODE;
...
FOR i IN 1 TO N_DIGITOS GENERATE
    latches[i].gate = z;
END GENERATE;

latches[N_DIGITOS..1].data[] = sal_cont[N_DIGITOS..1][];
qlatch[N_DIGITOS..1][] = latches[N_DIGITOS..1].q[];

```

3.2.2.5 Bloque multiplexor

El bloque multiplexor esta constituido por un cuádruple multiplexor 6:1, o sea un multiplexor 8:1, con solo habilitación de 6 entradas y cuya salida y dichas entradas son de 4 bits cada una.

```

multiplexor: lpm_mux WITH (LPM_WIDTH=4, LPM_SIZE=6, LPM_WIDTHS=3);

sal_mux[3..0]                                          : NODE;
...
multiplexor.data[0] [] = qlatch1[];
multiplexor.data[1] [] = qlatch2[];
multiplexor.data[2] [] = qlatch3[];
multiplexor.data[3] [] = qlatch4[];
multiplexor.data[4] [] = qlatch5[];
multiplexor.data[5] [] = qlatch6[];
multiplexor.sel[] = selec_decodig[];
sal_mux[] = multiplexor.result[];

```

para el caso general de una cantidad de dígitos dada:

```

constant N_DIGITOS = 6;                                % De modo que %
constant POT_K = ceil(log2(N_DIGITOS));                % 2^POT_K >= N_DIGITOS %
...
multiplexor: lpm_mux WITH (LPM_WIDTH=4, LPM_SIZE=N_DIGITOS, LPM_WIDTHS=POT_K);

sal_mux[3..0]                                          : NODE;
...
multiplexor.data[N_DIGITOS-1..0] [] = qlatch[N_DIGITOS..1][];
multiplexor.sel[] = selec_decodig[];
sal_mux[] = multiplexor.result[];

```

3.2.2.6 Bloque decodificador BCD 7 segmentos

En éste bloque debemos implementar un “traductor” del lenguaje binario de los contadores a un lenguaje “decimal” de los displays de 7 segmentos. Este lenguaje en los displays está representado mediante los leds a, b, c, d, e, f, y g; los cuales están distribuidos de la forma:

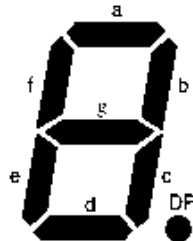


Figura 3.9: Distribución de los leds en un display de 7 segmentos.

Cabe aclarar que el led dp que corresponde al punto no influirá en la representación del dígito, y lo abordaremos luego para la ubicación del punto de acuerdo a la escala y base de tiempos utilizada.

Vamos ahora a construir una tabla en AHDL mediante la sentencia `TABLE` la cual represente al decodificador del lenguaje binario (el cual lo tomaremos en hexadecimal) al lenguaje del display (tomamos como nivel lógico 1 para encender led):

```

a, b, c, d, e, f, g                                     : OUTPUT;
...
bcd[3..0]                                              : NODE;
...
TABLE
    bcd[3..0] => a, b, c, d, e, f, g;

    H"0"  =>    1, 1, 1, 1, 1, 1, 0;
    H"1"  =>    0, 1, 1, 0, 0, 0, 0;
    H"2"  =>    1, 1, 0, 1, 1, 0, 1;
    H"3"  =>    1, 1, 1, 1, 0, 0, 1;
    H"4"  =>    0, 1, 1, 0, 0, 1, 1;
    H"5"  =>    1, 0, 1, 1, 0, 1, 1;
    H"6"  =>    1, 0, 1, 1, 1, 1, 1;
    H"7"  =>    1, 1, 1, 0, 0, 0, 0;
    H"8"  =>    1, 1, 1, 1, 1, 1, 1;
    H"9"  =>    1, 1, 1, 1, 0, 1, 1;
END TABLE;

bcd[3..0] = sal_mux[ ];

```

3.2.2.7 Bloque decodificador

El bloque decodificador es el que se encarga de la decodificación binaria de la secuencia repetitiva de 0 a 5 de la salida del contador `presc_sel_decodig` (`selec_decodig[2..0]`) a una secuencia repetitiva que permita seleccionar un dígito a la vez de la forma que muestra la tabla 3.2:

selec_decodig[2..0]	selec_digito[6..1]
"000"	"000001"
"001"	"000010"
"010"	"000100"
"011"	"001000"
"100"	"010000"
"101"	"100000"

Tabla 3.2: Decodificador

Para su implementación en lenguaje AHDL, nos valemos de la función parametrizada `lpm_decode`:

```

selec_digito5, selec_digito4, selec_digito3           : OUTPUT;
selec_digito2, selec_digito1, selec_digito0         : OUTPUT;
...
decodificador: lpm_decode WITH (LPM_WIDTH=3, LPM_DECODES=6);
...
decodificador.data[] = selec_decodig[];
selec_digito1 = decodificador.eq[0];
selec_digito2 = decodificador.eq[1];
selec_digito3 = decodificador.eq[2];
selec_digito4 = decodificador.eq[3];
selec_digito5 = decodificador.eq[4];
selec_digito6 = decodificador.eq[5];

```

o bien para el caso general de una cantidad de dígitos dada:

```

constant N_DIGITOS = 6;                               % De modo que %
constant POT_K = ceil(log2(N_DIGITOS));               % 2^POT_K >= N_DIGITOS %
...
selec_digito[N_DIGITOS..1]                           : OUTPUT;
...
decodificador: lpm_decode WITH (LPM_WIDTH=POT_K, LPM_DECODES=N_DIGITOS);
...
decodificador.data[] = selec_decodig[];
selec_digito[N_DIGITOS..1] = decodificador.eq[N_DIGITOS-1..0];

```

3.2.2.8 Bloque de ubicación del punto dp

Este bloque se encarga de la ubicación del punto dp en uno de los seis dígitos de acuerdo a la base de tiempos utilizada. Para cumplir con el rango de frecuencias a medir, tomaremos la visualización en los displays en la escala de KHz, ésto nos permitirá una correcta visualización del rango de frecuencias a medir, eligiendo la correcta base de tiempos.

Tomemos el caso de la base de tiempos con 1000ms = 1seg. en nivel alto (`selec_base[1..0] = "01" = 1`). Si tuviéramos que medir una frecuencia de 1KHz, su período sería de 1ms, con lo cual el grupo contadores llegaría a un valor de 1000 y la representación en los displays sería:

00 1000

con lo cual deberíamos ubicar el punto en el dígito 4, de forma que leamos 1KHz, y los display: quedarían:

00 1.000

Supongamos ahora el caso de la base de tiempos con 10000ms = 10seg. en nivel alto (`selec_base[1..0] = "00" = 0`), y la misma frecuencia de 1KHz en la señal de entrada. El grupo contadores llegaría a un valor de 10000 y la representación en los displays sería:

0 10000

con lo cual deberíamos ubicar el punto en el dígito 5, de forma que leamos 1KHz, y los display quedarían:

0 10000

Ahora para la misma frecuencia de la señal de entrada, tomamos el caso de la base de tiempos con 100ms = 0.1seg. en nivel alto (`selec_base[1..0] = "10" = 2`). El grupo contadores llegaría a un valor de 100 y la representación en los displays sería:

000 100

con lo cual en este caso para que leamos 1KHz deberíamos ubicar el punto en el dígito 3, de forma que los display se muestren:

000 100

Y nos queda el último caso, para la base de tiempos con 10ms = 0.01seg. en nivel alto (`selec_base[1..0] = "11" = 3`), el grupo de contadores llegaría a 10 (siempre con la misma frecuencia de 1KHz de la señal de entrada) con la siguiente visualización de los displays:

0000 10

y por lo tanto deberíamos ubicar el punto en el dígito 2, de forma que los display se muestren:

0000 1.0

Por lo tanto podemos confeccionar la tabla 3.3, que nos muestra el dígito que contiene el punto de acuerdo a la base seleccionada:

selec_base[1..0]	duración del nivel alto de la base de tiempos	dígito que contiene el punto
"00" = 0	10000 ms	5
"01" = 1	1000 ms	4
"10" = 2	100 ms	3
"11" = 3	10 ms	2

Tabla 3.3: Ubicación del punto dp

Volcando ésto al lenguaje AHDL, mediante la sentencia `CASE`, queda de la forma:

```

dp                                     : OUTPUT;
...
CASE selec_base[] IS
  WHEN B"00" =>
    dp = selec_digito[5];
  WHEN B"01" =>
    dp = selec_digito[4];
  WHEN B"10" =>
    dp = selec_digito[3];
  WHEN B"11" =>
    dp = selec_digito[2];
END CASE;
```

3.2.2.9 Bloque combinatorio + contadores + latches + multiplexor + decodificador BCD 7 segmentos + decodificador + ubicación del punto dp

Vamos ahora a implementar los bloques combinatorio, contadores, latches, decodificador BCD 7 segmentos, decodificador y ubicación del punto dp en un único bloque.

Para éste único bloque tendremos los siguientes pines:

entrada: clock10M, reset, base_tiempos, signal_in, selec_base[1..0] y selec_decodig[2..0]

salida: a, b, c, d, e, f, g, dp, selec_digito6, selec_digito5, selec_digito4, selec_digito3, selec_digito2 y selec_digito1.

Además agregaremos algunas salidas para verificar por simulación como:

salidas para simulación: dig6[3..0], dig5[3..0], dig4[3..0], dig3[3..0],
dig2[3..0], dig1[3..0], zout y wout.

El programa en lenguaje AHDL al que llamaremos *bloquef.tdf*:

```

INCLUDE "lpm_counter.inc";
INCLUDE "lpm_latch.inc";
INCLUDE "lpm_mux.inc";
INCLUDE "lpm_decode.inc";

SUBDESIGN bloquef
(
    clock10M, reset, base_tiempos, signal_in           : INPUT;
    selec_base[1..0], selec_decodig[2..0]             : INPUT;
    a, b, c, d, e, f, g, dp, zout, wout               : OUTPUT;
    selec_digito6, selec_digito5, selec_digito4       : OUTPUT;
    selec_digito3, selec_digito2, selec_digito1       : OUTPUT;
    dig1[3..0], dig2[3..0], dig3[3..0]                : OUTPUT;
    dig4[3..0], dig5[3..0], dig6[3..0]                : OUTPUT;
)

VARIABLE
    ss: MACHINE OF BITS (z) WITH STATES (s0 = 0, s1 = 1, s2 = 0);
    contador1: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
    contador2: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
    contador3: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
    contador4: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
    contador5: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
    contador6: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
    latch1: lpm_latch WITH (LPM_WIDTH=4);
    latch2: lpm_latch WITH (LPM_WIDTH=4);
    latch3: lpm_latch WITH (LPM_WIDTH=4);
    latch4: lpm_latch WITH (LPM_WIDTH=4);
    latch5: lpm_latch WITH (LPM_WIDTH=4);
    latch6: lpm_latch WITH (LPM_WIDTH=4);

    multiplexor: lpm_mux WITH (LPM_WIDTH=4, LPM_SIZE=6, LPM_WIDTHS=3);

    decodificador: lpm_decode WITH (LPM_WIDTH=3, LPM_DECODES=6);

    resetcont                                     : DFF;

    sal_cont1[3..0], sal_cont2[3..0], sal_cont3[3..0] : NODE;
    sal_cont4[3..0], sal_cont5[3..0], sal_cont6[3..0] : NODE;
    qlatch1[3..0], qlatch2[3..0], qlatch3[3..0]      : NODE;
    qlatch4[3..0], qlatch5[3..0], qlatch6[3..0]      : NODE;
    sal_mux[3..0], bcd[3..0], w                      : NODE;

BEGIN

% Comienzo del diseño del monoestable con maquina de estados %
ss.clk = clock10M;
ss.reset = reset;
TABLE
% estado entrada próximo %
% actual actual estado %
    ss, base_tiempos => ss;

    s0, 1 => s0;
    s0, 0 => s1;
    s1, 0 => s2;
    s1, 1 => s0;
    s2, 0 => s2;
    s2, 1 => s0;
END TABLE;
% Fin diseño del monoestable %

```

```

% Comienzo del diseño del decodificador BCD a 7 segmentos %
TABLE
    bcd[3..0] => a, b, c, d, e, f, g;

    H"0" => 1, 1, 1, 1, 1, 1, 0;
    H"1" => 0, 1, 1, 0, 0, 0, 0;
    H"2" => 1, 1, 0, 1, 1, 0, 1;
    H"3" => 1, 1, 1, 1, 0, 0, 1;
    H"4" => 0, 1, 1, 0, 0, 1, 1;
    H"5" => 1, 0, 1, 1, 0, 1, 1;
    H"6" => 1, 0, 1, 1, 1, 1, 1;
    H"7" => 1, 1, 1, 0, 0, 0, 0;
    H"8" => 1, 1, 1, 1, 1, 1, 1;
    H"9" => 1, 1, 1, 1, 0, 1, 1;
END TABLE;
% Fin diseño decodificador BCD a 7 segmentos %

% Generación de la señal w para resetear el contador %
resetcont.clrn = !base_tiempos;
resetcont.d = VCC;
resetcont.clk = !z;
w = resetcont.q;

% Contadores %
contador1.clock = signal_in;
contador2.clock = signal_in;
contador3.clock = signal_in;
contador4.clock = signal_in;
contador5.clock = signal_in;
contador6.clock = signal_in;

contador1.cnt_en = base_tiempos;
contador2.cnt_en = base_tiempos & contador1.eq[9];
contador3.cnt_en = base_tiempos & contador1.eq[9] & contador2.eq[9];
contador4.cnt_en = base_tiempos & contador1.eq[9] & contador2.eq[9] &
    contador3.eq[9];
contador5.cnt_en = base_tiempos & contador1.eq[9] & contador2.eq[9] &
    contador3.eq[9] & contador4.eq[9];
contador6.cnt_en = base_tiempos & contador1.eq[9] & contador2.eq[9] &
    contador3.eq[9] & contador4.eq[9] & contador5.eq[9];

contador1.aclr = w;
contador2.aclr = w;
contador3.aclr = w;
contador4.aclr = w;
contador5.aclr = w;
contador6.aclr = w;

sal_cont1[3..0] = contador1.q[];
sal_cont2[3..0] = contador2.q[];
sal_cont3[3..0] = contador3.q[];
sal_cont4[3..0] = contador4.q[];
sal_cont5[3..0] = contador5.q[];
sal_cont6[3..0] = contador6.q[];

% Latches %
latch1.gate = z;
latch2.gate = z;
latch3.gate = z;
latch4.gate = z;
latch5.gate = z;
latch6.gate = z;

latch1.data[] = sal_cont1[];
latch2.data[] = sal_cont2[];
latch3.data[] = sal_cont3[];
latch4.data[] = sal_cont4[];
latch5.data[] = sal_cont5[];

```

```

    latch6.data[] = sal_cont6[];

    qlatch1[] = latch1.q[];
    qlatch2[] = latch2.q[];
    qlatch3[] = latch3.q[];
    qlatch4[] = latch4.q[];
    qlatch5[] = latch5.q[];
    qlatch6[] = latch6.q[];

%   Multiplexor   %
multiplexor.data[0] [] = qlatch1[];
multiplexor.data[1] [] = qlatch2[];
multiplexor.data[2] [] = qlatch3[];
multiplexor.data[3] [] = qlatch4[];
multiplexor.data[4] [] = qlatch5[];
multiplexor.data[5] [] = qlatch6[];
multiplexor.sel[] = selec_decodig[];
sal_mux[] = multiplexor.result[];

bcd[3..0] = sal_mux[];

%   Decodificador   %
decodificador.data[] = selec_decodig[];
selec_digito1 = decodificador.eq[0];
selec_digito2 = decodificador.eq[1];
selec_digito3 = decodificador.eq[2];
selec_digito4 = decodificador.eq[3];
selec_digito5 = decodificador.eq[4];
selec_digito6 = decodificador.eq[5];

%   Salidas para simulación   %
dig1[]=qlatch1[];
dig2[]=qlatch2[];
dig3[]=qlatch3[];
dig4[]=qlatch4[];
dig5[]=qlatch5[];
dig6[]=qlatch6[];
zout = z;
wout = w;

%   Ubicación del punto (dp) de acuerdo a la base utilizada   %
CASE selec_base[] IS
    WHEN B"00" =>
        dp = selec_digito5;
    WHEN B"01" =>
        dp = selec_digito4;
    WHEN B"10" =>
        dp = selec_digito3;
    WHEN B"11" =>
        dp = selec_digito2;
END CASE;

END;
```

Y el programa para el caso general de una cantidad de dígitos dada, al que llamaremos *bloquef_digitos.tdf*:

```

constant N_DIGITOS = 6; %   De modo que   %
constant POT_K = ceil(log2(N_DIGITOS)); % 2^POT_K >= N_DIGITOS %

INCLUDE "lpm_counter.inc";
INCLUDE "lpm_latch.inc";
INCLUDE "lpm_mux.inc";
INCLUDE "lpm_decode.inc";
```

```

SUBDESIGN bloquef_digitos
(
    clock10M, reset, base_tiempos, signal_in           : INPUT;
    selec_base[1..0], selec_decodig[POT_K-1..0]       : INPUT;
    a, b, c, d, e, f, g, dp, zout, wout              : OUTPUT;
    selec_digito[N_DIGITOS..1]                       : OUTPUT;
    dig[N_DIGITOS..1][3..0]                          : OUTPUT;
)
VARIABLE
    ss: MACHINE OF BITS (z) WITH STATES (s0 = 0, s1 = 1, s2 = 0);
    contador[N_DIGITOS..1]: lpm_counter WITH (LPM_WIDTH=4, LPM_MODULUS=10);
    latches[N_DIGITOS..1]: lpm_latch WITH (LPM_WIDTH=4);
    multiplexor: lpm_mux WITH (LPM_WIDTH=4, LPM_SIZE=N_DIGITOS,
                              LPM_WIDTHS=POT_K);
    decodificador: lpm_decode WITH (LPM_WIDTH=POT_K, LPM_DECODES=N_DIGITOS);

    resetcont                                           : DFF;

    sal_cont[N_DIGITOS..1][3..0], qlatch[N_DIGITOS..1][3..0] : NODE;
    sal_mux[3..0], bcd[3..0], nodo[N_DIGITOS..1], w      : NODE;

BEGIN
% Comienzo del diseño del monoestable con maquina de estados %
ss.clk = clock10M;
ss.reset = reset;
TABLE
% estado entrada próximo %
% actual actual estado %
    ss, base_tiempos => ss;

    s0, 1 => s0;
    s0, 0 => s1;
    s1, 0 => s2;
    s1, 1 => s0;
    s2, 0 => s2;
    s2, 1 => s0;
END TABLE;
% Fin diseño del monoestable %

% Comienzo del diseño del decodificador BCD a 7 segmentos %
TABLE
    bcd[3..0] => a, b, c, d, e, f, g;

    H"0" => 1, 1, 1, 1, 1, 1, 0;
    H"1" => 0, 1, 1, 0, 0, 0, 0;
    H"2" => 1, 1, 0, 1, 1, 0, 1;
    H"3" => 1, 1, 1, 1, 0, 0, 1;
    H"4" => 0, 1, 1, 0, 0, 1, 1;
    H"5" => 1, 0, 1, 1, 0, 1, 1;
    H"6" => 1, 0, 1, 1, 1, 1, 1;
    H"7" => 1, 1, 1, 0, 0, 0, 0;
    H"8" => 1, 1, 1, 1, 1, 1, 1;
    H"9" => 1, 1, 1, 1, 0, 1, 1;
END TABLE;
% Fin diseño decodificador BCD a 7 segmentos %

% Generación de la señal w para resetear el contador %
resetcont.clrn = !base_tiempos;
resetcont.d = VCC;
resetcont.clk = !z;
w = resetcont.q;

% Contadores, latches y multiplexor %
FOR i IN 1 TO N_DIGITOS GENERATE
    contador[i].clock = signal_in;
    contador[i].aclr = w;
    latches[i].gate = z;
END GENERATE;

```

```

sal_cont[N_DIGITOS..1][3..0] = contador[N_DIGITOS..1].q[];
latches[N_DIGITOS..1].data[] = sal_cont[N_DIGITOS..1][];
qlatch[N_DIGITOS..1][] = latches[N_DIGITOS..1].q[];
multiplexor.data[N_DIGITOS-1..0][] = qlatch[N_DIGITOS..1][];

contador[1].cnt_en = base_tiempos;
nodo[1] = base_tiempos;
FOR n IN 2 TO N_DIGITOS GENERATE
    nodo[n] = nodo[n-1] & contador[n-1].eq[9];
    contador[n].cnt_en = nodo[n];
END GENERATE;

multiplexor.sel[] = selec_decodig[];
sal_mux[] = multiplexor.result[];

bcd[3..0] = sal_mux[];

%   Decodificador   %
decodificador.data[] = selec_decodig[];
selec_digito[N_DIGITOS..1] = decodificador.eq[N_DIGITOS-1..0];

%   Salidas para simulación   %
dig[N_DIGITOS..1][] = qlatch[N_DIGITOS..1][];
zout = z;
wout = w;

%   Ubicación del punto (dp) de acuerdo a la base utilizada   %
%   solo para el caso que N_DIGITOS >= 5   %
CASE selec_base[] IS
    WHEN B"00" =>
        dp = selec_digito[5];
    WHEN B"01" =>
        dp = selec_digito[4];
    WHEN B"10" =>
        dp = selec_digito[3];
    WHEN B"11" =>
        dp = selec_digito[2];
END CASE;

END;
```

Como podemos ver, para la ubicación del punto dp solo es válido para el caso de número de dígitos mayor o igual a 5, si el nº de dígitos es menor deberemos reformar la sentencia CASE.

3.2.3 Frecuencímetro en AHDL

Vamos ahora a implementar el frecuencímetro completo en AHDL. Para esto generaremos un símbolo con cada archivo de texto en AHDL (*GBdT.tdf* y *bloquef_digitos.tdf*), los conectaremos a través del editor gráfico y realizaremos la posterior simulación.

Como mencionamos anteriormente el primer paso es generar los archivos de símbolo (*GBdT.sym* y *bloquef_digitos.sym*), luego los cargaremos y conectaremos en el editor gráfico, de forma que quede como muestra la figura 3.10 y a éste archivo lo llamaremos *frecuencímetro.gdf*.

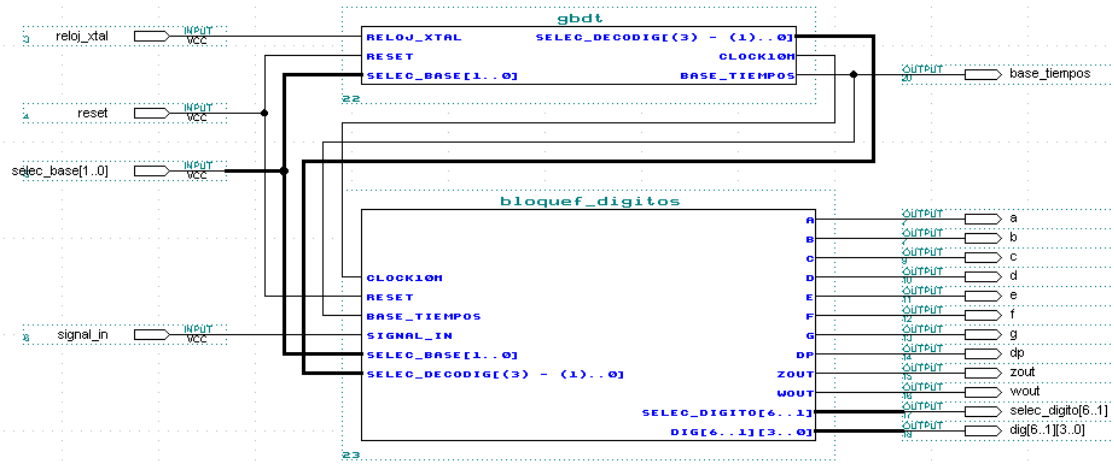


Figura 3.10: Editor gráfico

Ahora realizaremos una simulación funcional, para una frecuencia de entrada de 51.2KHz ($T = 19.53125\mu s$), con una base de tiempos con 10ms = 0.01seg. en nivel alto (selec_base[1..0] = "11" = 3), y verificaremos las salidas.

Inicialmente vamos a verificar la base de tiempos (10ms en alto), la señal z (zout), la señal w (wout) y las señales de selección de dígitos (selec_digito[6..1]):

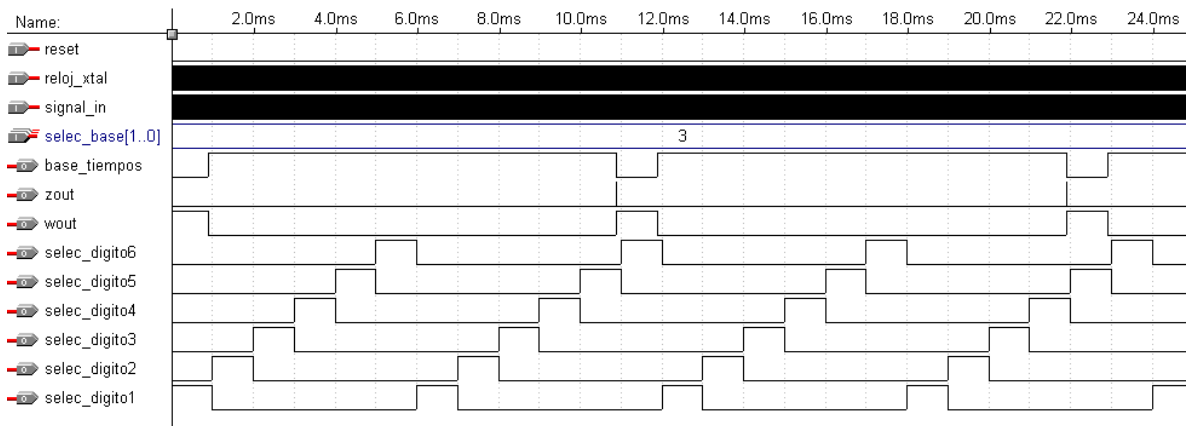


Figura 3.11: Simulación en el Editor de Formas de onda

Como vemos es la figura 3.11, se cumplen las señales pretendidas al crear los archivos.

Hacemos un zoom para verificar la correcta generación de la señal z :

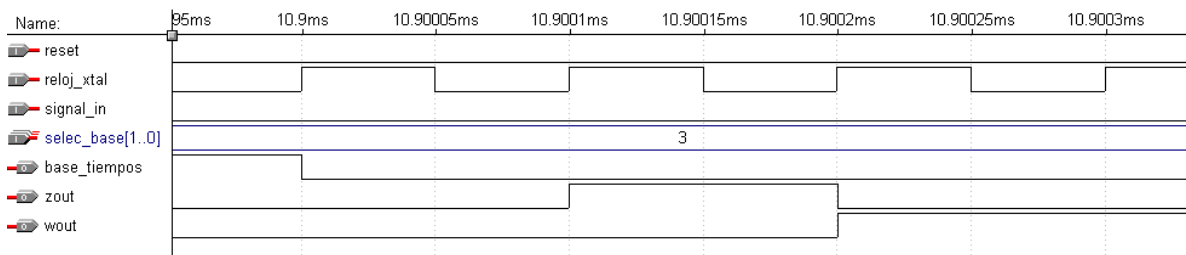


Figura 3.12: Simulación en el Editor de Formas de onda

De acuerdo a la base de tiempo utilizada deberíamos tener la siguiente visualización de los displays:

0005 1.2

Por lo tanto las salidas de los dígitos, luego de la primera carga de latches, debería ser:

dígito	valor decimal
dig[6][3..0]	0
dig[5][3..0]	0
dig[4][3..0]	0
dig[3][3..0]	5
dig[2][3..0]	1
dig[1][3..0]	2

Tabla 3.4: valor de los dígitos

lo verificamos, tal como muestra la figura 3.13:

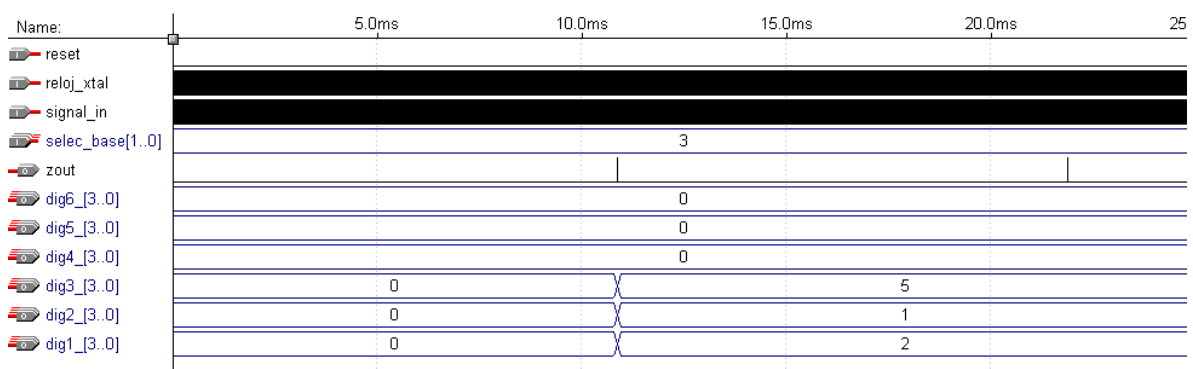


Figura 3.13: Simulación en el Editor de Formas de onda

Ahora debemos corroborar que en el momento de selección de cada dígito coincidan las salidas a, b, c, d, e, f, g y dp.

Para los dígitos 6, 5 y 4 deberíamos tener:

valor del dígito	a	b	c	d	e	f	g	dp
0	1	1	1	1	1	1	0	0

Tabla 3.5: valor de las salidas a-dp para dígito con valor 0

verificamos en la simulación (figura 3.14):

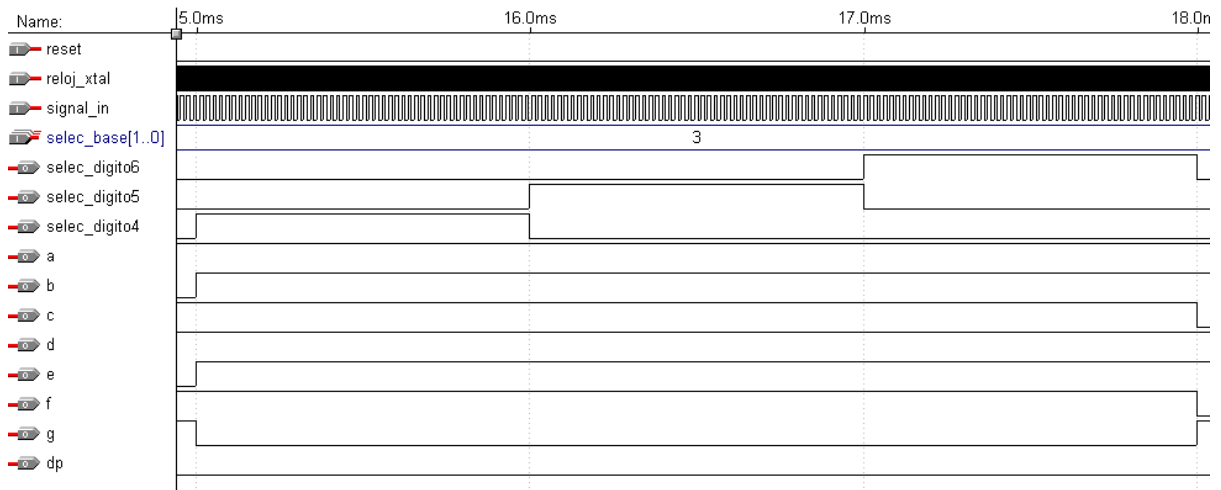


Figura 3.14: Simulación en el Editor de Formas de onda

para el dígito 3:

valor del dígito	a	b	c	d	e	f	g	dp
5	1	0	1	1	0	1	1	0

Tabla 3.6: valor de las salidas a-dp para dígito con valor 5

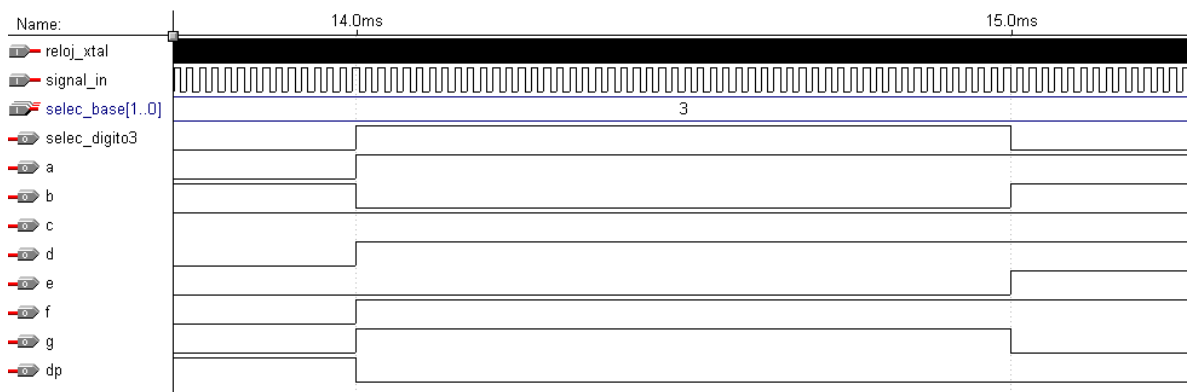


Figura 3.15: Simulación en el Editor de Formas de onda

para el dígito 2 (el que tiene el punto dp):

valor del dígito	a	b	c	d	e	f	g	dp
1.	0	1	1	0	0	0	0	1

Tabla 3.7: valor de las salidas a-dp para dígito con valor 1.

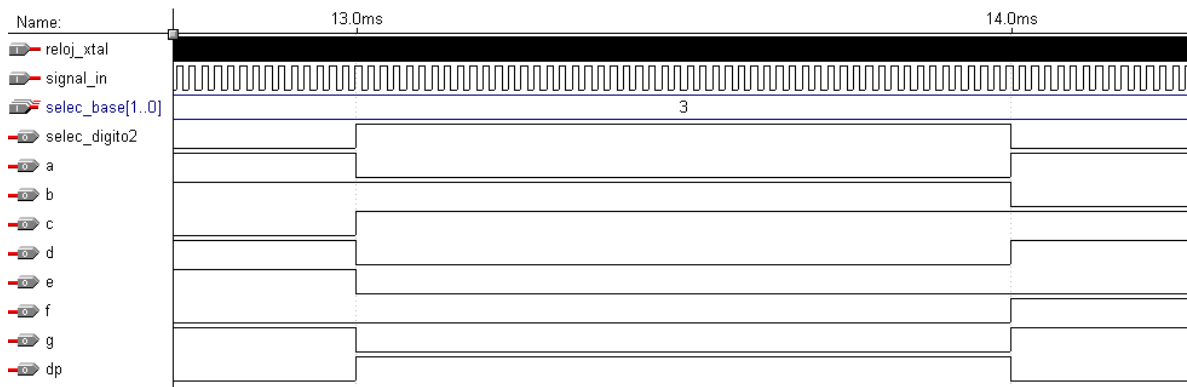


Figura 3.16: Simulación en el Editor de Formas de onda

y por último para el dígito 1:

valor del dígito	a	b	c	d	e	f	g	dp
2	1	1	0	1	1	0	1	0

Tabla 3.8: valor de las salidas a-dp para dígito con valor 2

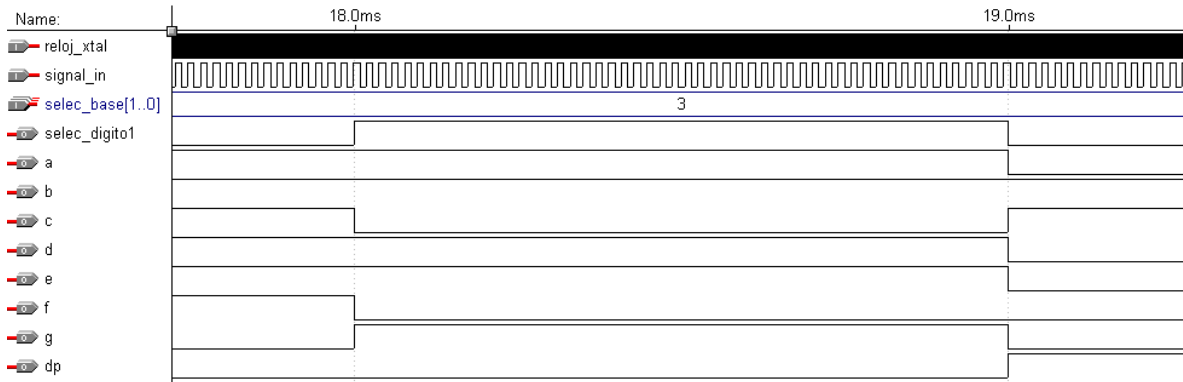


Figura 3.17: Simulación en el Editor de Formas de onda

3.3 Medidor de períodos

3.3.1 Arquitectura básica

Podemos definir a un medidor de períodos de la misma forma que lo hicimos con el frecuencímetro, como un contador de eventos cíclico (pero en éste caso, cuenta una serie de sucesos, que son pulsos de períodos definidos, dentro del período de la señal a medir, que aquí actúa como la base de tiempos), los presenta en un display, vuelve a cero y comienza a contar nuevamente.

En la figura 3.18 podemos ver un diagrama en bloques elemental de medidor de períodos digital, en forma análoga al frecuencímetro, como el que implementaremos:

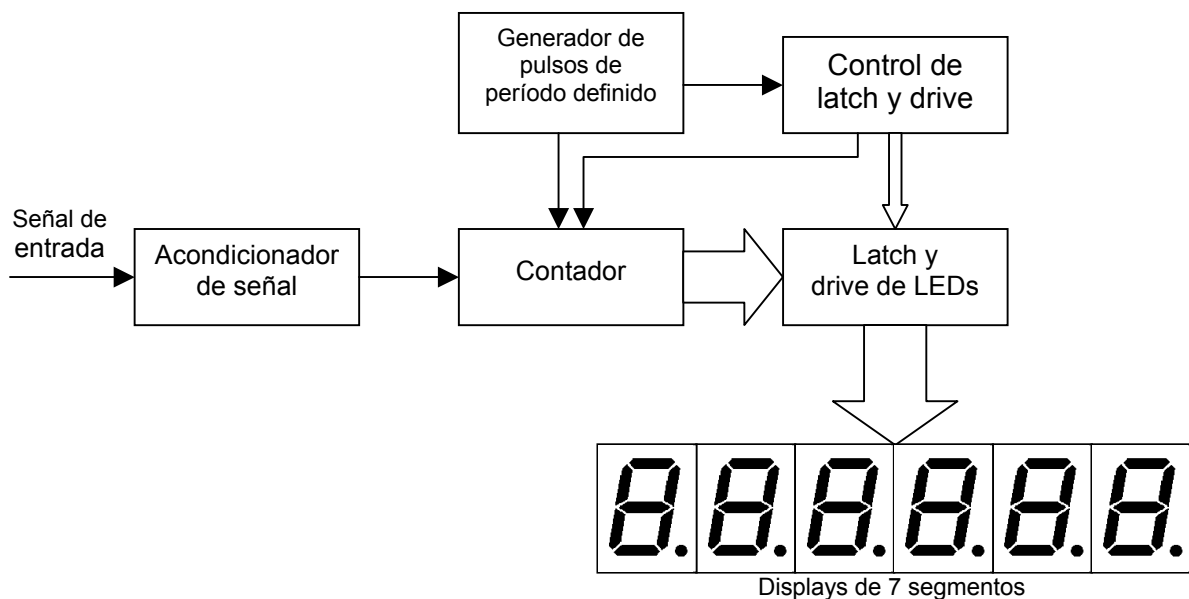


Figura 3.18: Diagrama en bloques del medidor de períodos digital

Como vemos el diagrama en bloques es similar al de medición de frecuencias, solo que aquí, como mencionamos anteriormente, el período de la señal a medir actúa como tiempo de medición (nivel en alto de la base de tiempos del frecuencímetro); y los pulsos a contar por el contador son de período preciso, luego:

$$T_{señal} = N^{\circ} \text{ de pulsos contados} * T_{pulsos}$$

O sea que dentro de este tiempo de medición (período de la señal) se activará el contador para contar los pulsos de período puntual, tal como se muestra en el diagrama de tiempos en la figura 3.19:

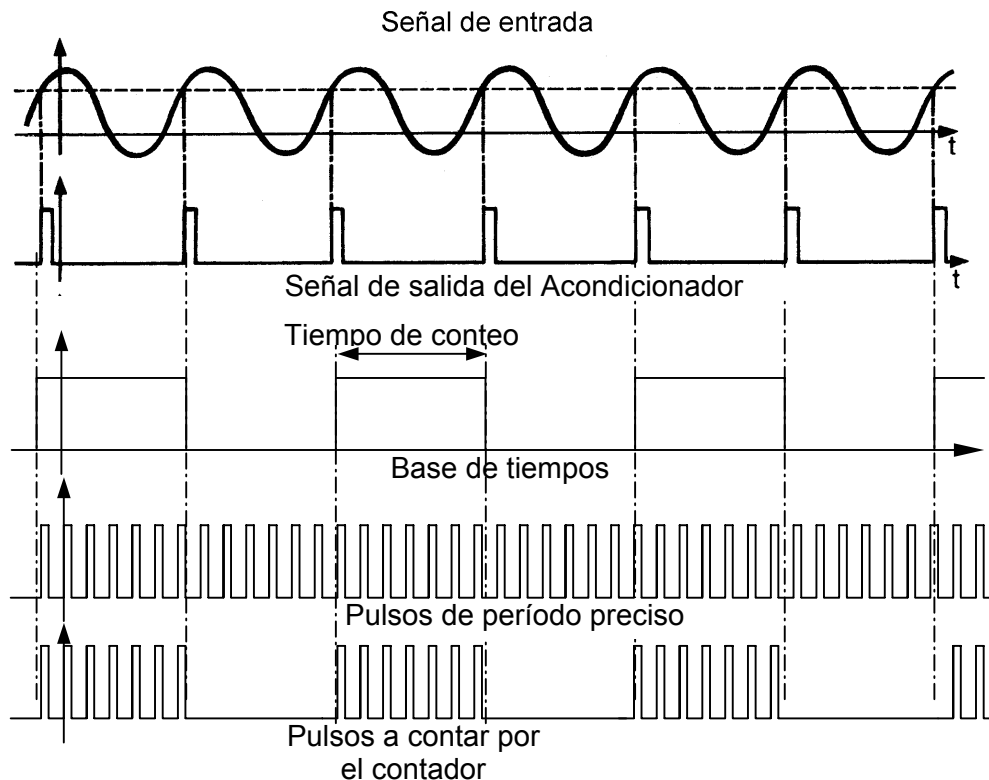


Figura 3.19: Diagrama de tiempos

Luego actúa exactamente igual al frecuencímetro, el valor tomado por el contador al finalizar el tiempo de conteo, es cargado en el latch (señal Z) y el contador puesto a cero (señal W). El valor cargado en el latch es representado para su visualización en el display a través de un decodificador BCD a 7 segmentos con el cual podemos excitar el display de 7 segmentos para la correcta representación del período medido.

3.3.2 Implementación en FPGA

Para nuestro caso, el medidor de períodos que implementaremos, en principio debe cumplir las siguientes especificaciones:

- Entrada de medición compatible con TTL.
- Rango: 100ns a 10seg.

Al igual que para el frecuencímetro, mediremos señales TTL (0 - 5V) de forma de onda cuadrada con lo cual no necesitaremos el acondicionador de señal que incluimos en la arquitectura básica.

La arquitectura utilizada es la misma que para el medidor de frecuencias, solo que reemplazaremos el generador de base de tiempos, por un generador de señales de pulsos con períodos definidos. Este generador, al que llamaremos Generador de pulsos, proveerá de 4 señales con distintos períodos, las cuales serán seleccionadas por líneas exteriores.

La figura 3.20 representa un diagrama en bloques de nuestro sistema:

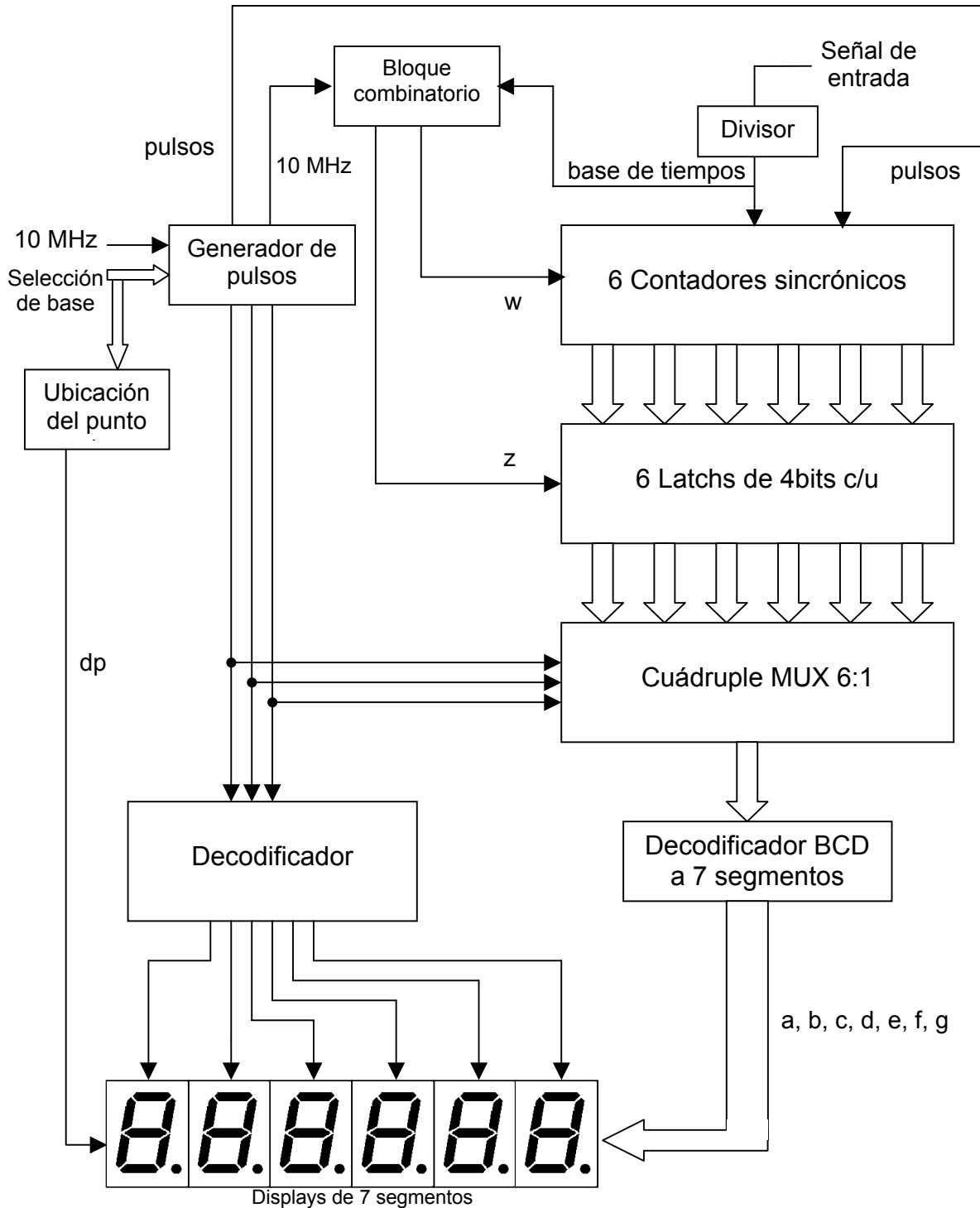


Figura 3.20: Diagrama en bloques del medidor de períodos en FPGA

Al igual que para el frecuencímetro, utilizamos un oscilador externo (de 10MHz), pero en éste caso es para la generación de los pulsos de período preciso. Cabe mencionar nuevamente que aquí la base de tiempos es generada a partir de la señal a medir, tal como se ve en la figura 3.20.

3.3.2.1 Bloque generador de pulsos.

El bloque generador de pulsos debe generar cuatro señales de distintos períodos precisos de acuerdo a las entradas externas de selección. Esta selección la realizaremos, al igual que para el medidor de frecuencias, mediante las líneas `selec_base[1..0]` de modo que :

<code>selec_base[1..0]</code>	Período
"00" = 0	0.1 us
"01" = 1	1us
"10" = 2	10 us
"11" = 3	100 us

Tabla 3.9: Pulsos de períodos definidos

Para conseguir estas señales y partiendo de la frecuencia del oscilador externo (10MHz), diseñaremos nuevamente un prescaler de cuatro etapas, donde cada una de ellas divide por 10. Cada etapa del prescaler será un contador de 4 bits, el cual haremos que cuente hasta 10, y con una conexión sincrónica entre ellos para lograr las 4 etapas y con lo cual contará el total hasta 10000.

Para generar las distintas señales nos valemos de la conexión sincrónica de los contadores y de la sentencia `CASE` de modo que:

```

selec_base[1..0], reloj_xtal, reset                : INPUT;
...
pulsos                                           : OUTPUT;
...
prescaler1: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
prescaler2: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
prescaler3: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
prescaler4: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
...
prescaler1.aclr = reset;
prescaler2.aclr = reset;
prescaler3.aclr = reset;
prescaler4.aclr = reset;
prescaler1.clock = reloj_xtal;
prescaler2.clock = reloj_xtal;
prescaler3.clock = reloj_xtal;
prescaler4.clock = reloj_xtal;
prescaler2.cnt_en = prescaler1.eq[9];
prescaler3.cnt_en = prescaler2.eq[9] & prescaler1.eq[9];
prescaler4.cnt_en = prescaler3.eq[9] & prescaler2.eq[9] & prescaler1.eq[9];

CASE selec_base[] IS
  WHEN B"00" =>
    pulsos = reloj_xtal;
  WHEN B"01" =>
    pulsos = prescaler1.eq[9];
  WHEN B"10" =>
    pulsos = prescaler2.eq[9];
  WHEN B"11" =>
    pulsos = prescaler3.eq[9];
END CASE;

```

Como vemos de acuerdo a la entrada seleccionada, tomaremos la salida de una de la etapa de los prescalers (que como sabemos cada una de ellas divide por 10) y por lo tanto obtendremos el período propuesto por la tabla 3.9. Finalmente y al igual que para medición de frecuencia, debemos generar el contador cuyas salidas `selec_decodig[2..0]` comandan los tiempos de la multiplexación de los dígitos a mostrar en el display y las líneas de selección del cuádruple MUX 6:1. Vamos a tomar el caso general para una cantidad de bits dada:

```
constant N_DIGITOS = 6;                % De modo que %
constant POT_K = ceil(log2(N_DIGITOS)); % 2^POT_K >= N_DIGITOS %
...
presc_sel_decodig: lpm_counter WITH ( LPM_WIDTH=POT_K, LPM_MODULUS=N_DIGITOS);
...
presc_sel_decodig.aclr = reset;
presc_sel_decodig.clock = reloj_xtal;
presc_sel_decodig.cnt_en = prescaler4.eq[9] & prescaler3.eq[9] &
                          prescaler2.eq[9] & prescaler1.eq[9];

selec_decodig[POT_K-1..0] = presc_sel_decodig.q[];
```

A continuación generaremos el programa completo el AHDL (*Gp.tdf*) que represente el bloque generador de base de pulsos, con todas las sentencias anteriores, cuyas entradas serán `reloj_xtal`, `reset` y `selec_base[1..0]`; y sus salidas `selec_decodig[2..0]`, `clock10M` y pulsos.

```
constant N_DIGITOS = 6;                % De modo que %
constant POT_K = ceil(log2(N_DIGITOS)); % 2^POT_K >= N_DIGITOS %

INCLUDE "lpm_counter.inc";

SUBDESIGN Gp
(
    reloj_xtal, reset, selec_base[1..0]           : INPUT;
    selec_decodig[POT_K-1..0], clock10M, pulsos   : OUTPUT;
)

VARIABLE
prescaler1: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
prescaler2: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
prescaler3: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
prescaler4: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
presc_sel_decodig: lpm_counter WITH ( LPM_WIDTH=POT_K,
                                     LPM_MODULUS=N_DIGITOS);

BEGIN
prescaler1.aclr = reset;
prescaler2.aclr = reset;
prescaler3.aclr = reset;
prescaler4.aclr = reset;
presc_sel_decodig.aclr = reset;
prescaler1.clock = reloj_xtal;
prescaler2.clock = reloj_xtal;
prescaler3.clock = reloj_xtal;
prescaler4.clock = reloj_xtal;
presc_sel_decodig.clock = reloj_xtal;
prescaler2.cnt_en = prescaler1.eq[9];
prescaler3.cnt_en = prescaler2.eq[9] & prescaler1.eq[9];
prescaler4.cnt_en = prescaler3.eq[9] & prescaler2.eq[9] &
                    prescaler1.eq[9];
presc_sel_decodig.cnt_en = prescaler4.eq[9] & prescaler3.eq[9] &
                           prescaler2.eq[9] & prescaler1.eq[9];
```

```

selec_decodig[POT_K-1..0] = presc_sel_decodig.q[];
clock5M = reloj_xtal;

CASE selec_base[] IS
    WHEN B"00" =>
        pulsos = reloj_xtal;
    WHEN B"01" =>
        pulsos = prescaler1.eq[9];
    WHEN B"10" =>
        pulsos = prescaler2.eq[9];
    WHEN B"11" =>
        pulsos = prescaler3.eq[9];
END CASE;
END;

```

Por último vamos a realizar una simulación funcional del programa a modo de verificación de las salidas del mismo para los distintos valores de entrada. Inicialmente vamos a verificar la correcta generación de la señal `pulsos` para distintos valores de las entradas `selec_base[1..0]`:

- para `selec_base[1..0] = "00" = 0` => $T_{\text{pulsos}} = 0.1\mu\text{s} = 100\text{ns}$

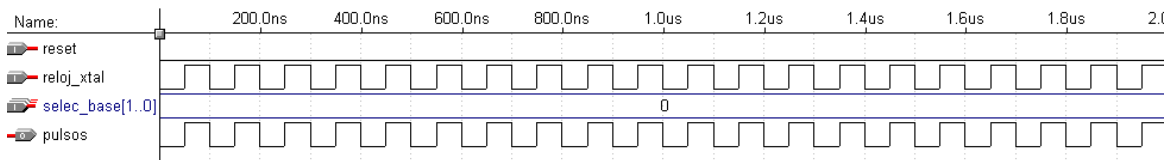


Figura 3.21: Simulación en el Editor de Formas de onda.

- para `selec_base[1..0] = "01" = 1` => $T_{\text{pulsos}} = 1\mu\text{s}$

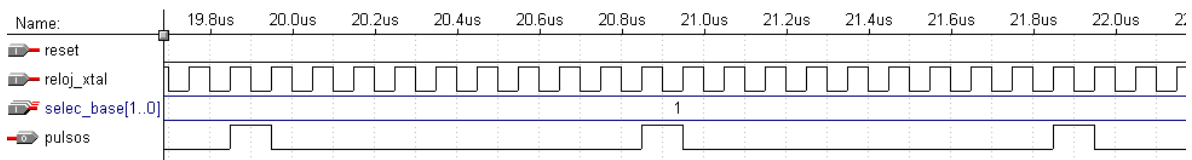


Figura 3.22: Simulación en el Editor de Formas de onda.

- para `selec_base[1..0] = "10" = 2` => $T_{\text{pulsos}} = 10\mu\text{s}$

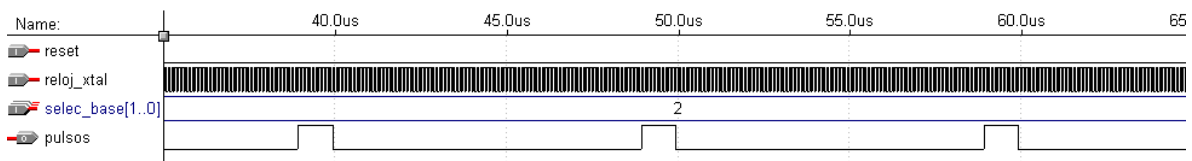


Figura 3.23: Simulación en el Editor de Formas de onda.

- para `selec_base[1..0] = "11" = 3` => $T_{\text{pulsos}} = 100\mu\text{s}$

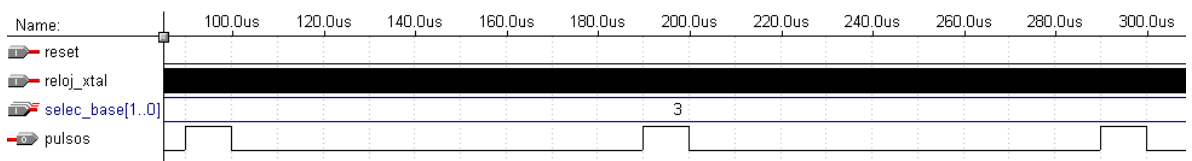


Figura 3.24: Simulación en el Editor de Formas de onda.

Vamos ahora a verificar la salida `selec_decodig[2..0]`:

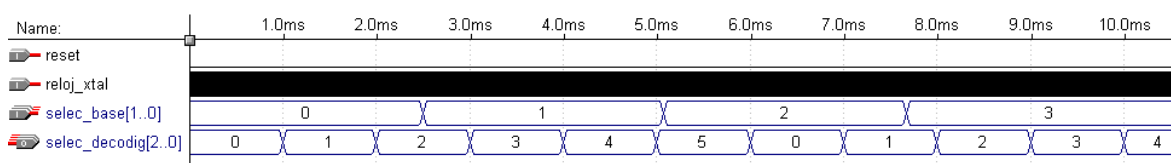


Figura 3.25: Simulación en el Editor de Formas de onda.

Podemos verificar la secuencia de la salida del contador `presc_sel_decodig` (`selec_decodig[2..0]`) y cuyo tiempo de duración en cada nivel es de 1ms.

Por último veremos la salida `clock10M` (cuya frecuencia es 10MHz) que nos servirá como entrada al bloque combinatorio que generará las señales de carga de latches y reseteo de contadores

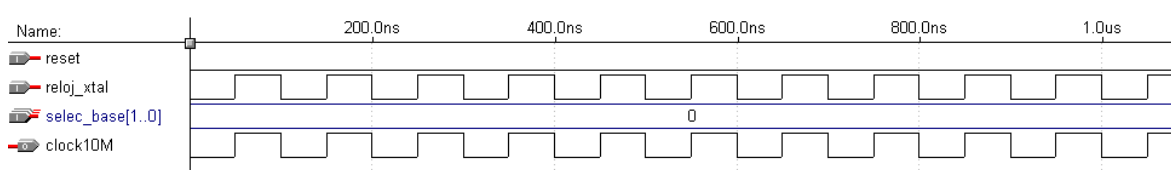


Figura 3.26: Simulación en el Editor de Formas de onda.

3.3.2.2 Bloque divisor.

El bloque divisor es el encargado de generar la base de tiempos a partir de la señal a medir. Simplemente consta de un divisor de frecuencias por dos el cual implementaremos por medio de un contador binario de un bit al cual llamaremos `divisor`. Al dividir la señal de entrada por dos obtendremos una señal cuyo tiempo de duración en alto es igual al período de la señal a medir, y por lo tanto nuestra base de tiempos, tal cual lo vimos en el diagrama tiempos de la figura 3.19.

```
divisor: lpm_counter WITH (LPM_WIDTH=1);

base_tiempos                                     : NODE;
...
divisor.clock = signal_in;
base_tiempos = divisor.q[];
```

3.3.2.3 Bloque combinatorio + contadores + latches + multiplexor + decodificador BCD 7 segmentos + decodificador

Debido a que el medidor de períodos sigue los mismos principios del medidor de frecuencias; o sea los contadores cuentan una dada cantidad de pulsos en un tiempo de conteo y luego mediante los latches, el decodificador BCD 7 segmentos y el decodificador, se representan sus valores en los displays; estos bloques son iguales a los de frecuencímetro, salvo que la entrada de reloj de los contadores ahora no es mas la señal a medir, sino los pulsos generados por el Bloque generador de pulsos.

3.3.2.4 Bloque de ubicación del punto dp

De igual forma que en el caso del frecuencímetro, éste bloque se encarga de la ubicación del punto dp en uno de los seis dígitos de acuerdo a la señal de pulsos utilizada. Para cumplir con el rango de períodos a medir, tomaremos la visualización en los displays en la escala de ms, ésto nos permitirá una correcta visualización del rango de períodos a medir, eligiendo la correcta señal de pulsos.

Tomemos el caso de la señal de pulsos con período 0.1us ($\text{selec_base}[1..0] = "00" = 0$). Si tuviéramos que medir una señal de período 2ms, el grupo contadores llegaría a un valor de 20000 y la representación en los displays sería:

020000

con lo cual deberíamos ubicar el punto en el dígito 5, de forma que leamos 2ms, con lo cual los display quedarían:

02.0000

Supongamos ahora el caso de la señal de pulsos con período 1us ($\text{selec_base}[1..0] = "01" = 1$) y la misma señal de período 2ms de entrada. El grupo contadores llegaría a un valor de 2000 y la representación en los displays sería:

002000

con lo cual deberíamos ubicar el punto en el dígito 4, de forma que leamos 2ms, con lo cual los display quedarían:

002.000

Ahora para la misma señal de entrada, tomamos el caso de la señal de pulsos con período 10us ($\text{selec_base}[1..0] = "10" = 2$). El grupo contadores llegaría a un valor de 200 y la representación en los displays sería:

000200

con lo cual en este caso para que leamos 2ms deberíamos ubicar el punto en el dígito 3, de forma que los display se muestren:

0002.00

Y nos queda el último caso, para la señal de pulsos con período 100us (selec_base[1..0] = "11" = 3), el grupo de contadores llegaría a 20 (siempre con la misma señal de entrada) con la siguiente visualización de los displays:

000020

y por lo tanto deberíamos ubicar el punto en el dígito 2, de forma que los display se muestren:

00002.0

Por lo tanto podemos confeccionar la tabla 3.10, que nos muestra el dígito que contiene el punto de acuerdo a la señal de pulsos seleccionada:

selec_base[1..0]	Período de la señal de pulsos	dígito que contiene el punto
"00" = 0	0.1us	5
"01" = 1	1us	4
"10" = 2	10us	3
"11" = 3	100us	2

Tabla 3.10: Ubicación del punto dp

O sea que nos ha quedado de la misma forma que para el medidor de frecuencias, y por lo tanto el programa en AHDL será el mismo:

```

dp                                     : OUTPUT;
...
CASE selec_base[] IS
  WHEN B"00" =>
    dp = selec_digito[5];
  WHEN B"01" =>
    dp = selec_digito[4];
  WHEN B"10" =>
    dp = selec_digito[3];
  WHEN B"11" =>
    dp = selec_digito[2];
END CASE;

```

3.3.2.5 Bloque combinatorio + contadores + latches + multiplexor + decodificador BCD 7 segmentos + decodificador + ubicación del punto dp

Vamos ahora a implementar los bloques combinatorio, contadores, latches, decodificador BCD 7 segmentos, decodificador y ubicación del punto dp en un único bloque.

Para éste único bloque tendremos los siguientes pines:

entrada: clock10M, reset, pulsos, signal_in, selec_base[1..0] y selec_decodig[2..0]

salida: a, b, c, d, e, f, g, dp, selec_digito6, selec_digito5, selec_digito4, selec_digito3, selec_digito2 y selec_digito1.

Nuevamente agregaremos algunas salidas para verificar por simulación como:

salidas para simulación: dig6[3..0], dig5[3..0], dig4[3..0], dig3[3..0], dig2[3..0], dig1[3..0], base_de_t, zout y wout.

El programa en lenguaje AHDL para el caso general de una cantidad de dígitos dada, al que llamaremos *bloquep_digitos.tdf*.

```
constant N_DIGITOS = 6; % De modo que %
constant POT_K = ceil(log2(N_DIGITOS)); % 2^POT_K >= N_DIGITOS %

INCLUDE "lpm_counter.inc";
INCLUDE "lpm_latch.inc";
INCLUDE "lpm_mux.inc";
INCLUDE "lpm_decode.inc";

SUBDESIGN bloquep_digitos
(
    clock10M, reset, pulsos, signal_in : INPUT;
    selec_base[1..0], selec_decodig[POT_K-1..0] : INPUT;
    a, b, c, d, e, f, g, dp, zout, wout : OUTPUT;
    selec_digito[N_DIGITOS..1] : OUTPUT;
    dig[N_DIGITOS..1][3..0], base_de_t : OUTPUT;
)

VARIABLE

ss: MACHINE OF BITS (z) WITH STATES (s0 = 0, s1 = 1, s2 = 0);
divisor: lpm_counter WITH (LPM_WIDTH=1);
contador[N_DIGITOS..1]: lpm_counter WITH (LPM_WIDTH=4, LPM_MODULUS=10);
latches[N_DIGITOS..1]: lpm_latch WITH (LPM_WIDTH=4);
multiplexor: lpm_mux WITH (LPM_WIDTH=4, LPM_SIZE=N_DIGITOS,
    LPM_WIDTHS=POT_K);
decodificador: lpm_decode WITH (LPM_WIDTH=POT_K, LPM_DECODES=N_DIGITOS);

resetcont : DFF;

sal_cont[N_DIGITOS..1][3..0], qlatch[N_DIGITOS..1][3..0] : NODE;
sal_mux[3..0], bcd[3..0], nodo[N_DIGITOS..1], w : NODE;
base_tiempos : NODE;
```

```

BEGIN
%   Conversión de período de la señal en nivel alto de base_tiempos   %
divisor.clock = signal_in;
base_tiempos = divisor.q[];

%   Comienzo del diseño del monoestable con maquina de estados       %
ss.clk = clock10M;
ss.reset = reset;
TABLE
%   estado   entrada   próximo   %
%   actual   actual   estado   %
%   ss,      base_tiempos =>   ss;
%
%   s0,      1       =>   s0;
%   s0,      0       =>   s1;
%   s1,      0       =>   s2;
%   s1,      1       =>   s0;
%   s2,      0       =>   s2;
%   s2,      1       =>   s0;
END TABLE;
%   Fin diseño del monoestable                                       %

%   Comienzo del diseño del decodificador BCD a 7 segmentos         %
TABLE
bcd[3..0] => a, b, c, d, e, f, g;
%
%   H"0"    =>   1, 1, 1, 1, 1, 1, 0;
%   H"1"    =>   0, 1, 1, 0, 0, 0, 0;
%   H"2"    =>   1, 1, 0, 1, 1, 0, 1;
%   H"3"    =>   1, 1, 1, 1, 0, 0, 1;
%   H"4"    =>   0, 1, 1, 0, 0, 1, 1;
%   H"5"    =>   1, 0, 1, 1, 0, 1, 1;
%   H"6"    =>   1, 0, 1, 1, 1, 1, 1;
%   H"7"    =>   1, 1, 1, 0, 0, 0, 0;
%   H"8"    =>   1, 1, 1, 1, 1, 1, 1;
%   H"9"    =>   1, 1, 1, 1, 0, 1, 1;
END TABLE;
%   Fin diseño decodificador BCD a 7 segmentos                       %

%   Generación de la señal w para resetear el contador              %
resetcont.clrn = !base_tiempos;
resetcont.d = VCC;
resetcont.clk = !z;
w = resetcont.q;

%   Contadores, latches y multiplexor                               %
FOR i IN 1 TO N_DIGITOS GENERATE
    contador[i].clock = pulsos;
    contador[i].aclr = w;
    latches[i].gate = z;
END GENERATE;
sal_cont[N_DIGITOS..1][3..0] = contador[N_DIGITOS..1].q[];
latches[N_DIGITOS..1].data[] = sal_cont[N_DIGITOS..1][];
qlatch[N_DIGITOS..1][] = latches[N_DIGITOS..1].q[];
multiplexor.data[N_DIGITOS-1..0] [] = qlatch[N_DIGITOS..1][];

contador[1].cnt_en = base_tiempos;
nodo[1] = base_tiempos;
FOR n IN 2 TO N_DIGITOS GENERATE
    nodo[n] = nodo[n-1] & contador[n-1].eq[9];
    contador[n].cnt_en = nodo[n];
END GENERATE;

multiplexor.sel[] = selec_decodig[];
sal_mux[] = multiplexor.result[];

bcd[3..0] = sal_mux[];

```

```

%   Decodificador                                     %
decodificador.data[] = selec_decodig[];
selec_digito[N_DIGITOS..1] = decodificador.eq[N_DIGITOS-1..0];

%   Salidas para simulación                           %
dig[N_DIGITOS..1][] = qlatch[N_DIGITOS..1][];
base_de_t = base_tiempos;
zout = z;
wout = w;

%   Ubicación del punto (dp) de acuerdo a la base utilizada %
%   solo para el caso que N_DIGITOS >= 5             %
CASE selec_base[] IS
  WHEN B"00" =>
    dp = selec_digito[5];
  WHEN B"01" =>
    dp = selec_digito[4];
  WHEN B"10" =>
    dp = selec_digito[3];
  WHEN B"11" =>
    dp = selec_digito[2];
END CASE;
END;
    
```

Al igual que para el frecuencímetro, la ubicación del punto dp solo es válida para el caso de número de dígitos mayor o igual a 5, si el nº de dígitos es menor deberemos reformar la sentencia CASE.

3.3.3 Medidor de períodos en AHDL

Vamos ahora a implementar el medidor de períodos completo en AHDL. Para esto, de la misma forma que hicimos para el de frecuencias, generaremos un símbolo con cada archivo de texto en AHDL (*Gp.tdf* y *bloquep_digitos.tdf*), los conectaremos a través del editor gráfico y realizaremos la posterior simulación.

El primer paso es generar los archivos de símbolo (*Gp.sym* y *bloquep_digitos.sym*), luego los cargaremos y conectaremos en el editor gráfico, de forma que quede como se muestra en la figura 3.27 y a éste archivo lo llamaremos *medperiodo.gdf*.

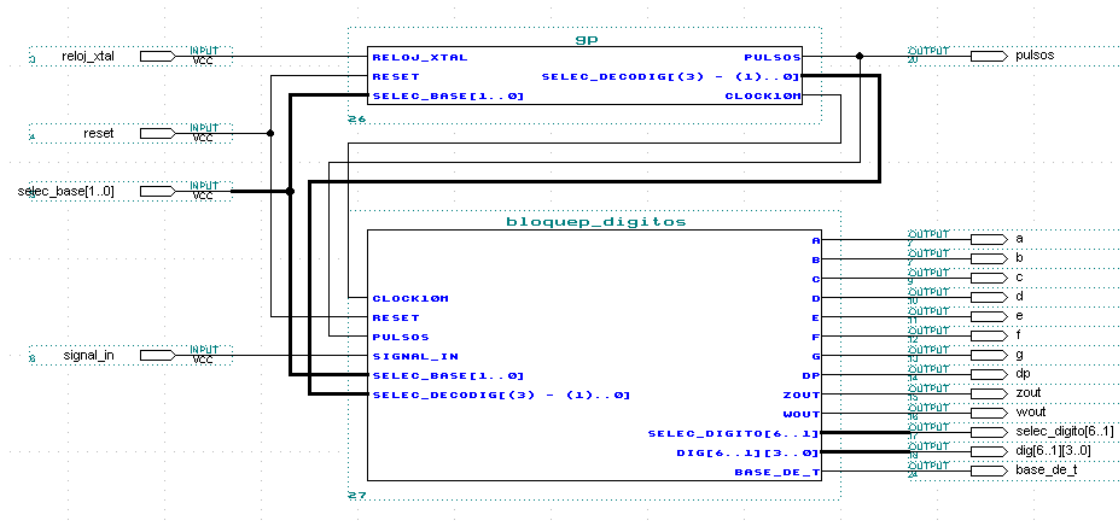


Figura 3.27: Editor gráfico

Ahora realizaremos una simulación funcional, para una señal de entrada cuyo período es 4.93ms ($f = 202.839\text{Hz}$), utilizando secuencialmente la cuatro señales de pulsos y verificaremos las salidas.

Inicialmente vamos a verificar la base de tiempos (1.42ms en alto), la señal z (zout), la señal w (wout), la base de tiempos(base_de_t) y las señales de selección de dígitos (selec_digito[6..1]):

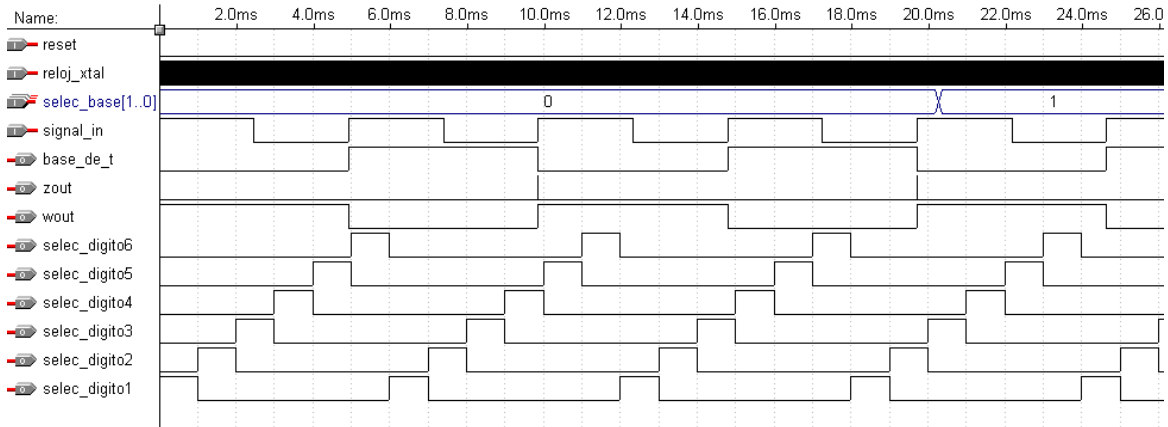


Figura 3.28: Simulación en el Editor de Formas de onda

En la figura 3.28 podemos verificar las señales, las cuales eran las esperadas.

Vamos ahora a verificar el valor de los dígitos de acuerdo a la señal de pulsos seleccionada. Para este caso deberemos leer la salida de los latches en su segunda carga (señal z) luego de la transición de las entradas selec_base[1..0], ya que esta puede haberse realizado cuando ya los contadores tenían un valor y por lo tanto en la primer carga de latches el valor es erróneo.

- Período de la señal de pulsos de 0.1us (selec_base[1..0] = "00" = 0).

De acuerdo a esta señal de pulsos, deberíamos tener una visualización de los display de la forma:

04.9300

Por lo tanto las salidas de los dígitos, luego de la segunda carga de latches, debería ser:

dígito	valor decimal
dig[6][3..0]	0
dig[5][3..0]	4
dig[4][3..0]	9
dig[3][3..0]	3
dig[2][3..0]	0
dig[1][3..0]	0

Tabla 3.11: valor de los dígitos

lo verificamos, tal como muestra la figura 3.29:

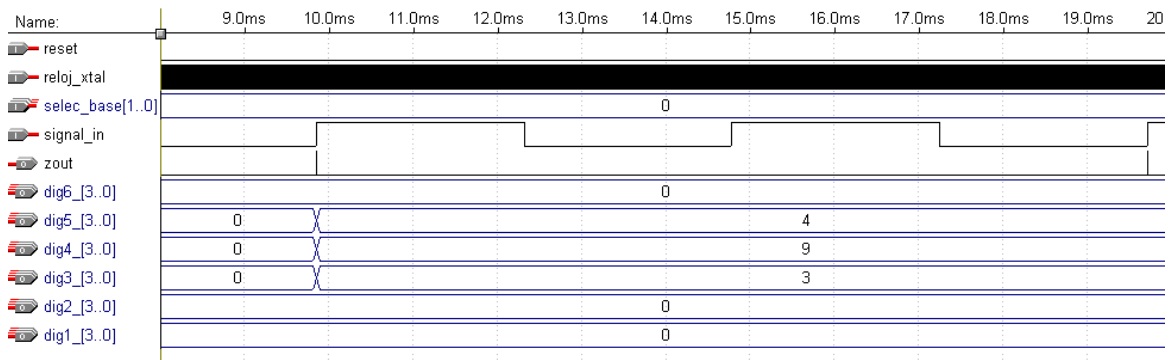


Figura 3.29: Simulación en el Editor de Formas de onda

Ahora vamos a corroborar que en el momento de selección de cada dígito coincidan las salidas a, b, c, d, e, f, g y dp.

Para los dígitos 6, 2 y 1 deberíamos tener la tabla 3.5.

Verificamos en la simulación el dígito 6 (figura 3.30):

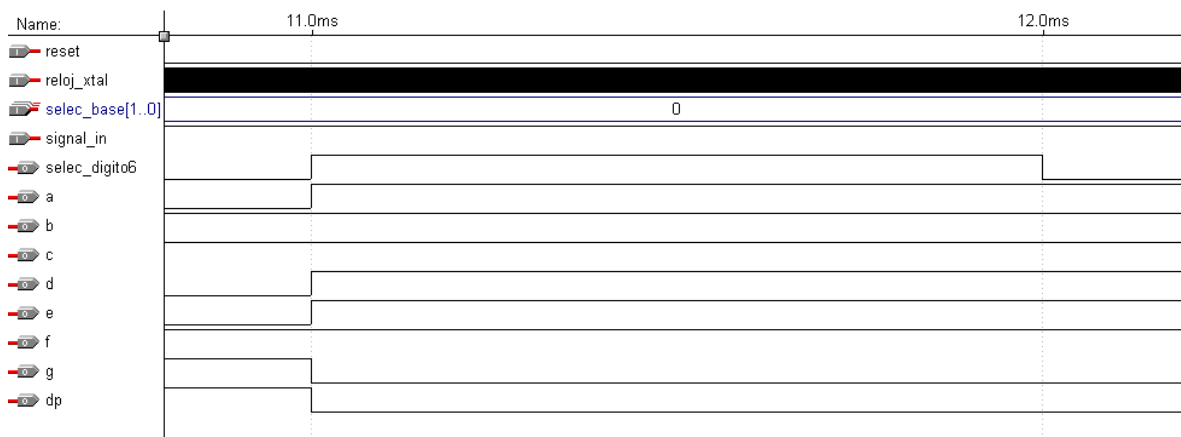


Figura 3.30: Simulación en el Editor de Formas de onda

y los dígitos 1 y 2:

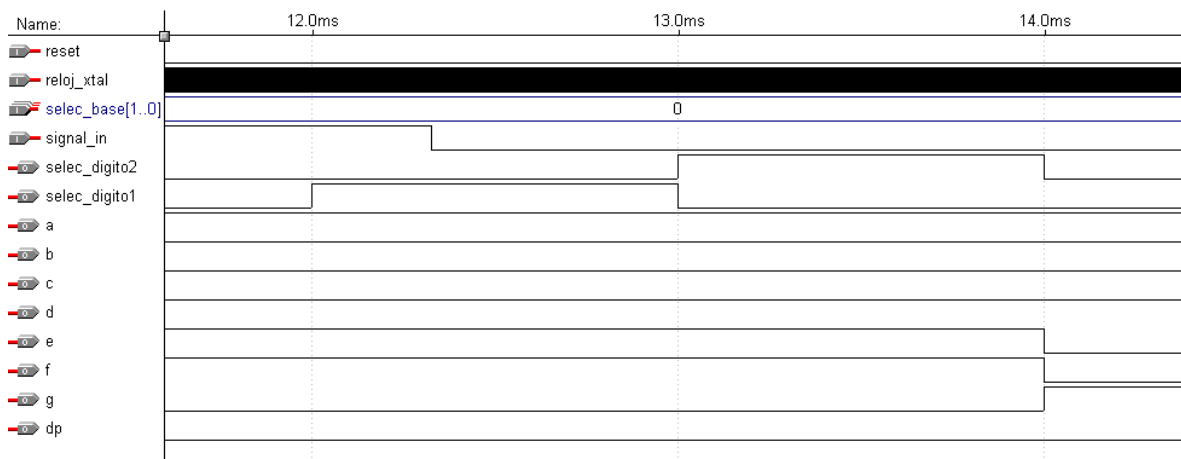


Figura 3.31: Simulación en el Editor de Formas de onda

Para el dígito 5 (el que tiene el punto dp):

valor del dígito	a	b	c	d	e	f	g	dp
4 .	0	1	1	0	0	1	1	1

Tabla 3.12: valor de las salidas a-dp para dígito con valor 4.

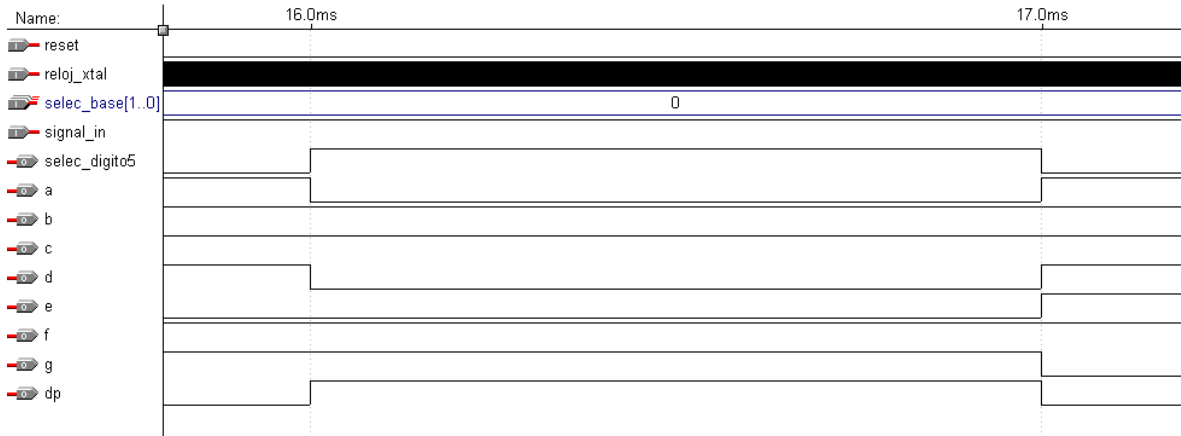


Figura 3.32: Simulación en el Editor de Formas de onda

Para el dígito 4:

valor del dígito	a	b	c	d	e	f	g	dp
9	1	1	1	1	0	1	1	0

Tabla 3.13: valor de las salidas a-dp para dígito con valor 9

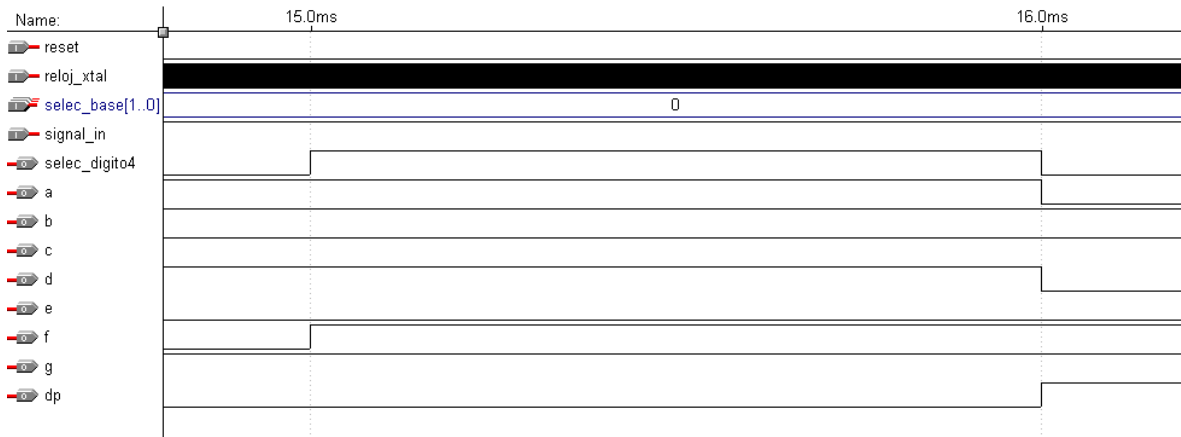


Figura 3.33: Simulación en el Editor de Formas de onda

Y para el dígito 3:

valor del dígito	a	b	c	d	e	f	g	dp
3	1	1	1	1	0	0	1	0

Tabla 3.14: valor de las salidas a-dp para dígito con valor 3

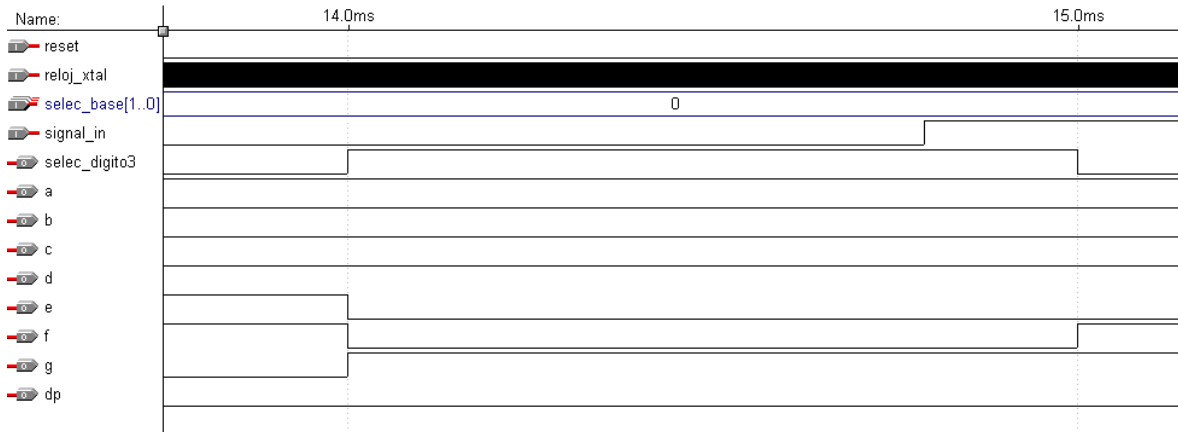


Figura 3.34: Simulación en el Editor de Formas de onda

- Período de la señal de pulsos de 1us ($selec_base[1..0] = "01" = 1$).

De acuerdo a esta señal de pulsos, deberíamos tener una visualización de los display de la forma:

004.930

Por lo tanto las salidas de los dígitos, luego de la segunda carga de latches, debería ser:

dígito	valor decimal
dig[6][3..0]	0
dig[5][3..0]	0
dig[4][3..0]	4
dig[3][3..0]	9
dig[2][3..0]	3
dig[1][3..0]	0

Tabla 3.15: valor de los dígitos

lo verificamos, tal como muestra la figura 3.35:

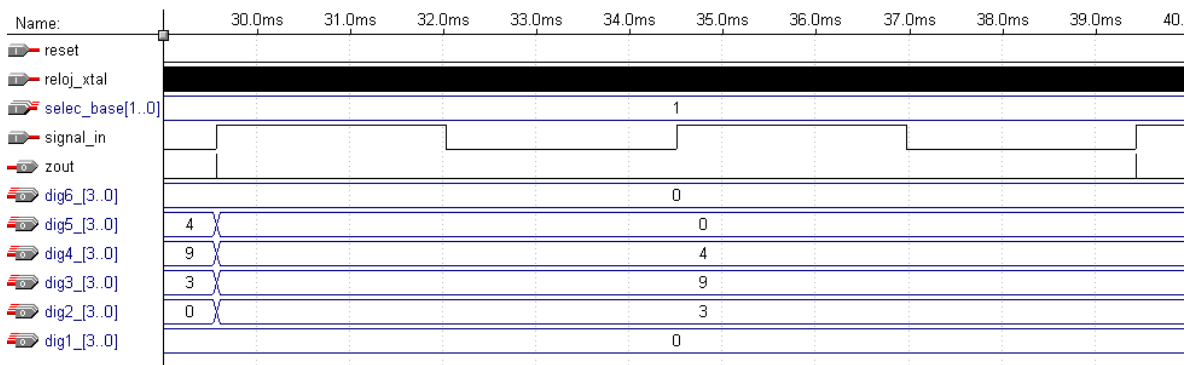


Figura 3.35: Simulación en el Editor de Formas de onda

Vamos a corroborar que en el momento de selección de cada dígito coincidan las salidas a, b, c, d, e, f, g y dp.

Para los dígitos 6, 5 y 1 deberíamos tener la tabla 3.5.

Verificamos para los dígitos 6 y 5:

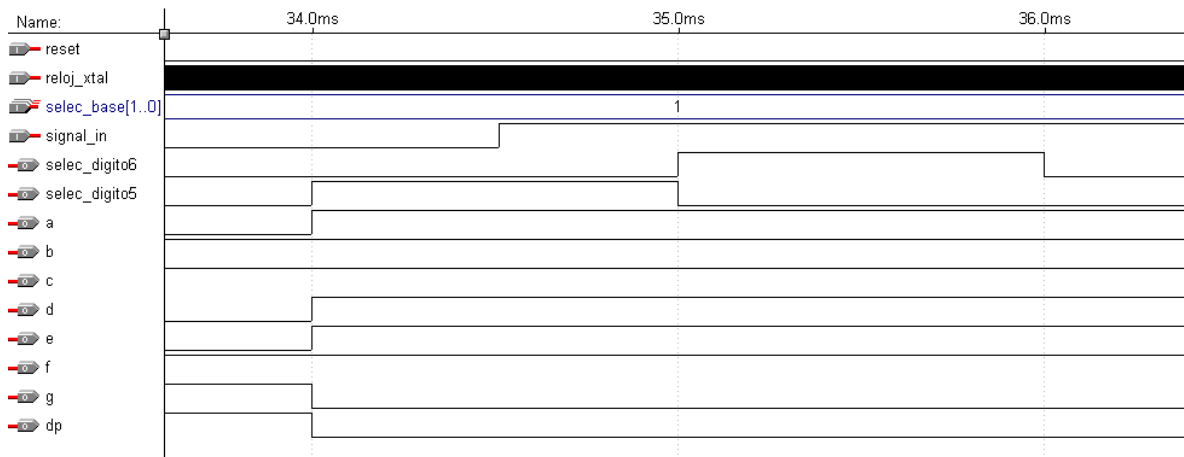


Figura 3.36: Simulación en el Editor de Formas de onda

y para el dígito 1:

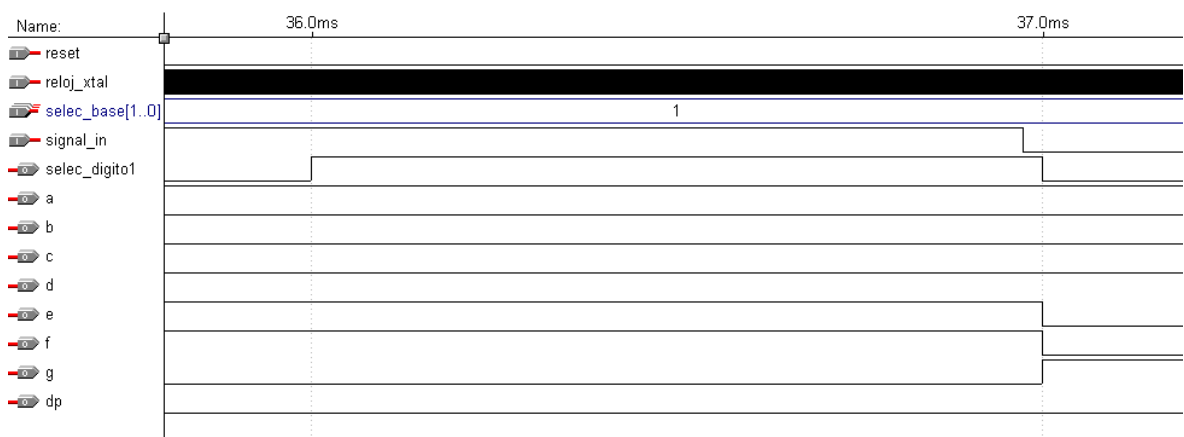


Figura 3.37: Simulación en el Editor de Formas de onda

Para el dígito 4 (el que tiene el punto dp), se debe verificar la tabla 3.12:

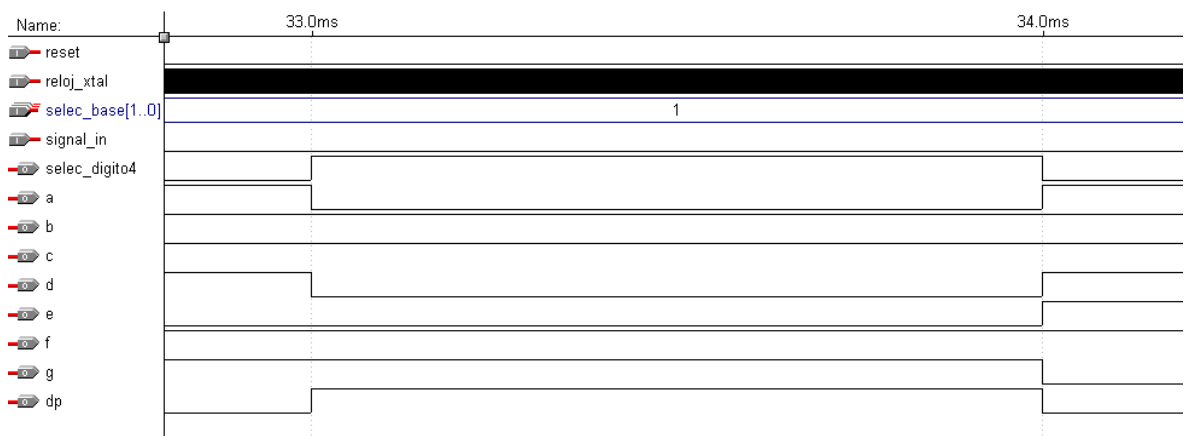


Figura 3.38: Simulación en el Editor de Formas de onda

El dígito 3 debe cumplir la tabla 3.13:

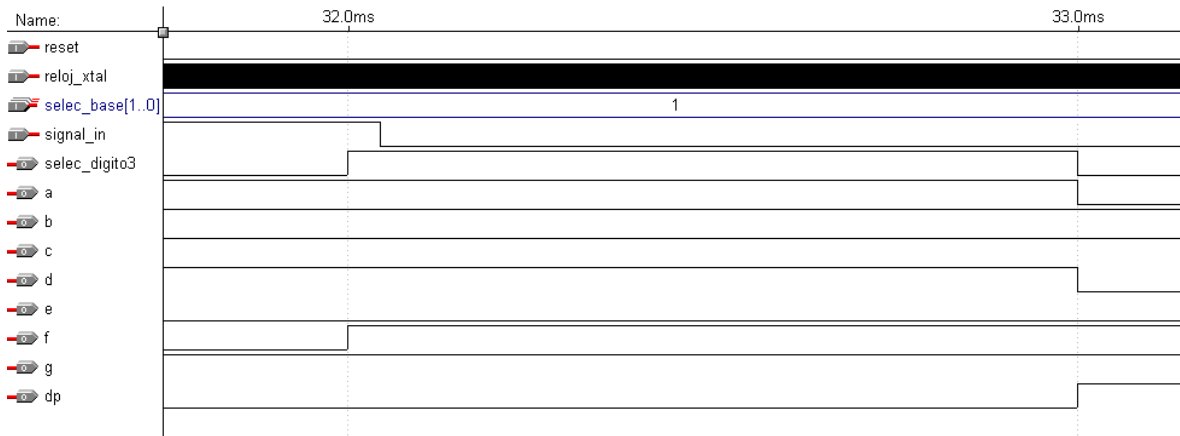


Figura 3.39: Simulación en el Editor de Formas de onda

Y el dígito 2, la tabla 3.14 :

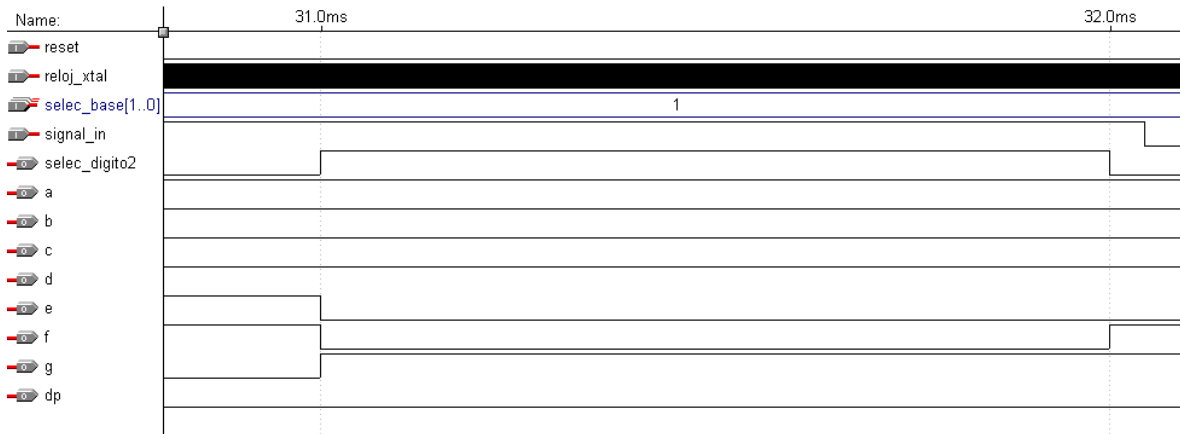


Figura 3.40: Simulación en el Editor de Formas de onda

- Período de la señal de pulsos de 10us ($selec_base[1..0] = "10" = 2$).

De acuerdo a esta señal de pulsos, deberíamos tener una visualización de los display de la forma:

0004.93

Por lo tanto las salidas de los dígitos, luego de la segunda carga de latches, debería ser:

dígito	valor decimal
dig[6][3..0]	0
dig[5][3..0]	0
dig[4][3..0]	0
dig[3][3..0]	4
dig[2][3..0]	9
dig[1][3..0]	3

Tabla 3.16: valor de los dígitos

lo verificamos, tal como muestra la figura 3.41:

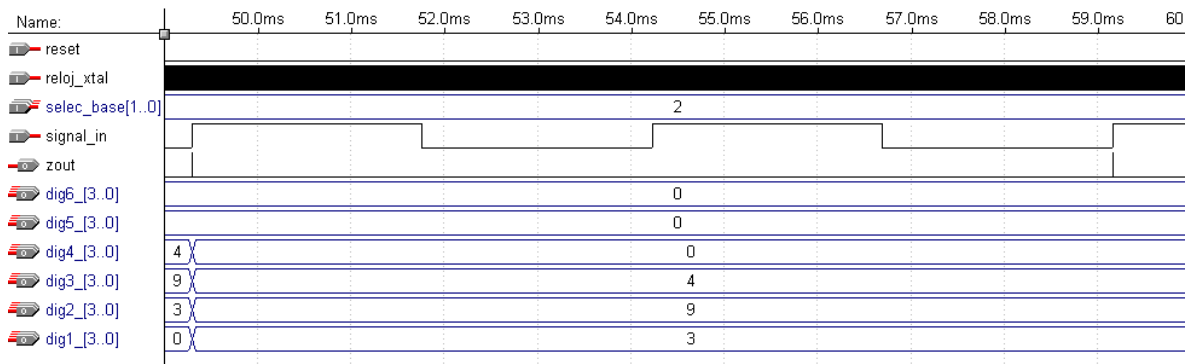


Figura 3.41: Simulación en el Editor de Formas de onda

Verificamos que en el momento de selección de cada dígito coincidan las salidas a, b, c, d, e, f, g y dp.

Para los dígitos 6, 5 y 4 deberíamos tener la tabla 3.5.

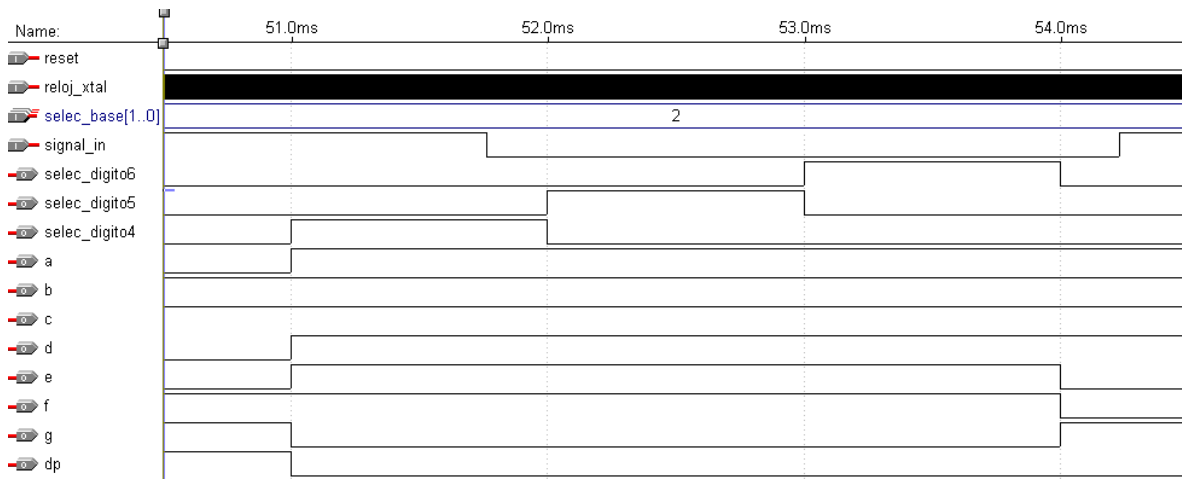


Figura 3.42: Simulación en el Editor de Formas de onda

Para el dígito 3 (el que tiene el punto dp), se debe verificar la tabla 3.12:

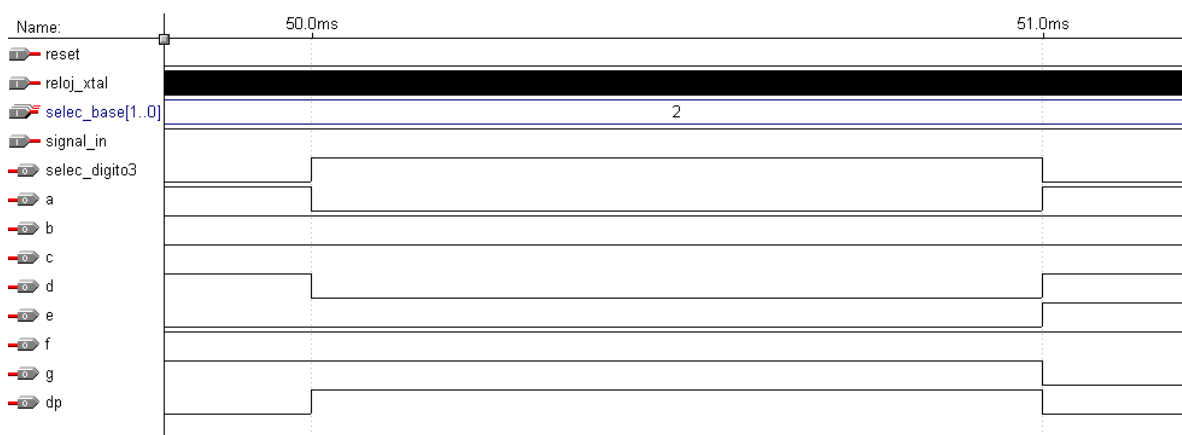


Figura 3.43: Simulación en el Editor de Formas de onda

El dígito 2 debe cumplir la tabla 3.13:

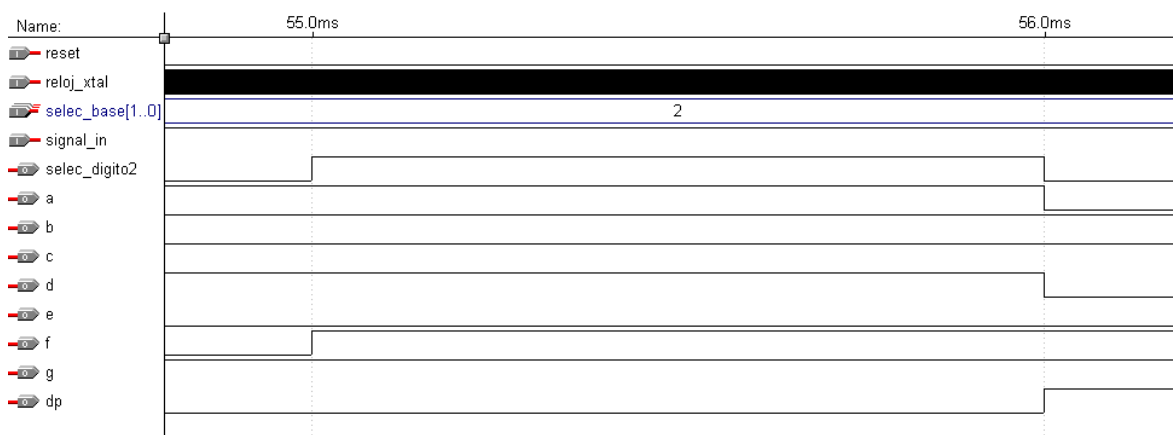


Figura 3.44: Simulación en el Editor de Formas de onda

Y el dígito 1, la tabla 3.14:

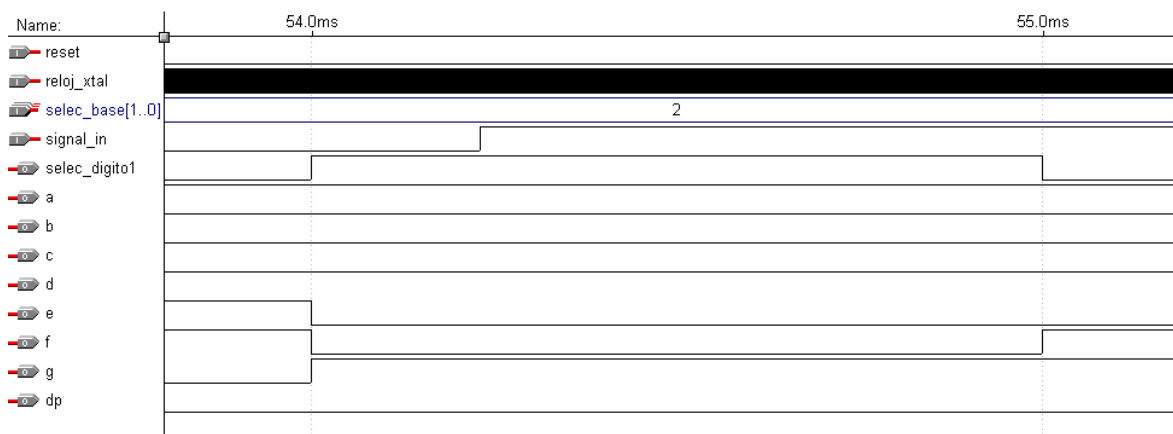


Figura 3.45: Simulación en el Editor de Formas de onda

- Período de la señal de pulsos de 100us ($selec_base[1..0] = "11" = 3$).

De acuerdo a esta señal de pulsos, deberíamos tener una visualización de los display de la forma:

00004.9

Por lo tanto las salidas de los dígitos, luego de la segunda carga de latches, debería ser:

dígito	valor decimal
dig[6][3..0]	0
dig[5][3..0]	0
dig[4][3..0]	0
dig[3][3..0]	0
dig[2][3..0]	4
dig[1][3..0]	9

Tabla 3.17: valor de los dígitos

lo verificamos, tal como muestra la figura 3.46:

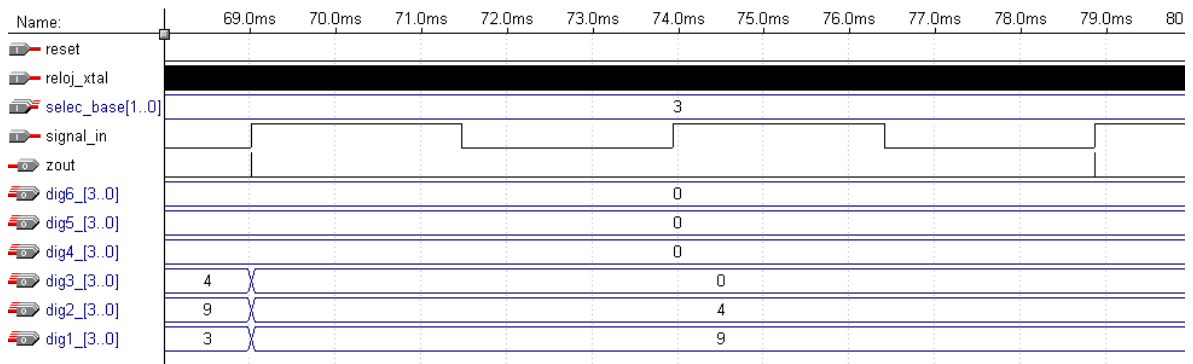


Figura 3.46: Simulación en el Editor de Formas de onda

Nuevamente verificamos que en el momento de selección de cada dígito coincidan las salidas a, b, c, d, e, f, g y dp.

Para los dígitos 6, 5, 4 y 3 deberíamos tener la tabla 3.5.



Figura 3.47: Simulación en el Editor de Formas de onda

Para el dígito 2 (el que tiene el punto dp), se debe verificar la tabla 3.12:

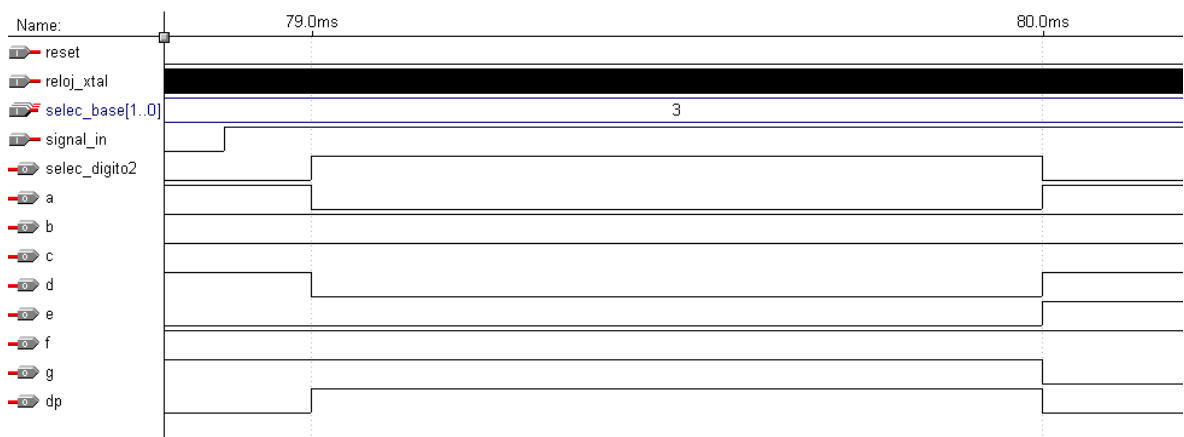


Figura 3.48: Simulación en el Editor de Formas de onda

Mientras que el dígito 1 debe cumplir la tabla 3.13:

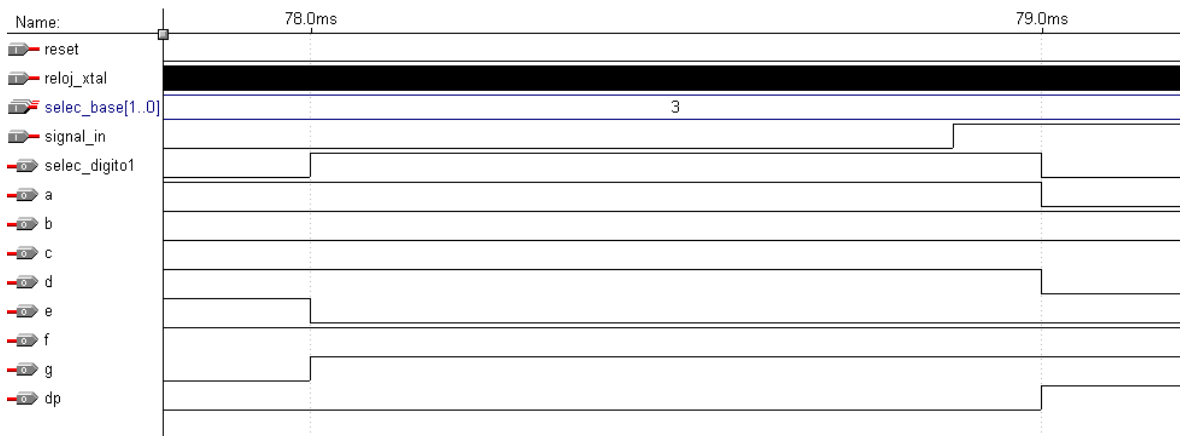


Figura 3.49: Simulación en el Editor de Formas de onda

3.4 Medidor de frecuencia y período en AHDL

3.4.1 Implementación del Medidor de frecuencia y período

Vamos a implementar ahora el medidor de frecuencias y el medidor de períodos en un único medidor de frecuencia y período en AHDL.

Como vimos anteriormente en el diagrama en bloques, ambos medidores tienen muchos bloques comunes como el bloque combinatorio, el de contadores, el de latches, el bloque multiplexor, el decodificador 7 segmentos, el decodificador y el bloque de ubicación del punto.

La principal diferencia entre ambos son las señales que entran a los contadores. En el medidor de frecuencias la señal de habilitación de los contadores es la base de tiempos generada por el Bloque Generador de bases de tiempos y la de reloj de los contadores es la señal cuya frecuencia queremos medir. Por su parte en el medidor de períodos, la señal de habilitación de los contadores es la base de tiempos resultante de la división por dos de la frecuencia de la señal cuyo período queremos obtener; y la de reloj de los contadores es la señal de pulsos con período definido generada por el Bloque Generador de pulsos.

Además se puede verificar que los bloques Generador de pulsos y Generador de Bases de tiempos tienen una estructura similar basada en un prescaler de cuatro etapas (lo cual no fue coincidencia y se tuvo en cuenta el conjunto al diseñar cada uno de ellos), y entradas de selección de igual nombre (`selec_base[1..0]`).

Deberíamos entonces inicialmente, tomar una entrada que nos defina que vamos a medir, si frecuencia o período. Esta entrada debería permitir elegir cual de las señales, de reloj y habilitación, comandan al grupo de contadores. A esta entrada la llamaremos `FOP`, y le asignaremos la siguiente característica:

FoP	Modo medición de
0	Período
1	Frecuencia

Tabla 3.18: Modo de medición de acuerdo a FoP

Vamos a comenzar definiendo la primera parte que será el generador de bases de tiempos y el generador de pulsos juntos:

```

constant MAX_COUNT = 10000;
constant N_DIGITOS = 6;
constant POT_K = ceil(log2(N_DIGITOS));
...
reloj_xtal, reset, selec_base[1..0] : INPUT;
...
cuenta: lpm_counter WITH (LPM_WIDTH=ceil(log2(MAX_COUNT)));
prescaler1: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
prescaler2: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
prescaler3: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
prescaler4: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
presc_sel_decodig: lpm_counter WITH ( LPM_WIDTH=POT_K, LPM_MODULUS=N_DIGITOS);
...
salida : DFF;
...
borrar, reloj_lms, clock10M, selec_decodig[POT_K-1..0], pulsos : NODE;
...
prescaler1.aclr = reset;
prescaler2.aclr = reset;
prescaler3.aclr = reset;
prescaler4.aclr = reset;
presc_sel_decodig.aclr = reset;
prescaler1.clock = reloj_xtal;
prescaler2.clock = reloj_xtal;
prescaler3.clock = reloj_xtal;
prescaler4.clock = reloj_xtal;
presc_sel_decodig.clock = reloj_xtal;
prescaler2.cnt_en = prescaler1.eq[9];
prescaler3.cnt_en = prescaler2.eq[9] & prescaler1.eq[9];
prescaler4.cnt_en = prescaler3.eq[9] & prescaler2.eq[9] & prescaler1.eq[9];
presc_sel_decodig.cnt_en = prescaler4.eq[9] & prescaler3.eq[9] &
prescaler2.eq[9] & prescaler1.eq[9];
reloj_lms = prescaler4.eq[9];
selec_decodig[POT_K-1..0] = presc_sel_decodig.q[];
clock10M = reloj_xtal;

salida.clrn = !reset;
salida.clk = reloj_lms;
cuenta.clock = reloj_lms;
cuenta.sclr = borrar;
CASE selec_base[] IS
  WHEN B"00" =>
    IF cuenta.q[] < 10000 THEN
      salida.d = VCC;
      borrar = GND;
    ELSE
      borrar = VCC;
      salida.d = GND;
    END IF;
  WHEN B"01" =>
    IF cuenta.q[] < 1000 THEN
      salida.d = VCC;
      borrar = GND;
    ELSE
      salida.d = GND;
      borrar = VCC;
    END IF;

```

```

    WHEN B"10" =>
        IF cuenta.q[] < 100 THEN
            salida.d = VCC;
            borrar = GND;
        ELSE
            borrar = VCC;
            salida.d = GND;
        END IF;
    WHEN B"11" =>
        IF cuenta.q[] < 10 THEN
            salida.d = VCC;
            borrar = GND;
        ELSE
            borrar = VCC;
            salida.d = GND;
        END IF;
END CASE;

CASE selec_base[] IS
    WHEN B"00" =>
        pulsos = reloj_xtal;
    WHEN B"01" =>
        pulsos = prescaler1.eq[9];
    WHEN B"10" =>
        pulsos = prescaler2.eq[9];
    WHEN B"11" =>
        pulsos = prescaler3.eq[9];
END CASE;

```

Vamos a agregar ahora el divisor por dos de la frecuencia de entrada, para el modo medición de períodos.

```

signal_in                                     : INPUT;
...
divisor: lpm_counter WITH (LPM_WIDTH=1);
...
divisor.clock = signal_in;

```

Ahora agregaremos la entrada `FoP`. También agregaremos dos nodos llamados `base_tiempos` y `reloj`, los cuales conectaremos mas adelante a la habilitación y reloj de los contadores respectivamente y mediante una sentencia `IF` (podría ser también una sentencia `CASE`), les asignamos:

```

FoP                                           : INPUT;
...
base_tiempos, reloj                          : NODE;
...
IF FoP THEN
    base_tiempos = salida.q;                  % Medición de %
    reloj = signal_in;                       % Frecuencia %
ELSE
    base_tiempos = divisor.q[];              % Medición de %
    reloj = pulsos;                          % Período %
END IF;

```

Es decir que para el caso de medición de frecuencias, conectamos la salida del `ff_salida` del Generador de bases de tiempos como base de tiempos y como reloj la señal a medir. Para el caso de medición de período, asignamos como base de tiempos la salida del divisor por dos de la frecuencia de la señal a medir y como reloj los pulsos generados por el Generador de pulsos. Luego los contadores, latches, decodificadores y ubicación del punto son iguales tanto para medir frecuencia como período:

```

a, b, c, d, e, f, g, dp                                     : OUTPUT;
selec_digito[N_DIGITOS..1]                               : OUTPUT;
...
ss: MACHINE OF BITS (z) WITH STATES      (s0 = 0, s1 = 1, s2 = 0);
contador[N_DIGITOS..1]: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
latches[N_DIGITOS..1]: lpm_latch WITH (LPM_WIDTH=4);
multiplexor: lpm_mux WITH (LPM_WIDTH=4, LPM_SIZE=N_DIGITOS, LPM_WIDTHS=POT_K);
decodificador: lpm_decode WITH (LPM_WIDTH=POT_K, LPM_DECODES=N_DIGITOS);
...
resetcont                                               : DFF;
...
sal_cont[N_DIGITOS..1][3..0], qlatch[N_DIGITOS..1][3..0] : NODE;
sal_mux[3..0], bcd[3..0], nodo[N_DIGITOS..1], w         : NODE;

```

la generación de la señal w:

```

ss.clk = clock10M;
ss.reset = reset;
TABLE
%   estado   entrada           próximo %
%   actual   actual             estado  %
%   ss,      base_tiempos =>      ss;

s0,      1      =>      s0;
s0,      0      =>      s1;
s1,      0      =>      s2;
s1,      1      =>      s0;
s2,      0      =>      s2;
s2,      1      =>      s0;
END TABLE;

```

el decodificador BCD a 7 segmentos:

```

TABLE
bcd[3..0] => a, b, c, d, e, f, g;
H"0" => 1, 1, 1, 1, 1, 1, 0;
H"1" => 0, 1, 1, 0, 0, 0, 0;
H"2" => 1, 1, 0, 1, 1, 0, 1;
H"3" => 1, 1, 1, 1, 0, 0, 1;
H"4" => 0, 1, 1, 0, 0, 1, 1;
H"5" => 1, 0, 1, 1, 0, 1, 1;
H"6" => 1, 0, 1, 1, 1, 1, 1;
H"7" => 1, 1, 1, 0, 0, 0, 0;
H"8" => 1, 1, 1, 1, 1, 1, 1;
H"9" => 1, 1, 1, 1, 0, 1, 1;
END TABLE;

```

la generación de la señal w:

```

resetcont.clrn = !base_tiempos;
resetcont.d = VCC;
resetcont.clk = !z;
w = resetcont.q;

```

los contadores latches y el MUX:

```

FOR i IN 1 TO N_DIGITOS GENERATE
  contador[i].clock = reloj;
  contador[i].aclr = w;
  latches[i].gate = z;
END GENERATE;
sal_cont[N_DIGITOS..1][3..0] = contador[N_DIGITOS..1].q[];
latches[N_DIGITOS..1].data[] = sal_cont[N_DIGITOS..1][];
qlatch[N_DIGITOS..1][] = latches[N_DIGITOS..1].q[];
multiplexor.data[N_DIGITOS-1..0] [] = qlatch[N_DIGITOS..1][];

```

```

contador[1].cnt_en = base_tiempos;
nodo[1] = base_tiempos;
FOR n IN 2 TO N_DIGITOS GENERATE
    nodo[n] = nodo[n-1] & contador[n-1].eq[9];
    contador[n].cnt_en = nodo[n];
END GENERATE;

```

```

multiplexor.sel[] = selec_decodig[];
sal_mux[] = multiplexor.result[];

```

```

bcd[3..0] = sal_mux[];

```

el decodificador:

```

decodificador.data[] = selec_decodig[];
selec_digito[N_DIGITOS..1] = decodificador.eq[N_DIGITOS-1..0];

```

y la ubicación del punto dp:

```

CASE selec_base[] IS
    WHEN B"00" =>
        dp = selec_digito[5];
    WHEN B"01" =>
        dp = selec_digito[4];
    WHEN B"10" =>
        dp = selec_digito[3];
    WHEN B"11" =>
        dp = selec_digito[2];
END CASE;

```

Luego el programa en AHDL completo al que llamaremos *med_fre_per.tdf* queda de la forma:

```

constant MAX_COUNT = 10000;
constant N_DIGITOS = 6;
constant POT_K = ceil(log2(N_DIGITOS));
% De modo que %
% 2^POT_K >= N_DIGITOS %

INCLUDE "lpm_counter.inc";
INCLUDE "lpm_latch.inc";
INCLUDE "lpm_mux.inc";
INCLUDE "lpm_decode.inc";

SUBDESIGN med_fre_per
(
    reloj_xtal, reset, selec_base[1..0], signal_in, FoP : INPUT;
    a, b, c, d, e, f, g, dp, selec_digito[N_DIGITOS..1] : OUTPUT;
    dig[N_DIGITOS..1][3..0], zout % para simulación % : OUTPUT;
)

VARIABLE

    cuenta: lpm_counter WITH (LPM_WIDTH=ceil(log2(MAX_COUNT)));
    prescaler1: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
    prescaler2: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
    prescaler3: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
    prescaler4: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
    presc_sel_decodig: lpm_counter WITH ( LPM_WIDTH=POT_K,
        LPM_MODULUS=N_DIGITOS);
    divisor: lpm_counter WITH (LPM_WIDTH=1);
    contador[N_DIGITOS..1]: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
    latches[N_DIGITOS..1]: lpm_latch WITH (LPM_WIDTH=4);
    multiplexor: lpm_mux WITH (LPM_WIDTH=4, LPM_SIZE=N_DIGITOS,
        LPM_WIDTHS=POT_K);
    decodificador: lpm_decode WITH (LPM_WIDTH=POT_K, LPM_DECODES=N_DIGITOS);
    ss: MACHINE OF BITS (z) WITH STATES (s0 = 0, s1 = 1, s2 = 0);

```

```

salida, resetcont                                     : DFF;

borrar, reloj_1ms, clock10M, selec_decodig[POT_K-1..0] : NODE;
base_tiempos, reloj, pulsos                          : NODE;
sal_cont[N_DIGITOS..1][3..0], qlatch[N_DIGITOS..1][3..0] : NODE;
sal_mux[3..0], bcd[3..0], nodo[N_DIGITOS..1], w      : NODE;

BEGIN

% Comienzo etapa prescaler %
prescaler1.aclr = reset;
prescaler2.aclr = reset;
prescaler3.aclr = reset;
prescaler4.aclr = reset;
presc_sel_decodig.aclr = reset;
prescaler1.clock = reloj_xtal;
prescaler2.clock = reloj_xtal;
prescaler3.clock = reloj_xtal;
prescaler4.clock = reloj_xtal;
presc_sel_decodig.clock = reloj_xtal;
prescaler2.cnt_en = prescaler1.eq[9];
prescaler3.cnt_en = prescaler2.eq[9] & prescaler1.eq[9];
prescaler4.cnt_en = prescaler3.eq[9] & prescaler2.eq[9] &
                    prescaler1.eq[9];
presc_sel_decodig.cnt_en = prescaler4.eq[9] & prescaler3.eq[9] &
                    prescaler2.eq[9] & prescaler1.eq[9];

reloj_1ms = prescaler4.eq[9];
selec_decodig[POT_K-1..0] = presc_sel_decodig.q[];
clock10M = reloj_xtal;
% Fin etapa prescaler %

% Base de tiempo para modo medición de frecuencias %
salida.clrn = !reset;
salida.clk = reloj_1ms;
cuenta.clock = reloj_1ms;
cuenta.sclr = borrar;
CASE selec_base[] IS
    WHEN B"00" =>
        IF cuenta.q[] < 10000 THEN
            salida.d = VCC;
            borrar = GND;
        ELSE
            borrar = VCC;
            salida.d = GND;
        END IF;
    WHEN B"01" =>
        IF cuenta.q[] < 1000 THEN
            salida.d = VCC;
            borrar = GND;
        ELSE
            salida.d = GND;
            borrar = VCC;
        END IF;
    WHEN B"10" =>
        IF cuenta.q[] < 100 THEN
            salida.d = VCC;
            borrar = GND;
        ELSE
            borrar = VCC;
            salida.d = GND;
        END IF;
    WHEN B"11" =>
        IF cuenta.q[] < 10 THEN
            salida.d = VCC;
            borrar = GND;
        ELSE
            borrar = VCC;

```

```

        salida.d = GND;
    END IF;
END CASE;

% Pulsos para modo de medición de período %
CASE selec_base[] IS
    WHEN B"00" =>
        pulsos = reloj_xtal;
    WHEN B"01" =>
        pulsos = prescaler1.eq[9];
    WHEN B"10" =>
        pulsos = prescaler2.eq[9];
    WHEN B"11" =>
        pulsos = prescaler3.eq[9];
END CASE;

% Base de tiempo para modo medición de períodos %
divisor.clock = signal_in;

% Conexión de los nodos base_tiempos y reloj según se mida frecuencia ó período %
%
IF FoP THEN
    base_tiempos = salida.q;           % Medición de %
    reloj = signal_in;                % Frecuencia %
ELSE
    base_tiempos = divisor.q[];       % Medición de %
    reloj = pulsos;                   % Período %
END IF;

% Monoestable con maquina de estados %
ss.clk = clock10M;
ss.reset = reset;
TABLE
% estado entrada próximo %
% actual actual estado %
ss, base_tiempos => ss;

s0, 1 => s0;
s0, 0 => s1;
s1, 0 => s2;
s1, 1 => s0;
s2, 0 => s2;
s2, 1 => s0;
END TABLE;

% Decodificador BCD a 7 segmentos %
TABLE
bcd[3..0] => a, b, c, d, e, f, g;
H"0" => 1, 1, 1, 1, 1, 1, 0;
H"1" => 0, 1, 1, 0, 0, 0, 0;
H"2" => 1, 1, 0, 1, 1, 0, 1;
H"3" => 1, 1, 1, 1, 0, 0, 1;
H"4" => 0, 1, 1, 0, 0, 1, 1;
H"5" => 1, 0, 1, 1, 0, 1, 1;
H"6" => 1, 0, 1, 1, 1, 1, 1;
H"7" => 1, 1, 1, 0, 0, 0, 0;
H"8" => 1, 1, 1, 1, 1, 1, 1;
H"9" => 1, 1, 1, 1, 0, 1, 1;
END TABLE;

% Generación de la señal w para resetear el contador %
resetcont.clrn = !base_tiempos;
resetcont.d = VCC;
resetcont.clk = !z;
w = resetcont.q;

```

```

%      Contadores, latches y multiplexor                                     %
FOR i IN 1 TO N_DIGITOS GENERATE
    contador[i].clock = reloj;
    contador[i].aclr = w;
    latches[i].gate = z;
END GENERATE;
sal_cont[N_DIGITOS..1][3..0] = contador[N_DIGITOS..1].q[];
latches[N_DIGITOS..1].data[] = sal_cont[N_DIGITOS..1][];
qlatch[N_DIGITOS..1][] = latches[N_DIGITOS..1].q[];
multiplexor.data[N_DIGITOS-1..0][] = qlatch[N_DIGITOS..1][];

contador[1].cnt_en = base_tiempos;
nodo[1] = base_tiempos;
FOR n IN 2 TO N_DIGITOS GENERATE
    nodo[n] = nodo[n-1] & contador[n-1].eq[9];
    contador[n].cnt_en = nodo[n];
END GENERATE;

multiplexor.sel[] = selec_decodig[];
sal_mux[] = multiplexor.result[];

bcd[3..0] = sal_mux[];

%      Decodificador                                                         %
decodificador.data[] = selec_decodig[];
selec_digito[N_DIGITOS..1] = decodificador.eq[N_DIGITOS-1..0];

%      Salidas para simulación                                               %
dig[N_DIGITOS..1][] = qlatch[N_DIGITOS..1][];
zout = z;

%      Ubicación del punto (dp) de acuerdo a la base utilizada              %
%      solo para el caso que N_DIGITOS >= 5                                 %
CASE selec_base[] IS
    WHEN B"00" =>
        dp = selec_digito[5];
    WHEN B"01" =>
        dp = selec_digito[4];
    WHEN B"10" =>
        dp = selec_digito[3];
    WHEN B"11" =>
        dp = selec_digito[2];
END CASE;

END;

```

Nuevamente vamos a realizar una simulación funcional para corroborar el correcto funcionamiento del sistema. Como pudimos apreciar en el programa, hemos agregado las salidas `dig[6..1][3..0]` y `zout` para verificar por simulación. Tomaremos como entrada, una señal cuya frecuencia es de 6.8KHz ($T = 147\mu s = 0.147ms$), a la cual le mediremos su período (FoP = 0) utilizando un período de la señal de pulsos de 1 μs (`selec_base[1..0] = "01" = 1`) y su frecuencia (FoP = 1) mediante una base de tiempos con 10ms = 0.01seg. en nivel alto (`selec_base[1..0] = "11" = 3`).

Vamos a verificar los resultados:

- Período: de acuerdo a la señal de pulsos utilizada, deberíamos tener una visualización de los displays de la forma:

000.147

por lo tanto las salidas de los dígitos, luego de la segunda carga de latches, debería ser:

dígito	valor decimal
dig[6][3..0]	0
dig[5][3..0]	0
dig[4][3..0]	0
dig[3][3..0]	1
dig[2][3..0]	4
dig[1][3..0]	7

Tabla 3.19: valor de los dígitos

lo verificamos en la simulación (figura 3.50):

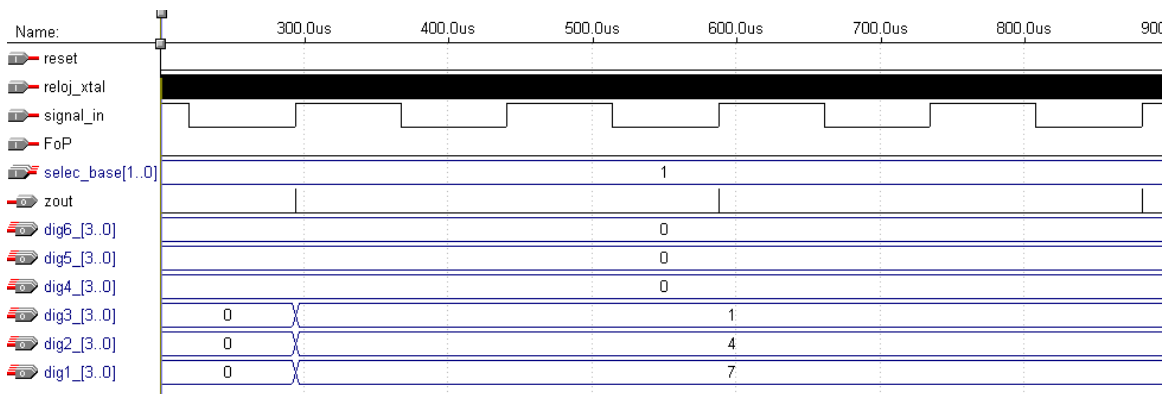


Figura 3.50: Simulación en el Editor de Formas de onda

Vamos a corroborar que en el momento de selección de cada dígito coincidan las salidas a, b, c, d, e, f, g y dp.

para los dígitos 6 y 5 deberíamos tener la tabla 3.5:

verificamos en la figura 3.51:

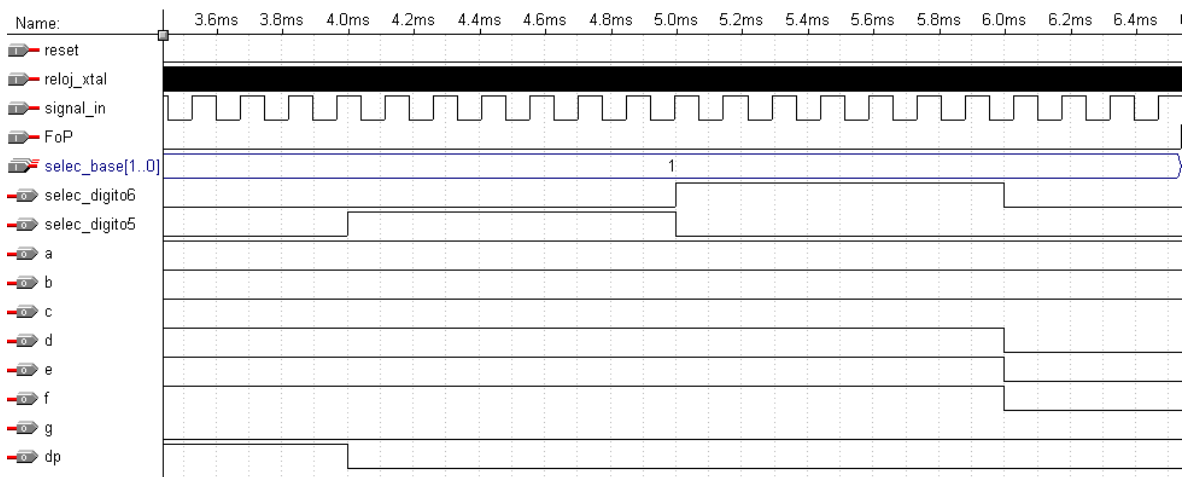


Figura 3.51: Simulación en el Editor de Formas de onda

para el dígito 4 (el que tiene el punto):

valor del dígito	a	b	c	d	e	f	g	dp
0.	1	1	1	1	1	1	0	1

Tabla 3.20: valor de las salidas a-dp para dígito con valor 0.

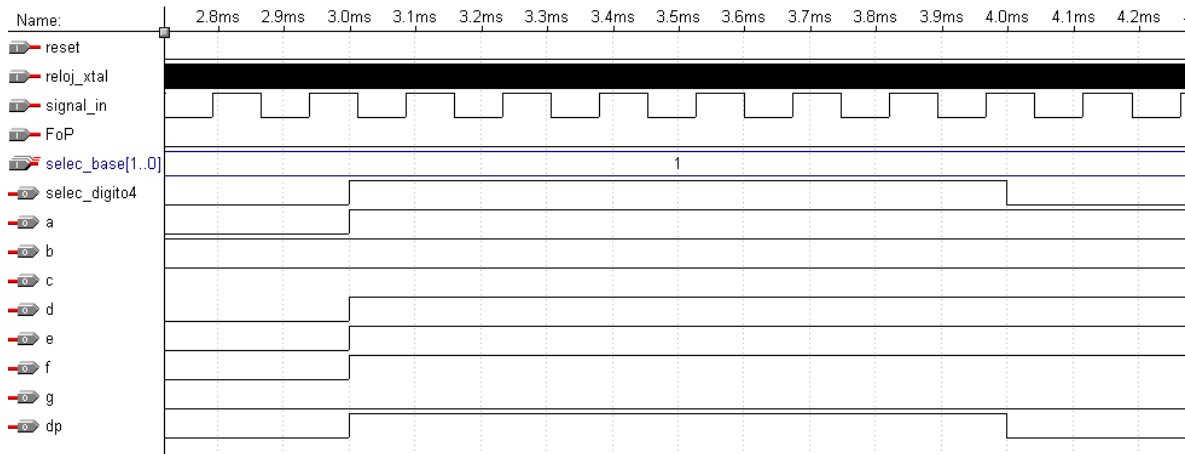


Figura 3.52: Simulación en el Editor de Formas de onda

para el dígito 3:

valor del dígito	a	b	c	d	e	f	g	dp
1	0	1	1	0	0	0	0	0

Tabla 3.21: valor de las salidas a-dp para dígito con valor 1

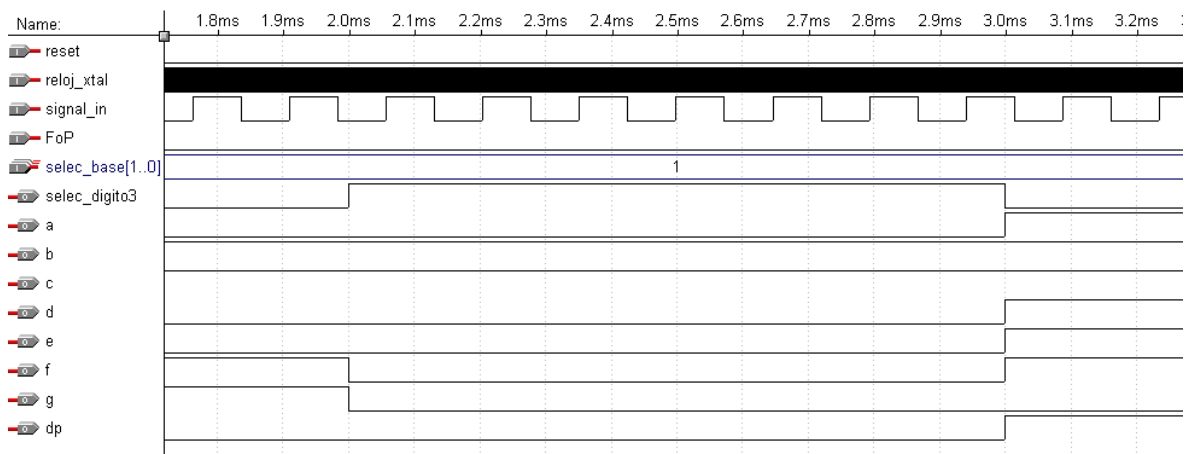


Figura 3.53: Simulación en el Editor de Formas de onda

para el dígito 2:

valor del dígito	a	b	c	d	e	f	g	dp
4	0	1	1	0	0	1	1	0

Tabla 3.22: valor de las salidas a-dp para dígito con valor 4

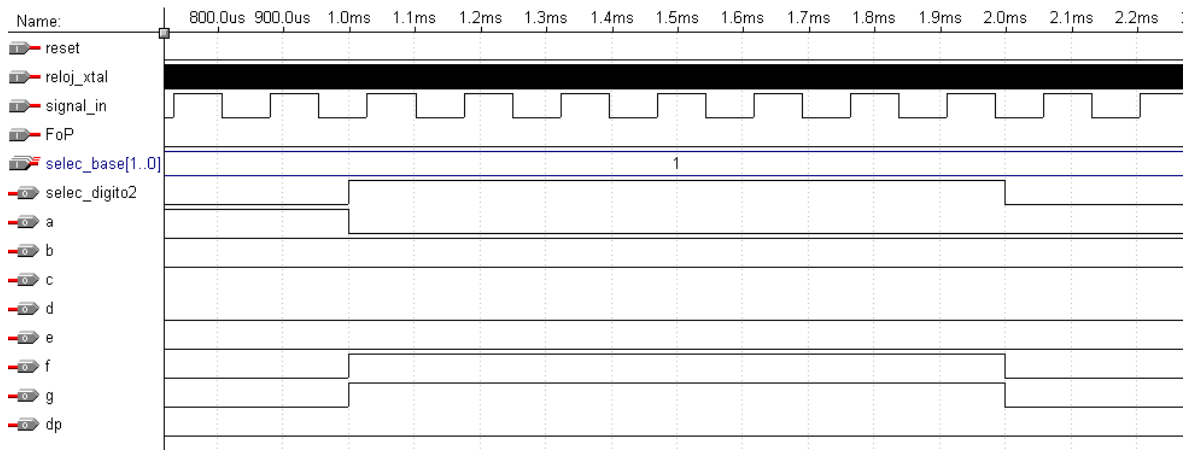


Figura 3.54: Simulación en el Editor de Formas de onda

y para el dígito 1:

valor del dígito	a	b	c	d	e	f	g	dp
7	1	1	1	0	0	0	0	0

Tabla 3.23: valor de las salidas a-dp para dígito con valor 7

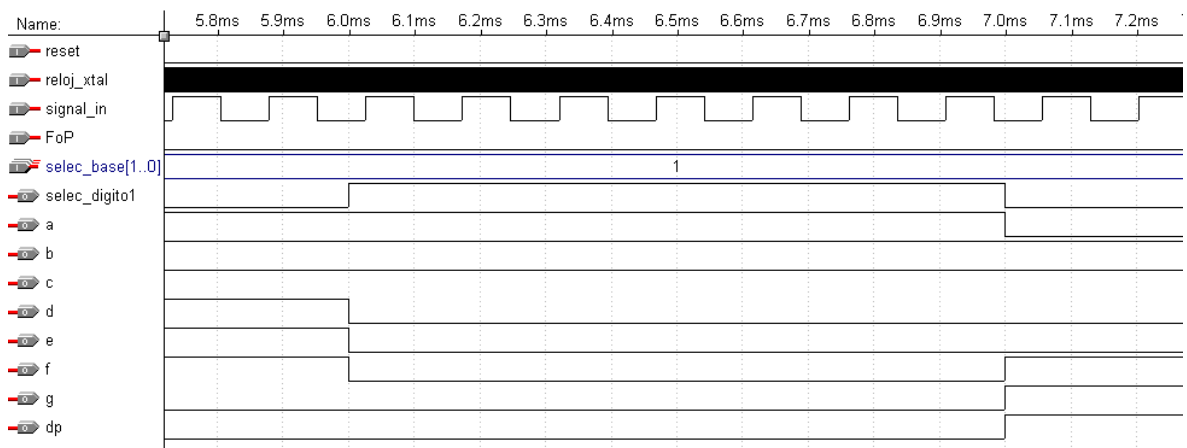


Figura 3.55: Simulación en el Editor de Formas de onda

- Frecuencia: de acuerdo a la base de tiempo utilizada, deberíamos tener una visualización de los displays de la forma:

00006.8

por lo tanto las salidas de los dígitos, luego de la segunda carga de latches, debería ser:

dígito	valor decimal
dig[6][3..0]	0
dig[5][3..0]	0
dig[4][3..0]	0

dig[3][3..0]	0
dig[2][3..0]	6
dig[1][3..0]	8

Tabla 3.24: valor de los dígitos

lo verificamos en la figura 3.56:

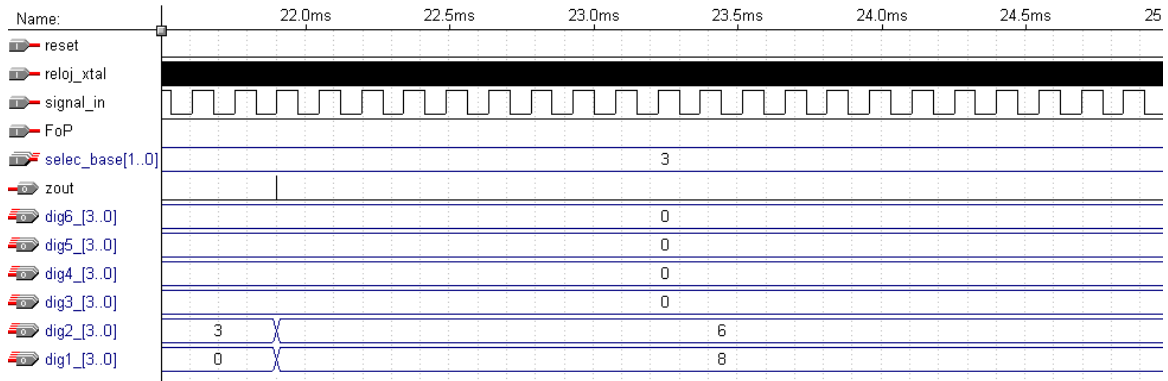


Figura 3.56: Simulación en el Editor de Formas de onda

Vamos a verificar que coincidan las salidas a, b, c, d, e, f, g y dp en el momento de selección de cada dígito.

para los dígitos 6, 5, 4 y 3 deberíamos tener la tabla 3.5:

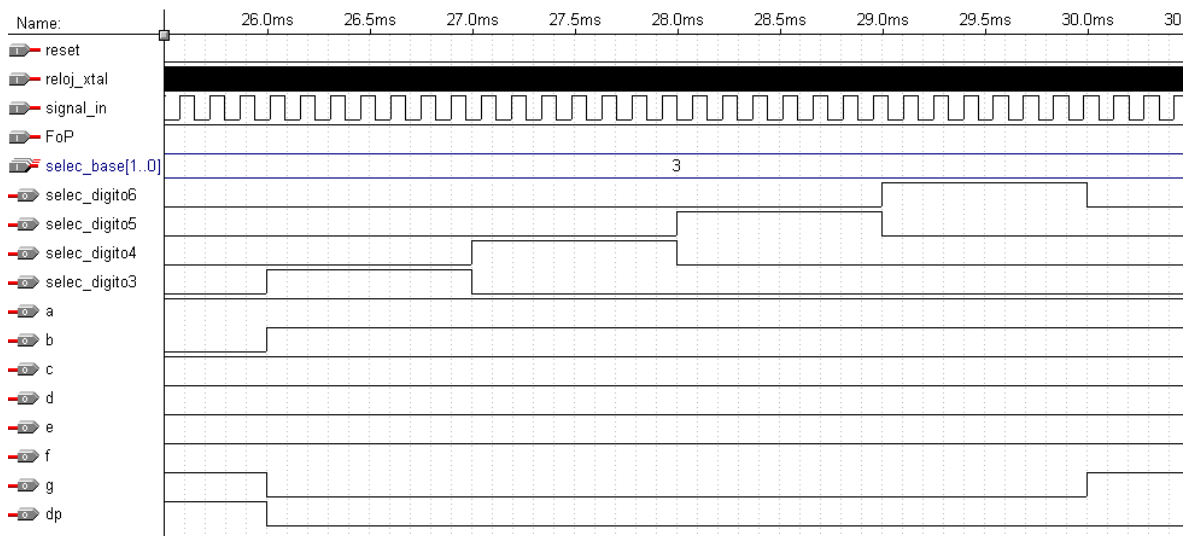


Figura 3.57: Simulación en el Editor de Formas de onda

para el dígito 2 (el que tiene el punto):

valor del dígito	a	b	c	d	e	f	g	dp
6.	1	0	1	1	1	1	1	1

Tabla 3.25: valor de las salidas a-dp para dígito con valor 6.

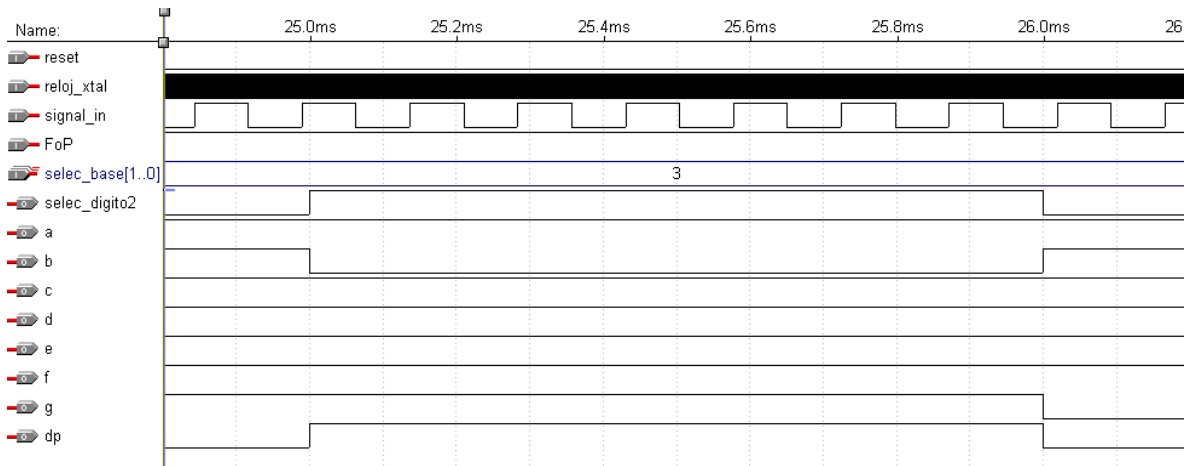


Figura 3.58: Simulación en el Editor de Formas de onda

y para el dígito 1:

valor del dígito	a	b	c	d	e	f	g	dp
8	1	1	1	1	1	1	1	0

Tabla 3.26: valor de las salidas a-dp para dígito con valor 8

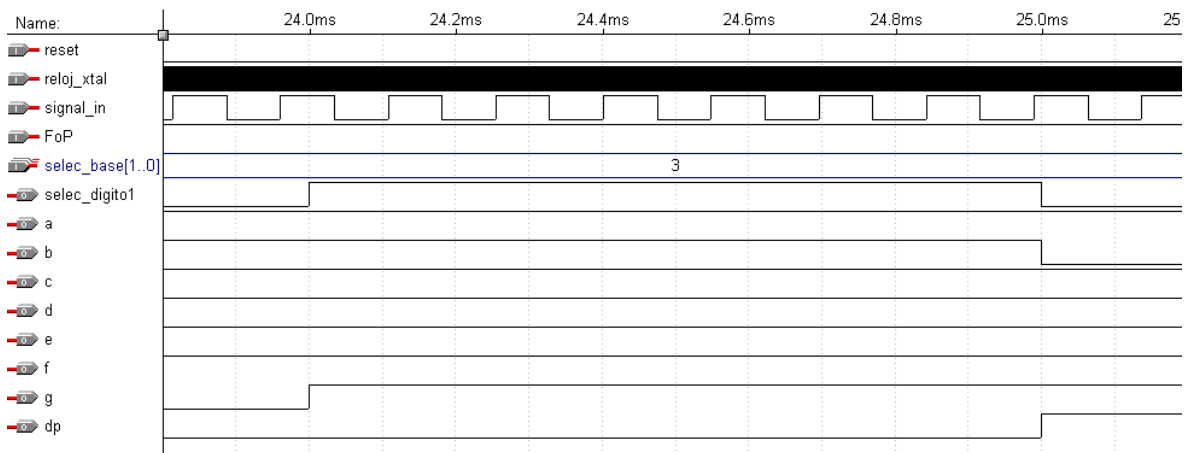


Figura 3.59: Simulación en el Editor de Formas de onda

3.4.2 Verificación del rango de frecuencias y períodos a medir

Como nombramos en los puntos anteriores, inicialmente nuestro medidor de frecuencias y períodos debe cumplir las siguientes especificaciones de rango:

Frecuencia: 1Hz a 100MHz.

Período: 100ns a 10seg

Es decir que mediante la correcta elección de la base de tiempos (o de la señal pulsos para el caso de períodos) debemos lograr medir señales dentro de estos rangos.

3.4.2.1 Rango de frecuencias

Recordemos inicialmente las distintas bases de tiempos generadas a partir de las entradas externas de selección `selec_base[1..0]`:

<code>selec_base[1..0]</code>	Duración del nivel alto de la base de tiempos
"00" = 0	10000 ms
"01" = 1	1000 ms
"10" = 2	100 ms
"11" = 3	10 ms

Tabla 3.1: Bases de tiempos

Luego vamos a agregarle los rangos de frecuencias medidos para cada una de las bases (sin tomar en cuenta los errores), recordando que:

$$f_{\text{señal}} = \frac{\text{N}^\circ \text{ de pulsos contados}}{\text{Tiempo de medición}}$$

donde:

Tiempo de medición: es el tiempo de duración en nivel alto de la base de tiempos.

Nº de pulsos contados: son la cantidad de pulsos a contar por el grupo de contadores.

Como tenemos un grupo de seis contadores conectados de forma que a cada uno contador le corresponde una potencia de 10, el rango de Nº de pulsos contados es de :

Nº pulsos contados: 1 – 999999 pulsos

Por lo tanto:

<code>selec_base[1..0]</code>	Duración del nivel alto de la base de tiempos	Rango de frecuencias a medir [Hz]
"00" = 0	10000 ms	0.1 a 99999.9
"01" = 1	1000 ms	1 a 999999
"10" = 2	100 ms	10 a 9999990
"11" = 3	10 ms	100 a 99999900

Tabla 3.27: Rango de frecuencias a medir para cada base de tiempos

Es decir que nuestro rango total de medición de frecuencias es de 0.1Hz a 99.9999MHz

3.4.2.2 Rango de períodos

Del mismo modo que para medición de frecuencias, repasemos las distintas señales de períodos definidos, generadas a partir de las entradas externas de selección `selec_base[1..0]`:

selec_base[1..0]	Período
"00" = 0	0.1 us
"01" = 1	1us
"10" = 2	10 us
"11" = 3	100 us

Tabla 3.9: Pulsos de períodos definidos

A continuación le agregaremos los rangos de períodos medidos para cada una de las señales (sin tomar en cuenta los errores), recordando que:

$$T_{señal} = N^{\circ} \text{ de pulsos contados} * T_{pulsos}$$

donde:

T_{PULSOS} : es el período de la señal de período preciso.

N° de pulsos contados: son la cantidad de pulsos a contar por el grupo de contadores.

Como mencionamos anteriormente, el rango de N° de pulsos contados es de :

N° pulsos contados: 1 – 999999 pulsos

Por lo tanto:

selec_base[1..0]	Período	Rango de períodos a medir [ms]
"00" = 0	0.1 us	0.0001 a 99.9999
"01" = 1	1us	0.001 a 999.999
"10" = 2	10 us	0.01 a 9999.99
"11" = 3	100 us	0.1 a 99999.9

Tabla 3.28: Rango de períodos a medir para cada señal de período definido

Es decir que nuestro rango total de medición de períodos es de 100ns a 99.9999 seg.

3.4.3 Calculo de errores

Para rangos de medición calculados en los puntos anteriores, no fueron tomados en cuenta los errores del medidor, los cuales podemos separar en dos tipos:

- Error de la base de tiempos o de la señal de período definido: este error esta dado pura y exclusivamente por la estabilidad del oscilador externo de 10MHz, el cual abordaremos en la parte de implementación de nuestro proyecto.
- Error en el número de pulsos contados: es el error del medidor implementado en AHDL que abordaremos en éste punto.

3.4.3.1 Error de N° de pulsos contados

Como ya hemos visto, los contadores cuentan dentro de un determinado tiempo de conteo (el cual está dado por la base de tiempos generada a partir del oscilador externo para medición de frecuencias, o de la división por 2 de la frecuencia de la señal de a medir para el caso de medición de períodos) de una dada cantidad de pulsos (que son los de la señal a medir para el caso de medición de frecuencias o los provistos por el generador de pulsos para el modo medición de períodos).

Como podemos intuir, estas dos señales en la gran mayoría de los casos no son sincrónicas, ya que una es generada por el oscilador de 10MHz y la otra a partir de una señal no conocida a medir, y es muy difícil que ambas estén en sincronismo.

Es por esto que en peor de los casos los contadores cuenten un pulso de más o bien uno de menos, con lo cual el error es de \pm el dígito menos significativo del display.

Los errores (en modo medición de frecuencias) para cada base de tiempos quedan de la forma:

selec_base[1..0]	Duración del nivel alto de la base de tiempos	Rango de frecuencias a medir [Hz]	error [Hz]
"00" = 0	10000 ms	0.1 a 99999.9	± 0.1
"01" = 1	1000 ms	1 a 999999	± 1
"10" = 2	100 ms	10 a 9999990	± 10
"11" = 3	10 ms	100 a 99999900	± 100

Tabla 3.29: Error para cada base de tiempos

Mientras que para cada señal de períodos definidos (modo medición de períodos)

selec_base[1..0]	Período	Rango de períodos a medir [ms]	error [us]
"00" = 0	0.1 us	0.0001 a 99.9999	± 0.1
"01" = 1	1us	0.001 a 999.999	± 1
"10" = 2	10 us	0.01 a 9999.99	± 10
"11" = 3	100 us	0.1 a 99999.9	± 100

Tabla 3.30: Error para cada señal de período definido

Es decir que como mínimo puedo medir una frecuencia con un error de $\pm 0.1\text{Hz}$ y un período con un error de $\pm 0.1\text{us} = \pm 100\text{ns}$.

Para que la medición sea medianamente correcta debemos tolerar un error que como máximo sea de un 10%, es decir que la mínima frecuencia a medir será de 1Hz y el mínimo período de 1us.

Como vemos la mínima frecuencia medible coincide con la propuesta por el rango de frecuencias, pero no así el mínimo período el cual es 10 veces mayor que el propuesto por el rango. Es por esto que deberemos implementar un prescaler que divida la frecuencia de la señal de entrada por 10, con lo cual mediremos a través del grupo de contadores, un período 10 veces mayor el original, y por lo tanto los errores con la activación de dicho prescaler para el

modo medición de períodos de la señal original serán 10 veces mas chicos, como muestra la tabla 3.31:

selec_base[1..0]	Período	Rango de períodos a medir [ms]	error [us]
"00" = 0	0.1 us	0.00001 a 9.99999	± 0.01
"01" = 1	1us	0.0001 a 99.9999	± 0.1
"10" = 2	10 us	0.001 a 999.999	± 1
"11" = 3	100 us	0.01 a 9999.99	± 10

Tabla 3.31: Error para cada señal de período definido con activación de prescaler

Mientras que para el modo medición de frecuencias, estaremos midiendo con el grupo de contadores una señal de frecuencia 10 veces menor que la propuesta, y por lo tanto los errores de frecuencias de la señal original serán 10 veces mayores:

selec_base[1..0]	Duración del nivel alto de la base de tiempos	Rango de frecuencias a medir [Hz]	error [Hz]
"00" = 0	10000 ms	1 a 999999	± 1
"01" = 1	1000 ms	10 a 9999990	± 10
"10" = 2	100 ms	100 a 99999900	± 100
"11" = 3	10 ms	1000 a 999999000	± 1000

Tabla 3.32: Error para cada base de tiempos con activación de prescaler

Luego los rangos de medición, con un error máximo del 10% quedan de la forma:

- Sin activación de prescaler
 - Frecuencia: 0.1Hz a 99.9999MHz
 - Período: 1us a 99.9999 seg.
- Con activación de prescaler
 - Frecuencia: 1Hz a 999.999MHz
 - Período: 100ns a 9.99999 seg.

Se puede apreciar que en modo medición de frecuencias con activación de prescaler, teóricamente podemos llegar a medir frecuencias de casi 1GHz, pero éste límite superior estará dado por la frecuencia máxima de trabajo de la FPGA a utilizar.

3.4.4 Implementación del prescaler en AHDL

En éste punto vamos a implementar el prescaler que divide la frecuencia de la señal a medir por 10. Para esto realizaremos algunas reformas al programa original *med_fre_per.tdf* y lo llamaremos *med_fre_per_presc.tdf*. Estas reformas serán inicialmente la adición de un prescaler que divida por 10 la frecuencia y luego la reubicación del punto dp en caso de activación de dicho prescaler. Además agregaremos una salida que se active cuando hay overflow

(desborde) de todos los contadores, es decir que nos permita darnos cuenta que debemos cambiar la base de tiempos (o la señal de período definido)

Primeramente vamos a definir el prescaler en AHDL a partir de un contador de 4 bits, que cuente hasta el valor 10; al cual llamaremos `prescaler_signal` y lo conectaremos como reloj la señal de entrada `signal_in`:

```
prescaler_signal: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
...
prescaler_signal.clock = signal_in;
```

Luego añadiremos una nueva señal de entrada llamada `prescaler_activo`, y un nodo llamado `reloj_in`. De acuerdo al valor de dicha entrada (`prescaler_activo`) el nodo adoptará como señal la de entrada o la de la salida `eq[9]` del prescaler (es decir la frecuencia de entrada dividido 10), de la forma que muestra la tabla 3.33:

<code>prescaler_activo</code>	<code>prescaler</code>	<code>reloj_in</code>
0	desactivado	<code>signal_in</code>
1	activado	<code>prescaler_signal.eq[9]</code>

Tabla 3.33: Activación de prescaler de acuerdo a `activa_prescaler`

Volcándolo en AHDL:

```
prescaler_activo                                     : INPUT;
...
reloj_in                                             : NODE;
...
IF prescaler_activo THEN                            % Si el prescaler esta activado %
    reloj_in = prescaler_signal.eq[9];              % La señal a medir es la %
                                                    % señal exterior/10 %
ELSE                                                 % Si el prescaler no esta activado %
    reloj_in = signal_in;                          % La señal a medir es la señal exterior %
END IF;
```

posteriormente éste nodo `reloj_in` es el que utilizaremos en lugar de `signal_in`, para el reloj de los contadores (en modo medición de frecuencia) o para la obtención de la base de tiempos (modo medición de períodos).

```
...
divisor.clock = reloj_in;
...
IF FoP THEN
    base_tiempos = salida.q;                        % Medición de %
    reloj = reloj_in;                              % Frecuencia %
ELSE
    base_tiempos = divisor.q[];                    % Medición de %
    reloj = pulsos;                               % Período %
END IF;
```

Por ultimo vamos a reubicar el punto `dp` en el caso de activación del prescaler. Si activamos el prescaler para el modo medición de frecuencias, los contadores estarán “midiendo” una señal de frecuencia 10 veces menor a la original, la cual será representada en el display. Es decir que para este caso deberemos correr el punto un lugar a la derecha con respecto al original (en el cual no había prescaler).

Si en cambio se activa en el modo medición de períodos, los contadores “medirán” y se representará en display un período 10 veces mayor que el original a medir; y por lo tanto deberemos correr el punto un lugar a la izquierda con respecto al modo sin prescaler.

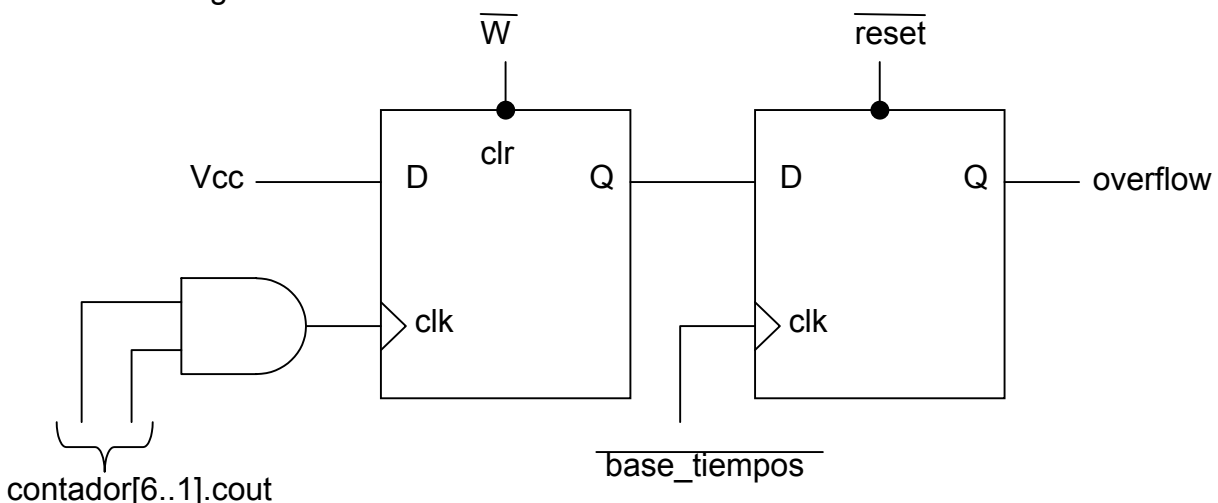
Es decir que el bloque de ubicación del punto dp en AHDL queda de la forma:

```

IF !prescaler_activo THEN                                     % Si no se activó prescaler %
  CASE selec_base[] IS
    WHEN B"00" =>
      dp = selec_digito[5];
    WHEN B"01" =>
      dp = selec_digito[4];
    WHEN B"10" =>
      dp = selec_digito[3];
    WHEN B"11" =>
      dp = selec_digito[2];
  END CASE;
ELSE                                                         % Si se activó prescaler %
  IF FoP THEN                                               % en modo medición Frecuencia %
    CASE selec_base[] IS                                     % corro punto a derecha %
      WHEN B"00" =>
        dp = selec_digito[5-1];
      WHEN B"01" =>
        dp = selec_digito[4-1];
      WHEN B"10" =>
        dp = selec_digito[3-1];
      WHEN B"11" =>
        dp = selec_digito[2-1];
    END CASE;
  ELSE                                                       % en modo medición Período %
    CASE selec_base[] IS                                     % corro punto a izquierda %
      WHEN B"00" =>
        dp = selec_digito[5+1];
      WHEN B"01" =>
        dp = selec_digito[4+1];
      WHEN B"10" =>
        dp = selec_digito[3+1];
      WHEN B"11" =>
        dp = selec_digito[2+1];
    END CASE;
  END IF;
END IF;
END IF;

```

Finalmente la señal de overflow la vamos a generar implementando el siguiente circuito lógico:



Al primer *flip flop* lo llamaremos `ffoverflow1` y al segundo `ffoverflow2`. Con esta configuración al generarse un carry en todos los contadores, inmediatamente la salida del 1º ff se pone en alto. Cuando la base de tiempos llega a su fin y pasa al nivel bajo, se copia el nivel alto en la salida del 2º ff y por lo tanto en la salida overflow. Luego al generarse la señal `w` de reseteo de contadores, ésta resetea el primer ff y lo deja listo para verificar si hay overflow en el próximo tiempo de conteo.

Implementándolo en AHDL:

```
overflow                                     : OUTPUT;
...
ffoverflow1, ffoverflow2                   : DFF;
...
ffoverflow1.clrn = !w;
ffoverflow1.d = VCC;
ffoverflow1.clk = contador[6].cout & contador[5].cout & contador[4].cout &
                contador[3].cout & contador[2].cout & contador[1].cout;
ffoverflow2.clrn = !reset;
ffoverflow2.d = ffoverflow1.q;
ffoverflow2.clk = !base_tiempos;
overflow = ffoverflow2.q;
```

Luego el programa en AHDL completo `med_fre_per_presc.tdf` queda de la forma:

```
constant MAX_COUNT = 10000;
constant N_DIGITOS = 6;                                     % De modo que %
constant POT_K = ceil(log2(N_DIGITOS));                   % 2^POT_K >= N_DIGITOS %

INCLUDE "lpm_counter.inc";
INCLUDE "lpm_latch.inc";
INCLUDE "lpm_mux.inc";
INCLUDE "lpm_decode.inc";

SUBDESIGN med_fre_per_presc
(
    reloj_xtal, reset, selec_base[1..0]                  : INPUT;
    signal_in, FoP, prescaler_activo                     : INPUT;
    a, b, c, d, e, f, g, dp                             : OUTPUT;
    overflow, selec_digito[N_DIGITOS..1]                 : OUTPUT;
)

VARIABLE

    cuenta: lpm_counter WITH (LPM_WIDTH=ceil(log2(MAX_COUNT)));
    prescaler1: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
    prescaler2: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
    prescaler3: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
    prescaler4: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
    prescaler_signal: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
    presc_sel_decodig: lpm_counter WITH ( LPM_WIDTH=POT_K,
                                         LPM_MODULUS=N_DIGITOS);
    divisor: lpm_counter WITH (LPM_WIDTH=1);
    contador[N_DIGITOS..1]: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
    latches[N_DIGITOS..1]: lpm_latch WITH (LPM_WIDTH=4);
    multiplexor: lpm_mux WITH (LPM_WIDTH=4, LPM_SIZE=N_DIGITOS,
                              LPM_WIDTHS=POT_K);
    decodificador: lpm_decode WITH (LPM_WIDTH=POT_K, LPM_DECODES=N_DIGITOS);
    ss: MACHINE OF BITS (z) WITH STATES (s0 = 0, s1 = 1, s2 = 0);

    salida, resetcont, ffoverflow1, ffoverflow2         : DFF;
```

```

borrar, reloj_1ms, clock10M, selec_decodig[POT_K-1..0]      : NODE;
base_tiempos, reloj, pulsos, reloj_in                     : NODE;
sal_cont[N_DIGITOS..1][3..0], qlatch[N_DIGITOS..1][3..0] : NODE;
sal_mux[3..0], bcd[3..0], nodo[N_DIGITOS..1], w          : NODE;

BEGIN
%   Comienzo etapa prescaler                               %
prescaler1.aclr = reset;
prescaler2.aclr = reset;
prescaler3.aclr = reset;
prescaler4.aclr = reset;
presc_sel_decodig.aclr = reset;
prescaler1.clock = reloj_xtal;
prescaler2.clock = reloj_xtal;
prescaler3.clock = reloj_xtal;
prescaler4.clock = reloj_xtal;
presc_sel_decodig.clock = reloj_xtal;
prescaler2.cnt_en = prescaler1.eq[9];
prescaler3.cnt_en = prescaler2.eq[9] & prescaler1.eq[9];
prescaler4.cnt_en = prescaler3.eq[9] & prescaler2.eq[9] &
                    prescaler1.eq[9];
presc_sel_decodig.cnt_en = prescaler4.eq[9] & prescaler3.eq[9] &
                           prescaler2.eq[9] & prescaler1.eq[9];

reloj_1ms = prescaler4.eq[9];
selec_decodig[POT_K-1..0] = presc_sel_decodig.q[];
clock10M = reloj_xtal;
%   Fin etapa prescaler                                   %
%   Base de tiempo para modo medición de frecuencias    %
salida.clrn = !reset;
salida.clk = reloj_1ms;
cuenta.clock = reloj_1ms;
cuenta.sclr = borrar;
CASE selec_base[] IS
    WHEN B"00" =>
        IF cuenta.q[] < 10000 THEN
            salida.d = VCC;
            borrar = GND;
        ELSE
            borrar = VCC;
            salida.d = GND;
        END IF;
    WHEN B"01" =>
        IF cuenta.q[] < 1000 THEN
            salida.d = VCC;
            borrar = GND;
        ELSE
            salida.d = GND;
            borrar = VCC;
        END IF;
    WHEN B"10" =>
        IF cuenta.q[] < 100 THEN
            salida.d = VCC;
            borrar = GND;
        ELSE
            borrar = VCC;
            salida.d = GND;
        END IF;
    WHEN B"11" =>
        IF cuenta.q[] < 10 THEN
            salida.d = VCC;
            borrar = GND;
        ELSE
            borrar = VCC;
            salida.d = GND;
        END IF;
END CASE;

```

```

% Pulsos para modo de medición de período %
CASE selec_base[] IS
    WHEN B"00" =>
        pulsos = reloj_xtal;
    WHEN B"01" =>
        pulsos = prescaler1.eq[9];
    WHEN B"10" =>
        pulsos = prescaler2.eq[9];
    WHEN B"11" =>
        pulsos = prescaler3.eq[9];
END CASE;

% Activación del prescaler que divide por 10 %
prescaler_signal.clock = signal_in;
IF prescaler_activo THEN % Si el prescaler esta activado %
    reloj_in = prescaler_signal.eq[9]; % La señal a medir es la %
    % señal exterior/10 %
ELSE % Si el prescaler no esta activado %
    reloj_in = signal_in; % La señal a medir es la señal exterior %
END IF;

% Base de tiempo para modo medición de períodos %
divisor.clock = reloj_in;

% Conexión de los nodos base_tiempos y reloj según se %
% mida frecuencia ó período %
IF FoP THEN
    base_tiempos = salida.q; % Medición de %
    reloj = reloj_in; % Frecuencia %
ELSE
    base_tiempos = divisor.q[]; % Medición de %
    reloj = pulsos; % Período %
END IF;

% Monoestable con maquina de estados %
ss.clk = clock10M;
ss.reset = reset;
TABLE
% estado entrada próximo %
% actual actual estado %
ss, base_tiempos => ss;

s0, 1 => s0;
s0, 0 => s1;
s1, 0 => s2;
s1, 1 => s0;
s2, 0 => s2;
s2, 1 => s0;
END TABLE;

% Decodificador BCD a 7 segmentos %
TABLE
bcd[3..0] => a, b, c, d, e, f, g;
H"0" => 1, 1, 1, 1, 1, 1, 0;
H"1" => 0, 1, 1, 0, 0, 0, 0;
H"2" => 1, 1, 0, 1, 1, 0, 1;
H"3" => 1, 1, 1, 1, 0, 0, 1;
H"4" => 0, 1, 1, 0, 0, 1, 1;
H"5" => 1, 0, 1, 1, 0, 1, 1;
H"6" => 1, 0, 1, 1, 1, 1, 1;
H"7" => 1, 1, 1, 0, 0, 0, 0;
H"8" => 1, 1, 1, 1, 1, 1, 1;
H"9" => 1, 1, 1, 1, 0, 1, 1;
END TABLE;

```

```

% Generación de la señal w para resetear el contador %
resetcont.clrn = !base_tiempos;
resetcont.d = VCC;
resetcont.clk = !z;
w = resetcont.q;

% Contadores, latches y multiplexor %
FOR i IN 1 TO N_DIGITOS GENERATE
    contador[i].clock = reloj;
    contador[i].aclr = w;
    latches[i].gate = z;
END GENERATE;
sal_cont[N_DIGITOS..1][3..0] = contador[N_DIGITOS..1].q[];
latches[N_DIGITOS..1].data[] = sal_cont[N_DIGITOS..1][];
qlatch[N_DIGITOS..1][] = latches[N_DIGITOS..1].q[];
multiplexor.data[N_DIGITOS-1..0][] = qlatch[N_DIGITOS..1][];

contador[1].cnt_en = base_tiempos;
nodo[1] = base_tiempos;
FOR n IN 2 TO N_DIGITOS GENERATE
    nodo[n] = nodo[n-1] & contador[n-1].eq[9];
    contador[n].cnt_en = nodo[n];
END GENERATE;

multiplexor.sel[] = selec_decodig[];
sal_mux[] = multiplexor.result[];

bcd[3..0] = sal_mux[];

% Decodificador %
decodificador.data[] = selec_decodig[];
selec_digito[N_DIGITOS..1] = decodificador.eq[N_DIGITOS-1..0];

% Generación de señal overflow %
ffoverflow1.clrn = !w;
ffoverflow1.d = VCC;
ffoverflow1.clk = contador[6].cout & contador[5].cout & contador[4].cout
    & contador[3].cout & contador[2].cout & contador[1].cout;
ffoverflow2.clrn = !reset;
ffoverflow2.d = ffoverflow1.q;
ffoverflow2.clk = !base_tiempos;
overflow = ffoverflow2.q;

% Ubicación del punto (dp) de acuerdo a la base y modo utilizado %
% solo para el caso que N_DIGITOS >= 5 %
IF !prescaler_activo THEN % Si no se activó prescaler %
    CASE selec_base[] IS
        WHEN B"00" =>
            dp = selec_digito[5];
        WHEN B"01" =>
            dp = selec_digito[4];
        WHEN B"10" =>
            dp = selec_digito[3];
        WHEN B"11" =>
            dp = selec_digito[2];
    END CASE;
ELSE % Si se activó prescaler %
    IF FoP THEN % en modo medición Frecuencia %
        CASE selec_base[] IS % corro punto a derecha %
            WHEN B"00" =>
                dp = selec_digito[5-1];
            WHEN B"01" =>
                dp = selec_digito[4-1];
            WHEN B"10" =>
                dp = selec_digito[3-1];
            WHEN B"11" =>
                dp = selec_digito[2-1];
        END CASE;
    END IF;
END IF;

```

```

ELSE
CASE selec_base[] IS
  WHEN B"00" =>
    dp = selec_digito[5+1];
  WHEN B"01" =>
    dp = selec_digito[4+1];
  WHEN B"10" =>
    dp = selec_digito[3+1];
  WHEN B"11" =>
    dp = selec_digito[2+1];
END CASE;
END IF;
END IF;
END;

```

3.4.5 Comunicación del medidor de Frecuencias y períodos con PC

Ahora vamos a realizar un control y transferencia de datos de nuestro medidor de frecuencias y períodos con la PC a través del puerto paralelo en modo SPP (*Standard Parallel Port*).

La idea es de poder controlar el Medidor de forma externa a través de entradas por pulsadores e interruptores y salida al display y de forma con comunicación a PC, donde podamos liberarnos del control externo y displays; y manejar los modos, bases, prescalers, y visualización de los resultados en la computadora. Para esto crearemos un nuevo programa en AHDL, al que llamaremos *med_fre_per_presc_PC.tdf*, el cual se basará en el programa *med_fre_per_presc.tdf* con el agregado de las nuevas sentencias para comunicación con PC.

3.4.5.1 Bits del puerto paralelo para comunicación

En el modo SPP tomaremos los 8 bits del puerto de datos (D7..D0) como salida de la PC, y cuatro del puerto de estados (S6..S3) como entrada.

El diseño de la comunicación se basa en la siguiente tabla de asignación de los pines del puerto paralelo:

pin	puerto	sentido	Función
D7	datos	salida	activa_prescaler
D6	datos	salida	FoP
D5	datos	salida	selec_base[1]
D4	datos	salida	selec_base[0]
D3	datos	salida	congela latches
(D2..D0)	datos	salida	selección dígito
(S6..S3)	estado	entrada	overflow y valor dígito

Tabla 3.34: Asignación de pines de puerto paralelo

Como vemos los pines (D7..D4) se encargan de comandar las señales de activación de prescaler, elegir en modo frecuencia o período y seleccionar la base (o señal de pulsos de período definido) a utilizar.

Por su parte los pines (D2..D0) comandan que valor debe la FPGA colocar en los pines (S6..S3) de forma como muestra la tabla 3.35:

(D2..D0)	(S6..S3)
"000" = 0	Overflow
"001" = 1	valor dígito 1
"010" = 2	valor dígito 2
"011" = 3	valor dígito 3
"100" = 4	valor dígito 4
"101" = 5	valor dígito 5
"110" = 6	valor dígito 6

Tabla 3.35: Valores de las entradas (S6..S3) de acuerdo a las salidas (D2..D0).

El bit D3 por su parte, se encarga de congelar los latches para la correcta carga de los valores de todos los dígitos (es decir que no se produzca una carga de latches en el momento de la lectura por parte de la PC, la cual genere una carga de algunos dígitos de una medición de los contadores y otros de otra).

Por lo tanto la secuencia será:

1. A través de los pines (D7..D4), comandamos las señales de activación de prescaler, modo frecuencia o período y selección base en forma continua.
2. Se activa la carga de valores mediante el bit D3 para congelar latches.
3. Se genera en (D2..D0) la secuencia de 0 a 6 dentro de la cual se van cargando los valores de overflow y de los distintos dígitos.
4. Se desactiva la carga de valores para permitir el refresco de los latches
5. Se repiten los puntos 2 - 5 en la medida necesaria

Ésta secuencia se puede generar en un programa, como por ejemplo Visual Basic, para comandar los valores de los pines de salida y obtener visualmente los valores de los pines de entrada.

3.4.5.2 Programa en AHDL

Debemos ahora generar el programa en AHDL que permita que nuestro medidor responda en forma correcta a las señales como muestran las tablas 3.34 y 3.35.

Para esto inicialmente debemos considerar una nueva entrada, a la que llamaremos `manual_PC`, la cual nos permita elegir entre el modo de utilización del Medidor en forma manual, externa a través de interruptores y pulsadores (valor 1) o con control mediante PC (valor 0). Luego las entradas llamadas `selec_base[1..0]`, `FoP` y `prescaler_activo` pasarán ser nodos y estas entradas externas se llamarán `sel_base_ext[1..0]`, `Frec_o_Period` y `activa_prescaler` respectivamente. A su vez agregaremos las nuevas entradas del puerto paralelo `status_port[6..3]` y `data_port[7..0]`.

```
Frec_o_Period, manual_PC, activa_prescaler, sel_base_ext[1..0] : INPUT;
data_port[7..0] : INPUT;
status_port[6..3] : OUTPUT;
```



```

...
selec_base[1..0], FoP, prescaler_activo           : NODE;
...

CASE manual_PC IS
  WHEN B"0" =>                                     %      Modo PC      %
    prescaler_activo = data_port[7];
    FoP = data_port[6];
    selec_base[] = data_port[5..4];
  WHEN B"1" =>                                     %      Modo manual  %
    prescaler_activo = activa_prescaler;
    FoP = Frec_o_Period;
    selec_base[] = sel_base_ext[1..0];
END CASE;

```

Ahora deberemos generar la señal que congela los prescalers. Para esto tomaremos la señal que sale del monoestable como $z1$; la señal z que carga los latches pasara a ser un nodo cuyo valor estará dado por la salida de una compuerta *and* que tiene como entradas la señal $z1$ (salida del monoestable) y /D3, de modo que cuando se active el bit D3 ('1'), z valga siempre '0', y cuando D3 sea '0', z adquiera el valor de $z1$.

```

ss: MACHINE OF BITS (z1) WITH STATES    (s0 = 0, s1 = 1, s2 = 0);
...
z                                     : NODE;
...
z = z1 & !data_port[3];

```

cave aquí aclarar que en el caso de no tener el puerto conectado, la entrada D3 ($data_port[3]$) quedaría flotante y puede que no se carguen los latches con la periodicidad correcta, es por eso que conviene aquí poner una resistencia de *pull-down* a fin de no congelar los latches en el caso de modo manual sin conexión del puerto.

Finalmente vamos a crear una sentencia para la correcta asignación de los valores de (S6..S3) de acuerdo a los valores de (D2..D0) como requiere la tabla 3.35:

```

CASE data_port[2..0] IS
  WHEN 0 =>
    status_port[3] = overflow;
  WHEN 1 =>
    status_port[6..3] = qlatch[1][];
  WHEN 2 =>
    status_port[6..3] = qlatch[2][];
  WHEN 3 =>
    status_port[6..3] = qlatch[3][];
  WHEN 4 =>
    status_port[6..3] = qlatch[4][];
  WHEN 5 =>
    status_port[6..3] = qlatch[5][];
  WHEN 6 =>
    status_port[6..3] = qlatch[6][];
END CASE;

```

El programa en AHDL completo *med_fre_per_presc_PC.tdf* queda de la forma:

```

constant MAX_COUNT = 10000;
constant N_DIGITOS = 6;                                     % De modo que %
constant POT_K = ceil(log2(N_DIGITOS));                   % 2^POT_K >= N_DIGITOS %

INCLUDE "lpm_counter.inc";
INCLUDE "lpm_latch.inc";
INCLUDE "lpm_mux.inc";
INCLUDE "lpm_decode.inc";
SUBDESIGN med_fre_per_presc_PC
(
    reloj_xtal, reset, sel_base_ext[1..0], manual_PC      : INPUT;
    signal_in, Frec_o_Period, activa_prescaler, data_port[7..0] : INPUT;
    a, b, c, d, e, f, g, dp, status_port[6..3]           : OUTPUT;
    overflow, selec_digito[N_DIGITOS..1]                 : OUTPUT;
)
VARIABLE

    cuenta: lpm_counter WITH (LPM_WIDTH=ceil(log2(MAX_COUNT)));
    prescaler1: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
    prescaler2: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
    prescaler3: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
    prescaler4: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
    prescaler_signal: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
    presc_sel_decodig: lpm_counter WITH ( LPM_WIDTH=POT_K,
                                         LPM_MODULUS=N_DIGITOS);
    divisor: lpm_counter WITH (LPM_WIDTH=1);
    contador[N_DIGITOS..1]: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
    latches[N_DIGITOS..1]: lpm_latch WITH (LPM_WIDTH=4);
    multiplexor: lpm_mux WITH (LPM_WIDTH=4, LPM_SIZE=N_DIGITOS,
                              LPM_WIDTHS=POT_K);
    decodificador: lpm_decode WITH (LPM_WIDTH=POT_K, LPM_DECODES=N_DIGITOS);
    ss: MACHINE OF BITS (z1) WITH STATES (s0 = 0, s1 = 1, s2 = 0);

    salida, resetcont, ffoverflow1, ffoverflow2          : DFF;

    borrar, reloj_lms, clock10M, selec_decodig[POT_K-1..0] : NODE;
    base_tiempos, reloj, pulsos, reloj_in                : NODE;
    sal_cont[N_DIGITOS..1][3..0], qlatch[N_DIGITOS..1][3..0] : NODE;
    sal_mux[3..0], bcd[3..0], nodo[N_DIGITOS..1], w       : NODE;
    selec_base[1..0], FoP, prescaler_activo, z            : NODE;

BEGIN
% Comienzo etapa prescaler %
prescaler1.aclr = reset;
prescaler2.aclr = reset;
prescaler3.aclr = reset;
prescaler4.aclr = reset;
presc_sel_decodig.aclr = reset;
prescaler1.clock = reloj_xtal;
prescaler2.clock = reloj_xtal;
prescaler3.clock = reloj_xtal;
prescaler4.clock = reloj_xtal;
presc_sel_decodig.clock = reloj_xtal;
prescaler2.cnt_en = prescaler1.eq[9];
prescaler3.cnt_en = prescaler2.eq[9] & prescaler1.eq[9];
prescaler4.cnt_en = prescaler3.eq[9] & prescaler2.eq[9] &
                    prescaler1.eq[9];
presc_sel_decodig.cnt_en = prescaler4.eq[9] & prescaler3.eq[9] &
                           prescaler2.eq[9] & prescaler1.eq[9];

reloj_lms = prescaler4.eq[9];
selec_decodig[POT_K-1..0] = presc_sel_decodig.q[];
clock10M = reloj_xtal;
% Fin etapa prescaler %

```

```

% Selección entre modo manual o PC %
CASE manual_PC IS
    WHEN B"0" => % Modo PC %
        prescaler_activo = data_port[7];
        FoP = data_port[6];
        selec_base[] = data_port[5..4];
    WHEN B"1" => % Modo manual %
        prescaler_activo = activa_prescaler;
        FoP = Frec_o_Period;
        selec_base[] = sel_base_ext[1..0];
END CASE;

% Base de tiempo para modo medición de frecuencias %
salida.clrn = !reset;
salida.clk = reloj_1ms;
cuenta.clock = reloj_1ms;
cuenta.sclr = borrar;
CASE selec_base[] IS
    WHEN B"00" =>
        IF cuenta.q[] < 10000 THEN
            salida.d = VCC;
            borrar = GND;
        ELSE
            borrar = VCC;
            salida.d = GND;
        END IF;
    WHEN B"01" =>
        IF cuenta.q[] < 1000 THEN
            salida.d = VCC;
            borrar = GND;
        ELSE
            salida.d = GND;
            borrar = VCC;
        END IF;
    WHEN B"10" =>
        IF cuenta.q[] < 100 THEN
            salida.d = VCC;
            borrar = GND;
        ELSE
            borrar = VCC;
            salida.d = GND;
        END IF;
    WHEN B"11" =>
        IF cuenta.q[] < 10 THEN
            salida.d = VCC;
            borrar = GND;
        ELSE
            borrar = VCC;
            salida.d = GND;
        END IF;
END CASE;

% Pulsos para modo de medición de período %
CASE selec_base[] IS
    WHEN B"00" =>
        pulsos = reloj_xtal;
    WHEN B"01" =>
        pulsos = prescaler1.eq[9];
    WHEN B"10" =>
        pulsos = prescaler2.eq[9];
    WHEN B"11" =>
        pulsos = prescaler3.eq[9];
END CASE;

% Activación del prescaler que divide por 10 %
prescaler_signal.clock = signal_in;

```

```

IF prescaler_activo THEN          % Si el prescaler esta activado %
    reloj_in = prescaler_signal.eq[9];      % La señal a medir es la %
                                           % señal exterior/10 %
ELSE                                % Si el prescaler no esta activado %
    reloj_in = signal_in; % La señal a medir es la señal exterior %
END IF;

% Base de tiempo para modo medición de períodos %
divisor.clock = reloj_in;

% Conexión de los nodos base_tiempos y reloj según se %
% mida frecuencia ó período %
IF FoP THEN
    base_tiempos = salida.q;          % Medición de %
    reloj = reloj_in;                % Frecuencia %
ELSE
    base_tiempos = divisor.q[];      % Medición de %
    reloj = pulsos;                  % Período %
END IF;

% Monoestable con maquina de estados %
ss.clk = clock10M;
ss.reset = reset;
TABLE
% estado entrada próximo %
% actual actual estado %
    ss, base_tiempos => ss;

    s0, 1 => s0;
    s0, 0 => s1;
    s1, 0 => s2;
    s1, 1 => s0;
    s2, 0 => s2;
    s2, 1 => s0;
END TABLE;

%Inhabilitación de la señal z, para congelar latches, cuando leo por puerto%
z = z1 & !data_port[3];

% Decodificador BCD a 7 segmentos %
TABLE
    bcd[3..0] => a, b, c, d, e, f, g;
    H"0" => 1, 1, 1, 1, 1, 1, 0;
    H"1" => 0, 1, 1, 0, 0, 0, 0;
    H"2" => 1, 1, 0, 1, 1, 0, 1;
    H"3" => 1, 1, 1, 1, 0, 0, 1;
    H"4" => 0, 1, 1, 0, 0, 1, 1;
    H"5" => 1, 0, 1, 1, 0, 1, 1;
    H"6" => 1, 0, 1, 1, 1, 1, 1;
    H"7" => 1, 1, 1, 0, 0, 0, 0;
    H"8" => 1, 1, 1, 1, 1, 1, 1;
    H"9" => 1, 1, 1, 1, 0, 1, 1;
END TABLE;

% Generación de la señal w para resetear el contador %
resetcont.clrn = !base_tiempos;
resetcont.d = VCC;
resetcont.clk = !z1;
w = resetcont.q;

% Contadores, latches y multiplexor %
FOR i IN 1 TO N_DIGITOS GENERATE
    contador[i].clock = reloj;
    contador[i].aclr = w;
    latches[i].gate = z;
END GENERATE;
sal_cont[N_DIGITOS..1][3..0] = contador[N_DIGITOS..1].q[];
latches[N_DIGITOS..1].data[] = sal_cont[N_DIGITOS..1][];

```

```

qlatch[N_DIGITOS..1][] = latches[N_DIGITOS..1].q[];
multiplexor.data[N_DIGITOS-1..0] [] = qlatch[N_DIGITOS..1][];

contador[1].cnt_en = base_tiempos;
nodo[1] = base_tiempos;
FOR n IN 2 TO N_DIGITOS GENERATE
    nodo[n] = nodo[n-1] & contador[n-1].eq[9];
    contador[n].cnt_en = nodo[n];
END GENERATE;

multiplexor.sel[] = selec_decodig[];
sal_mux[] = multiplexor.result[];

bcd[3..0] = sal_mux[];

% Decodificador %
decodificador.data[] = selec_decodig[];
selec_digito[N_DIGITOS..1] = decodificador.eq[N_DIGITOS-1..0];

% Generación de señal overflow %
ffoverflow1.clrn = !w;
ffoverflow1.d = VCC;
ffoverflow1.clk = contador[6].cout & contador[5].cout & contador[4].cout
    & contador[3].cout & contador[2].cout & contador[1].cout;
ffoverflow2.clrn = !reset;
ffoverflow2.d = ffoverflow1.q;
ffoverflow2.clk = !base_tiempos;
overflow = ffoverflow2.q;

% Ubicación del punto (dp) de acuerdo a la base y modo utilizado %
% solo para el caso que N_DIGITOS >= 5 %
IF !prescaler_activo THEN % Si no se activó prescaler %
    CASE selec_base[] IS
        WHEN B"00" =>
            dp = selec_digito[5];
        WHEN B"01" =>
            dp = selec_digito[4];
        WHEN B"10" =>
            dp = selec_digito[3];
        WHEN B"11" =>
            dp = selec_digito[2];
    END CASE;
ELSE % Si se activó prescaler %
    IF FoP THEN % en modo medición Frecuencia %
        CASE selec_base[] IS % corro punto a derecha %
            WHEN B"00" =>
                dp = selec_digito[5-1];
            WHEN B"01" =>
                dp = selec_digito[4-1];
            WHEN B"10" =>
                dp = selec_digito[3-1];
            WHEN B"11" =>
                dp = selec_digito[2-1];
        END CASE;
    ELSE % en modo medición Período %
        CASE selec_base[] IS % corro punto a izquierda %
            WHEN B"00" =>
                dp = selec_digito[5+1];
            WHEN B"01" =>
                dp = selec_digito[4+1];
            WHEN B"10" =>
                dp = selec_digito[3+1];
            WHEN B"11" =>
                dp = selec_digito[2+1];
        END CASE;
    END IF;
END IF;

```

```
% Salida hacia el puerto de estado (status_port) según la entrada por el puerto de datos (data_port)
CASE data_port[2..0] IS
  WHEN 0 =>
    status_port[3] = overflow;
  WHEN 1 =>
    status_port[6..3] = qlatch[1][];
  WHEN 2 =>
    status_port[6..3] = qlatch[2][];
  WHEN 3 =>
    status_port[6..3] = qlatch[3][];
  WHEN 4 =>
    status_port[6..3] = qlatch[4][];
  WHEN 5 =>
    status_port[6..3] = qlatch[5][];
  WHEN 6 =>
    status_port[6..3] = qlatch[6][];
END CASE;

END;
```

3.5 Adquisidor de datos autónomo

3.5.1 Arquitectura

La idea aquí es controlar un convertor analógico digital (el ADC0820 para nuestro caso) y almacenar las muestras obtenidas de modo que puedan ser utilizadas por una PC. Si para lograr esto utilizamos el convertor comandado directamente por el puerto paralelo de la PC, tendremos una limitación de frecuencia de muestreo dada por la velocidad máxima de dicho puerto. Es por esto que utilizamos una FPGA, de modo que emule una memoria RAM del tipo FIFO (*First In First Out*). Mediante ésta lógica programable nos permitimos controlar el convertor, almacenar una determinada cantidad de muestras del mismo y luego transferirlas a la PC mediante el puerto paralelo en modo SPP. La velocidad del convertor ya no es problema puesto que con éstas FPGAs podemos rondar las decenas de MHz de frecuencia.

Por su parte el convertor ADC0820 es un convertor analógico digital de 8 bits, con un tiempo de conversión mínimo de 1.5us (frecuencia de muestreo máxima de 650Khz aproximadamente) en el modo WR-RD. Es decir que nuestra FPGA actuará como una memoria FIFO de 8 bits, con una determinada cantidad de muestras y con una transmisión de datos a PC mediante 5 bits, que son los que soporta el puerto en modo SPP como entradas. Cabe aclarar que la cantidad de bits de la memoria, así como la cantidad de muestras a cargar dependen de la capacidad de implementación de RAM que tenga el dispositivo FPGA elegido para la implementación, los valores aquí tomados (principalmente el de cantidad de muestras) son a modo de ejemplo.

La figura 3.60 muestra un diagrama de la comunicación entre la FPGA, el convertor y los pines de entrada y de salida que se conectaran al puerto paralelo de la PC:

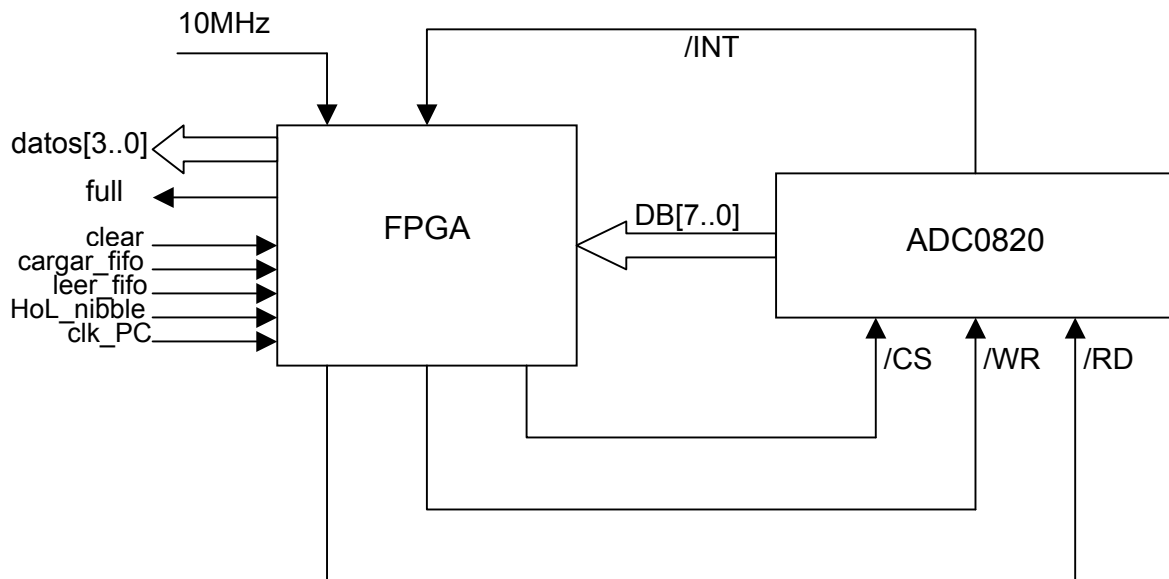


Figura 3.60: Diagrama adquisidor autónomo de datos

Como vemos los pines de entrada a la FPGA provenientes del puerto son:

clear: limpia el contenido de la FIFO.

cargar_fifo: habilita la carga de la FIFO con las muestras del conversor.

leer_fifo: habilita la lectura de la FIFO por parte de la PC.

HoL_nibble: como la transferencia de los datos a PC es en modo SPP, solo se hará a través de palabras de 4 bits (la memoria es de 8 bits), con lo cual mediante esta entrada seleccionaremos secuencialmente los 4 bits mas y menos significativos de la muestra.

clk_PC: es el reloj que comandará el clock de la FIFO cuando se estén transfiriendo datos a la PC.

Los pines de salida de la FPGA al puerto:

full: indica a la PC que la memoria está llena.

datos[3..0]: transfiere los cuatro bits mas o menos significativos de la muestra.

Además tenemos una entrada en la FPGA proveniente de un oscilador externo (10MHz).

Para comunicar la FPGA con el conversor, nos valemos de los pines:

/WR: (modo WR-RD) cuando */CS* está en bajo, la conversión comienza con el flanco descendente de */WR*.

/RD: (modo WR-RD) cuando */CS* está en bajo, las salidas de datos *DB[7..0]* se activan cuando */RD* pasa al nivel bajo.

/INT: (modo WR-RD) cuando */INT* pasa al nivel alto indica que la conversión se completó y que los datos se encuentran cargados en los latches. */INT* es reseteado con el flanco ascendente de */RD* o de */CS*.

/CS: cuando está en bajo reconoce las entradas */WR* y */RD*.

DB[7..0]: salida de datos de bits 7, 6, 5, 4, 3, 2, 1 y 0 de tres estados.

La figura 3.61 muestra un diagrama de tiempos de las señales de los pines del conversor ADC0820 descritos anteriormente en el modo WR-RD (el cual es el que utilizaremos)

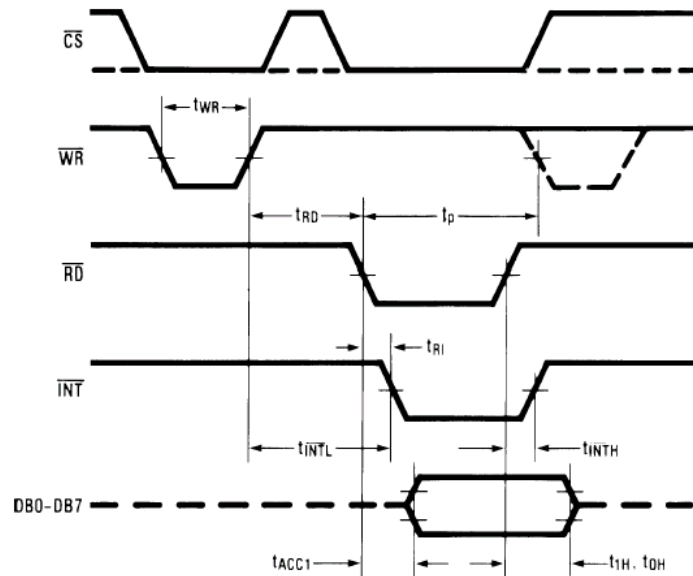


Figura 3.61: Diagrama de tiempos del ADC0820 en modo WR-RD

donde:

- t_{WR} : tiempo de escritura.
- t_{RD} : tiempo de lectura.
- t_{INTL} : retardo entre el flanco ascendente de /WR y el descendente de /INT
- t_{INTH} : retardo entre el flanco ascendente de /RD y el ascendente de /INT.
- t_p : retardo entre fin de conversión y la próxima conversión.
- t_{RI} : retardo entre /RD e /INT.
- t_{ACC1} : tiempo de acceso (retardo entre el flanco descendente de /RD y la aparición de los datos válidos en DB[7..0]).
- t_{1H}, t_{0H} : retardo entre el flanco ascendente de /RD y la puesta en TRI-STATE de las salidas.

3.5.2 Implementación de la memoria FIFO en AHDL

Vamos ahora a implementar la memoria *First In First Out* en el lenguaje AHDL, utilizando la función parametrizada `lpm_fifo`. Mediante la ayuda podemos ver en las tablas 3.36 y 3.37 sus puertos de entrada y salida respectivamente:

Nombre del puerto	Requerido?	Descripción	Comentario
<code>data[]</code>	Si	Entrada de datos a la <code>lpm_fifo</code>	Puerto de Entrada de ancho <code>LPM_WIDTH</code>
<code>wrreq</code>	Si	Control de petición de escritura. El puerto <code>data[]</code> es escrito en la <code>lpm_fifo</code>	La escritura es deshabilitada si <code>full = 1</code> .
<code>rdreq</code>	Si	Control de petición de lectura. El dato mas viejo de	La lectura es deshabilitada si

clock	Si	la lpm_fifo va al puerto q[].	empty = 1.
Aclr	No	Reloj de activación por flanco ascendente	
Sclr	No	Entrada de Clear asincrónico. Resetea la lpm_fifo a empty.	
		Entrada de Clear sincrónico. Resetea la lpm_fifo a empty.	

Tabla 3.36: Puertos de entrada

Nombre del puerto	Requerido?	Descripción	Comentario
q[]	Si	Salida de datos de la lpm_fifo.	Puerto de Salida de ancho LPM_WIDTH.
full	No	Indica que la lpm_fifo está llena y deshabilita el puerto wrreq.	Es ratificado cuando usedw[] = LPM_NUMWORDS
Empty	No	Indica que la lpm_fifo está vacía y deshabilita el puerto rdreq.	Es ratificado cuando usedw[] = 0.
Usedw[]	No	Número de palabras que hay actualmente en la lpm_fifo	Puerto de Salida de ancho [LPM_WIDTH-1..0].

Tabla 3.37: Puertos de salida

Y en la tabla 3.38 sus parámetros:

Parámetro	Tipo	requerido?	Descripción
LPM_WIDTH	Entero	Si	El ancho de los puertos q[] y data[].
LPM_NUMWORDS	Entero	Si	Profundidad de la lpm_fifo. Numero de palabras que son almacenadas en la memoria, cuyo valor es usualmente una potencia de 2.
LPM_WIDTHU	Entero	Si	El valor recomendado es CEIL(LOG2(LPM_NUMWORDS)). Ancho del puerto usedw[].
LPM_SHOWAHEAD	Cadena	No	Permite a los datos aparecer inmediatamente en q[] sin esperar a rdreq, cuando se encuentra en "ON". Los valores son "ON" y "OFF", por default el valor es "OFF".

Tabla 3.38: Parámetros de la FIFO

Vamos ahora a generar la memoria en lenguaje AHDL mediante la función parametrizada `lpm_fifo`, para esto tomaremos a modo de ejemplo una memoria de en la cual almacenemos 5 muestras:

```

constant N_MUESTRAS = 5;                                % De modo que %
constant POT_M = ceil(log2(N_MUESTRAS));                % 2^POT_M >= N_MUESTRAS %
...
fifo: lpm_fifo WITH (LPM_WIDTH=8, LPM_NUMWORDS=N_MUESTRAS, LPM_WIDTHU=POT_M);
...

```

crearemos los nodos llamados de la misma forma que los pines de entrada y salida del puerto nombrados anteriormente:

```

datos[3..0], full, clear, cargar_fifo                    : NODE;
leer_fifo, HoL_nibble, clk_PC                           : NODE;

```

y los pines de entrada y salida de la FPGA:

```

reloj_xtal, /INT, DB[7..0], data_port[4..0]            : INPUT;
/CS, /RD, /WR, status_port[7..3]                       : OUTPUT;

```

A continuación vamos a conectar los nodos con los puertos de entrada y salida de la FIFO:

```

% Los pines de entrada %
fifo.rdreq = leer_fifo;
fifo.aclr = clear;

% Los pines de salida %
/CS = !cargar_fifo;
full = fifo.full;

```

Seguidamente vamos a generar las señales $/RD$ y $/WR$ de modo de cumplir con el diagrama de tiempos de la figura 3.61. Aquí no buscaremos la frecuencia máxima de muestreo, sino que tomaremos una frecuencia menor para corroborar que la FPGA comanda de forma adecuada al convertor. Luego en la implementación trataremos de acercarnos a la máxima frecuencia del ADC. De la hoja de datos del convertor tenemos que:

	máximo	mínimo
t_{WR}	50us	600ns
t_{RD}		600ns
t_P		500ns
t_{INTH}		225ns

Tabla 3.39: Tiempos máx. y min. del ADC0820

Tomaremos entonces los siguientes valores

$$\begin{aligned}
 t_{WR} &= 800\text{ns} \\
 t_{RD} &= 800\text{ns} \\
 t_P &= 800\text{ns} \\
 t_{INTH} &= 300\text{ns}
 \end{aligned}$$

Por lo tanto generaremos las siguientes señales, tal como muestra la figura 3.62:

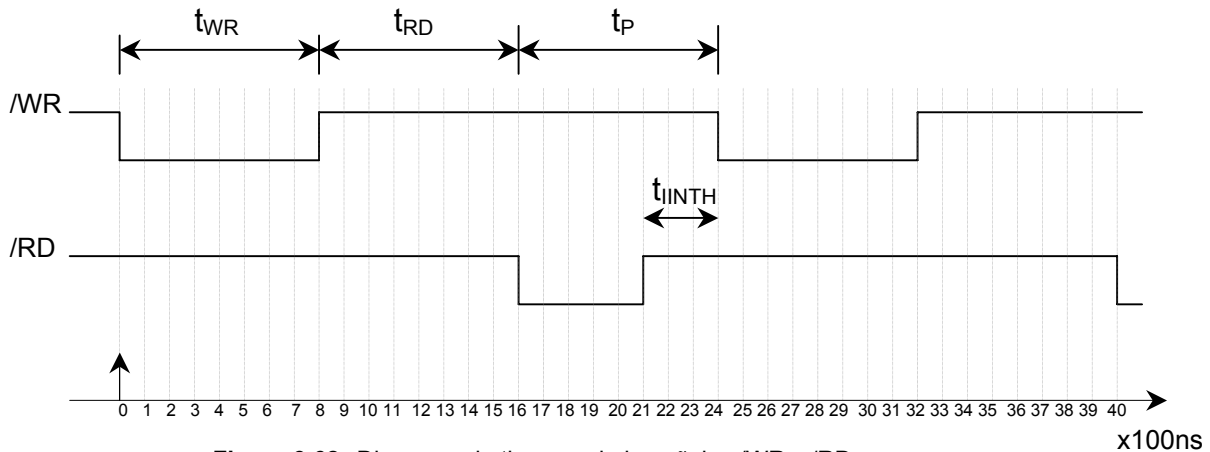


Figura 3.62: Diagrama de tiempos de la señales /WR y /RD a generar

En AHDL:

```

constant MAX_CONT = 24;
...
contador: lpm_counter WITH ( LPM_WIDTH= ceil(log2(MAX_CONT)),
                             LPM_MODULUS=MAX_CONT);
...
salidard, salidawr                                     : DFF;
...
contador.aclr = clear;
contador.clock = reloj_xtal;
salidawr.clnr = !clear;
salidard.clnr = !clear;
salidawr.clk = reloj_xtal;
salidard.clk = reloj_xtal;

IF contador.q[] < 8 THEN
    salidawr.d = GND;
    salidard.d = VCC;
ELSE IF contador.q[] < 16 THEN
    salidawr.d = VCC;
    salidard.d = VCC;
ELSE IF contador.q[] < 21 THEN
    salidawr.d = VCC;
    salidard.d = GND;
ELSE IF contador.q[] < 24 THEN
    salidawr.d = VCC;
    salidard.d = VCC;
ELSE
    salidawr.d = GND;
    salidard.d = VCC;
END IF;
END IF;
END IF;
END IF;

/WR = salidawr.q;
/RD = salidard.q;

```

Para la señal de escritura en la memoria ($wrreq$) utilizaremos la señal /INT negada proveniente del conversor, ya que en ese instante tenemos el dato en los latches del ADC (ver figura 3.61):

```
fifo.wrreq = !/INT;
```

Luego debemos seleccionar las señales de reloj de la memoria se acuerdo a si estamos escribiendo en ella o leyéndola:

```
IF cargar_fifo THEN                                % Si está cargando datos del ADC           %
    fifo.clock = salidard.q;                       % El reloj será el generado por el contador %
ELSE                                                % Si va a cargar datos en PC               %
    fifo.clock = clk_PC;                           % El reloj será el generado por la PC      %
END IF;
```

Como podemos ver cuando cargamos la memoria, el reloj para la FIFO es la misma señal que /RD, (ver figura 3.61).

A continuación, mediante un multiplexor vamos a elegir entre los cuatro bits mas o menos significativos de la palabra de 8 bits de acuerdo a la entrada HoL_nibble, para enviar hacia la PC; de modo que se cumpla la tabla 3.40:

nibble	HoL_nibble	datos[3..0]
High	0	q[7..4]
Low	1	q[3..0]

Tabla 3.40: Valor de datos[3..0] de acuerdo a entrada HoL_nibble

en AHDL:

```
multiplexor: lpm_mux WITH (LPM_WIDTH=4, LPM_SIZE=2, LPM_WIDTHS=1);
...
multiplexor.data[0][] = fifo.q[7..4];
multiplexor.data[1][] = fifo.q[3..0];
multiplexor.sel[] = HoL_nibble;
datos[] = multiplexor.result[];
```

Finalmente vamos a conectar los nodos con los pines del puerto paralelo de modo que quede como muestra la tabla 3.41:

pin	puerto	sentido	nodo
D0	datos	salida	clear
D1	datos	salida	cargar_fifo
D2	datos	salida	leer_fifo
D3	datos	salida	HoL_nibble
D4	datos	salida	clk_PC
S7	estado	entrada	full
(S6..S3)	estado	entrada	datos[3..0]

Tabla 3.41: Asignación de pines de puerto paralelo

en AHDL:

```
clear = data_port[0];
cargar_fifo = data_port[1];
leer_fifo = data_port[2];
HoL_nibble = data_port[3];
clk_PC = data_port[4];
status_port[7] = full;
status_port[6..3] = datos[3..0];
```

La secuencia que deberá generar el programa de la PC que comande el sistema FPGA - conversor (que puede ser por ejemplo un programa en Visual Basic) será:

1. Puesta en '1' del bit D0 para limpiar la memoria.
2. Se activa la carga de la memoria mediante el bit D1 hasta que el bit S7 se ponga en '1' (memoria llena).
3. Se activa la lectura de la memoria mediante el bit D2.
4. Con el bit D3 en bajo, se genera un pulso de reloj en el bit D4 para cargar los cuatro bits mas altos de la muestra a través de los bits (S6..S3).
5. Se pone el bit D3 en alto y se cargan los cuatro bits menos significativos de la muestra a través de los bits (S6..S3).
6. Se repiten los puntos 4 - 5 hasta cargar en la PC, todas las muestras que estaban almacenadas en la memoria.

Luego el programa en AHDL completo al que llamaremos *adquisidor.tdf* queda de la forma:

```
constant MAX_CONT = 24;
constant N_MUESTRAS = 500;
constant POT_M = ceil(log2(N_MUESTRAS));

INCLUDE "lpm_fifo.inc";
INCLUDE "lpm_counter.inc";
INCLUDE "lpm_mux.inc";

SUBDESIGN adquisidor
(
    reloj_xtal, /INT, DB[7..0], data_port[4..0] : INPUT;
    /CS, /RD, /WR, status_port[7..3] : OUTPUT;
)

VARIABLE

    fifo: lpm_fifo WITH (LPM_WIDTH=8, LPM_NUMWORDS=N_MUESTRAS,
                        LPM_WIDTHU=POT_M);
    contador: lpm_counter WITH ( LPM_WIDTH= ceil(log2(MAX_CONT)),
                                LPM_MODULUS=MAX_CONT);
    multiplexor: lpm_mux WITH (LPM_WIDTH=4, LPM_SIZE=2, LPM_WIDTHS=1);

    salidard, salidawr : DFF;

    datos[3..0], full, clear, cargar_fifo : NODE;
    leer_fifo, HoL_nibble, clk_PC : NODE;

BEGIN

% Conexión del puerto con los nodos %
clear = data_port[0];
cargar_fifo = data_port[1];
leer_fifo = data_port[2];
HoL_nibble = data_port[3];
clk_PC = data_port[4];
status_port[7] = full;
status_port[6..3] = datos[3..0];

% Obtención de las salidas /RD y /WR %
contador.aclr = clear;
contador.clock = reloj_xtal;
salidawr.clrn = !clear;
salidard.clrn = !clear;
```

```

salidawr.clk = reloj_xtal;
salidard.clk = reloj_xtal;

IF contador.q[] < 8 THEN
    salidawr.d = GND;
    salidard.d = VCC;
ELSE IF contador.q[] < 16 THEN
    salidawr.d = VCC;
    salidard.d = VCC;
ELSE IF contador.q[] < 21 THEN
    salidawr.d = VCC;
    salidard.d = GND;
ELSE IF contador.q[] < 24 THEN
    salidawr.d = VCC;
    salidard.d = VCC;
ELSE
    salidawr.d = GND;
    salidard.d = VCC;
END IF;
END IF;
END IF;

/WR = salidawr.q;
/RD = salidard.q;

% Los pines de entrada %
fifo.rdreq = leer_fifo;
fifo.aclr = clear;
fifo.wrreq = !/INT;
fifo.data[] = DB[];

% Los pines de salida %
/CS = !cargar_fifo;
full = fifo.full;

% De acuerdo a el modo en que esté trabajando la FIFO %
IF cargar_fifo THEN % Si está cargando datos del ADC %
    fifo.clock = salidard.q; % Reloj generado por el contador %
ELSE % Si va a cargar datos en PC %
    fifo.clock = clk_PC; % Reloj generado por la PC %
END IF;

% El multiplexor divide las datos de salida para cargar por %
% puerto "datos[3..0]" %
% cuando HoL_nibble = 0 => datos[3..0] = q[7..4] High Nibble %
% HoL_nibble = 1 => datos[3..0] = q[3..0] Low Nibble %
multiplexor.data[0][] = fifo.q[7..4];
multiplexor.data[1][] = fifo.q[3..0];
multiplexor.sel[] = HoL_nibble;
datos[] = multiplexor.result[];

END;

```

3.5.3 Simulación del Adquisidor

Vamos ahora a realizar una simulación funcional para verificar el correcto funcionamiento de nuestro sistema. Para esto simularemos las entradas /INT y DB[7..0] provenientes del conversor de modo que cumplan el diagrama de tiempos de la figura 3.61.

Inicialmente simulamos solo fijando la entrada del oscilador de 10MHz tal como muestra la figura 3.63:

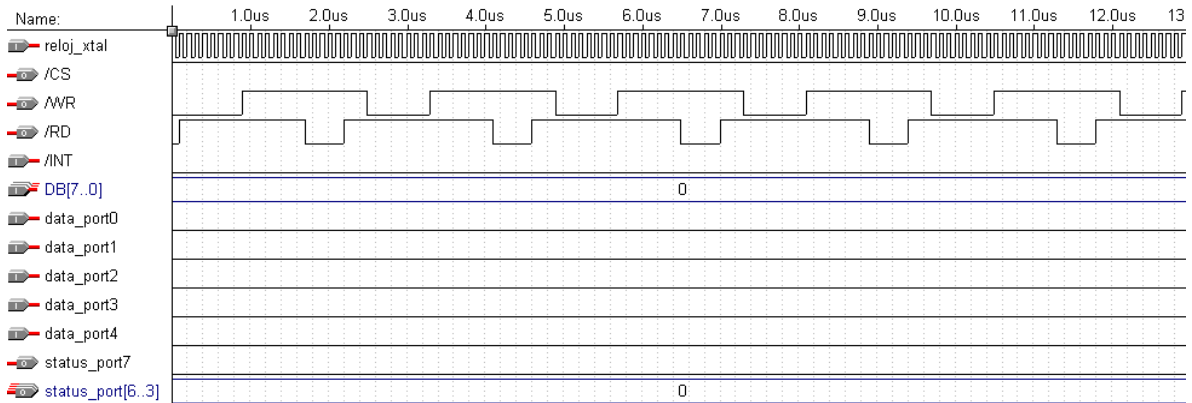


Figura 3.63: Simulación en el Editor de Formas de onda.

A partir de aquí generaremos las señales /INT y DB[7..0] cumpliendo con el diagrama de tiempos de la figura 3.61, para la toma de las 5 primeras muestras, dando como valores de dichas muestras las que muestra la tabla 3.42:

muestra nº	DB[7..0]	high nibble	low nibble
1	“00001101” = 13	“0000” = 0	“1101” = 13
2	“00010010” = 18	“0001” = 1	“0010” = 2
3	“10010101” = 149	“1001” = 9	“0101” = 5
4	“01100100” = 100	“0110” = 6	“0100” = 4
5	“10000111” = 135	“1000” = 8	“0111” = 7

Tabla 3.42: Valores de las 5 muestras supuestas del convertor

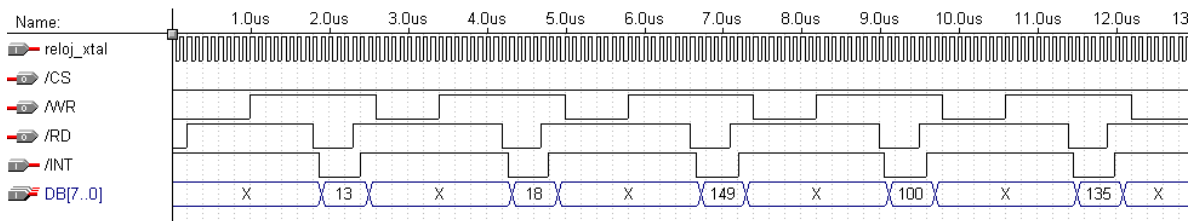


Figura 3.64: Simulación en el Editor de Formas de onda

Ahora procederemos a cargar la memoria, la cual se debería llenar con las 5 muestras (`status_port[7] = full = 1`):

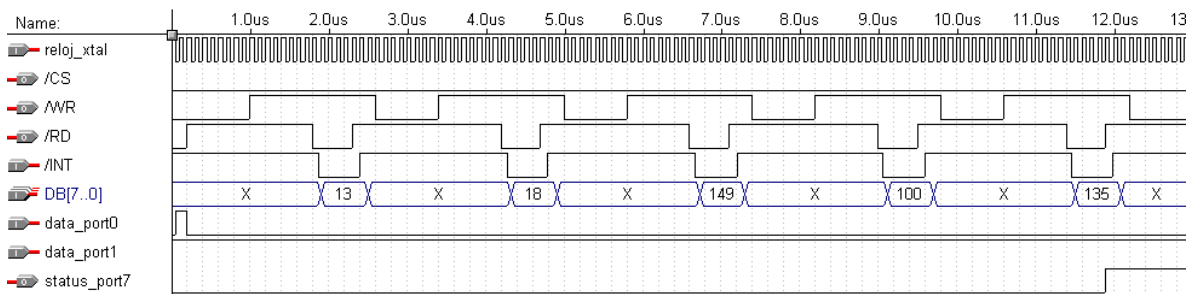


Figura 3.65: Simulación en el Editor de Formas de onda

Finalmente vamos a leer los datos de la memoria:

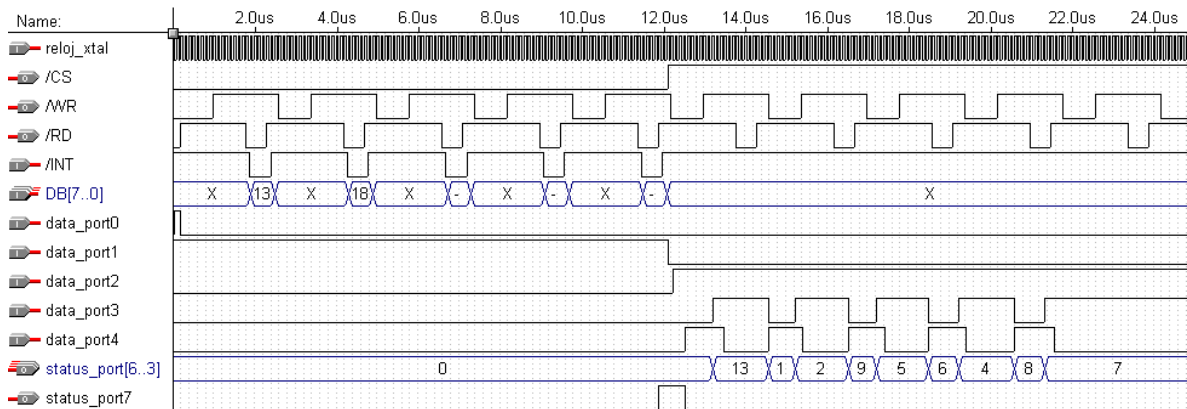


Figura 3.66: Simulación en el Editor de Formas de onda

Efectuando un zoom en la zona de lectura de la memoria:

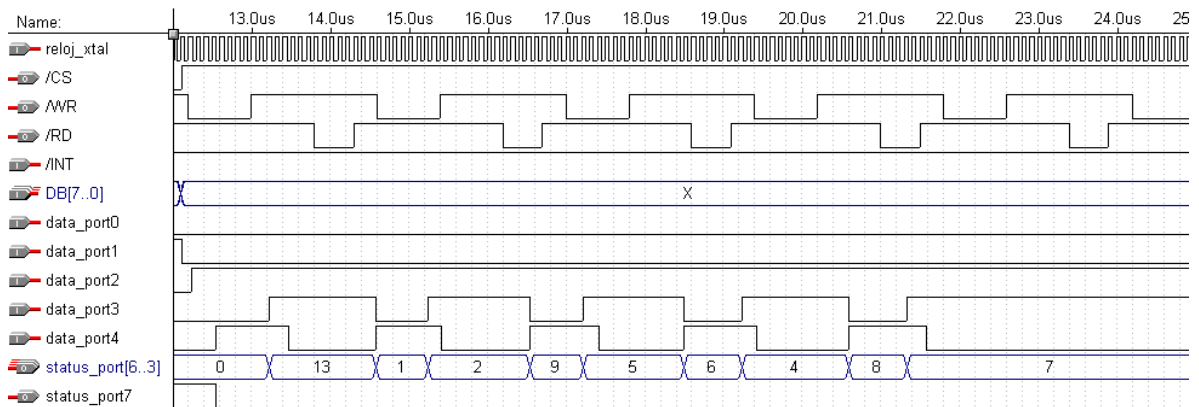


Figura 3.67: Simulación en el Editor de Formas de onda

Podemos ver que los valores que se cargaran por puerto (`status_port[6..3]`) de acuerdo a la elección del high o low nibble por medio de la entrada `data_port[3]` son los esperados según se pueden constatar con la tabla 3.42.

Capítulo 4 Implementación

4.1 Introducción

En éste capítulo vamos a implementar los proyectos del Medidor de frecuencia y período y el Adquisidor de datos autónomo en un único proyecto el cual implementaremos mediante lenguaje AHDL en una FPGA. El dispositivo elegido es el EPF10K10LC84-3 de la familia FLEX10K de Altera. La elección de dicho dispositivo se basa en que la cátedra de *Introducción a los Sistemas Lógicos y Digitales* (para la cual está destinado éste proyecto) posee una plaqueta experimental Upx10K10 apta para desarrollos con las FPGA EPF10K10LC84; y además es el dispositivo que mejor se ajusta a nuestra implementación (lo cual no es coincidencia, ya que en el desarrollo del proyecto se tuvo en cuenta que éste debía ser implementado en la FPGA FLEX10K10). Luego diseñaremos las plaquetas que se conectarán a la UPx10K10, es decir la implementación del hardware de nuestro proyecto, con sus respectivos programas en Visual Basic para que, una vez comunicado el dispositivo con la PC a través del puerto paralelo en modo SPP, podamos realizar una interfase física de la PC con el usuario (aunque en el caso del Medidor de frecuencias y períodos, esta interfaz también se podrá realizar a través de la plaqueta).

4.2 Dispositivo EPF10K10LC84-3 y plaqueta Upx10K10

4.2.1 Dispositivo EPF10K10LC84-3

Como vimos anteriormente los dispositivos *FLEX10K* son dispositivos lógicos programable basados en tablas de *look-up* con programación tipo SRAM. El dispositivo EPF10K10LC84-3 opera a 5V y posee 576 elementos lógicos (LEs) agrupados internamente en LABs de 8 LEs cada uno, con 3 bloques (EABs) de 2Kbits de memoria RAM interna cada uno y una matriz de ruteado global de alta velocidad (*Fast Track*), de 3 filas con 144 canales por fila y 24 columnas con 24 canales por columna, que permite la interconexión entre los LEs y las celdas de entrada/salida (I/O elements).

La FPGA utilizada se encuentra encapsulada en un PLCC (*Plastic Leadless Chip Carrier*) de 84 patas, que es para el que esta diseñada la plaqueta experimental.



Figura 4.1: Dispositivo EPF10K10LC84-3 de Altera

4.2.2 Paqueta experimental Upx10K10

La Upx10K10 es una plaqueta experimental diseñada por el Ingeniero Guillermo Jaquenod, cuya finalidad es ser utilizada en tareas de enseñanza de lógica programable. Conjuntamente con el software para PC MAX+Plus II de Altera, provee los recursos necesarios par crear y verificar diseños digitales de complejidad media.

Las características principales de la Upx10K10 son:

- En forma de kit para armar.
- Apta para desarrollos con la FPGA EPF10K10LC84 de Altera, en encapsulado PLCC de 84 patas.
- Plaqueta doble faz de dimensiones reducidas.
- Con un regulador de tensión incorporado de 1 Ampere (U5).
- Con un generador de clock (U3) incorporado de 8MHz (que puede ser habilitado o no)
- Con dos conectores de expansión (CON1 y CON2) que permiten tener acceso a las patas del chip dentro de la plaqueta.
- Con un *ByteBlaster* incorporado, que sirve tanto para configurar la FPGA como para el test, mediante BST (*Boundary Scan Testing*) de dispositivos accesibles a través de los conectores de expansión.
- Con una memoria EPC2 (U1), para programar al dispositivo a través de memoria EPROM (dispositivo SRAM).
- Con conexión directa a la PC a través de un cable de impresora paralela estándar tipo CENTRONICS (J3).
- Con una serie de jumpers (J1 y J2) que permiten bypasear la EPC2; realizar expansiones a través de los conectores CON1 y CON2, para programar mas de una plaqueta; y conectar o no la salida del oscilador local de 8MHz (U3) a la entrada de reloj global (GCLK0, pin 1) de la EPF10K10LC84 y al pin 1 de los conectores de expansión.

La figura 4.2 muestra la distribución de los componentes en la Upx10K10:

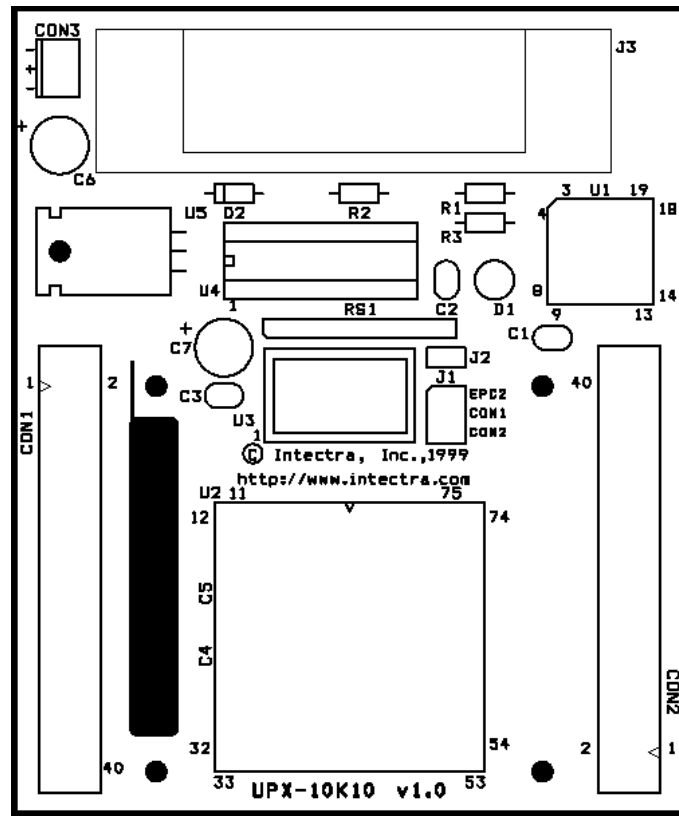


Figura 4.2: Distribución de los componentes en la UPx10K10

La tabla 4.1 muestra la equivalencia de los pines de los conectores de expansión, con los pines del EPF10K10LC84.

CON1	Conectado con	CON2	Conectado con
1	MCLK*: EPF10K10#1	1	MCLK*: EPF10K10#1
2	IO0*: EPF10K10#16	2	IO18: EPF10K10#47
3	TDO (#1) de la EPC2	3	TDO (#18) de CON1
4	IO1* : EPF10K10#17	4	IO19: EPF10K10#48
5	GND*	5	GND*
6	GND*	6	GND*
7	IO2: EPF10K10#18	7	IO20: EPF10K10#49
8	IO3: EPF10K10#19	8	IO21: EPF10K10#50
9	IO4: EPF10K10#21	9	IO22: EPF10K10#51
10	IO5: EPF10K10#22	10	IO23: EPF10K10#52
11	DEDIN0: EPF10K10#2	11	DEDIN2: EPF10K10#44
12	DEDIN1: EPF10K10#42	12	DEDIN3: EPF10K10#84
13	IO6: EPF10K10#23	13	IO24: EPF10K10#53
14	GCLEAR*: EPF10K10#3	14	GCLEAR*: EPF10K10#3
15	IO7: EPF10K10#24	15	IO25: EPF10K10#54
16	IO8: EPF10K10#25	16	IO26: EPF10K10#58
17	IO9: EPF10K10#27	17	IO27: EPF10K10#59
18	TDI (#3) de CON2	18	TDO del ByteBlaster
19	IO10: EPF10K10#28	19	IO28: EPF10K10#60
20	IO11: EPF10K10#29	20	IO29: EPF10K10#61
21	VCC*	21	VCC*
22	VCC*	22	VCC*
23	IO12: EPF10K10#30	23	IO30: EPF10K10#62

24	IO13: EPF10K10#35	24	IO31: EPF10K10#64
25	IO14: EPF10K10#36	25	IO32: EPF10K10#65
26	TCK*: EPF10K10#77	26	TCK*: EPF10K10#77
27	IO15: EPF10K10#37	27	IO33: EPF10K10#66
28	IO16: EPF10K10#38	28	IO34: EPF10K10#67
29	IO17: EPF10K10#39	29	IO35*: EPF10K10#71
30	IO37: EPF10K10#5	30	IO36*: EPF10K10#72
31	IO38: EPF10K10#6	31	IO45: EPF10K10#70
32	IO39: EPF10K10#7	32	IO46: EPF10K10#73
33	IO40: EPF10K10#8	33	IO47: EPF10K10#78
34	IO41: EPF10K10#9	34	IO48: EPF10K10#79
35	IO42: EPF10K10#10	35	IO49: EPF10K10#80
36	IO43: EPF10K10#11	36	IO50: EPF10K10#81
37	IO44: EPF10K10#69	37	IO51: EPF10K10#83
38	IO35*: EPF10K10#71	38	IO00*: EPF10K10#16
39	IO36*: EPF10K10#72	39	IO01*: EPF10K10#17
40	TMS*: EPF10K10#57	40	TMS*: EPF10K10#57

Tabla 4.1: Equivalencia de los pines de CON1 y CON2 con los de la EPF10K10LC84

Los pines marcados con (*) están disponibles en ambos conectores

Debido al rango de períodos a medir por nuestro Medidor de períodos, en un momento debemos generar una señal de pulsos de período preciso de valor 0.1us (10MHz), por lo tanto debimos cambiar el oscilador de 8MHz por uno de 10MHz, con lo cual la entrada de oscilador externo a la FPGA es la prevista en el capítulo anterior.

La figura 4.3 muestra a la plaqueta utilizada.

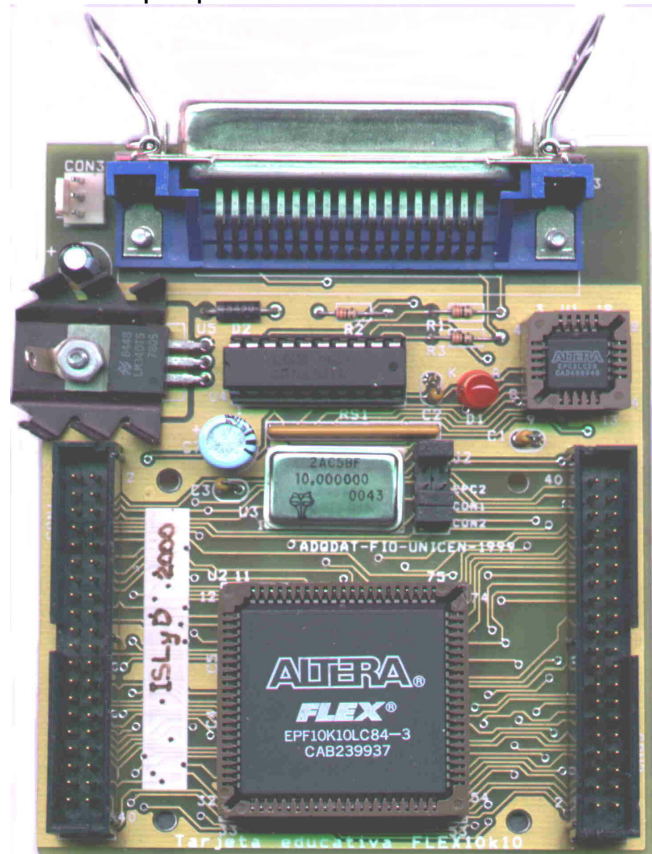


Figura 4.3: Plaqueta UPx10K10 utilizada.

4.3 Programa inicial en AHDL del proyecto Medidor de frecuencias y períodos + Adquisidor de datos autónomo

Vamos ahora a generar el programa en AHDL que comprenda en un solo proyecto a los subproyectos Medidor de frecuencias y períodos y al Adquisidor autónomo de datos, basándonos en los programas *med_fre_per_presc_PC.tdf* y *adquisidor.tdf* del capítulo anterior.

4.3.1 Pines comunes a ambos subproyectos

Éste nuevo programa tendrá como entradas y salidas comunes al Medidor y Adquisidor, aquellas que se conecten el puerto paralelo de la computadora (*data_port[7..0]* y *status_port[7..3]*). Luego mediante un nuevo pin de entrada llamado *Frec_o_Adqui* seleccionaremos entre cual de los dos subproyectos estamos ejecutando de forma como muestra la tabla 4.2:

Frec_o_Adqui	subproyecto
0	Adquisidor de datos autónomo
1	Medidor de frecuencias y períodos

Tabla 4.2: Modo de ejecución de acuerdo a *Frec_o_Adqui*

Es por esto que en AHDL crearemos nodos correspondientes a los pines de comunicación con puerto de cada uno de los proyectos y el nuevo pin de entrada:

```
Frec_o_Adqui           : INPUT;
...
data_port_F[7..0], status_port_F[6..3] : NODE;
data_port_A[4..0], status_port_A[7..3]  : NODE;
```

Luego para cumplir con la tabla 2:

```
CASE Frec_o_Adqui IS
  WHEN B"0" =>
    %                               Modo Adquisidor           %
    data_port_A[4..0] = data_port[4..0];
    status_port[7..3] = status_port_A[7..3];
  WHEN B"1" =>
    %                               Modo Frecuencímetro       %
    data_port_F[7..0] = data_port[7..0];
    status_port[6..3] = status_port_F[6..3];
END CASE;
```

4.3.2 Modificaciones al Medidor de frecuencias y períodos

4.3.2.1 Pin de selección de base

Como podemos recordar, el medidor de frecuencias y períodos tiene para la selección externa de una de las cuatro bases de tiempos, una entrada llamada `sel_base_ext[1..0]`, la cual es de 2 bits.

Para simplificar el hardware de manejo del Medidor, es conveniente utilizar un pulsador para cambio de base. Este pulsador (`pulsador_selbase`) comandará un contador de 2 bits (`contadorbase`) interno, el cual para cada pulso incrementará el ahora nodo `sel_base_ext[1..0]`.

```
pulsador_selbase           : INPUT;
...
contadorbase : lpm_counter WITH ( LPM_WIDTH=2, LPM_MODULUS=4);
...
sel_base_ext    : NODE;
...
contadorbase.clock = pulsador_selbase;
sel_base_ext[1..0] = contadorbase.q[];
```

4.3.2.2 Circuito anti-rebote para el pulsador de selección de base

Al utilizar un pulsador externo, en el momento del accionamiento del mismo, el contacto llega al punto de cierre y rebota varias veces hasta que llega al nivel deseado. Estos rebotes producen una ráfaga de pulsos al accionar el mismo, mientras que nosotros queremos un solo pulso por cada accionamiento.

Para salvar este inconveniente vamos a generar en la FPGA un circuito lógico, cuya salida sea `pulsos_selbase` y su entrada `pulsador_selbase`; y responda como muestra el diagrama de tiempos de la figura 4.4 :

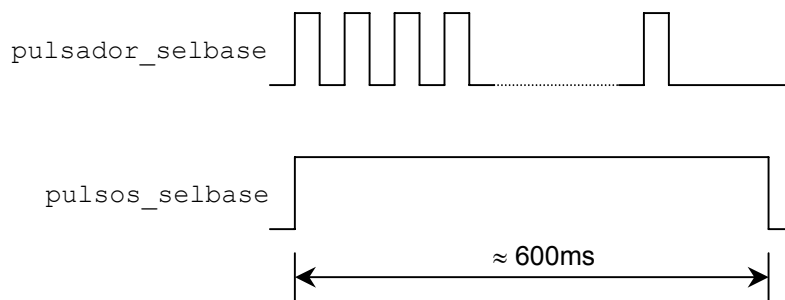


Figura 4.4: Diagrama de tiempos del anti-rebote.

Es decir que al activar el pulsador, se generará un único pulso en `pulsos_selbase` cuya duración es de aproximadamente 0.6seg. De ésta forma se ignorarán todos los rebotes que se produzcan dentro de éste tiempo que es mayor al del tren de pulsos generado por el pulsador.

Para lograr esto implementaremos el siguiente circuito lógico:

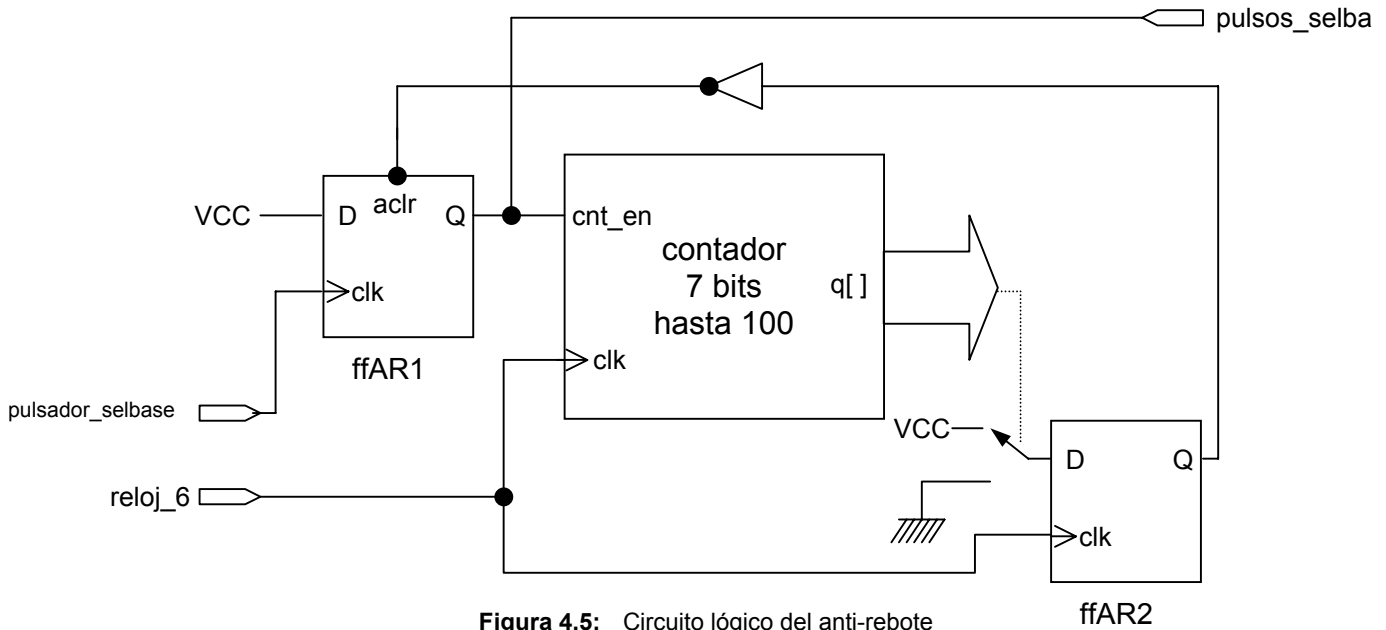


Figura 4.5: Circuito lógico del anti-rebote

La señal proveniente del pulsador (`pulsador_selbase`) activa el primer *flip flop* llamado `ffAR1`. Éste copia la señal `VCC` a su salida y por lo tanto pone en alto el pin de salida `pulsos_selbase` y habilita al contador. Dicho contador al llegar a 100 (o sea 600ms, ya que su señal de reloj es de 6ms de período) y mediante una lógica, pone en `VCC` la entrada del segundo *flip flop* (`ffAR2`), el cual permite eliminar los *glitches* provenientes de la salida `q[]` del contador. Éste segundo ff resetea al primer ff, deshabilitando al contador y poniendo en bajo la salida `pulsos_selbase`.

Volcando éste circuito en lenguaje AHDL, queda de la forma:

```
cont_anti_rebote: lpm_counter WITH (LPM_WIDTH=7, LPM_MODULUS=100);
...
ffar1, ffar2                                     : DFF;
...
pulsos_selbase, reloj_6ms                         : NODE;
...
reloj_6ms = presc_sel_decodig.eq[5];
...
ffar1.d = VCC;
ffar1.clk = pulsador_selbase;
ffar2.clk = reloj_6ms;
cont_anti_rebote.clock = reloj_6ms;
cont_anti_rebote.cnt_en = ffar1.q;
IF cont_anti_rebote.q[] == 99 THEN
    ffar2.d = VCC;
ELSE
    ffar2.d = GND;
END IF;
ffar1.cln = !ffar2.q;
pulsos_selbase = ffar1.q;
...
contadorbase.clock = pulsos_selbase;
```

4.3.3 Programa en AHDL

El programa completo en AHDL que incluya al Medidor de frecuencias y períodos (*med_fre_per_presc_PC.tdf*), con el pulsador para selección de base y el circuito antirebote; y al Adquisidor autónomo de datos (*adquisidor.tdf*) tomando 500 muestras; lo llamaremos *proyecto.tdf* y queda de la forma:

```

constant MAX_COUNT = 10000;
constant N_DIGITOS = 6;                                     % De modo que %
constant POT_K = ceil(log2(N_DIGITOS));                    % 2^POT_K >= N_DIGITOS %
constant MAX_CONT = 24;
constant N_MUESTRAS = 500;                                % De modo que %
constant POT_M = ceil(log2(N_MUESTRAS));                  % 2^POT_M >= N_MUESTRAS %

INCLUDE "lpm_counter.inc";
INCLUDE "lpm_latch.inc";
INCLUDE "lpm_mux.inc";
INCLUDE "lpm_decode.inc";
INCLUDE "lpm_fifo.inc";

SUBDESIGN proyecto
(
    reloj_xtal, reset, manual_PC, pulsador_selbase          : INPUT;
    signal_in, Frec_o_Period, activa_prescaler              : INPUT;
    Frec_o_Adqui, /INT, DB[7..0], data_port[7..0]          : INPUT;
    a, b, c, d, e, f, g, dp                                : OUTPUT;
    overflow, selec_digito[N_DIGITOS..1]                   : OUTPUT;
    /CS, /RD, /WR, status_port[7..3]                       : OUTPUT;
)
VARIABLE

cuenta1: lpm_counter WITH (LPM_WIDTH=ceil(log2(MAX_COUNT)));
cuenta2: lpm_counter WITH ( LPM_WIDTH= ceil(log2(MAX_CONT)),
                           LPM_MODULUS=MAX_CONT);
cont_anti_rebote: lpm_counter WITH (LPM_WIDTH=7, LPM_MODULUS=100);
contadorbase : lpm_counter WITH ( LPM_WIDTH=2, LPM_MODULUS=4);
prescaler1: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
prescaler2: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
prescaler3: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
prescaler4: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
prescaler_signal: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
presc_sel_decodig: lpm_counter WITH ( LPM_WIDTH=POT_K,
                                      LPM_MODULUS=N_DIGITOS);
divisor: lpm_counter WITH (LPM_WIDTH=1);
contador[N_DIGITOS..1]: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
latches[N_DIGITOS..1]: lpm_latch WITH (LPM_WIDTH=4);
multiplexor: lpm_mux WITH (LPM_WIDTH=4, LPM_SIZE=N_DIGITOS,
                           LPM_WIDTHS=POT_K);
mux_HoL_nibble: lpm_mux WITH (LPM_WIDTH=4, LPM_SIZE=2, LPM_WIDTHS=1);
decodificador: lpm_decode WITH (LPM_WIDTH=POT_K, LPM_DECODES=N_DIGITOS);
fifo: lpm_fifo WITH (LPM_WIDTH=8, LPM_NUMWORDS=N_MUESTRAS,
                    LPM_WIDTHU=POT_M);
ss: MACHINE OF BITS (z1) WITH STATES (s0 = 0, s1 = 1, s2 = 0);

salida, salidard, salidawr, resetcont                      : DFF;
ffar1, ffar2, ffoverflow1, ffoverflow2                   : DFF;

borrar, reloj_lms, clock10M, selec_decodig[POT_K-1..0]    : NODE;
base_tiempos, reloj, pulsos, reloj_in                     : NODE;
sal_cont[N_DIGITOS..1][3..0], qlatch[N_DIGITOS..1][3..0] : NODE;
sal_mux[3..0], bcd[3..0], nodo[N_DIGITOS..1], w          : NODE;
selec_base[1..0], FoP, prescaler_activo, z               : NODE;
datos[3..0], full, clear, cargar_fifo                    : NODE;
leer_fifo, HoL_nibble, clk_PC                            : NODE;
data_port_F[7..0], status_port_F[6..3]                   : NODE;

```



```

data_port_A[4..0], status_port_A[7..3]           : NODE;
sel_base_ext[1..0], pulsos_selbase, reloj_6ms  : NODE;

BEGIN

%%%%% Asignación de pines de acuerdo al modo de utilización de la FPGA %%%%%

CASE Frec_o_Adqui IS
    WHEN B"0" => %           Modo Adquisidor           %
        data_port_A[4..0] = data_port[4..0];
        status_port[7..3] = status_port_A[7..3];
    WHEN B"1" => %           Modo Frecuencímetro       %
        data_port_F[7..0] = data_port[7..0];
        status_port[6..3] = status_port_F[6..3];
END CASE;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Medidor de Frecuencias y Periodos %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%           Comienzo etapa prescaler           %
prescaler1.aclr = reset;
prescaler2.aclr = reset;
prescaler3.aclr = reset;
prescaler4.aclr = reset;
presc_sel_decodig.aclr = reset;
prescaler1.clock = reloj_xtal;
prescaler2.clock = reloj_xtal;
prescaler3.clock = reloj_xtal;
prescaler4.clock = reloj_xtal;
presc_sel_decodig.clock = reloj_xtal;
prescaler2.cnt_en = prescaler1.eq[9];
prescaler3.cnt_en = prescaler2.eq[9] & prescaler1.eq[9];
prescaler4.cnt_en = prescaler3.eq[9] & prescaler2.eq[9] &
                    prescaler1.eq[9];
presc_sel_decodig.cnt_en = prescaler4.eq[9] & prescaler3.eq[9] &
                    prescaler2.eq[9] & prescaler1.eq[9];

reloj_1ms = prescaler4.eq[9];
reloj_6ms = presc_sel_decodig.eq[5];
selec_decodig[POT_K-1..0] = presc_sel_decodig.q[];
clock10M = reloj_xtal;
%           Fin etapa prescaler           %

%           Circuito anti-rebotes para convertir el pulso de "pulsador_selbase" %
%           en un pulso "pulso_selbase" de 600ms de duración aproximadamente %
ffar1.d = VCC;
ffar1.clk = pulsador_selbase;
ffar2.clk = reloj_6ms;
cont_anti_rebote.clock = reloj_6ms;
cont_anti_rebote.cnt_en = ffar1.q;
IF cont_anti_rebote.q[] == 99 THEN
    ffar2.d = VCC;
ELSE
    ffar2.d = GND;
END IF;
ffar1.clrn = !ffar2.q;
pulsos_selbase = ffar1.q;

%           Circuito para incrementar con la entrada "pulsos_selbase" %
%           el contador de elección de la base de tiempos "contadorbase" %
contadorbase.clock = pulsos_selbase;
sel_base_ext[1..0] = contadorbase.q[];

```

```

% Selección entre modo manual o PC %
CASE manual_PC IS
    WHEN B"0" => % Modo PC %
        prescaler_activo = data_port_F[7];
        FoP = data_port_F[6];
        selec_base[] = data_port_F[5..4];
    WHEN B"1" => % Modo manual %
        prescaler_activo = activa_prescaler;
        FoP = Frec_o_Period;
        selec_base[] = sel_base_ext[1..0];
END CASE;

% Base de tiempo para modo medición de frecuencias %
salida.clrn = !reset;
salida.clk = reloj_1ms;
cuenta.clock = reloj_1ms;
cuenta.sclr = borrar;
CASE selec_base[] IS
    WHEN B"00" =>
        IF cuenta1.q[] < 10000 THEN
            salida.d = VCC;
            borrar = GND;
        ELSE
            borrar = VCC;
            salida.d = GND;
        END IF;
    WHEN B"01" =>
        IF cuenta1.q[] < 1000 THEN
            salida.d = VCC;
            borrar = GND;
        ELSE
            salida.d = GND;
            borrar = VCC;
        END IF;
    WHEN B"10" =>
        IF cuenta1.q[] < 100 THEN
            salida.d = VCC;
            borrar = GND;
        ELSE
            borrar = VCC;
            salida.d = GND;
        END IF;
    WHEN B"11" =>
        IF cuenta1.q[] < 10 THEN
            salida.d = VCC;
            borrar = GND;
        ELSE
            borrar = VCC;
            salida.d = GND;
        END IF;
END CASE;

% Pulsos para modo medición de período %
CASE selec_base[] IS
    WHEN B"00" =>
        pulsos = reloj_xtal;
    WHEN B"01" =>
        pulsos = prescaler1.eq[9];
    WHEN B"10" =>
        pulsos = prescaler2.eq[9];
    WHEN B"11" =>
        pulsos = prescaler3.eq[9];
END CASE;

```

```

%   Activación del prescaler que divide por 10                                     %
prescaler_signal.clock = signal_in;
IF prescaler_activo THEN % Si el prescaler esta activado %
    reloj_in = prescaler_signal.eq[9]; % La señal a medir es la %
                                     % señal exterior/10 %
ELSE % Si el prescaler no esta activado %
    reloj_in = signal_in; % La señal a medir es la señal exterior %
END IF;

%   Base de tiempo para modo medición de periodos                               %
divisor.clock = reloj_in;

%   Conexión de los nodos base_tiempos y reloj según se                          %
%   mida frecuencia ó periodo                                                  %
IF FoP THEN
    base_tiempos = salida.q; % Medición de %
    reloj = reloj_in; % Frecuencia %
ELSE
    base_tiempos = divisor.q[]; % Medición de %
    reloj = pulsos; % Período %
END IF;

%   Monoestable con maquina de estados                                         %
ss.clk = clock10M;
ss.reset = reset;
TABLE
%   estado  entrada      próximo %
%   actual  actual      estado  %
    ss,    base_tiempos =>  ss;

    s0,    1      =>  s0;
    s0,    0      =>  s1;
    s1,    0      =>  s2;
    s1,    1      =>  s0;
    s2,    0      =>  s2;
    s2,    1      =>  s0;
END TABLE;

%Inhabilitación de la señal z, para congelar latches, cuando leo por puerto%
z = z1 & !data_port_F[3];

%   Decodificador BCD a 7 segmentos                                             %
TABLE
bcd[3..0] => a, b, c, d, e, f, g;
H"0" => 1, 1, 1, 1, 1, 1, 0;
H"1" => 0, 1, 1, 0, 0, 0, 0;
H"2" => 1, 1, 0, 1, 1, 0, 1;
H"3" => 1, 1, 1, 1, 0, 0, 1;
H"4" => 0, 1, 1, 0, 0, 1, 1;
H"5" => 1, 0, 1, 1, 0, 1, 1;
H"6" => 1, 0, 1, 1, 1, 1, 1;
H"7" => 1, 1, 1, 0, 0, 0, 0;
H"8" => 1, 1, 1, 1, 1, 1, 1;
H"9" => 1, 1, 1, 1, 0, 1, 1;
END TABLE;

%   Generación de la señal w para resetear el contador                         %
resetcont.clrn = !base_tiempos;
resetcont.d = VCC;
resetcont.clk = !z1;
w = resetcont.q;

%   Contadores, latches y multiplexor                                          %
FOR i IN 1 TO N_DIGITOS GENERATE
    contador[i].clock = reloj;
    contador[i].aclr = w;
    latches[i].gate = z;
END GENERATE;

```

```

sal_cont[N_DIGITOS..1][3..0] = contador[N_DIGITOS..1].q[];
latches[N_DIGITOS..1].data[] = sal_cont[N_DIGITOS..1][];
qlatch[N_DIGITOS..1][] = latches[N_DIGITOS..1].q[];
multiplexor.data[N_DIGITOS-1..0][] = qlatch[N_DIGITOS..1][];
contador[1].cnt_en = base_tiempos;
nodo[1] = base_tiempos;
FOR n IN 2 TO N_DIGITOS GENERATE
    nodo[n] = nodo[n-1] & contador[n-1].eq[9];
    contador[n].cnt_en = nodo[n];
END GENERATE;

multiplexor.sel[] = selec_decodig[];
sal_mux[] = multiplexor.result[];

bcd[3..0] = sal_mux[];

%   Decodificador   %
decodificador.data[] = selec_decodig[];
selec_digito[N_DIGITOS..1] = decodificador.eq[N_DIGITOS-1..0];

%   Generación de señal overflow   %
ffoverflow1.clrn = !w;
ffoverflow1.d = VCC;
ffoverflow1.clk = contador[6].cout & contador[5].cout & contador[4].cout
                & contador[3].cout & contador[2].cout & contador[1].cout;
ffoverflow2.clrn = !reset;
ffoverflow2.d = ffoverflow1.q;
ffoverflow2.clk = !base_tiempos;
overflow = ffoverflow2.q;

%   Ubicación del punto (dp) de acuerdo a la base y modo utilizado   %
%   solo para el caso que N_DIGITOS >= 5   %
IF !prescaler_activo THEN           % Si no se activó prescaler %
    CASE selec_base[] IS
        WHEN B"00" =>
            dp = selec_digito[5];
        WHEN B"01" =>
            dp = selec_digito[4];
        WHEN B"10" =>
            dp = selec_digito[3];
        WHEN B"11" =>
            dp = selec_digito[2];
    END CASE;
ELSE                                 % Si se activó prescaler   %
    IF FoP THEN                     % en modo medición Frecuencia %
        CASE selec_base[] IS       % corro punto a derecha   %
            WHEN B"00" =>
                dp = selec_digito[5-1];
            WHEN B"01" =>
                dp = selec_digito[4-1];
            WHEN B"10" =>
                dp = selec_digito[3-1];
            WHEN B"11" =>
                dp = selec_digito[2-1];
        END CASE;
    ELSE                             % en modo medición Período   %
        CASE selec_base[] IS       % corro punto a izquierda %
            WHEN B"00" =>
                dp = selec_digito[5+1];
            WHEN B"01" =>
                dp = selec_digito[4+1];
            WHEN B"10" =>
                dp = selec_digito[3+1];
            WHEN B"11" =>
                dp = selec_digito[2+1];
        END CASE;
    END IF;
END IF;

```

```

% Salida hacia el puerto de estado (status_port) según la %
% entrada por el puerto de datos (data_port) %
CASE data_port_F[2..0] IS
    WHEN 0 =>
        status_port_F[3] = overflow;
    WHEN 1 =>
        status_port_F[6..3] = qlatch[1][];
    WHEN 2 =>
        status_port_F[6..3] = qlatch[2][];
    WHEN 3 =>
        status_port_F[6..3] = qlatch[3][];
    WHEN 4 =>
        status_port_F[6..3] = qlatch[4][];
    WHEN 5 =>
        status_port_F[6..3] = qlatch[5][];
    WHEN 6 =>
        status_port_F[6..3] = qlatch[6][];
END CASE;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Memoria FIFO para el adquisidor %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Conexión del puerto con los nodos %
clear = data_port_A[0];
cargar_fifo = data_port_A[1];
leer_fifo = data_port_A[2];
HoL_nibble = data_port_A[3];
clk_PC = data_port_A[4];
status_port_A[7] = full;
status_port_A[6..3] = datos[3..0];

% Obtención de las salidas /RD y /WR %
cuenta2.aclr = clear;
cuenta2.clock = reloj_xtal;
salidawr.clnr = !clear;
salidard.clnr = !clear;
salidawr.clk = reloj_xtal;
salidard.clk = reloj_xtal;

IF cuenta2.q[] < 8 THEN
    salidawr.d = GND;
    salidard.d = VCC;
ELSE IF cuenta2.q[] < 16 THEN
    salidawr.d = VCC;
    salidard.d = VCC;
ELSE IF cuenta2.q[] < 21 THEN
    salidawr.d = VCC;
    salidard.d = GND;
ELSE IF cuenta2.q[] < 24 THEN
    salidawr.d = VCC;
    salidard.d = VCC;
ELSE
    salidawr.d = GND;
    salidard.d = VCC;
END IF;
END IF;
END IF;

/WR = salidawr.q;
/RD = salidard.q;

% Los pines de entrada %
fifo.rdreq = leer_fifo;
fifo.aclr = clear;
fifo.wrreq = !/INT;
fifo.data[] = DB[];

```

```

%      Los pines de salida          %
/CS = !cargar_fifo;
full = fifo.full;

%      De acuerdo a el modo en que esté trabajando la FIFO          %
IF cargar_fifo THEN          % Si está cargando datos del ADC          %
    fifo.clock = salidard.q; % Reloj generado por el contador          %
ELSE                          % Si va a cargar datos en PC          %
    fifo.clock = clk_PC;      % Reloj generado por la PC          %
END IF;

%      El multiplexor divide las datos de salida para cargar por          %
%      puerto "datos[3..0]"          %
%      cuando HoL_nibble = 0 => datos[3..0] = q[7..4] High Nibble          %
%      HoL_nibble = 1 => datos[3..0] = q[3..0] Low Nibble          %
mux_HoL_nibble.data[0][] = fifo.q[7..4];
mux_HoL_nibble.data[1][] = fifo.q[3..0];
mux_HoL_nibble.sel[] = HoL_nibble;
datos[] = mux_HoL_nibble.result[];

END;

```

4.4 Diseño del Hardware del proyecto

En éste punto vamos a diseñar las plaquetas que se conectarán con la Upx10K10 a través de los conectores CON1 y CON2, para tener acceso a los pines de la FLEX10K10.

4.4.1 Diseño general de las plaquetas

Una de las posibilidades es la de generar tres plaquetas:

- Una plaqueta de comunicación con el puerto paralelo, conectada al conector CON2.
- Una plaqueta correspondiente al Medidor de frecuencias y períodos; con sus respectivos interruptores, pulsadores, displays, leds; conectada al conector CON1.
- Una plaqueta correspondiente al Adquisidor autónomo de datos; con el conversor; conectada al conector CON1.

Luego la plaqueta del puerto estará siempre conectada al CON2, mientras que de acuerdo a cual de los dos subproyecto queramos ejecutar, conectaremos al CON1 la plaqueta del Medidor de frecuencias y períodos o la del Adquisidor.

La figura 4.6 muestra un croquis de las tres plaquetas conectadas a la plaqueta experimental Upx10K10.

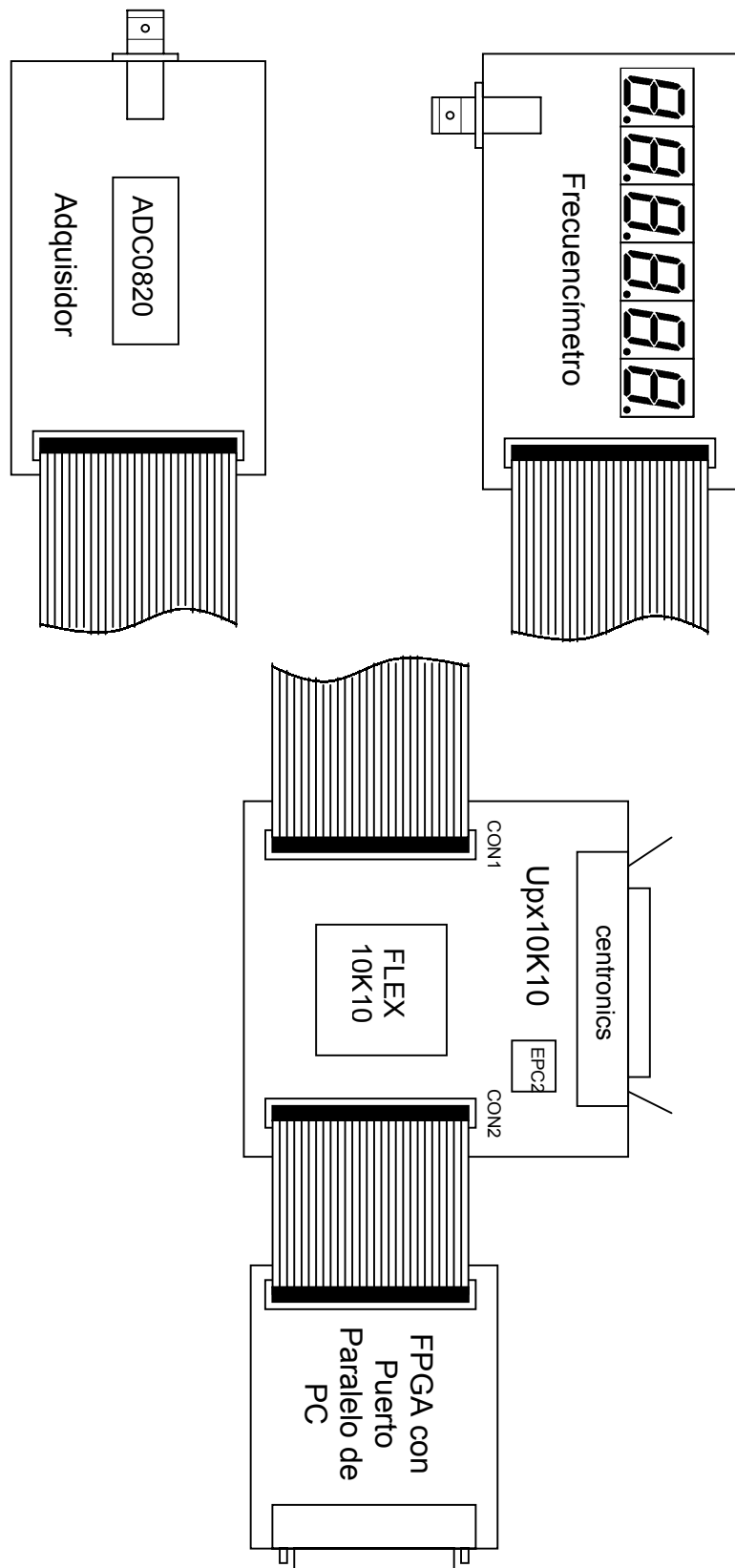


Figura 4.6: Croquis de plaquetas

Como vemos tenemos dos placas conectadas al CON1 y una única conectada al CON2 de la Upx10K10. Es por esto que debemos tener en cuenta en la

asignación de los pines a la FLEX, que todos los del Frecuencímetro se encuentren en el CON1, como así también los del adquisidor; mientras que los de conexión al puerto en el CON2.

4.4.2 Asignación de los pines

Los pines de la plaqueta del Medidor de frecuencias y período son:

- **Salida:** `reset`, `pulsador_selbase`, `signal_in`, `Frec_o_Period`, `activa_prescaler` y `Frec_o_Adqui`.
- **Entrada:** `a`, `b`, `c`, `d`, `e`, `f`, `g`, `dp`, `overflow` y `selec_digito[6..1]`.

Los pines de la plaqueta del Adquisidor son:

- **Salida:** `Frec_o_Adqui`, `/INT` y `DB[7..0]`.
- **Entrada:** `/CS`, `/RD` y `/WR`.

Y los pines de la plaqueta de comunicación a puerto paralelo:

- **Salida:** `manual_PC` y `data_port[7..0]`.
- **Entrada:** `status_port[7..3]`.

Luego además tenemos el oscilador (`reloj_xtal`), pero éste es generado por el oscilador interno de la UPx10K10 (pin1 del CON1 y CON2, ver tabla 4.1).

Vamos ahora a asignar al CON1 los pines del Medidor de frecuencia y período según muestra la tabla 4.3:

CON1	Conectado con	Pin del proyecto
1	MCLK*: EPF10K10#1	
2	IO00*: EPF10K10#16	<code>pulsador_selbase</code>
3	TDO (#1) de la EPC2	
4	IO01*: EPF10K10#17	<code>Frec_o_Period</code>
5	GND*	
6	GND*	
7	IO02: EPF10K10#18	<code>sel_digito1</code>
8	IO03: EPF10K10#19	<code>activa_prescaler</code>
9	IO04: EPF10K10#21	<code>overflow</code>
10	IO05: EPF10K10#22	<code>sel_digito2</code>
11	DEDIN0: EPF10K10#2	<code>Frec_o_Adqui</code>
12	DEDIN1: EPF10K10#42	<i>reservado para reset</i>
13	IO06: EPF10K10#23	<code>sel_digito3</code>
14	GCLEAR*: EPF10K10#3	<code>reset</code>
15	IO07: EPF10K10#24	<code>sel_digito4</code>
16	IO08: EPF10K10#25	<i>reservado para reset</i>
17	IO09: EPF10K10#27	<code>sel_digito5</code>
18	TDI (#3) de CON2	

19	IO10: EPF10K10#28	sel_digito6
20	IO11: EPF10K10#29	dp
21	VCC*	
22	VCC*	
23	IO12: EPF10K10#30	c
24	IO13: EPF10K10#35	e
25	IO14: EPF10K10#36	d
26	TCK*: EPF10K10#77	
27	IO15: EPF10K10#37	b
28	IO16: EPF10K10#38	a
29	IO17: EPF10K10#39	g
30	IO37: EPF10K10#5	f
31	IO38: EPF10K10#6	
32	IO39: EPF10K10#7	
33	IO40: EPF10K10#8	
34	IO41: EPF10K10#9	
35	IO42: EPF10K10#10	
36	IO43: EPF10K10#11	
37	IO44: EPF10K10#69	signal_in
38	IO35*: EPF10K10#71	
39	IO36*: EPF10K10#72	
40	TMS*: EPF10K10#57	

Tabla 4.3: Asignación de pines del Frecuencímetro al CON1 y EPF10K10LC84

Como vemos hemos dejado dos pines mas reservados para `reset` (pin 12 y 16 del CON1), debido a que en la prueba de la plaqueta implementándola con *protoboard*, se detectó una pequeño problema de ruido en la señal de reset, el cual cambiaba si se utilizaba un pin de entrada dedicada (pin 12), un pin de *global clear* (pin 14) o una entrada/salida (pin 16) de la FPGA. Es por esto que se diseñará la placa reservando estos tres pines y luego mediante un puente se seleccionará el óptimo.

Luego asignaremos al CON1 los pines del Adquisidor según muestra la tabla 4.4:

CON1	Conectado con	Pin del proyecto
1	MCLK*: EPF10K10#1	
2	IO00*: EPF10K10#16	
3	TDO (#1) de la EPC2	
4	IO01* : EPF10K10#17	/INT
5	GND*	
6	GND*	
7	IO02: EPF10K10#18	
8	IO03: EPF10K10#19	
9	IO04: EPF10K10#21	
10	IO05: EPF10K10#22	
11	DEDIN0: EPF10K10#2	Frec_o_Adqui
12	DEDIN1: EPF10K10#42	
13	IO06: EPF10K10#23	
14	GCLEAR*: EPF10K10#3	
15	IO07: EPF10K10#24	
16	IO08: EPF10K10#25	
17	IO09: EPF10K10#27	
18	TDI (#3) de CON2	
19	IO10: EPF10K10#28	

20	IO11: EPF10K10#29	
21	VCC*	
22	VCC*	
23	IO12: EPF10K10#30	/WR
24	IO13: EPF10K10#35	
25	IO14: EPF10K10#36	
26	TCK*: EPF10K10#77	
27	IO15: EPF10K10#37	/RD
28	IO16: EPF10K10#38	/CS
29	IO17: EPF10K10#39	
30	IO37: EPF10K10#5	
31	IO38: EPF10K10#6	DB7
32	IO39: EPF10K10#7	DB6
33	IO40: EPF10K10#8	DB5
34	IO41: EPF10K10#9	DB4
35	IO42: EPF10K10#10	DB3
36	IO43: EPF10K10#11	DB2
37	IO44: EPF10K10#69	
38	IO35*: EPF10K10#71	DB1
39	IO36*: EPF10K10#72	DB0
40	TMS*: EPF10K10#57	

Tabla 4.4: Asignación de pines del Adquisidor al CON1 y EPF10K10LC84

Como podemos ver de la tablas 4.3 y 4.4, tenemos una cantidad de pines del Frecuencímetro y el Adquisidor asignados a los mismos pines del CON1 (pin 4, 23, 27 y 28). Esto se debe a la limitación de la cantidad de pines de la FPGA a los que podemos acceder por el CON1.

Para salvar esta complicación vamos a reformar el programa en AHDL de modo que se cumpla la siguiente tabla:

pin de entada o salida	Frec_o_Adqui = 0 (Adquisidor)	Frec_o_Adqui = 1 (Frecuencímetro)
Frec_o_Period_o_/INT	/INT	Frec_o_Period
c_o_/WR	/WR	c
b_o_/RD	/RD	b
a_o_/CS	/CS	a

Tabla 4.5: Asignación de pines de acuerdo a Frec_o_Adqui

Luego en AHDL:

```

...
reloj_xtal, reset, manual_PC, pulsador_selbase           : INPUT;
signal_in, Frec_o_Period_o_/INT, activa_prescaler       : INPUT;
Frec_o_Adqui, DB[7..0], data_port[7..0]                 : INPUT;
a_o_/CS, b_o_/RD, c_o_/WR, d, e, f, g, dp, overflow    : OUTPUT;
selec_digito[N_DIGITOS..1], status_port[7..3]         : OUTPUT;
...
a, b, c, /CS, /RD, /WR, /INT, Frec_o_Period           : NODE;
...
CASE Frec_o_Adqui IS
  WHEN B"0" => %                                     Modo Adquisidor %
    data_port_A[4..0] = data_port[4..0];
    status_port[7..3] = status_port_A[7..3];
    /INT = Frec_o_Period_O_/INT;
    a_o_/CS = /CS;
    b_o_/RD = /RD;
    c_o_/WR = /WR;

```

```

WHEN B"1" =>
    data_port_F[7..0] = data_port[7..0];
    status_port[6..3] = status_port_F[6..3];
    Frec_o_Period = Frec_o_Period_O_/INT;
    a_o_/CS = a;
    b_o_/RD = b;
    c_o_/WR = c;
END CASE;

```

Luego, por lo tanto la asignación de pines al CON1 queda de la forma:

CON1	Conectado con	Pin del proyecto
1	MCLK*: EPF10K10#1	
2	IO00*: EPF10K10#16	pulsador selbase
3	TDO (#1) de la EPC2	
4	IO01*: EPF10K10#17	Frec_o_Period_o_/INT
5	GND*	
6	GND*	
7	IO02: EPF10K10#18	sel_digito1
8	IO03: EPF10K10#19	activa_prescaler
9	IO04: EPF10K10#21	overflow
10	IO05: EPF10K10#22	sel_digito2
11	DEDIN0: EPF10K10#2	Frec_o_Adqui
12	DEDIN1: EPF10K10#42	<i>reservado para reset</i>
13	IO06: EPF10K10#23	sel_digito3
14	GCLEAR*: EPF10K10#3	reset
15	IO07: EPF10K10#24	sel_digito4
16	IO08: EPF10K10#25	<i>reservado para reset</i>
17	IO09: EPF10K10#27	sel_digito5
18	TDI (#3) de CON2	
19	IO10: EPF10K10#28	sel_digito6
20	IO11: EPF10K10#29	dp
21	VCC*	
22	VCC*	
23	IO12: EPF10K10#30	c_o_/WR
24	IO13: EPF10K10#35	e
25	IO14: EPF10K10#36	d
26	TCK*: EPF10K10#77	
27	IO15: EPF10K10#37	b_o_/RD
28	IO16: EPF10K10#38	a_o_/CS
29	IO17: EPF10K10#39	g
30	IO37: EPF10K10#5	f
31	IO38: EPF10K10#6	DB7
32	IO39: EPF10K10#7	DB6
33	IO40: EPF10K10#8	DB5
34	IO41: EPF10K10#9	DB4
35	IO42: EPF10K10#10	DB3
36	IO43: EPF10K10#11	DB2
37	IO44: EPF10K10#69	signal_in
38	IO35*: EPF10K10#71	DB1
39	IO36*: EPF10K10#72	DB0
40	TMS*: EPF10K10#57	

Tabla 4.6: Asignación de pines al CON1 y EPF10K10LC84

Vamos finalmente a asignar al CON2 los pines de la plaqueta de comunicación a puerto paralelo:

CON2	Conectado con	Pin del proyecto
1	MCLK*: EPF10K10#1	
2	IO18: EPF10K10#47	data_port0
3	TDO (#18) de CON1	
4	IO19: EPF10K10#48	data_port1
5	GND*	
6	GND*	
7	IO20: EPF10K10#49	data_port2
8	IO21: EPF10K10#50	data_port3
9	IO22: EPF10K10#51	data_port4
10	IO23: EPF10K10#52	data_port5
11	DEDIN2: EPF10K10#44	
12	DEDIN3: EPF10K10#84	
13	IO24: EPF10K10#53	data_port6
14	GCLEAR*: EPF10K10#3	
15	IO25: EPF10K10#54	data_port7
16	IO26: EPF10K10#58	status_port3
17	IO27: EPF10K10#59	status_port4
18	TDO del ByteBlaster	
19	IO28: EPF10K10#60	status_port5
20	IO29: EPF10K10#61	status_port6
21	VCC*	
22	VCC*	
23	IO30: EPF10K10#62	status_port7
24	IO31: EPF10K10#64	
25	IO32: EPF10K10#65	manual_PC
26	TCK*: EPF10K10#77	
27	IO33: EPF10K10#66	
28	IO34: EPF10K10#67	
29	IO35*: EPF10K10#71	
30	IO36*: EPF10K10#72	
31	IO45: EPF10K10#70	
32	IO46: EPF10K10#73	
33	IO47: EPF10K10#78	
34	IO48: EPF10K10#79	
35	IO49: EPF10K10#80	
36	IO50: EPF10K10#81	
37	IO51: EPF10K10#83	
38	IO00*: EPF10K10#16	
39	IO01*: EPF10K10#17	
40	TMS*: EPF10K10#57	

Tabla 4.7: Asignación de pines al CON2 y EPF10K10LC84

4.4.3 Diseño de la plaqueta de comunicación con el puerto paralelo de la PC.

Vamos a continuación a realizar el diseño completo de la plaqueta que permitirá la conexión entre el puerto paralelo de la PC y el CON2 que comunicará con la FPGA; a la que llamaremos *FPGA con Puerto Paralelo de PC*.

Inicialmente vamos a construir el esquemático del circuito en el programa Protel 99SE, para luego crear el PCB (*Printed Circuit Board*) del mismo.

Este circuito consta simplemente de los siguientes componentes:

- ✓ 1 conector macho de 2 x 40.
- ✓ 1 conector DB25 macho a 90°.
- ✓ 2 74LS241.
- ✓ 2 capacitores de 0.1uF.
- ✓ 2 resistencias de 4.7K.

El primer conector (J1) servirá para conectar al puerto paralelo de la PC y el segundo para conectar mediante un cable plano al conector CON2 de la Upx10K10.

Los 74LS241 (U1 y U2) son *buffers* que permitirán proteger tanto al puerto como a la FPGA y los capacitores (C1 y C2) se utilizarán para proteger sus alimentaciones de ruido. Una de las resistencias (Rpd) se utilizarán como *pull-down* en la entrada D3 (`data_port[3]`), ya que en el modo Frecuencímetro (como vimos en el capítulo anterior), en el caso de no tener el puerto conectado, ésta quedaría flotante y puede que no se carguen los latches con la periodicidad correcta.

La otra resistencia (Rpu) se utilizará como *pull-up* para el caso de modo manual o PC del Medidor de Frecuencias y Períodos. Cuando el cable que conecte al puerto esté desconectado, ésta resistencia forzará la entrada manual_PC a '1' y por lo tanto trabajará en modo manual. Cuando se conecte el cable al puerto, éste forzará la entrada a '0'.

4.4.3.1 Asignación de pines al puerto paralelo

Inicialmente vamos a ver cual es la asignación de pines del puerto paralelo en modo SPP.

La figura 4.7 muestra la disposición de las patas de un conector DB25 hembra:

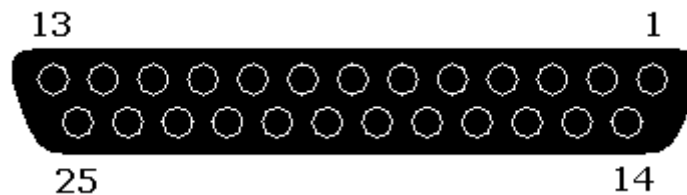


Figura 4.7: Disposición de los pines conector DB25 hembra

La tabla 4.8 muestra la descripción de los pines:

pin	descripción	dirección
1	/Strobe	salida
2	data 0	salida
3	data 1	salida
4	data 2	salida
5	data 3	salida
6	data 4	salida
7	data 5	salida

8	data 6	salida
9	data 7	salida
10	status 6	entrada
11	/status 7	entrada
12	status 5	entrada
13	status 4	entrada
14	/control 1	salida
15	status 3	entrada
16	control 2	salida
17	/control 3	salida
18	GND	---
19	GND	---
20	GND	---
21	GND	---
22	GND	---
23	GND	---
24	GND	---
25	GND	---

Tabla 4.8: Asignación de pines al DB25

4.4.3.2 Esquemático de la placa 'FPGA con Puerto Paralelo de PC'.

El diseño del esquemático queda como muestra la siguiente figura:

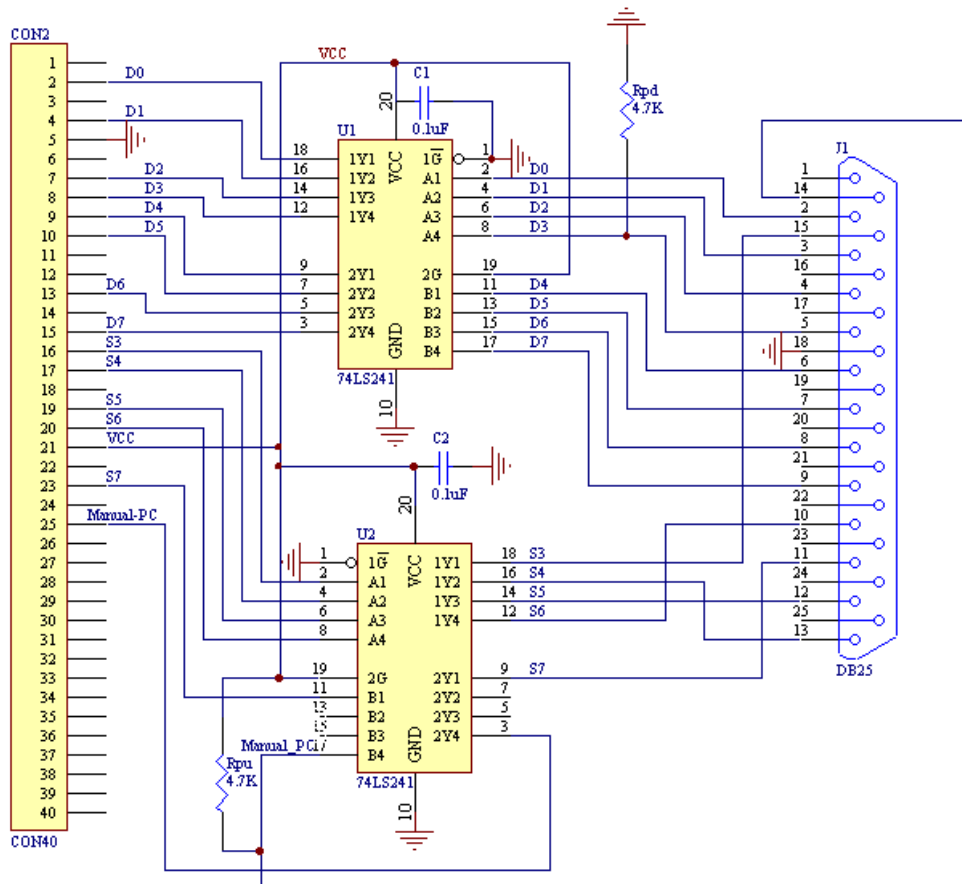


Figura 4.8: Esquemático de placa 'FPGA con puerto paralelo de PC'.

4.4.3.3 PCB de la plaqueta 'FPGA con Puerto Paralelo de PC'.

La ubicación de los componentes en una plaqueta cuadrada simple faz de 71.12mm de lado queda:

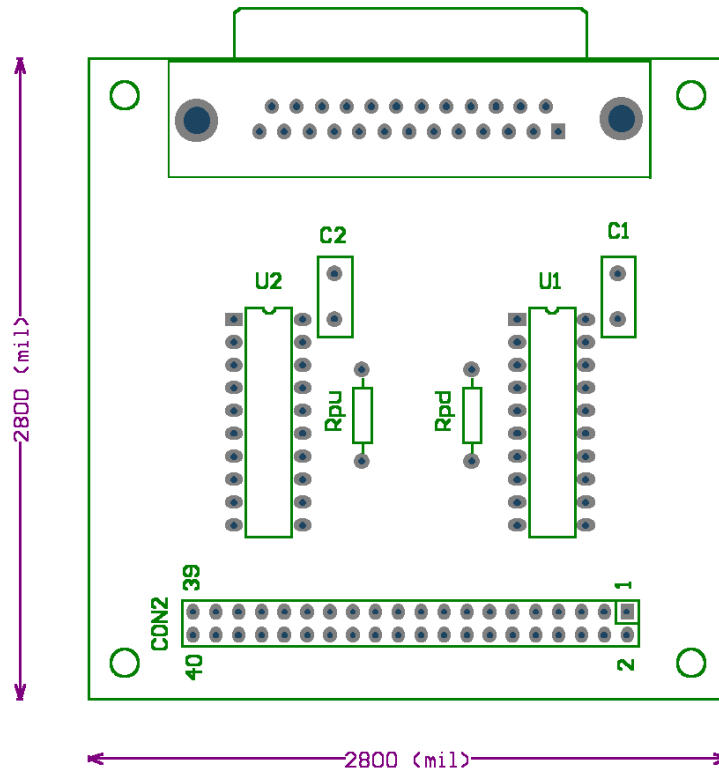


Figura 4.9: Ubicación de los componentes en la plaqueta 'FPGA con puerto paralelo de PC'.

luego las pistas quedan de la forma:

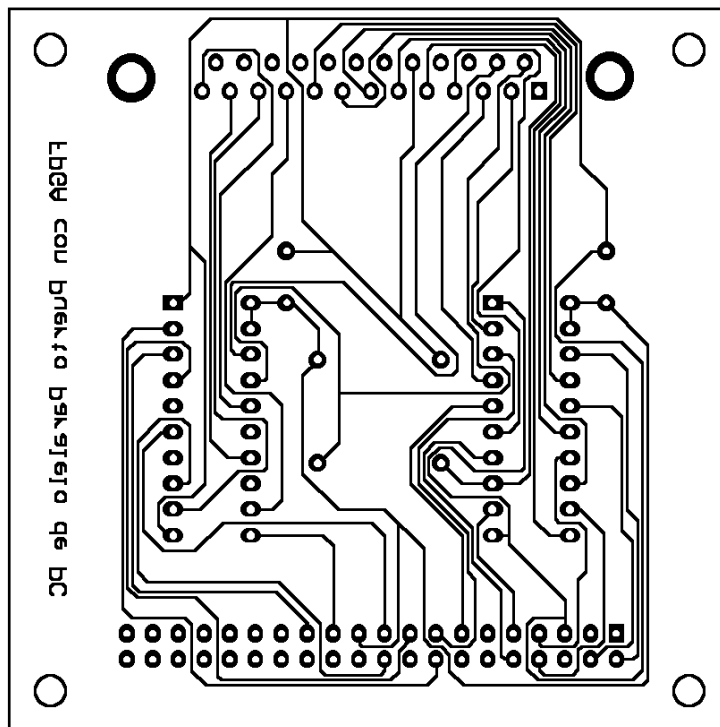


Figura 4.10: Ubicación de las pistas en la plaqueta 'FPGA con puerto paralelo de PC'.

Finalmente la plaqueta con los componentes y las pistas, se muestran en la figura 4.11:

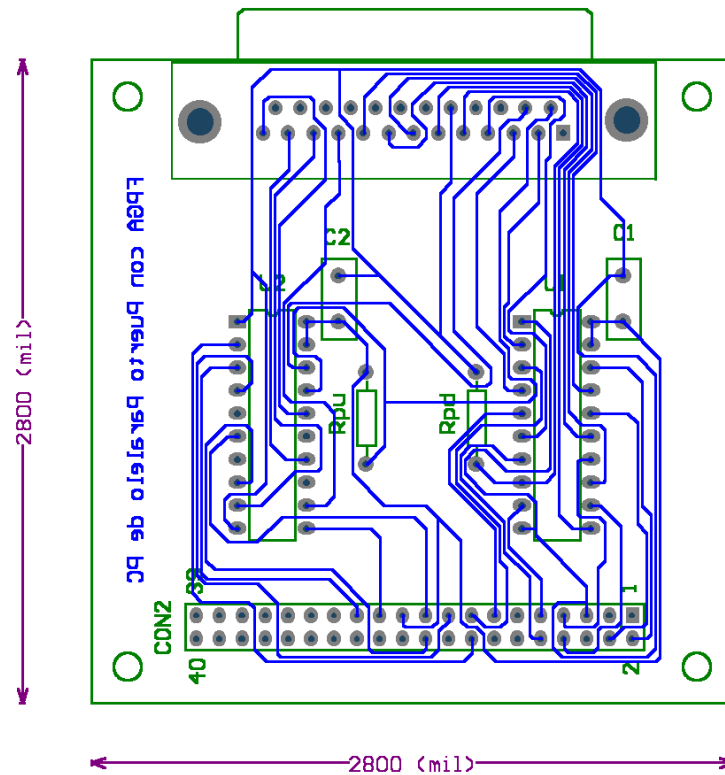


Figura 4.11: Ubicación de los componentes y las pistas en la plaqueta 'FPGA con puerto paralelo de PC'.

4.4.3.4 Plaqueta 'FPGA con Puerto Paralelo de PC' final.

La figura 4.12 nos muestra a la plaqueta FPGA con puerto paralelo de PC armada para éste proyecto.

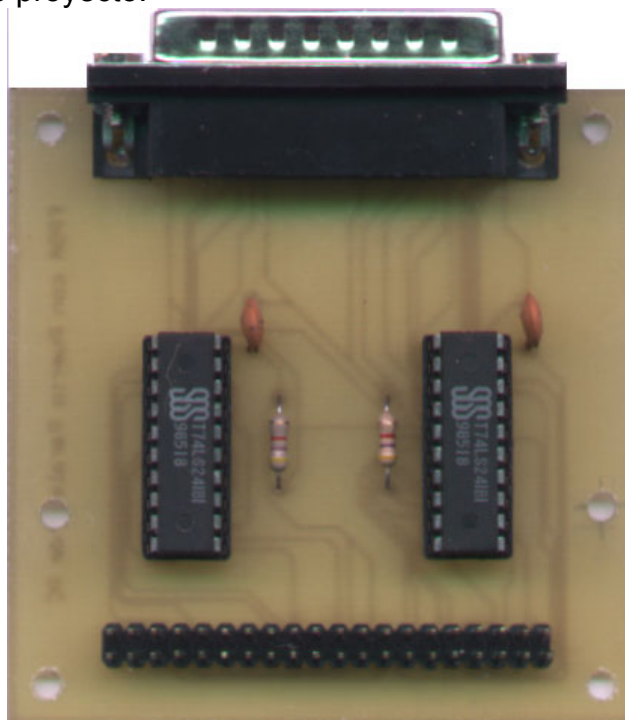


Figura 4.12: Plaqueta 'FPGA con puerto paralelo de PC' armada para el proyecto.

Luego ésta plaqueta se conectará al puerto paralelo de la PC a través de un cable, el cual tendrá internamente conectada la salida `manual_PC` con GND (tierra). Esto permitirá que en cuando ejecutemos el Medidor de Frecuencias y Períodos, al conectar el cable que va al puerto, trabaje en modo PC (`manual_PC = 0`), y cuando se desconecte el mismo trabaje en modo manual (`manual_PC = 1`), forzado por la resistencia de *pull-up* Rpu).

4.4.4 Diseño de la plaqueta del Medidor de frecuencias y períodos

Posteriormente vamos a diseñar plaqueta que contendrá los displays, pulsadores, leds y otros componentes del Medidor de frecuencias y períodos; a la que llamaremos *Frecuencímetro*.

El circuito del Frecuencímetro está conformado por los siguientes componentes:

- ✓ 1 conector macho de 2 x 40.
- ✓ 1 conector BNC hembra.
- ✓ 1 ULN2803.
- ✓ 2 leds rojos de 3mm.
- ✓ 2 leds rojos de 5mm.
- ✓ 6 displays de 7 segmentos de 13mm (AC).
- ✓ 6 transistores PNP 2N3906.
- ✓ 6 resistencias de 1.5K.
- ✓ 13 resistencias de 220.
- ✓ 1 resistencia de 4.7K.
- ✓ 1 capacitor de 0.047uF.
- ✓ 2 pulsadores.
- ✓ 2 pulsadores con retención.

El primer conector servirá para conectar, mediante un cable plano, al conector CON1 de la Upx10K10; mientras que el segundo (BNC) para conectar la señal de entrada a medir.

El ULN2803 (U1) permite no tomar el consumo de los leds de los displays de los pines de la FPGA, sino de la fuente de la plaqueta experimental

Los leds de 3mm (ms y kHz) son para diferenciar entre medición de período (se enciende el de ms) y medición de frecuencias (se enciende el de kHz). Los de 5mm son para indicar cuando el prescaler está activado y cuando hay overflow.

Los seis display (Digito6..Digito1) son para representar la frecuencia o período medidos. La selección de cada uno de éstos displays se realiza a través de los seis transistores PNP (Q6..Q1), cuyas bases se encuentran excitadas por las entradas `sel_digito[6..1]`, pasando por seis resistencias de 1.5K (R6..R1) para limitar la corriente de la base.

Doce resistencias de 220 son para limitar la corriente de los leds a 15mA aproximadamente (son 8 para los leds del display (Ra, Rb,..,Rdp), 2 para los leds de 3mm (Rms, RkHz) y 2 para los de 5mm (Roverf y Rpres)).

La resistencia restante de 220 (Rclear) junto con el capacitor de 0.047uF (Cc) permiten realizar una señal de reset en el momento que se enciende el circuito.

La resistencia de 4.7K (Rbase) actúa como *pull-down* en la entrada de cambio de base.

Los dos pulsadores son para señal de reset y cambio de base respectivamente; mientras que los con retención son para activar el prescaler y elegir entre modo frecuencia o período.

Además se fijará el pin `Frec_o_Adqui` a `VCC` para seleccionar el modo Medidor de Frecuencias y Períodos.

4.4.4.1 Esquemático de la plaqueta Frecuencímetro.

El diseño del esquemático queda como muestra la siguiente figura:

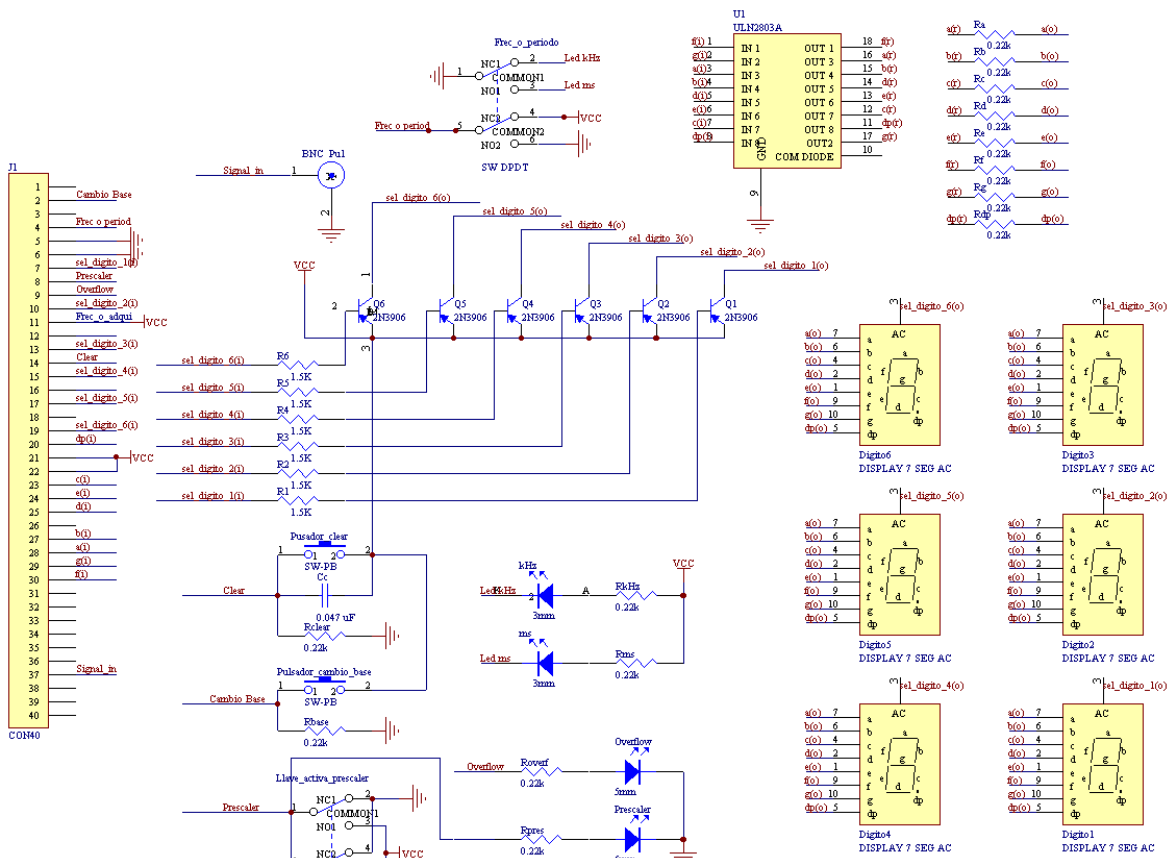


Figura 4.13: Esquemático de plaqueta Frecuencímetro.

Como vemos se han utilizado displays ánodo común, y por lo tanto transistores PNP para la selección de dichos dígitos; es por esto que luego en el programa final deberemos invertir las salidas de la FPGA de selección de dígitos.

4.4.4.2 PCB de la plaqueta Frecuencímetro.

La ubicación de los componentes en una plaqueta rectangular simple faz de 107 x 163mm queda:

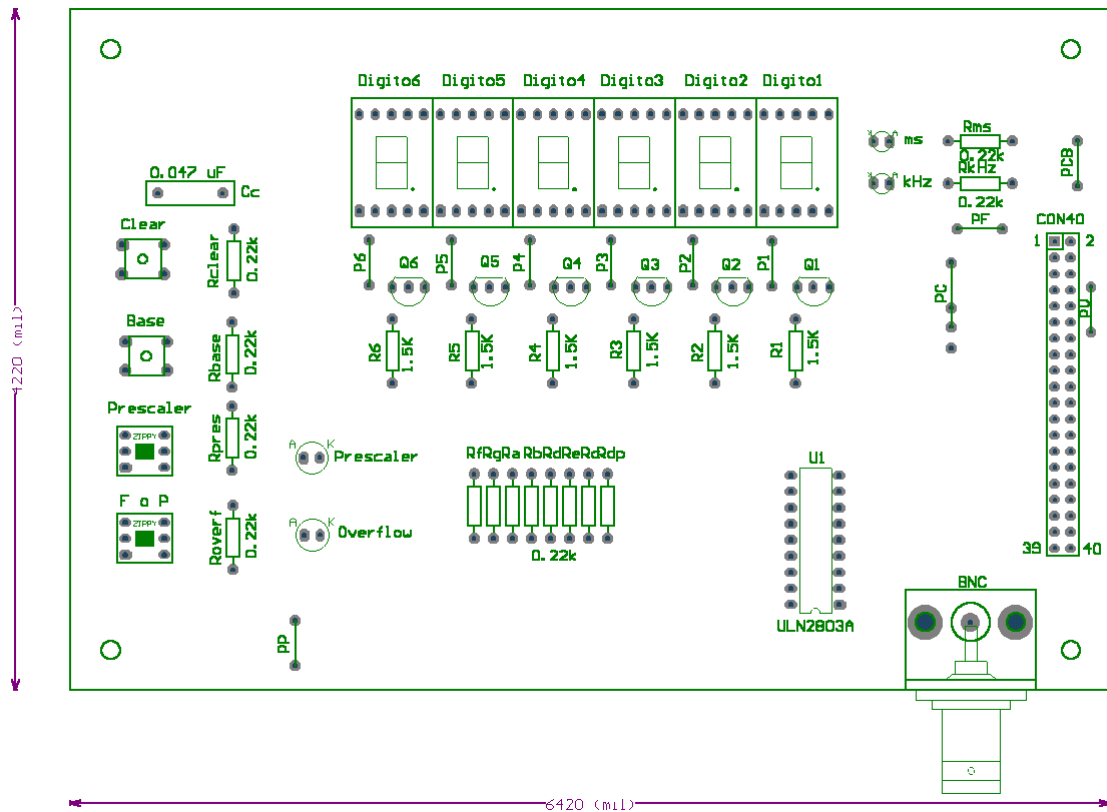


Figura 4.14: Ubicación de los componentes en la plaqueta Frecuencímetro.

Donde los componentes P1, P2, P3, P4, P5, P6, PCB, PV, PF, PC y PP son puentes. Debido a que la plaquetas se realizan en simple faz, éstos puentes permiten que sus respectivas pistas no se crucen con las otras. En particular el puente PC (correspondiente a la señal de reset) permite interconectar la pista proveniente del pulsador Clear con tres pistas distintas correspondientes a los pines 12, 14 o 16 del conector de 40 por el problema mencionado en el punto 4.4.2.

Las pistas entonces quedan de la forma:

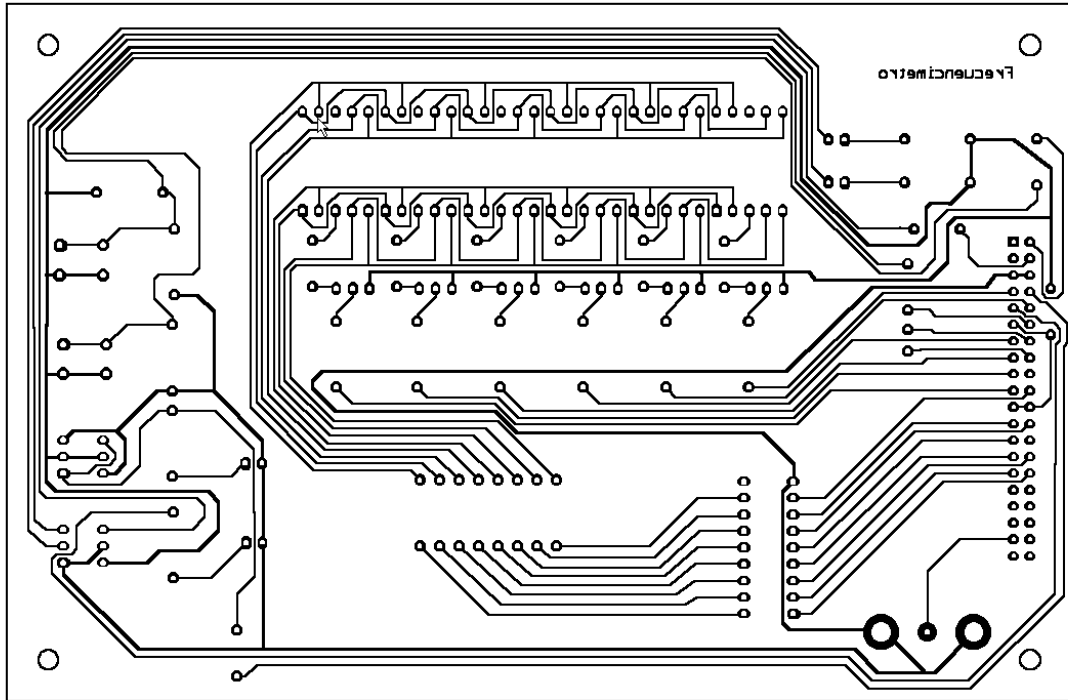


Figura 4.15: Ubicación de las pistas en la placa Frecuencímetro.

Y la placa con los componentes y las pistas, se muestran en la figura 4.16:

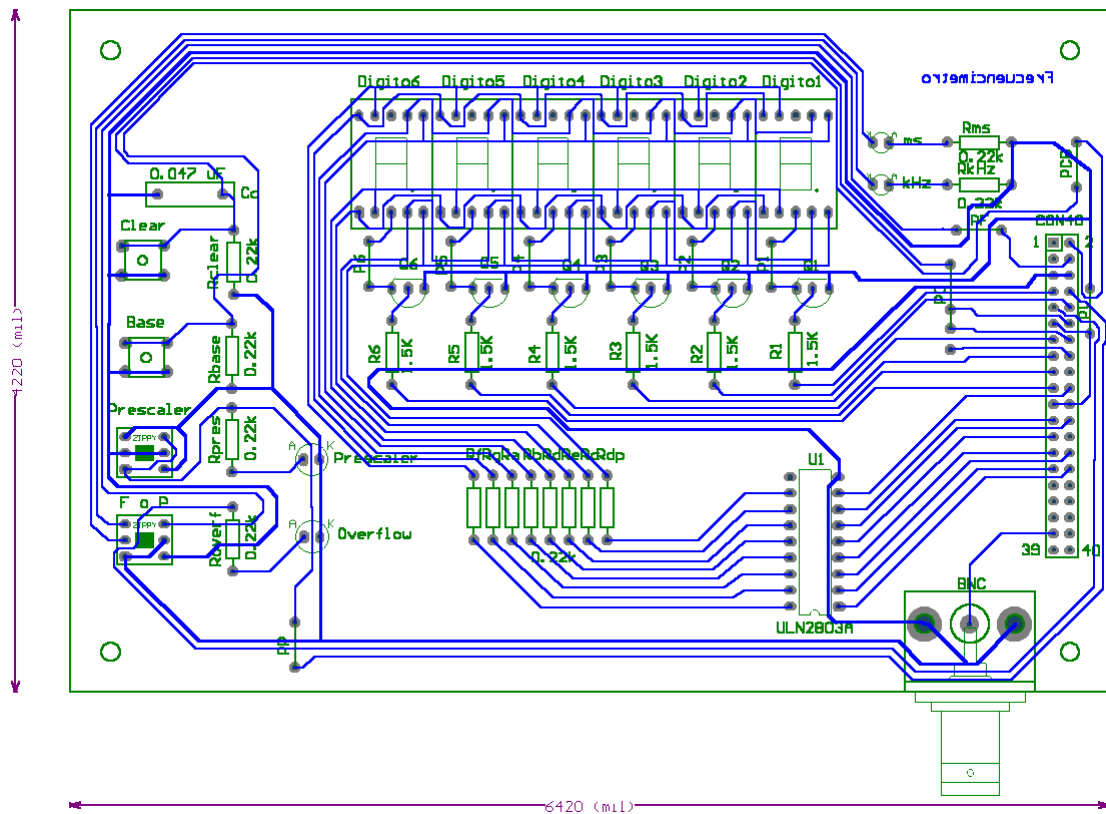


Figura 4.16: Ubicación de los componentes y las pistas en la placa Frecuencímetro.

4.4.4.3 Plaqueta Frecuencímetro final.

La figura 4.17 nos muestra a la plaqueta Frecuencímetro armada para éste proyecto.

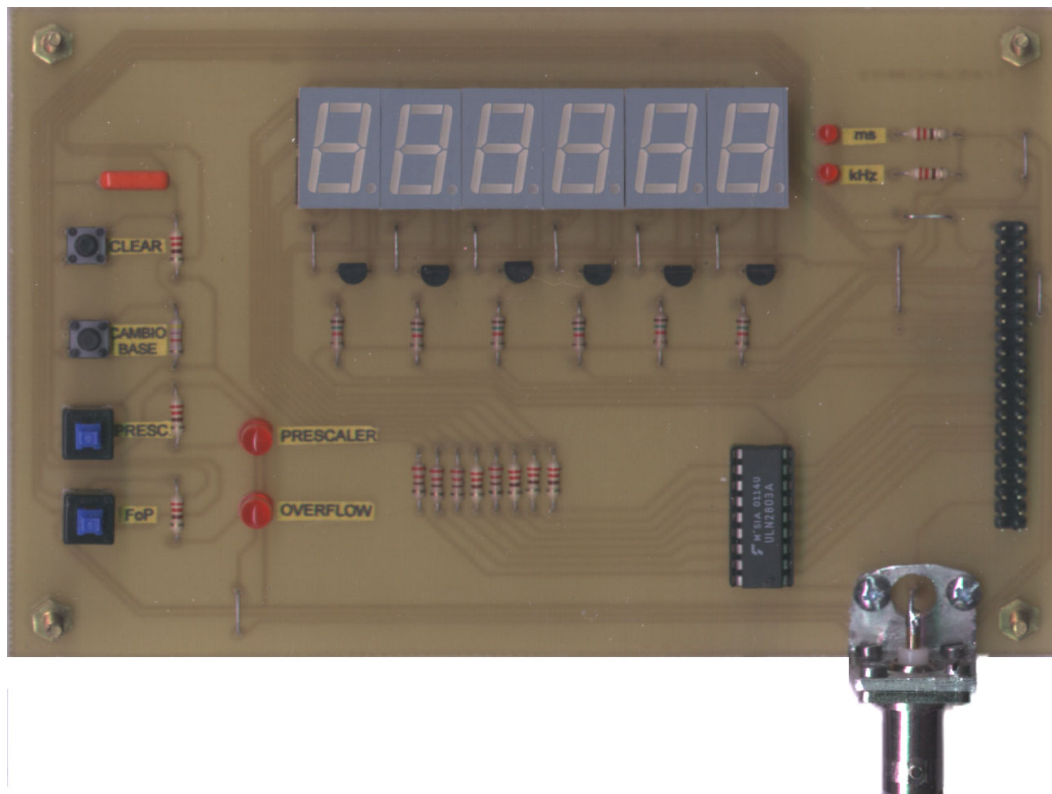


Figura 4.17: Plaqueta Frecuencímetro armada para el proyecto.

4.4.5 Diseño de la plaqueta del Adquisidor autónomo de datos

A continuación vamos a diseñar plaqueta que contendrá el conversor y otros componentes del Adquisidor autónomo de datos; a la que llamaremos *Adquisidor*.

El circuito del Adquisidor está conformado por los siguientes componentes:

- ✓ 1 conector macho de 2 x 40.
- ✓ 1 conector BNC hembra.
- ✓ 1 ADC0820.
- ✓ 1 LF351.
- ✓ 1 7660.
- ✓ 1 LM336 de 2.5V.
- ✓ 1 potenciómetro vertical de 200K.
- ✓ 1 resistencia de 560.
- ✓ 3 capacitores de 100nF.
- ✓ 4 capacitores de 10uF (25 o 50V).

El primer conector servirá para conectar, mediante un cable plano, al conector CON1 de la Upx10K10; mientras que el segundo (BNC) para conectar la señal de entrada a muestrear.

El ADC0820 (U3) es el conversor analógico digital que utilizaremos, debido a que es el que posee la cátedra de *Introducción a los Sistemas Lógicos y Digitales*.

El LF351 es un amplificador operacional que conectado como *buffer* (realimentación unitaria) posee un ancho de banda de 4MHz. Éste permitirá que el circuito Adquisidor tenga una impedancia de entrada muy grande y no sobrecargue al que contiene la señal a medir.

Puesto que el LF351 trabaja con fuente partida de +5V y -5V, y la plaqueta experimental sólo nos suministra +5V, utilizamos el 7660 que mediante el agregado de un capacitor de 10uF nos provee de una fuente partida de +/-5V.

El LM336 en conjunto con el potenciómetro de 200K y la resistencia de 560 para limitar la corriente, permiten fijar con exactitud la tensión de referencia positiva del conversor a 2.5V (mientras que la negativa es tierra).

Los capacitores de 100nF y los tres restantes de 10uF se utilizan para filtrar las fuentes que alimentan el ADC0820, el 7660 y el LF351.

Para esta plaqueta se fijará el pin `Frec_o_Adqui` a GND para seleccionar el modo Adquisidor autónomo de datos

Como vemos las tensiones de referencia del adquisidor son +2.5V y 0V, por lo tanto la señal a muestrear deberá estar comprendida en éste rango.

4.4.5.1 Esquemático de la plaqueta Adquisidor.

El diseño del esquemático del adquisidor queda como muestra la siguiente figura:

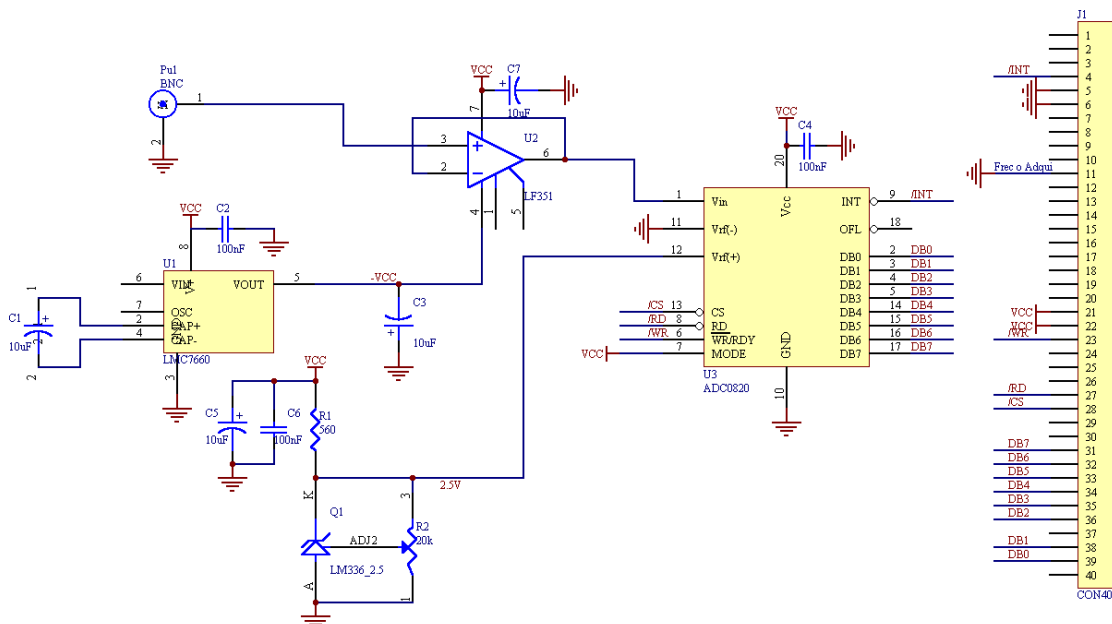


Figura 4.18: Esquemático de plaqueta Adquisidor.

4.4.5.2 PCB de la plaqueta Adquisidor.

La ubicación de los componentes en una plaqueta simple faz de 63.5 x 91.44mm queda:

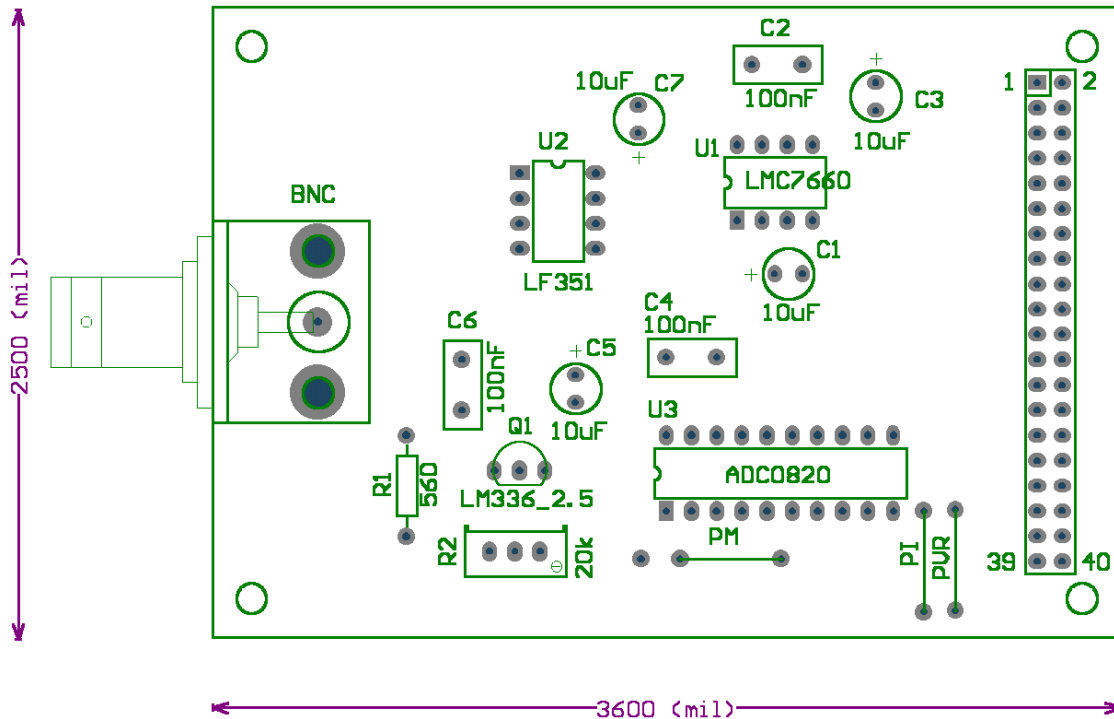
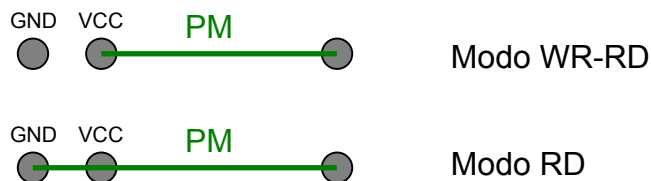


Figura 4.19: Ubicación de los componentes en la plaqueta Adquisidor.

Nuevamente aparecen aquí puentes como PI, PVR, y PM que permiten que sus respectivas pistas no se crucen con las otras.

En particular el puente PM es el que alimenta el pin *MODE* del ADC0820. Éste pin cuando se alimenta con VCC (nuestro caso) permite al adquisidor trabajar en modo WR-RD; pero además se puede conectar a GND (tierra) con lo cual el adquisidor puede trabajar en modo RD en un futuro proyecto.



Las pistas de la plaqueta Adquisidor quedan de la forma:

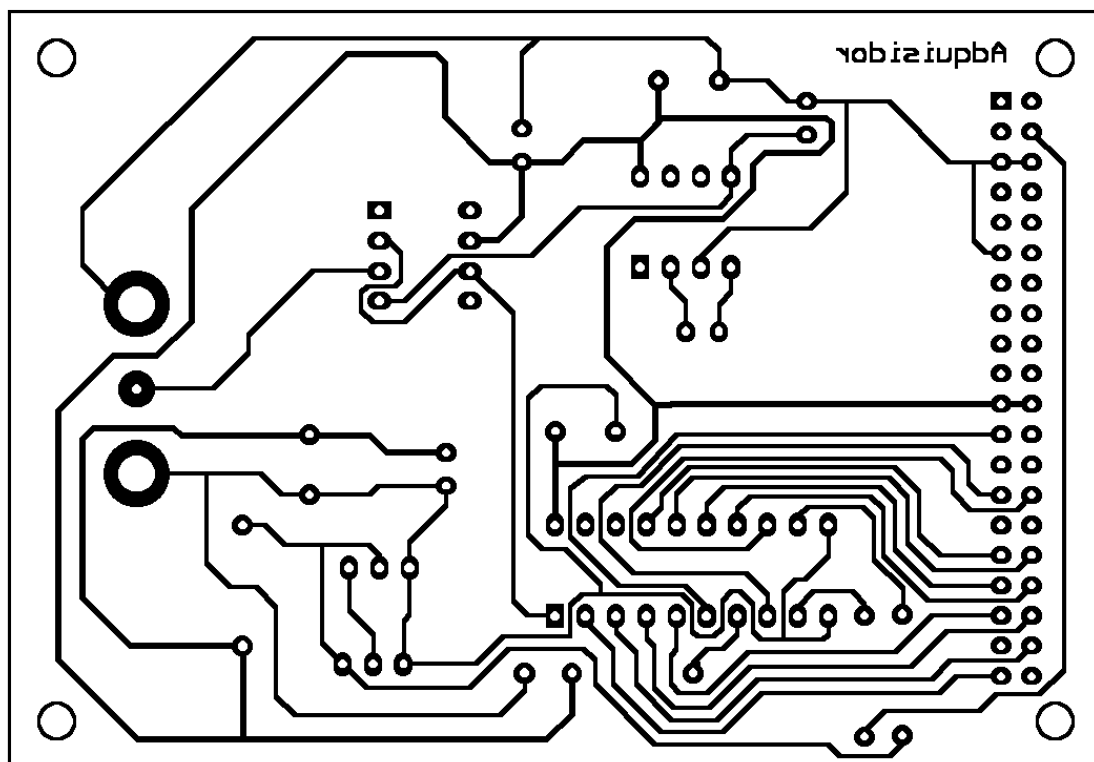


Figura 4.20: Ubicación de las pistas en la plaqueta Adquisidor.

Y la plaqueta con los componentes y las pistas, se muestran en la figura 4.21:

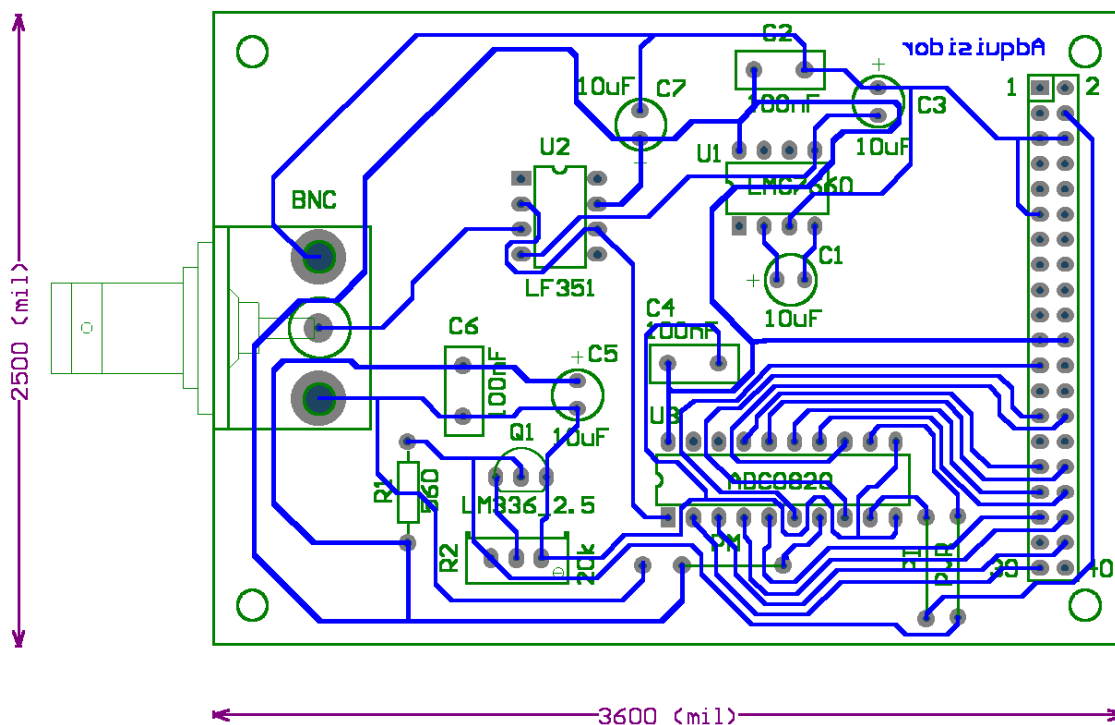


Figura 4.21: Ubicación de los componentes y las pistas en la plaqueta Adquisidor.

4.4.5.3 Plaqueta Adquisidor final.

La figura 4.22 nos muestra a la plaqueta Adquisidor armada para éste proyecto.

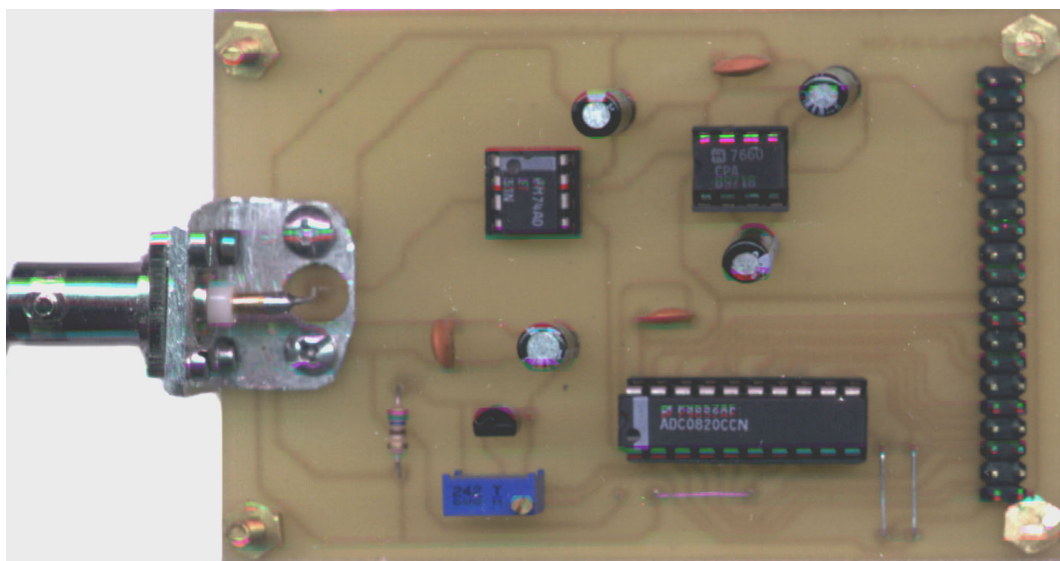


Figura 4.22: Plaqueta Adquisidor armada para el proyecto.

4.5 Diseño del Software del proyecto

Aquí vamos ahora a diseñar los programas en Visual Basic que permitan realizar una interfaz entre el usuario y la PC para manejar el Medidor de Frecuencias y Períodos y el Adquisidor autónomo de datos.

4.5.1 Programa para el manejo del Medidor de Frecuencias y Períodos.

Como podemos recordar del capítulo anterior el diseño de la comunicación se basa en la siguiente tabla de asignación de los pines del puerto paralelo:

pin	puerto	sentido	Función
D7	datos	salida	activa_prescaler
D6	datos	salida	FoP
D5	datos	salida	selec_base[1]
D4	datos	salida	selec_base[0]
D3	datos	salida	congela latches
(D2..D0)	datos	salida	selección dígito
(S6..S3)	estado	entrada	overflow y valor dígito

Tabla 4.9: Asignación de pines de puerto paralelo

Donde los pines (D7..D4) se encargan de comandar las señales de activación de prescaler, elegir en modo frecuencia o período y seleccionar la base (o señal de pulsos de período definido) a utilizar.

Por su parte los pines (D2..D0) comandan que valor debe la FPGA colocar en los pines (S6..S3) de forma como muestra la tabla 4.10:

(D2..D0)	(S6..S3)
"000" = 0	overflow
"001" = 1	valor dígito 1
"010" = 2	valor dígito 2
"011" = 3	valor dígito 3
"100" = 4	valor dígito 4
"101" = 5	valor dígito 5
"110" = 6	valor dígito 6

Tabla 4.10: Valores de las entradas (S6..S3) de acuerdo a las salidas (D2..D0).

Además el bit D3 se encarga de congelar los latches para la correcta carga de los valores de todos los dígitos

En Visual Basic inicialmente vamos a generar una ventana (formulario) llamado *Form1* con todos los controles posibles para manejar el Medidor:

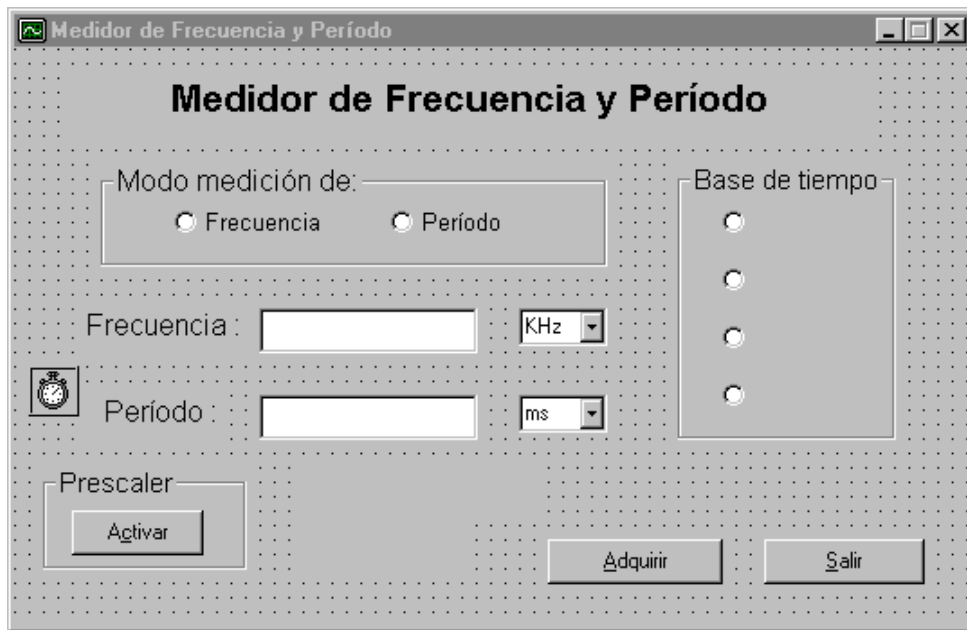


Figura 4.23: Formulario para el Medidor de Frecuencia y Período.

Ésta ventana posee, entre otros, los siguientes controles:

Control	tipo	nombre
Adquirir	Botón de comando	<i>Command1</i>
Activar	Botón de comando	<i>Command2</i>
Salir	Botón de comando	<i>Command4</i>
Modo medición de Frecuencia	Botón de opción	<i>Option1(0)</i>
Modo medición de Período	Botón de opción	<i>Option1(1)</i>
Base de Tiempo 1º	Botón de opción	<i>Option2(0)</i>
Base de Tiempo 2º	Botón de opción	<i>Option2(1)</i>
Base de Tiempo 3º	Botón de opción	<i>Option2(2)</i>
Base de Tiempo 4º	Botón de opción	<i>Option2(3)</i>

Valor de Frecuencia	Cuadro de texto	<i>Text1</i>
Valor de Período	Cuadro de texto	<i>Text2</i>
Unidades de frecuencia	Cuadro de lista (Hz, kHz y MHz)	<i>Combo2</i>
Unidades de período	Cuadro de lista (us, ms y s)	<i>Combo3</i>
Cartel de prescaler activo	Rotulo	<i>Label4</i>
Cartel de overflow	Rótulo	<i>Label5</i>
	Reloj	<i>Timer1</i>

Tabla 4.11: Controles del formulario

Inicialmente vamos a declarar las variables en un módulo llamado *Variables.bas*:

```
Global DATA As Integer
Global STATUS As Integer
Global FoP As Byte
Global Base As Byte
Global ActPres As Byte
Global Valor As Single
Global Frec As Single
Global Per As Single
Global PortAddress As Integer
Global Overflow As Byte
```

Luego declararemos la función para poder comunicarnos con el puerto paralelo, con su respectivo archivo de registro, y la guardaremos como un módulo llamado *Declaración de la DLL.bas*.

```
'Declare Inp and Out for port I/O
Public Declare Function Inp Lib "inpout32.dll" _
Alias "Inp32" (ByVal PortAddress As Integer) As Integer
Public Declare Sub Out Lib "inpout32.dll" _
Alias "Out32" (ByVal PortAddress As Integer, ByVal Value As Integer)
```

Y declararemos además otra función que nos permitirá desplazar los bits de un número binario hacia derecha (*Función.bas*).

```
'Declaración para correr n bits a la derecha
Function Shift(Dato As Byte, n As Integer) As Integer
    Shift = Fix(Dato / (2 ^ n))
End Function
```

A continuación vamos a construir las sentencias del programa. Comenzaremos con la de la carga del mismo que contendrá la asignación de variables y como va a arrancar el programa:

```
Private Sub Form_Load()
    Option1(0).Value = True 'Inicializo en modo Frecuencia
    Option2(0).Value = True 'Con la 1° base de tiempos
    Combo2.Enabled = True 'Habilito la selección de unidades de frecuencia
    Combo3.Enabled = True 'Habilito la selección de unidades de Período
    FoP = 1
    Base = 0
    ActPres = 0
    DATA = &H378 'Dirección del Data Port
    STATUS = &H379 'Dirección del Status Port
    Out DATA, Shift(ActPres, -7) + Shift(FoP, -6) + Shift(Base, -4)
End Sub
```

los botones de opción entre frecuencia y período:

```
Private Sub Option1_Click(Index As Integer)
```

```

Timer1.Enabled = False           'Detengo si estoy cargando datos
Command1.Caption = "&Adquirir"
If Option1(0).Value = True Then
    Option1(1).Value = False
    FoP = 1
    Option2(0).Caption = "10seg"
    Option2(1).Caption = "1seg"
    Option2(2).Caption = "0.1seg"
    Option2(3).Caption = "0.01seg"
Else
    Option1(0).Value = False
    FoP = 0
    Option2(0).Caption = "0.1us"
    Option2(1).Caption = "1us"
    Option2(2).Caption = "10us"
    Option2(3).Caption = "100us"
End If
Out DATA, Shift(ActPres, -7) + Shift(FoP, -6) + Shift(Base, -4)
Text1.Text = ""                 'Como cambio de modo de medición, vacío
Text2.Text = ""                 ' los valores de Frec Y Period
Label5.Caption = ""            ' vacío el Overflow
Combo2.Enabled = False 'Deshabilito la selección de unidades de frecuencia
Combo3.Enabled = False 'Deshabilito la selección de unidades de Período
End Sub

```

los de elección de la base de tiempos:

```

Private Sub Option2_Click(Index As Integer)
    Timer1.Enabled = False           'Detengo si estoy cargando datos
    Command1.Caption = "&Adquirir"
    If Option2(0).Value = True Then
        Option2(1).Value = False
        Option2(2).Value = False
        Option2(3).Value = False
        Base = 0
    End If

    If Option2(1).Value = True Then
        Option2(0).Value = False
        Option2(2).Value = False
        Option2(3).Value = False
        Base = 1
    End If

    If Option2(2).Value = True Then
        Option2(0).Value = False
        Option2(1).Value = False
        Option2(3).Value = False
        Base = 2
    End If

    If Option2(3).Value = True Then
        Option2(0).Value = False
        Option2(1).Value = False
        Option2(2).Value = False
        Base = 3
    End If

    Out DATA, Shift(ActPres, -7) + Shift(FoP, -6) + Shift(Base, -4)
    Text1.Text = ""                 'Como cambio de escala vacío los valores de Frec Y Period
    Text2.Text = ""
    Label5.Caption = ""            ' vacío el Overflow
    Combo2.Enabled = False 'Deshabilito la selección de unidades de frecuencia
    Combo3.Enabled = False 'Deshabilito la selección de unidades de Período
End Sub

```

Para activar y desactivar el prescaler:

```

Private Sub Command2_Click()
    Timer1.Enabled = False 'Detengo si estoy cargando datos
    Command1.Caption = "&Adquirir"
    If ActPres = 0 Then
        ActPres = 1
        Command2.Caption = "Desac&tivar"
        Label4.Caption = "Prescaler Activo"
    Else
        ActPres = 0
        Command2.Caption = "A&ctivar"
        Label4.Caption = ""
    End If
    Out DATA, Shift(ActPres, -7) + Shift(FoP, -6) + Shift(Base, -4)
    Text1.Text = "" 'Como cambio de escala vacio los valores de Frec Y Period
    Text2.Text = ""
    Label5.Caption = "" ' vacio el Overflow
    Combo2.Enabled = False 'Deshabilito la selección de unidades de frecuencia
    Combo3.Enabled = False 'Deshabilito la selección de unidades de Período
End Sub

```

La selección de las unidades de frecuencia:

```

Private Sub Combo2_Click()
    Select Case Combo2.ListIndex
    Case 0
        Text1.Text = CStr(Frec * 1000)
    Case 1
        Text1.Text = CStr(Frec)
    Case 2
        Text1.Text = CStr(Frec / 1000)
    End Select
End Sub

```

las de período:

```

Private Sub Combo3_Click()
    Select Case Combo3.ListIndex
    Case 0
        Text2.Text = CStr(Per * 1000)
    Case 1
        Text2.Text = CStr(Per)
    Case 2
        Text2.Text = CStr(Per / 1000)
    End Select
End Sub

```

La activación de la carga de datos:

```

Private Sub Command1_Click()
    If Timer1.Enabled = False Then
        Timer1.Enabled = True
        Command1.Caption = "&Detener"
        Combo2.Enabled = True 'Habilito la selección de unidades de frecuencia
        Combo3.Enabled = True 'Habilito la selección de unidades de Período
    Else
        Timer1.Enabled = False
        Command1.Caption = "&Adquirir"
    End If
    Out DATA, Shift(ActPres, -7) + Shift(FoP, -6) + Shift(Base, -4)
End Sub

```

Como vemos al habilitar la carga, se habilita al reloj el cual deberá cumplir la siguiente secuencia:

1. Activar la carga de valores mediante el bit D3 para congelar latches.

2. Generar en (D2..D0) la secuencia de 0 a 6 dentro de la cual se van cargando los valores de overflow y de los distintos dígitos.
3. Desactivar la carga de valores para permitir el refresco de los latches

Ésta secuencia es repetida cada 2 segundos (valor cargado en la propiedad Interval del reloj *Timer1*) mientras esté activada la carga por el botón *Command1*.

La sentencia del reloj, por lo tanto será:

```
Private Sub Timer1_Timer()
'Fuerzo 0 en bits 0, 1, y 2 para leer si hay overflow, 1 en el bit 3 para
'congelar latches, base en los bits 4 y 5, FoP en bit6, y ActPres en bit 7
Out DATA, Shift(ActPres, -7) + Shift(FoP, -6) + Shift(Base, -4) + Shift(1, -3) + 0
'Cargo en Overflow solo el bit 3 del SP previa inversión del bit 7
Overflow = (Shift(Inp(STATUS) Xor &H80, 3) And &H1)
'Fuerzo 1 en bits 0, 1 y 2 para leer el 1° digito en el SP, 1 en el bit 3
'para congelar latches, base en los bits 4 y 5 y FoP en el 6 y ActPres en el 7
Out DATA, Shift(ActPres, -7) + Shift(FoP, -6) + Shift(Base, -4) + Shift(1, -3) + 1
'Cargo en Valor solo los bits 3, 4, 5 y 6 del SP previa inversión del bit 7
Valor = Shift(Inp(STATUS) Xor &H80, 3) And &HF
'Fuerzo 2 en bits 0, 1 y 2 para leer el 2° digito en el SP, 1 en el bit 3
'para congelar latches, base en los bits 4 y 5 y FoP en el 6 y ActPres en el 7
Out DATA, Shift(ActPres, -7) + Shift(FoP, -6) + Shift(Base, -4) + Shift(1, -3) + 2
Valor = Valor + (Shift(Inp(STATUS) Xor &H80, 3) And &HF) * 10
'Fuerzo 3 en bits 0, 1 y 2 para leer el 3° digito en el SP, 1 en el bit 3
'para congelar latches, base en los bits 4 y 5 y FoP en el 6 y ActPres en el 7
Out DATA, Shift(ActPres, -7) + Shift(FoP, -6) + Shift(Base, -4) + Shift(1, -3) + 3
Valor = Valor + (Shift(Inp(STATUS) Xor &H80, 3) And &HF) * 100
'Fuerzo 4 en bits 0, 1 y 2 para leer el 4° digito en el SP, 1 en el bit 3
'para congelar latches, base en los bits 4 y 5 y FoP en el 6 y ActPres en el 7
Out DATA, Shift(ActPres, -7) + Shift(FoP, -6) + Shift(Base, -4) + Shift(1, -3) + 4
Valor = Valor + (Shift(Inp(STATUS) Xor &H80, 3) And &HF) * 1000
'Fuerzo 5 en bits 0, 1 y 2 para leer el 5° digito en el SP, 1 en el bit 3
'para congelar latches, base en los bits 4 y 5 y FoP en el 6 y ActPres en el 7
Out DATA, Shift(ActPres, -7) + Shift(FoP, -6) + Shift(Base, -4) + Shift(1, -3) + 5
Valor = Valor + (Shift(Inp(STATUS) Xor &H80, 3) And &HF) / 1 * 10000
'Fuerzo 6 en bits 0, 1 y 2 para leer el 6° digito en el SP, 1 en el bit 3
'para congelar latches, base en los bits 4 y 5 y FoP en el 6 y ActPres en el 7
Out DATA, Shift(ActPres, -7) + Shift(FoP, -6) + Shift(Base, -4) + Shift(1, -3) + 6
Valor = Valor + (Shift(Inp(STATUS) Xor &H80, 3) And &HF) * 100000
Out DATA, Shift(ActPres, -7) + Shift(FoP, -6) + Shift(Base, -4)'Desactivo congela
latches

Select Case Base 'De acuerdo a la base de tiempos que usemos es la escala
Case 0
    Valor = Valor / 10000
Case 1
    Valor = Valor / 1000
Case 2
    Valor = Valor / 100
Case 3
    Valor = Valor / 10
End Select

If FoP = 1 Then 'Medición de Frecuencia
    If ActPres = 1 Then 'Si actúa el prescaler exterior
        Valor = Valor * 10 'El valor real de frec. es 10 veces el leído
    End If
    Frec = Valor
    If Valor = 0 Then
        Per = 0
    Else
        Per = 1 / Valor
    End If
Else 'Medición de Período
    If ActPres = 1 Then 'Si actúa el prescaler exterior
        Valor = Valor / 10 'El valor real de Per. es 0.1 veces el leído
    End If
End If
```

```

Per = Valor
If Valor = 0 Then
    Frec = 0
Else
    Frec = 1 / Valor
End If
End If
If Overflow Then
    Label5.Caption = "Overflow" 'Si hubo Overflow
Else
    Label5.Caption = "" 'Si no hubo Overflow
End If

Select Case Combo2.ListIndex
Case 0
    Text1.Text = Frec * 1000
Case 1, -1
    Text1.Text = Frec
Case 2
    Text1.Text = Frec / 1000
End Select

Select Case Combo3.ListIndex
Case 0
    Text2.Text = CStr(Per * 1000)
Case 1, -1
    Text2.Text = CStr(Per)
Case 2
    Text2.Text = CStr(Per / 1000)
End Select

End Sub

```

Y finalmente para salir del programa:

```

Private Sub Command4_Click()
    End
End Sub

```

Luego la figura 4.24 muestra un ejemplo de la medición de una frecuencia con overflow, prescaler activo y base de tiempos de 1segundo:

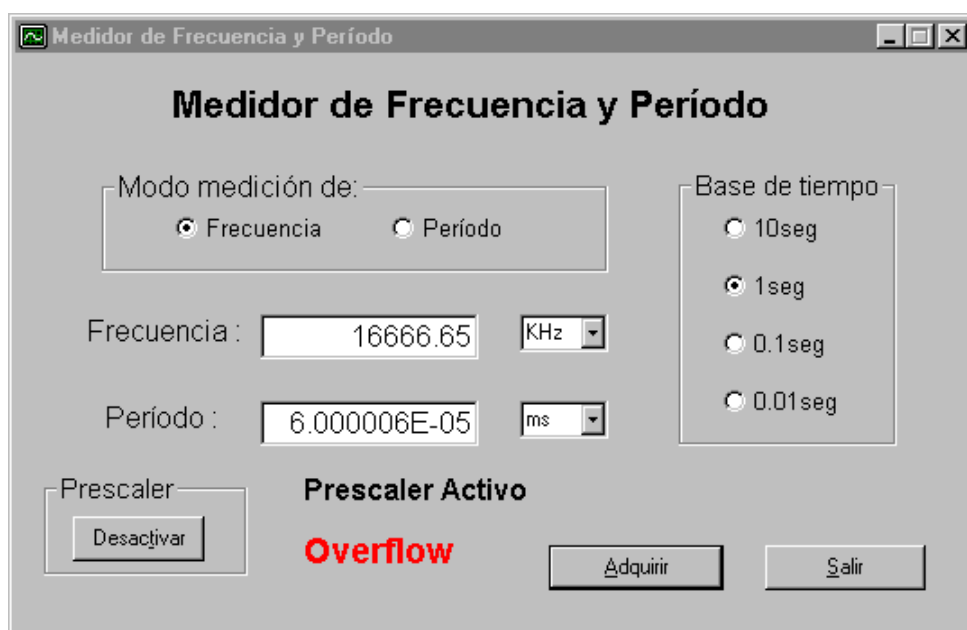


Figura 4.24: Ejemplo de medición de frecuencia.

4.5.2 Programa para el manejo del Adquisidor autónomo de datos.

Del capítulo anterior, el diseño de la comunicación se basa en la siguiente tabla de asignación de los pines del puerto paralelo:

pin	puerto	sentido	nodo
D0	datos	salida	clear
D1	datos	salida	cargar_fifo
D2	datos	salida	leer_fifo
D3	datos	salida	HoL_nibble
D4	datos	salida	clk_PC
S7	estado	entrada	full
(S6..S3)	estado	entrada	datos[3..0]

Tabla 4.12: Asignación de pines de puerto paralelo

En Visual Basic inicialmente vamos nuevamente a generar una ventana (formulario) llamado *Form1* con todos los controles posibles para manejar el Adquisidor:

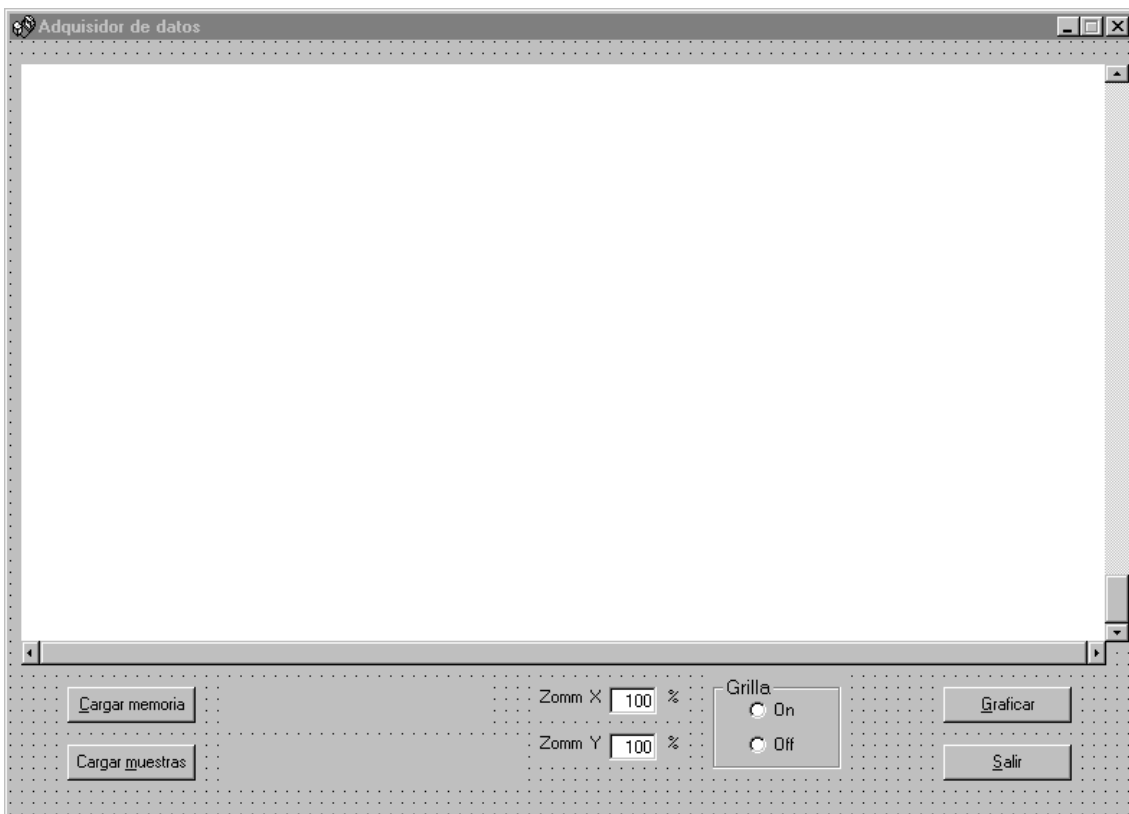


Figura 4.25: Formulario para el Adquisidor autónomo de datos.

Ésta ventana posee, entre otros, los siguientes controles:

Control	tipo	nombre
Salir	Botón de comando	<i>Command1</i>
Graficar	Botón de comando	<i>Command2</i>
Cargar memoria	Botón de comando	<i>Command3</i>

Cargar muestras	Botón de comando	<i>Command4</i>
Grilla on	Botón de opción	<i>Option1(0)</i>
Grilla off	Botón de opción	<i>Option1(1)</i>
Ventana de gráfico	Cuadro de imagen	<i>Picture1</i>
Barra de desplazamiento horizontal	Barra de desplazamiento H	<i>Hscroll1</i>
Barra de desplazamiento vertical	Barra de desplazamiento V	<i>Vscroll1</i>
Zoom X	Cuadro de texto	<i>Text3</i>
Zoom Y	Cuadro de texto	<i>Text1</i>
Cartel de Memoria de FPGA llena	Rotulo	<i>Label3</i>
Cartel de Muestras cargadas en PC	Rótulo	<i>Label2</i>

Tabla 4.13: Controles del formulario

De la misma forma que para el Medidor de Frecuencias y períodos, declaramos las variables en un módulo llamado *Variables.bas*:

```
Global Const NumMuestra = 500
Global Valor (NumMuestra - 1) As Integer
Global n As Double
Global DATA As Integer
Global STATUS As Integer
Global Muestras As Double
Global Full As Integer
Global Gridon As Boolean
Global ValorX As Double
Global ValorY As Double
```

Y cargamos los mismos módulos *Declaración de la DLL.bas.* y *Función.bas*:

```
'Declare Inp and Out for port I/O
Public Declare Function Inp Lib "inpout32.dll" _
Alias "Inp32" (ByVal PortAddress As Integer) As Integer
Public Declare Sub Out Lib "inpout32.dll" _
Alias "Out32" (ByVal PortAddress As Integer, ByVal Value As Integer)

'Declaración para correr n bits a la derecha
Function Shift (Dato As Byte, n As Integer) As Integer
    Shift = Fix (Dato / (2 ^ n))
End Function
```

Posteriormente vamos a construir las sentencias del programa. Nuevamente comenzamos con la de la carga del mismo que contendrá la asignación de variables y como va a arrancar el programa:

```
Private Sub Form_Load()
    Form1.Left = 100
    Form1.Top = 100
    DATA = &H378
    STATUS = &H379
    Full = 0
    Option1(0).Value = True
    Command2.Enabled = False
    Text1.Enabled = False
    Text3.Enabled = False
    Option1(0).Enabled = False
    Option1(1).Enabled = False
    Command4.Enabled = False
End Sub
```

Las opciones de grilla:

```
Private Sub Option1_Click (Index As Integer)
    If Option1(0).Value = True Then
```

```

        Option1(1).Value = False
        Gridon = True
    Else
        Option1(0).Value = False
        Gridon = False
    End If
    Call Command2_Click
End Sub

```

Luego a través del botón de carga de memoria deberemos generar la siguiente secuencia:

1. Puesta en '1' del bit D0 para limpiar la memoria.
2. Se activa la carga de la memoria mediante el bit D1 hasta que el bit S7 se ponga en '1' (memoria llena).

En Visual Basic:

```

Private Sub Command3_Click()
    Label2.Caption = "" 'Todavía no se cargaron en la pc
    Picture1.Cls 'Borro el gráfico
    Command3.Caption = "Cargando"
    Out DATA, 1 'Pongo clear en 1 (bit D0) para limpiar la memoria
    Out DATA, Shift(1, -1) 'Pongo cargar_fifo en 1 (bit D1) para cargar memoria
    Do
        Full = (Shift(Inp(STATUS) Xor &H80, 7) And &H1) 'Leo si está llena en bit S7
    Loop While Full = 0
    Command3.Caption = "&Cargar memoria"
    Label3.Caption = "Memoria FPGA Llena"
    Command2.Enabled = False
    Text1.Enabled = False
    Text3.Enabled = False
    Option1(0).Enabled = False
    Option1(1).Enabled = False
    Command4.Enabled = True
End Sub

```

Una vez cargada la memoria, procedemos a cargar las muestras en la PC, con lo cual la secuencia debe seguir de la forma:

3. Se activa la lectura de la memoria mediante el bit D2.
4. Con el bit D3 en bajo, se genera un pulso de reloj en el bit D4 para cargar los cuatro bits mas altos de la muestra a través de los bits (S6..S3).
5. Se pone el bit D3 en alto y se cargan los cuatro bits menos significativos de la muestra a través de los bits (S6..S3).

Volcándolo a Visual Basic:

```

Private Sub Command4_Click()
    Label3.Caption = "" 'Ya que la FPGA no está mas llena
    For n = 0 To NumMuestra - 1
        Out DATA, Shift(1, -2)
        'Bit D2 en 1 para leer FIFO y bit D3 en 0 para nivel bajo en clock
        Out DATA, Shift(1, -2) + Shift(1, -4)
        'Bit D2 en 1 para leer FIFO y bit D4 en 1 para nivel alto en clock
        Valor(n) = Shift(Inp(STATUS) Xor &H80, -1) And &HF0 'Leo el High Nibble
        Out DATA, Shift(1, -2) + Shift(1, -3)
        'Bit D2 en 1 para leer FIFO, bit D4 en 0 para nivel bajo en clock y bit D3
        'en 1 para Low Nibble
        Valor(n) = Valor(n) + (Shift(Inp(STATUS) Xor &H80, 3) And &HF)
        'Leo el Low Nibble
    Next n
    Label2.Caption = "Muestras cargadas en PC"
    Command4.Enabled = False
    Command2.Enabled = True
End Sub

```

A continuación a través del botón Graficar, debemos graficar las muestras en el cuadro de imágenes:

```

Private Sub Command2_Click()
    Picture1.Cls
    ZoomX = Val(Text3.Text) / 100
    ZoomY = Val(Text1.Text) / 100

    If ZoomX > 1 Then
        HScroll1.Visible = True
    Else
        HScroll1.Visible = False
    End If

    If ZoomY > 1 Then
        VScroll1.Visible = True
    Else
        VScroll1.Visible = False
    End If

    HScroll1.Max = (ZoomX - 1) * 10
    VScroll1.Min = -(ZoomY - 1) * 10

    ` EJE X
    For n = 0 To 10000 Step (10000 / 12) 'tengo que meter 12 pasos en 10000 pixels
        Picture1.Line (ZoomX * n + 500 - 1000 * ValorX, 5550)-(ZoomX * n + 500 -
1000 * ValorX, 5450)
        Picture1.Line (ZoomX * n + 400 - 1000 * ValorX, 5600)-(ZoomX * n + 401 -
1000 * ValorX, 5600)
        Muestras = n / 20 'ya que en 10000/(500 muestras) = 20
        Picture1.Print (Muestras * 0.0024)
        'Ya que tenemos una muestra cada 2.4us, y la escala es en ms
        If Gridon = True Then
            Picture1.Line (ZoomX * n + 500 - 1000 * ValorX, 300)-(ZoomX * n + 500 -
1000 * ValorX, 5500), &HC0C0C0 'Grilla en y color gris
        End If
    Next n

    ` EJE Y
    For n = 0 To 5000 Step 1000
        Picture1.Line (450, 5500 - ZoomY * n - 500 * ValorY)-(550, 5500 - ZoomY * n
- 500 * ValorY)
        Picture1.Line (50, 5400 - ZoomY * n - 500 * ValorY)-(50, 5400 - ZoomY * n -
500 * ValorY)
        Picture1.Print 2.5 * n / 5000
        If Gridon = True Then
            Picture1.Line (500, 5500 - ZoomY * n - 500 * ValorY)-(11000, 5500 -
ZoomY * n - 500 * ValorY), &HC0C0C0 'Grilla en x color gris
        End If
    Next n

    Picture1.Line (500 - 1000 * ValorX, 200)-(500 - 1000 * ValorX, 5500) 'Eje Y
    Picture1.Line (500, 5500 - 500 * ValorY)-(11000, 5500 - 500 * ValorY) 'Eje X
    Picture1.Line (10800, 5750)-(10801, 5750)
    Picture1.Print "[ms]"

    For n = 0 To NumMuestra - 1
        Picture1.PSet ((10 * ZoomX * n) * 2 + 500 - 1000 * ValorX, 5500 - ZoomY *
5000 / 255 * Valor(n) - 500 * ValorY), &HFF
    Next n

    Text1.Enabled = True
    Text3.Enabled = True
    Option1(0).Enabled = True
    Option1(1).Enabled = True

End Sub

```

Las barras de desplazamiento vertical y horizontal:

```
Private Sub VScroll1_Change()  
    ValorY = VScroll1.Value  
    Call Command2_Click  
End Sub
```

```
Private Sub HScroll1_Change()  
    ValorX = HScroll1.Value  
    Call Command2_Click  
End Sub
```

Los cuadros de zoom de X y de Y:

```
Private Sub Text3_Change()  
    Call Command2_Click  
End Sub
```

```
Private Sub Text1_Change()  
    Call Command2_Click  
End Sub
```

Y finalmente el botón de salida del programa:

```
Private Sub Command1_Click()  
    End  
End Sub
```

Vamos ahora a modo de ejemplo de funcionamiento del programa, a cargar las muestras de una señal de continua de 2.5V.

Inicialmente abrimos el programa:

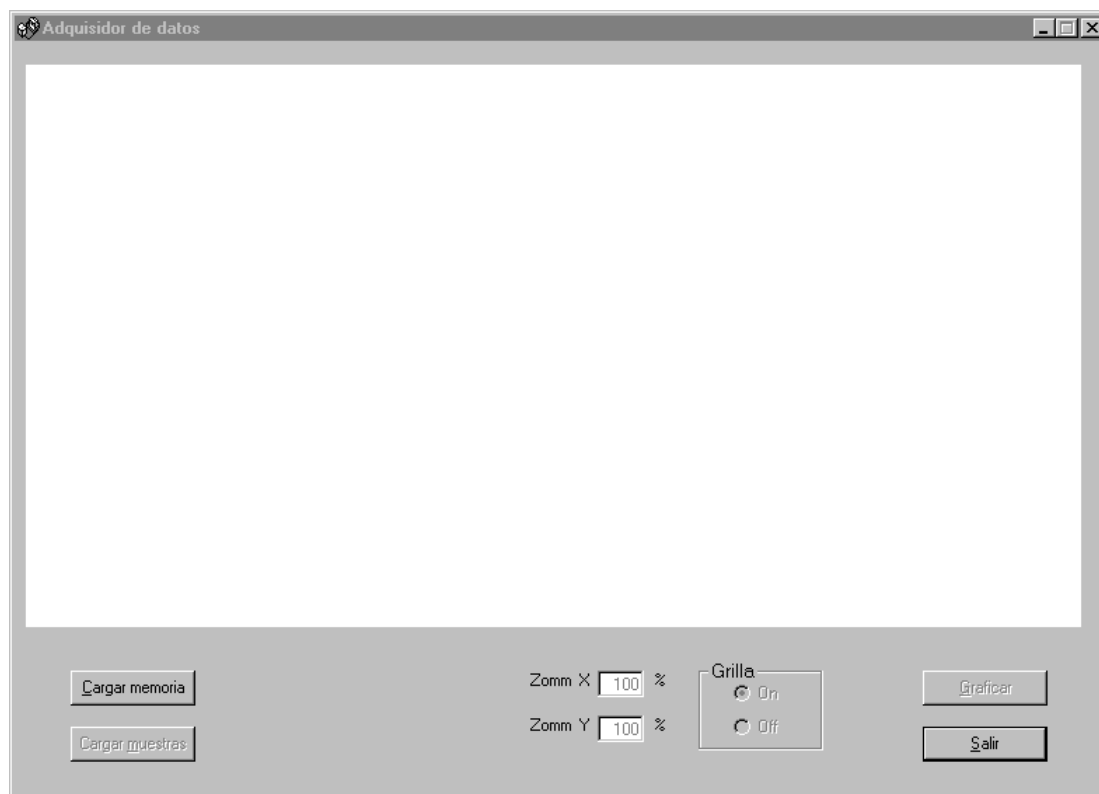


Figura 4.26: Formulario para el Adquisidor autónomo de datos.

luego cargamos los valores de las muestras en la FPGA hasta que la misma se llene:

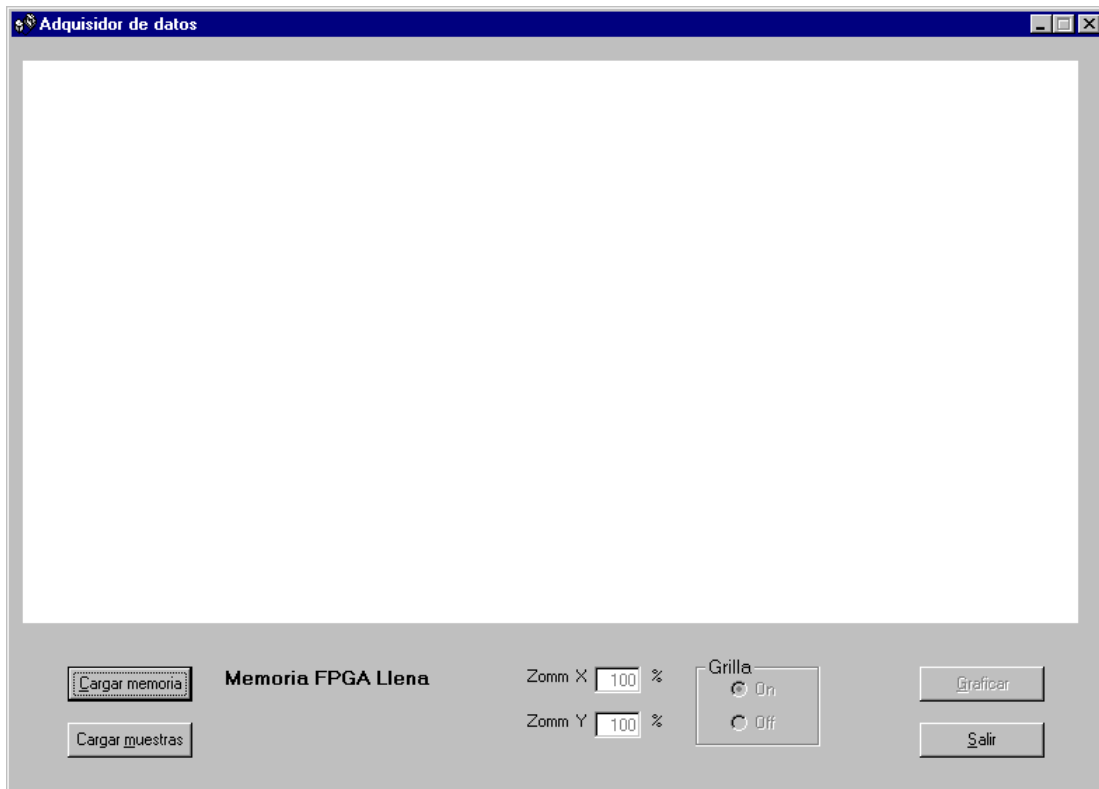


Figura 4.27: Formulario para el Adquisidor autónomo de datos.

Posteriormente cargamos las muestras en la PC:

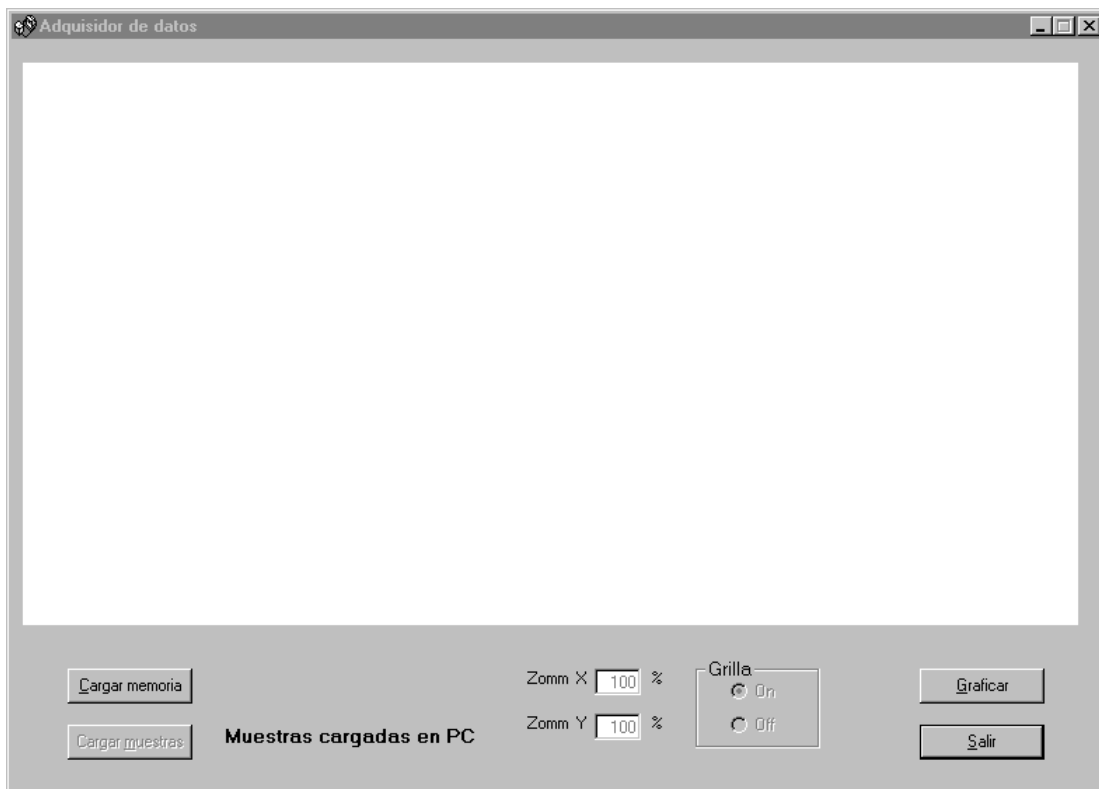


Figura 4.28: Formulario para el Adquisidor autónomo de datos.

graficamos las muestras:

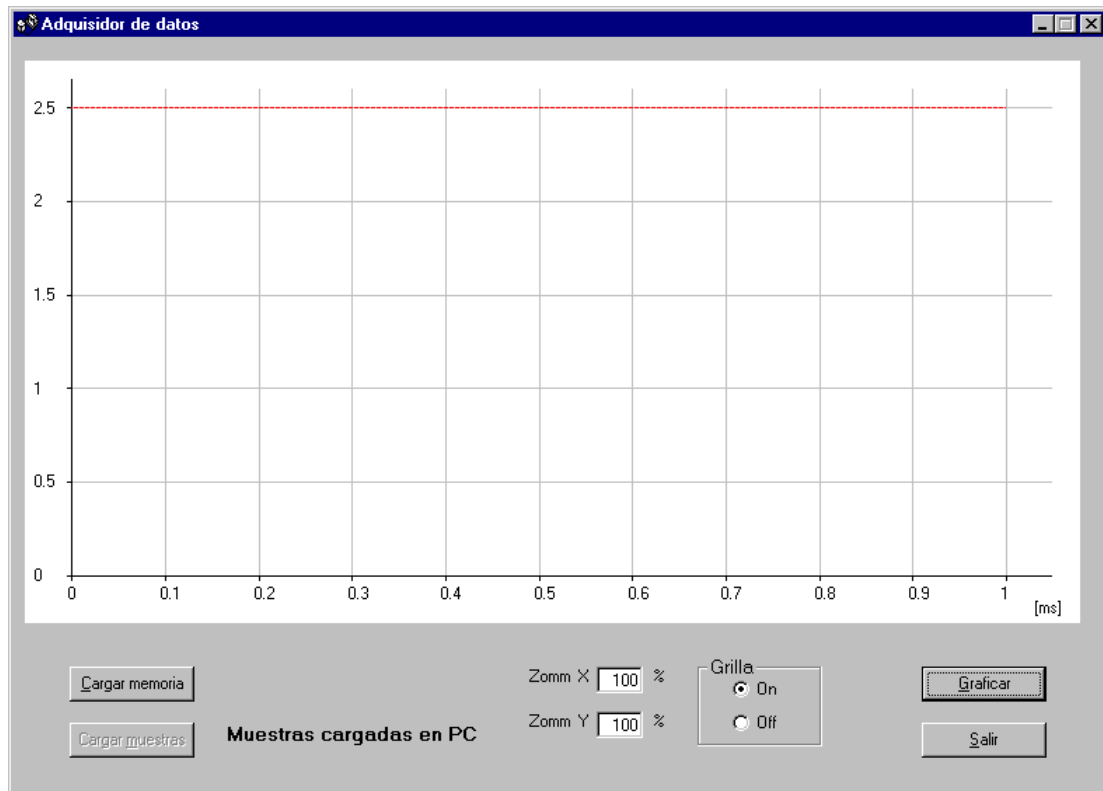


Figura 4.29: Formulario para el Adquisidor autónomo de datos.

Y finalmente efectuamos un zoom:

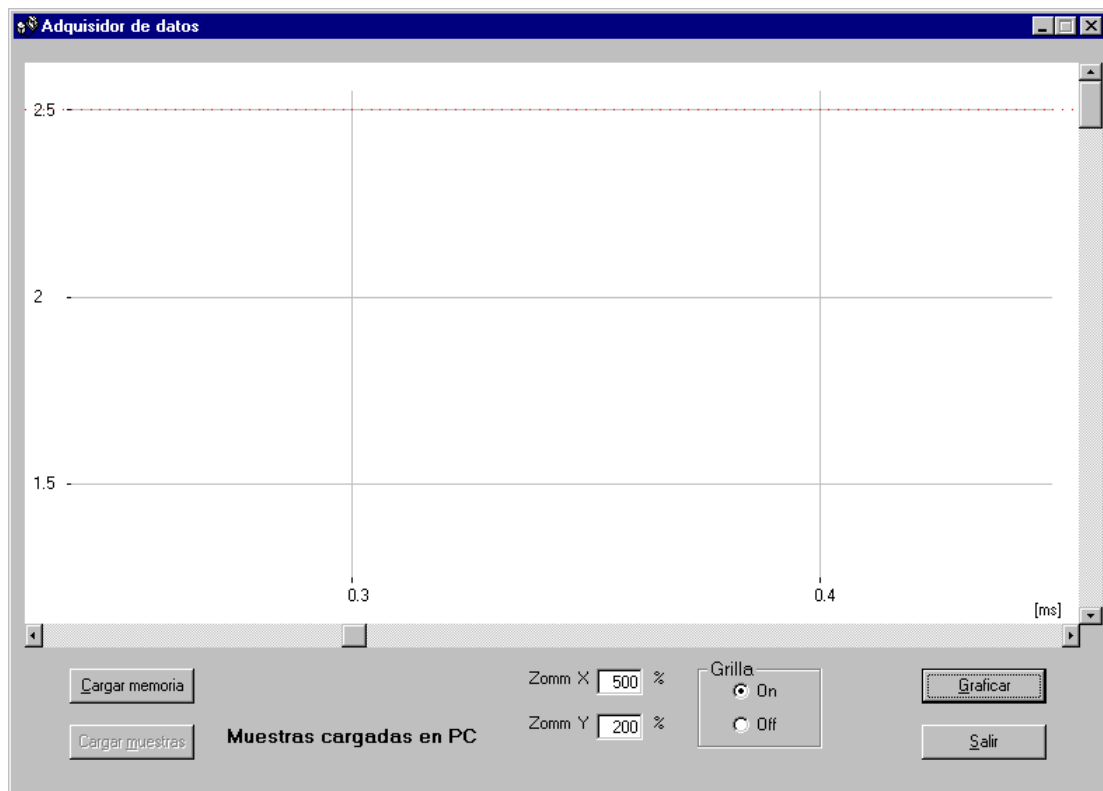


Figura 4.30: Formulario para el Adquisidor autónomo de datos.

4.6 Armado y verificación del proyecto.

En este punto se procede a la compilación del proyecto y programación en la FPGA, el ensamblado de las placas y la verificación del correcto funcionamiento.

4.6.1 Compilación del proyecto y programación de la FPGA a través del software MAX+plus II

4.6.1.1 Compilación del proyecto.

Vamos inicialmente a compilar el proyecto llamado *proyecto1.tdf*, el cual es el archivo *proyecto.tdf* con las reformas del punto 4.4.2; asignándole el dispositivo EPF10K10LC84-3 de la familia FLEX10K.

Del archivo de reporte podemos verificar la cantidad de lógica utilizada:

```

** DEVICE SUMMARY **

Chip/
POF      Device          Input Output Bidir  Memory  Memory          LCs
          POF          Pins  Pins  Pins  Bits % Utilized  LCs  % Utilized

proyecto1
          EPF10K10LC84-3    24    20    0   4096    66 %    552    95 %

User Pins:          24    20    0

Total dedicated input pins used:          2/6    ( 33%)
Total I/O pins used:          42/53    ( 79%)
Total logic cells used:          552/576    ( 95%)
Total embedded cells used:          16/24    ( 66%)
Total EABs used:          2/3    ( 66%)
Average fan-in:          3.27/4    ( 81%)
Total fan-in:          1806/2304    ( 78%)

```

como vemos hemos utilizado el 95% de las celdas lógicas (552 de 576) y 2 EABs de 3 que posee (para implementar la FIFO)

4.6.1.2 Asignación de los pines a la FPGA

Vamos ahora a asignar los pines del proyecto a las patas del EPF10K10LC84-3 a través del *Floorplan Editor* del MAX+plus II. Esta asignación debe cumplir con las tabla 6 y 7, las cuales reordenándolas según las patas de la FPGA, queda como muestra la tabla 4.14:

EPF10K10	CON1	CON2	Pin name	Pin del proyecto
1	1	1	Dedicated Clock 1	reloj_xtal
2	11	-	Dedicated Input 0	Frec o Adqui
3	14	14	Global Clear	reset
4	-	-	VCCINT	
5	30	-	I/O 37	f
6	31	-	I/O 38	DB7
7	32	-	I/O 39	DB6
8	33	-	I/O 40	DB5
9	34	-	I/O 41	DB4
10	35	-	I/O 42	DB3
11	36	-	I/O 43	DB2
12	-	-	DATA 0	
13	-	-	DCLK	
14	-	-	NCE	
15	-	-	TDI	
16	2	38	I/O 00	pulsador_selbase
17	4	39	I/O 01	Frec o Period o /INT
18	7	-	I/O 02	sel_digito1
19	8	-	I/O 03	activa_prescaler
20	-	-	VCCINT	
21	9	-	I/O 04	overflow
22	10	-	I/O 05	sel_digito2
23	13	-	I/O 06	sel_digito3
24	15	-	I/O 07	sel_digito4
25	16	-	I/O 08	
26	-	-	GNDINT	
27	17	-	I/O 09	sel_digito5
28	19	-	I/O 10	sel_digito6
29	20	-	I/O 11	dp
30	23	-	I/O 12	c_o_/WR
31	-	-	MSEL 0	
32	-	-	MSEL 1	
33	-	-	VCCINT	
34	-	-	nCONFIG	
35	24	-	I/O 13	e
36	25	-	I/O 14	d
37	27	-	I/O 15	b_o_/RD
38	28	-	I/O 16	a_o_/CS
39	29	-	I/O 17	g
40	-	-	VCCINT	
41	-	-	GNDINT	
42	12	-	Dedicated Input 1	
43	-	-	Dedicated Clock 2	
44	-	11	Dedicated Input 2	
45	-	-	VCCINT	
46	-	-	GNDINT	
47	-	2	I/O 18	data_port0
48	-	4	I/O 19	data_port1
49	-	7	I/O 20	data_port2
50	-	8	I/O 21	data_port3
51	-	9	I/O 22	data_port4

52	-	10	I/O 23	data_port5
53	-	13	I/O 24	data_port6
54	-	15	I/O 25	data_port7
55	-	-	nSTATUS	
56	-	-	TRST	
57	40	40	TMS	
58	-	16	I/O 26	status_port3
59	-	17	I/O 27	status_port4
60	-	19	I/O 28	status_port5
61	-	20	I/O 29	status_port6
62	-	23	I/O 30	status_port7
63	-	-	VCCINT	
64	-	24	I/O 31	
65	-	25	I/O 32	manual_PC
66	-	27	I/O 33	
67	-	28	I/O 34	
68	-	-	GNDINT	
69	37	-	I/O 44	signal_in
70	-	31	I/O 45	
71	38	29	I/O 35	DB1
72	39	30	I/O 36	DB0
73	-	32	I/O 46	
74	-	-	TDO	
75	-	-	nCEO	
76	-	-	CONF_DONE	
77	26	26	TCK	
78	-	33	I/O 47	
79	-	34	I/O 48	
80	-	35	I/O 49	
81	-	36	I/O 50	
82	-	-	GNDINT	
83	-	37	I/O 51	
84	-	12	Dedicated Input 3	

Tabla 4.14: Asignación de pines a la EPF10K10LC84

Volcándolo al proyecto a través del *Foorplan Editor*, el chip queda con sus pines distribuidos de la forma como muestra la figura 4.31:

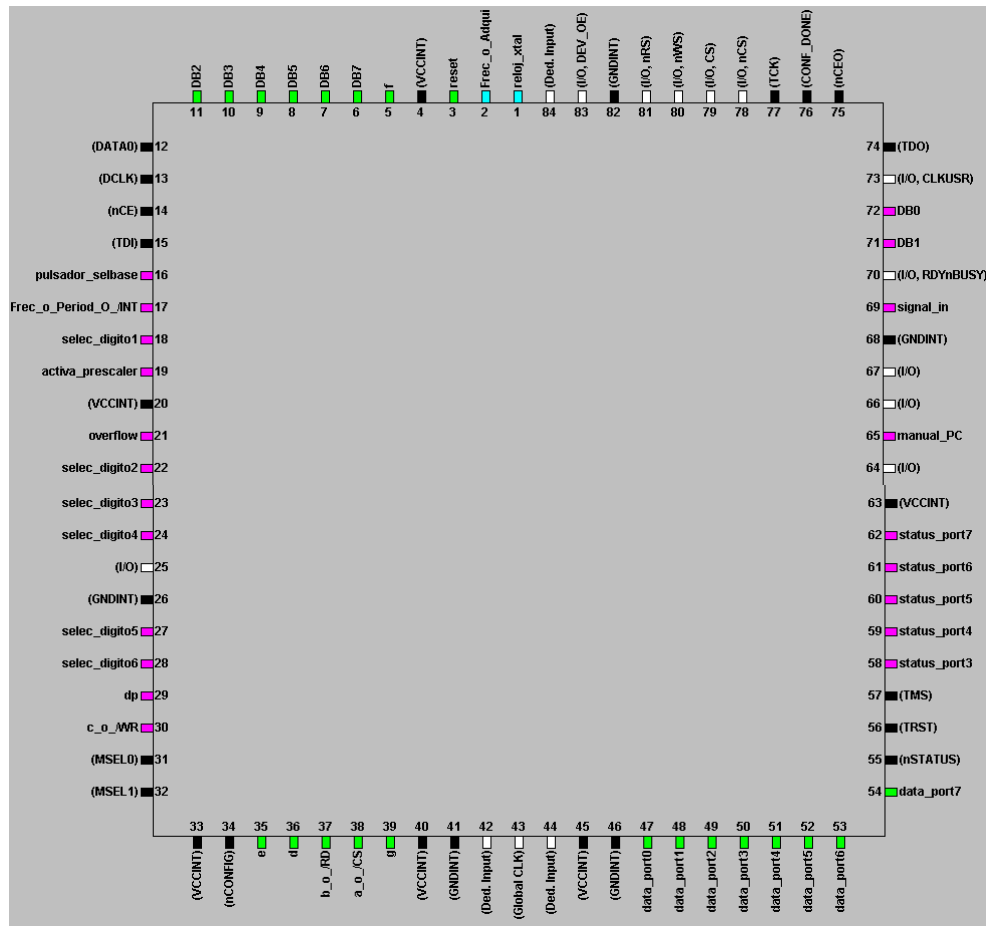


Figura 4.31: Asignación de pines en el Foorplan Editor.

Luego debemos recompilar el proyecto para que tome la asignación de las patas del chip.

4.6.1.3 Programación de la FPGA.

Posteriormente vamos a proceder a programar la FPGA. Para esto vamos a programar la EPC2 de la UPx10K10 a través de la interfase *ByteBlaster*. Los pasos a seguir son los siguientes:

1. Verificamos que la EPC2 está en su zócalo, y que el jumper J1-EPC2 no está colocado.
2. Como no hay plaquetas enchufadas en CON1 o CON2, verificamos que también los jumpers J1-CON1 y J1-CON2, respectivamente, estén colocados.
3. Conectamos el cable de impresora entre la PC y la UPx10K10, y conectamos la fuente comprobando que tiene energía (LED encendido).
4. Generamos el archivo .pof de programación de la EPC2. Para ello:
 - Con la ventana del *Compiler* abierta, vamos a la opción *File* de la línea superior y elija *Convert SRAM Object File*.
 - Elegimos el archivo .sof a convertir y lo ingresamos mediante *Add* a la lista

- Mediante el botón *Output File Options* abrimos la ventana de igual nombre, eligiendo como EPROM a la EPC2LC20, activando la opción *Use Configuration Eprom Pull Up Resistor*, y luego *OK*.
 - Cerramos la ventana *Convert SRAM Object File* apretando *OK*.
 - El MAX+PLUS II preguntará si queremos sobrescribir un eventual .pof preexistente; para lo cual elegimos *YES*
5. Elegimos el *Programmer* en el MAX+PLUS II, vamos a *OPTIONS*→*Hardware Setup*, y seleccionamos como programador el *ByteBlaster*.
 6. Aún en el *Programmer* en el MAX+PLUS II, colocamos en *ON* la opción *MultiDevice JTAG Chain* en el sub-menú *JTAG*.
 7. Elegimos *MultiDevice JTAG Chain Setup* en ese submenú.
 8. Seleccionamos *EPC2* en la ventana *Device Name*.
 9. Escribimos el nombre del archivo de programación en la ventana *Programming File Name* (para esta tarea también podemos usar el botón *Select Programming File*). Este archivo puede tener la extensión .pof, aunque también .jam o .jbc.
 10. Una vez elegidos el dispositivo y el archivo de programación, apretamos el botón *Add* para incorporar ambos a la ventana *Device Names & Programming File Names*.
 11. Seleccionamos ahora *EPF10K10* en la ventana *Device Name*.
 12. Borrarnos el contenido de la ventana *Programming File Name* y apretamos el botón *Add* para incorporar ambos a la ventana *Device Names & Programming File Names*. En la zona correspondiente al archivo asociado a la EPF10K10 aparecerá como texto *<none>*.
 13. Si todo se ha realizado correctamente, en la ventana *Device Names & Programming File Names* debe estar en el puesto 1º la EPF10K10 (con *<none>*) y en 2º lugar la EPC2 (con su archivo .pof) como muestra la figura 4.32.

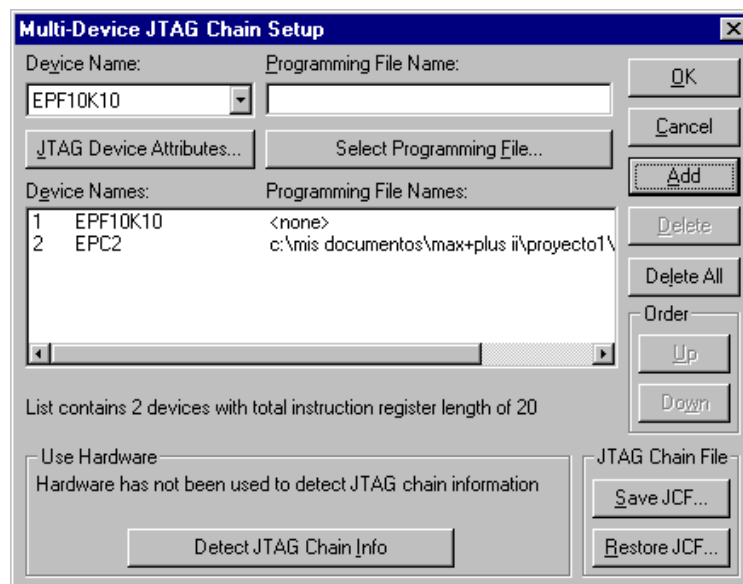


Figura 4.32: Ventana de MultiDevice JTAG Chain.

14. Apretamos el botón *Detect JTAG Chain Info* para verificar si todo funciona bien. A través del *ByteBlaster* interno de la UPx10K10 el MAX+PLUS II debe detectar la EPF10K10 y la EPC2.

15. Archivamos estas opciones mediante el botón Save .JCF (*JTAG Configuration File*).
16. Finalmente volvemos al *Programmer* en el MAX+PLUS II, y elegimos *Program*.

4.6.2 Armado y verificación del sistema Medidor de Frecuencias y Períodos

4.6.2.1 Armado del sistema Medidor de Frecuencia y Período.

Vamos ahora a proceder al armado del sistema Medidor de frecuencia y período. Para esto vamos a conectar la plaqueta Frecuencímetro al CON1 de la Upx10K10, y la placa FPGA con puerto paralelo de PC al CON2. La figura 4.33 muestra una imagen del sistema armado.

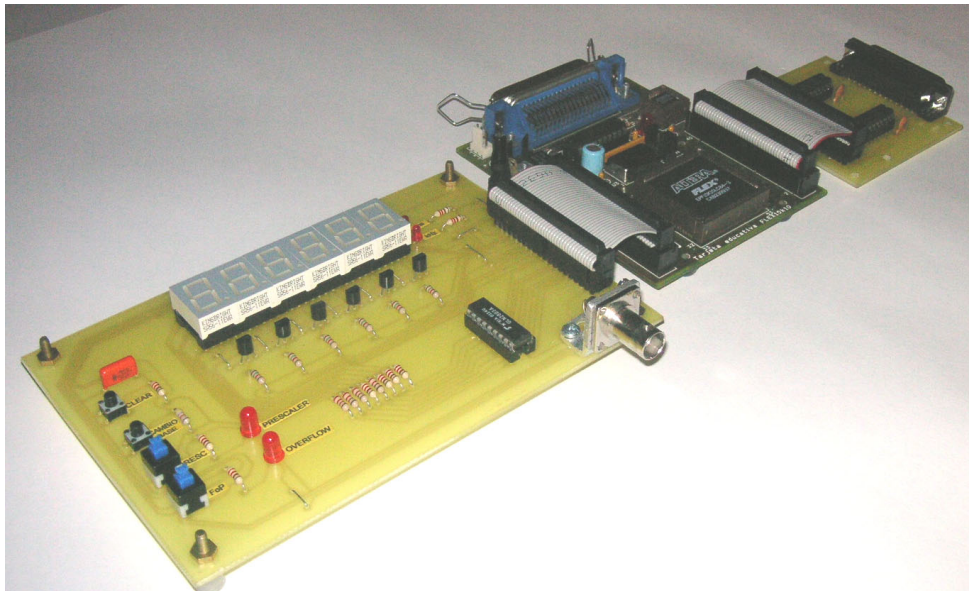


Figura 4.33: Sistema Medidor de frecuencia y período en modo manual.

La figura anterior muestra al sistema en modo manual. Para pasar al modo PC debemos conectar el cable que va al puerto paralelo de la PC como muestra la figura 4.34.

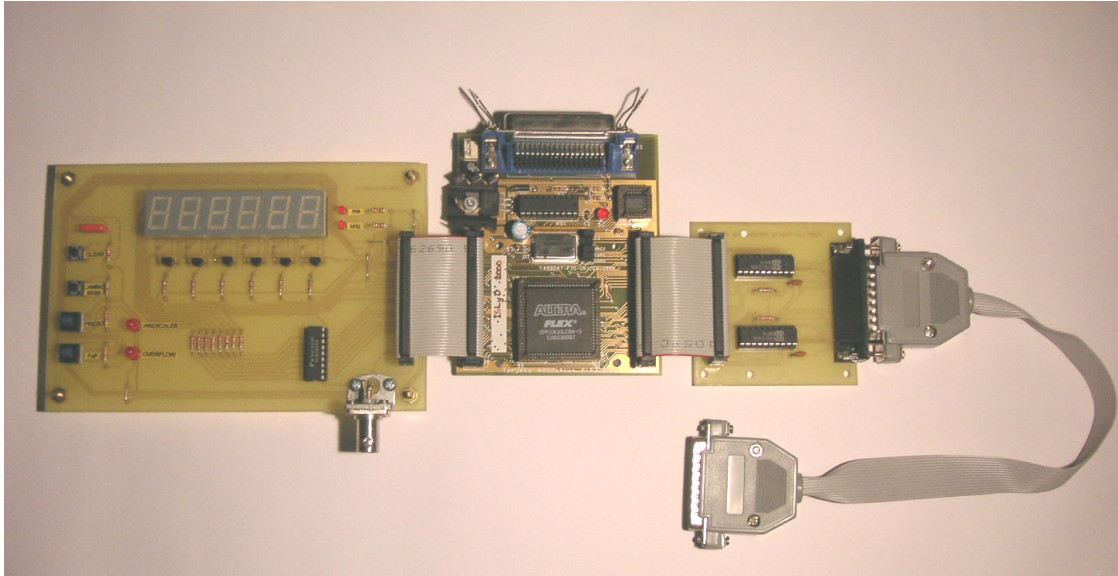


Figura 4.34: Sistema Medidor de frecuencia y período en modo PC.

4.6.2.2 Cálculo de errores del sistema Medidor de frecuencia y período.

Como se detalló en el capítulo anterior, existen dos tipos de errores del medidor:

- Error de la base de tiempos o de la señal de período definido: este error esta dado pura y exclusivamente por la estabilidad del oscilador externo de 10MHz.
- Error en el número de pulsos contados.

El error en el número de pulsos contados fue abordado en el capítulo anterior llegando a la conclusión que en peor de los casos, los contadores cuentan un pulso de más o bien uno de menos, con lo cual el error es de \pm el dígito menos significativo del display. Es decir que según la base elegida, este error tome los valores que muestran las tablas 4.15, 4.16, 4.17 y 4.18.

selec_base[1..0]	Duración del nivel alto de la base de tiempos	error [Hz]
"00" = 0	10000 ms	± 0.1
"01" = 1	1000 ms	± 1
"10" = 2	100 ms	± 10
"11" = 3	10 ms	± 100

Tabla 4.15: Error en frecuencias para cada base de tiempos, sin prescaler

selec_base[1..0]	Período	error [us]
"00" = 0	0.1 us	± 0.1
"01" = 1	1us	± 1
"10" = 2	10 us	± 10
"11" = 3	100 us	± 100

Tabla 4.16: Error para cada señal de período definido, sin prescaler

selec_base[1..0]	Duración del nivel alto de la base de tiempos	error [Hz]
"00" = 0	10000 ms	± 1
"01" = 1	1000 ms	± 10
"10" = 2	100 ms	± 100
"11" = 3	10 ms	± 1000

Tabla 4.17: Error en frecuencias para cada base de tiempos con activación de prescaler

selec_base[1..0]	Período	error [us]
"00" = 0	0.1 us	± 0.01
"01" = 1	1us	± 0.1
"10" = 2	10 us	± 1
"11" = 3	100 us	± 10

Tabla 4.18: Error para cada señal de período definido con activación de prescaler

El error en la base de tiempos, está dado por la estabilidad del oscilador externo. Par nuestro caso el oscilador utilizado es el JITO[®]-2AC5BF 10.000000 de **FOX Electronics**, cuyas características son las siguientes:

- Encapsulado: A = 14 pin DIP.
- Tipo: C = HCMOS.
- Tensión de alimentación: 5 = +5.0V.
- Estabilidad: B = ±50ppm.
- Rango de temperatura: F = -20°C a +70°C.
- Frecuencia: 10.000000 = 10MHz.

Por lo tanto se desprende de aquí que el error en nuestra base de tiempos es de 50 partes por millón (*ppm*). Luego para el modo medición de frecuencias el error en las bases será del 0.005%.

Finalmente el error total de medición para frecuencia será:

$$\boxed{ErrorF = \pm[0.005\% * F_{medida} + 1dígito]}$$

y para la medición de período:

$$\boxed{ErrorP = \pm[0.005\% * P_{medido} + 1dígito]}$$

4.6.2.3 Verificación del sistema Medidor de Frecuencias y períodos

Se procedió a la verificación del correcto funcionamiento del sistema Medidor de frecuencias y períodos, utilizando para esto distintos valores de frecuencias y períodos a medir.

Se constató que el sistema en modo frecuencia funciona correctamente hasta valores de frecuencia de 20MHz aproximadamente sin prescaler y 100MHz con

prescaler. Esto se debe a que internamente se producen retardos que generan error cuando superamos valores de frecuencias del orden de los 20 o 30MHz. En cambio al utilizar el prescaler, ésta frecuencia es rápidamente dividida y por lo tanto ya no tenemos su totalidad en todo el dispositivo; sino su décima parte. En modo período no existieron problemas, salvo el máximo valor de período ($\approx 100\text{seg}$), debido a la falta de instrumental para generar dicha señal (0.01Hz). Además los valores de período menores a 100ns, deben medirse con activación de prescaler tal cual se vio en el capítulo anterior.

Luego los rangos de medición, con un error máximo del 10% quedan de la forma:

- Sin activación de prescaler
 - Frecuencia: 0.1Hz a 20MHz
 - Período: 1 μs a 10 seg.
- Con activación de prescaler
 - Frecuencia: 1Hz a 100MHz
 - Período: 100ns a 9.99999 seg.

4.6.3 Armado y verificación del sistema Adquisidor autónomo de datos

4.6.3.1 Armado del sistema Adquisidor autónomo de datos.

A continuación vamos a proceder al armado del sistema Adquisidor autónomo de datos. Para esto vamos a conectar la plaqueta Adquisidor al CON1 de la Upx10K10, y la placa FPGA con puerto paralelo de PC al CON2. La figura 4.35 muestra una imagen del sistema armado.

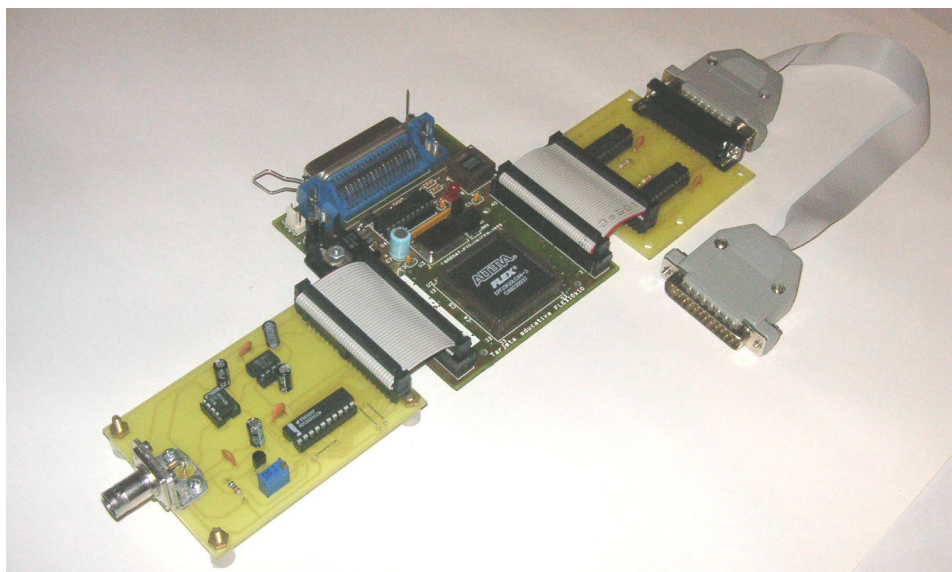


Figura 4.35: Sistema Adquisidor autónomo de datos.

4.6.3.2 Verificación del sistema Adquisidor autónomo de datos

Se procedió a la verificación del correcto funcionamiento del sistema Adquisidor autónomo de datos.

Aquí surgió un problema que consistía en el reseteo de la memoria en el momento de la lectura por parte de la PC.

El error surgía cuando se modificaba la entrada de selección del multiplexor interno `mux_HoL_nibble` a través de la entrada `HoL_nibble` para poder cargar en la PC la palabra de 8bits como dos de 4bits.

Luego entonces se trató de salvar el problema generando una FIFO de 4bits y 1000 palabras, para leer directamente la palabra completa de la memoria, pero el problema persistía.

Se volvió al sistema original pero colocando aquí un multiplexor externo, apareciendo nuevamente el mismo error.

Finalmente y luego de varias pruebas se llegó a la conclusión que el problema provenía de *glitches* generados por el puerto en el cable que lo comunica con la placa 'FPGA con puerto paralelo de PC'.

Para salvar esto se procedió a conectar directamente la plaqueta 'FPGA con puerto paralelo a PC' con el puerto; sin utilizar el cable. A partir de aquí el sistema funcionó correctamente.

La siguiente figura muestra como quedó armado nuestro sistema Adquisidor autónomo de datos.

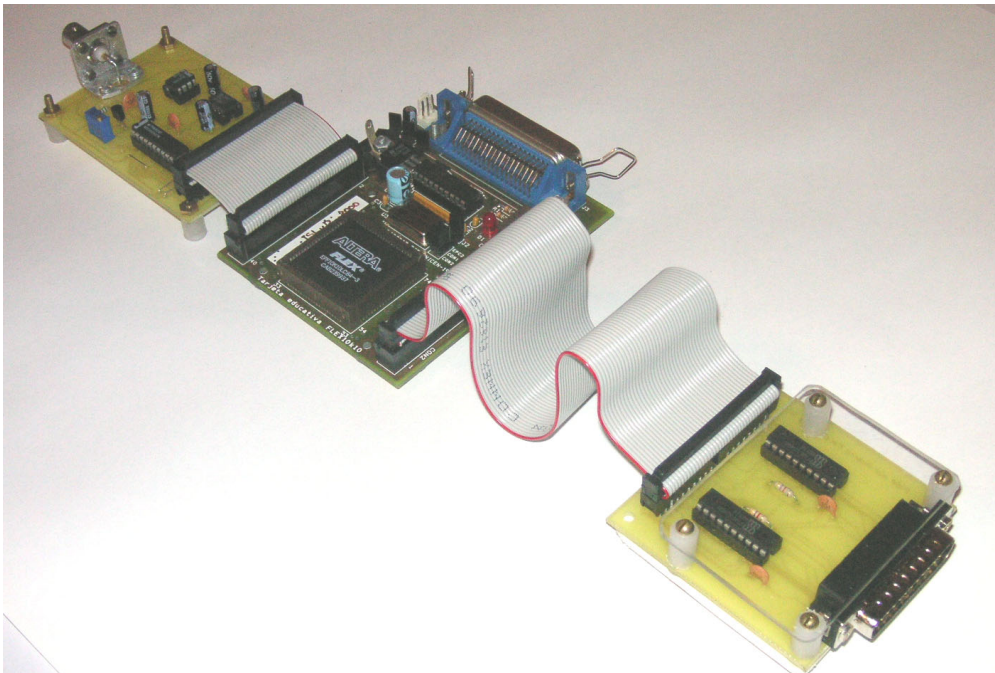


Figura 4.36: Sistema Adquisidor autónomo de datos final

4.6.3.3 Modificación de la plaqueta 'FPGA con puerto paralelo'.

Debido a la no utilización del cable que conecta al puerto, se debió realizar una reforma a la plaqueta 'FPGA con puerto paralelo de PC'.

Como podemos recordar del punto 4.4.3.4, éste cable tenía un puente entre el pin `manual_PC` y GND, para que cuando se conecte al sistema Medición de Frecuencias y Períodos, éste trabaje en modo manual.

Al no utilizar más éste cable deberemos realizar el puente en la plaqueta uniendo el pin `manual_PC` (pin 14 del DB25) con un pin GND del puerto (pin 19 del DB25), de modo que cuando se conecte la plaqueta al puerto el Medidor trabaje en modo PC.

La plaqueta con el puente (cable blanco) se muestra en la figura 4.37.

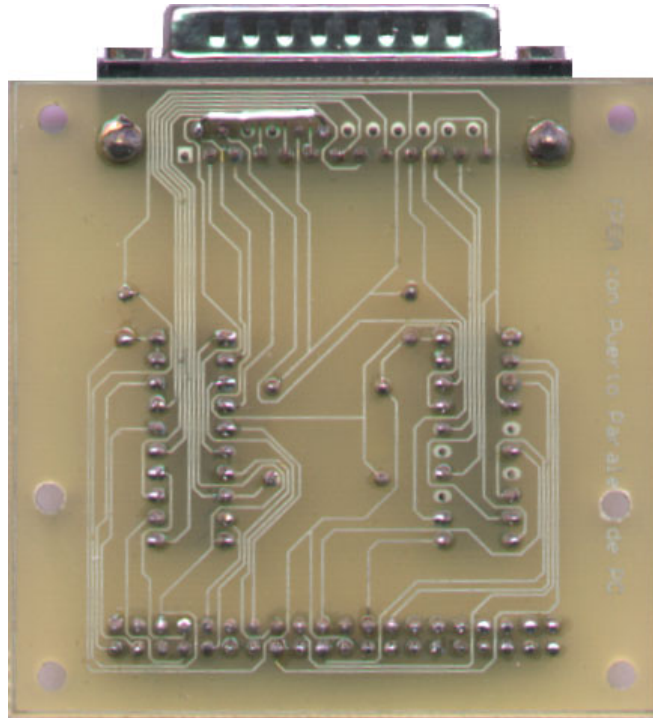


Figura 4.37: Plaqueta FPGA con puerto paralelo de PC con puente

4.6.3.4 Optimización del sistema Adquisidor autónomo de datos.

Vamos ahora a aumentar la frecuencia del convertor a 500Khz ($T = 2\mu s$), y verificar si el sistema funciona correctamente.

Por lo tanto generaremos ahora las señales que muestra la figura 4.38:

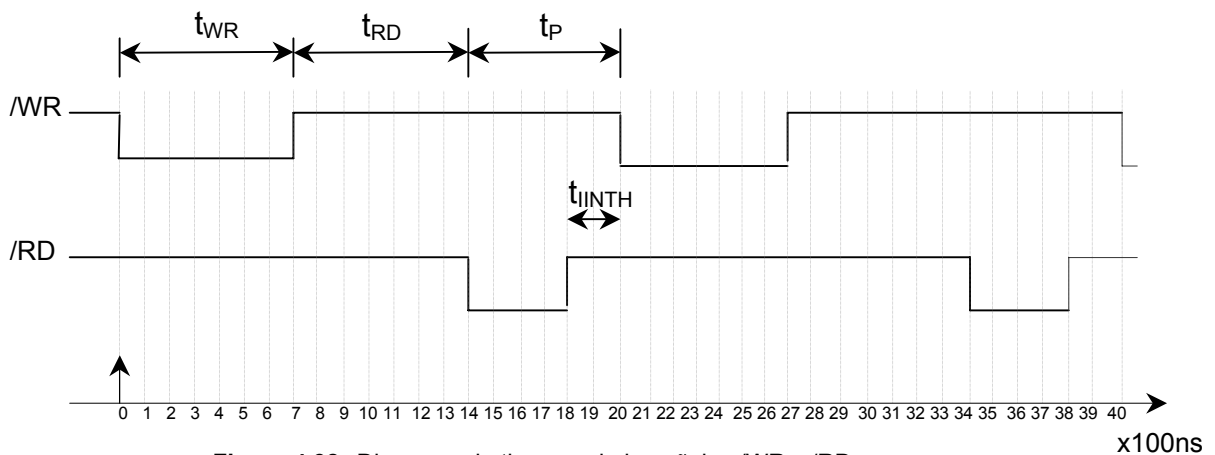


Figura 4.38: Diagrama de tiempos de la señales `/WR` y `/RD` a generar

Luego en AHDL debemos reformar la generación de las señales /WR y /RD:

```

...
cuenta2.aclr = clear;
cuenta2.clock = reloj_xtal;
salidawr.clnr = !clear;
salidard.clnr = !clear;
salidawr.clk = reloj_xtal;
salidard.clk = reloj_xtal;

IF cuenta2.q[] < 7 THEN
    salidawr.d = GND;
    salidard.d = VCC;
ELSE IF cuenta2.q[] < 14 THEN
    salidawr.d = VCC;
    salidard.d = VCC;
    ELSE IF cuenta2.q[] < 18 THEN
        salidawr.d = VCC;
        salidard.d = GND;
        ELSE IF cuenta2.q[] < 20 THEN
            salidawr.d = VCC;
            salidard.d = VCC;
        ELSE
            salidawr.d = GND;
            salidard.d = VCC;
        END IF;
    END IF;
END IF;

/WR = salidawr.q;
/RD = salidard.q;

```

En el programa en Visual, se debe reformar la sentencia para graficar:

```

Private Sub Command2_Click()
    Picture1.Cls
    ZoomX = Val(Text3.Text) / 100
    ZoomY = Val(Text1.Text) / 100

    If ZoomX > 1 Then
        HScroll1.Visible = True
    Else
        HScroll1.Visible = False
    End If

    If ZoomY > 1 Then
        VScroll1.Visible = True
    Else
        VScroll1.Visible = False
    End If

    HScroll1.Max = (ZoomX - 1) * 10
    VScroll1.Min = -(ZoomY - 1) * 10

    ` EJE X
    For n = 0 To 10000 Step (10000 / 10) 'tengo que meter 10 pasos en 10000 pixels
        Picture1.Line (ZoomX * n + 500 - 1000 * ValorX, 5550)-(ZoomX * n + 500 -
1000 * ValorX, 5450)
        Picture1.Line (ZoomX * n + 400 - 1000 * ValorX, 5600)-(ZoomX * n + 401 -
1000 * ValorX, 5600)
        Muestras = n / 20 'ya que en 10000/(500 muestras) = 20
        Picture1.Print (Muestras * 0.002)
        'Ya que tenemos una muestra cada 2us, y la escala es en ms
        If Gridon = True Then
            Picture1.Line (ZoomX * n + 500 - 1000 * ValorX, 300)-(ZoomX * n + 500 -
1000 * ValorX, 5500), &HC0C0C0 'Grilla en y color gris
        End If
    Next n

```

```

Next n

` EJE Y
For n = 0 To 5000 Step 1000
  Picture1.Line (450, 5500 - ZoomY * n - 500 * ValorY)-(550, 5500 - ZoomY * n - 500 * ValorY)
  Picture1.Line (50, 5400 - ZoomY * n - 500 * ValorY)-(50, 5400 - ZoomY * n - 500 * ValorY)
  Picture1.Print 2.5 * n / 5000
  If Gridon = True Then
    Picture1.Line (500, 5500 - ZoomY * n - 500 * ValorY)-(11000, 5500 - ZoomY * n - 500 * ValorY), &HC0C0C0 'Grilla en x color gris
  End If
Next n

Picture1.Line (500 - 1000 * ValorX, 200)-(500 - 1000 * ValorX, 5500) 'Eje Y
Picture1.Line (500, 5500 - 500 * ValorY)-(11000, 5500 - 500 * ValorY) 'Eje X
Picture1.Line (10800, 5750)-(10801, 5750)
Picture1.Print "[ms]"

For n = 0 To NumMuestra - 1
  Picture1.PSet ((10 * ZoomX * n) * 2 + 500 - 1000 * ValorX, 5500 - ZoomY * 5000 / 255 * Valor(n) - 500 * ValorY), &HFF
Next n

Text1.Enabled = True
Text3.Enabled = True
Option1(0).Enabled = True
Option1(1).Enabled = True

End Sub

```

Con lo cual por ejemplo para una señal sinusoidal de 8KHz, con una tensión pico a pico de 1.5V y un offset de 1.25V, la ventana del programa queda como muestra la figura 4.39.

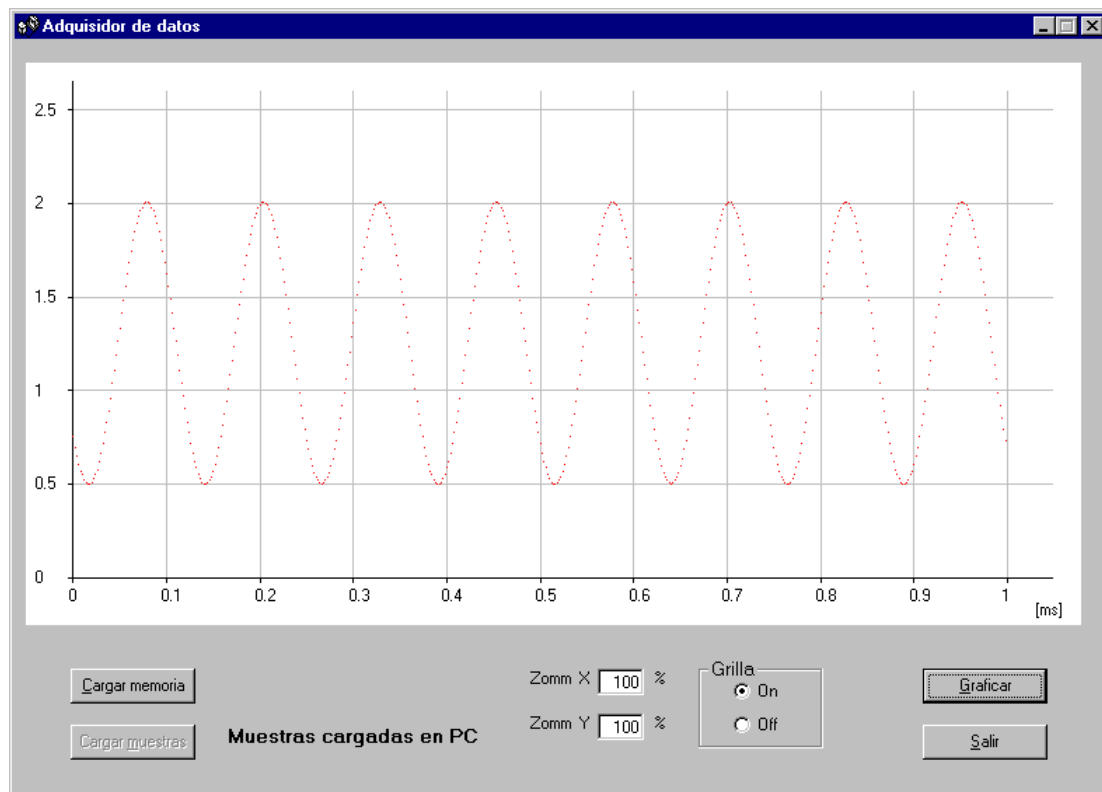


Figura 4.39: Formulario para el Adquisidor autónomo de datos.

Finalmente se procedió a la verificación del funcionamiento del Adquisidor para ésta frecuencia de muestreo (500kHz), la cual resultó óptima.

4.7 Programa final en AHDL del proyecto Medidor de frecuencias y períodos + Adquisidor de datos autónomo

El programa final en AHDL al que llamaremos *proyectofinal.tdf* queda de la forma

```
constant MAX_COUNT = 10000;
constant N_DIGITOS = 6;                                % De modo que %
constant POT_K = ceil(log2(N_DIGITOS));                % 2^POT_K >= N_DIGITOS %
constant MAX_CONT = 24;
constant N_MUESTRAS = 500;                             % De modo que %
constant POT_M = ceil(log2(N_MUESTRAS));              % 2^POT_M >= N_MUESTRAS %

INCLUDE "lpm_counter.inc";
INCLUDE "lpm_latch.inc";
INCLUDE "lpm_mux.inc";
INCLUDE "lpm_decode.inc";
INCLUDE "lpm_fifo.inc";

SUBDESIGN proyectofinal
(
    reloj_xtal, reset, manual_PC, pulsador_selbase      : INPUT;
    signal_in, Frec_o_Period_o_/INT, activa_prescaler    : INPUT;
    Frec_o_Adqui, DB[7..0], data_port[7..0]             : INPUT;

    a_o_/CS, b_o_/RD, c_o_/WR, d, e, f, g, dp, overflow : OUTPUT;
    selec_digito[N_DIGITOS..1], status_port[7..3]      : OUTPUT;
)

VARIABLE

cuenta1: lpm_counter WITH (LPM_WIDTH=ceil(log2(MAX_COUNT)));
cuenta2: lpm_counter WITH ( LPM_WIDTH= ceil(log2(MAX_CONT)),
                           LPM_MODULUS=MAX_CONT);
cont_anti_rebote: lpm_counter WITH (LPM_WIDTH=7, LPM_MODULUS=100);
contadorbase : lpm_counter WITH ( LPM_WIDTH=2, LPM_MODULUS=4);
prescaler1: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
prescaler2: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
prescaler3: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
prescaler4: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
prescaler_signal: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
presc_sel_decodig: lpm_counter WITH ( LPM_WIDTH=POT_K,
                                       LPM_MODULUS=N_DIGITOS);
divisor: lpm_counter WITH (LPM_WIDTH=1);
contador[N_DIGITOS..1]: lpm_counter WITH ( LPM_WIDTH=4, LPM_MODULUS=10);
latches[N_DIGITOS..1]: lpm_latch WITH (LPM_WIDTH=4);
multiplexor: lpm_mux WITH (LPM_WIDTH=4, LPM_SIZE=N_DIGITOS,
                           LPM_WIDTHS=POT_K);
mux_HoL_nibble: lpm_mux WITH (LPM_WIDTH=4, LPM_SIZE=2, LPM_WIDTHS=1);
decodificador: lpm_decode WITH (LPM_WIDTH=POT_K, LPM_DECODES=N_DIGITOS);
fifo: lpm_fifo WITH (LPM_WIDTH=8, LPM_NUMWORDS=N_MUESTRAS,
                    LPM_WIDTHU=POT_M);
ss: MACHINE OF BITS (z1) WITH STATES (s0 = 0, s1 = 1, s2 = 0);

salida, salidard, salidawr, resetcont                : DFF;
ffar1, ffar2, ffoverflow1, ffoverflow2              : DFF;
```

```

borrar, reloj_1ms, clock10M, selec_decodig[POT_K-1..0] : NODE;
base_tiempos, reloj, pulsos, reloj_in : NODE;
sal_cont[N_DIGITOS..1][3..0], qlatch[N_DIGITOS..1][3..0] : NODE;
sal_mux[3..0], bcd[3..0], nodo[N_DIGITOS..1], w : NODE;
selec_base[1..0], FoP, prescaler_activo, z : NODE;
datos[3..0], full, clear, cargar_fifo : NODE;
leer_fifo, HoL_nibble, clk_PC : NODE;
data_port_F[7..0], status_port_F[6..3] : NODE;
data_port_A[4..0], status_port_A[7..3] : NODE;
sel_base_ext[1..0], pulsos_selbase, reloj_6ms : NODE;
a, b, c, /CS, /RD, /WR, /INT, Frec_o_Period : NODE;

```

```
BEGIN
```

```
%%%% Asignación de pines de acuerdo al modo de utilización de la FPGA %%%
```

```

CASE Frec_o_Adqui IS
  WHEN B"0" => % Modulo Adquisidor %
    data_port_A[4..0] = data_port[4..0];
    status_port[7..3] = status_port_A[7..3];
    /INT = Frec_o_Period_O_/INT;
    a_o_/CS = /CS;
    b_o_/RD = /RD;
    c_o_/WR = /WR;

  WHEN B"1" => % Modulo Frecuencimetro %
    data_port_F[7..0] = data_port[7..0];
    status_port[6..3] = status_port_F[6..3];
    Frec_o_Period = Frec_o_Period_O_/INT;
    a_o_/CS = a;
    b_o_/RD = b;
    c_o_/WR = c;
END CASE;

```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Medidor de Frecuencias y Periodos %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```

% Comienzo etapa prescaler %
prescaler1.aclr = reset;
prescaler2.aclr = reset;
prescaler3.aclr = reset;
prescaler4.aclr = reset;
presc_sel_decodig.aclr = reset;
prescaler1.clock = reloj_xtal;
prescaler2.clock = reloj_xtal;
prescaler3.clock = reloj_xtal;
prescaler4.clock = reloj_xtal;
presc_sel_decodig.clock = reloj_xtal;
prescaler2.cnt_en = prescaler1.eq[9];
prescaler3.cnt_en = prescaler2.eq[9] & prescaler1.eq[9];
prescaler4.cnt_en = prescaler3.eq[9] & prescaler2.eq[9] &
  prescaler1.eq[9];
presc_sel_decodig.cnt_en = prescaler4.eq[9] & prescaler3.eq[9] &
  prescaler2.eq[9] & prescaler1.eq[9];

reloj_1ms = prescaler4.eq[9];
reloj_6ms = presc_sel_decodig.eq[5];
selec_decodig[POT_K-1..0] = presc_sel_decodig.q[];
clock10M = reloj_xtal;
% Fin etapa prescaler %

% Circuito anti-rebotes para convertir el pulso de "pulsador_selbase" %
% en un pulso "pulso_selbase" de 600ms de duración aproximadamente %
ffar1.d = VCC;
ffar1.clk = pulsador_selbase;

```

```

ffar2.clk = reloj_6ms;
cont_anti_rebote.clock = reloj_6ms;
cont_anti_rebote.cnt_en = ffar1.q;
IF cont_anti_rebote.q[] == 99 THEN
    ffar2.d = VCC;
ELSE
    ffar2.d = GND;
END IF;
ffar1.clrn = !ffar2.q;
pulsos_selbase = ffar1.q;

% Circuito para incrementar con la entrada "pulsos_selbase" %
% el contador de elección de la base de tiempos "contadorbse" %
contadorbse.clock = pulsos_selbase;
sel_base_ext[1..0] = contadorbse.q[];

% Selección entre modo manual o PC %
CASE manual_PC IS
    WHEN B"0" => % Modo PC %
        prescaler_activo = data_port_F[7];
        FoP = data_port_F[6];
        selec_base[] = data_port_F[5..4];
    WHEN B"1" => % Modo manual %
        prescaler_activo = activa_prescaler;
        FoP = Frec_o_Period;
        selec_base[] = sel_base_ext[1..0];
END CASE;

% Base de tiempo para modo medición de frecuencias %
salida.clrn = !reset;
salida.clk = reloj_1ms;
cuenta.clock = reloj_1ms;
cuenta.sclr = borrar;
CASE selec_base[] IS
    WHEN B"00" =>
        IF cuenta1.q[] < 10000 THEN
            salida.d = VCC;
            borrar = GND;
        ELSE
            borrar = VCC;
            salida.d = GND;
        END IF;
    WHEN B"01" =>
        IF cuenta1.q[] < 1000 THEN
            salida.d = VCC;
            borrar = GND;
        ELSE
            salida.d = GND;
            borrar = VCC;
        END IF;
    WHEN B"10" =>
        IF cuenta1.q[] < 100 THEN
            salida.d = VCC;
            borrar = GND;
        ELSE
            borrar = VCC;
            salida.d = GND;
        END IF;
    WHEN B"11" =>
        IF cuenta1.q[] < 10 THEN
            salida.d = VCC;
            borrar = GND;
        ELSE
            borrar = VCC;
            salida.d = GND;
        END IF;
END CASE;

```

```

% Pulsos para modo medición de período %
CASE selec_base[] IS
    WHEN B"00" =>
        pulsos = reloj_xtal;
    WHEN B"01" =>
        pulsos = prescaler1.eq[9];
    WHEN B"10" =>
        pulsos = prescaler2.eq[9];
    WHEN B"11" =>
        pulsos = prescaler3.eq[9];
END CASE;

% Activación del prescaler que divide por 10 %
prescaler_signal.clock = signal_in;
IF prescaler_activo THEN % Si el prescaler esta activado %
    reloj_in = prescaler_signal.eq[9]; % La señal a medir es la %
    % señal exterior/10 %
ELSE % Si el prescaler no esta activado %
    reloj_in = signal_in; % La señal a medir es la señal exterior %
END IF;

% Base de tiempo para modo medición de períodos %
divisor.clock = reloj_in;

% Conexión de los nodos base_tiempos y reloj según se %
% mida frecuencia ó período %
IF FoP THEN
    base_tiempos = salida.q; % Medición de %
    reloj = reloj_in; % Frecuencia %
ELSE
    base_tiempos = divisor.q[]; % Medición de %
    reloj = pulsos; % Período %
END IF;

% Monoestable con maquina de estados %
ss.clk = clock10M;
ss.reset = reset;
TABLE
% estado entrada próximo %
% actual actual estado %
    ss, base_tiempos => ss;

    s0, 1 => s0;
    s0, 0 => s1;
    s1, 0 => s2;
    s1, 1 => s0;
    s2, 0 => s2;
    s2, 1 => s0;
END TABLE;

%Inhabilitación de la señal z, para congelar latches, cuando leo por puerto%
z = z1 & !data_port_F[3];

% Decodificador BCD a 7 segmentos %
TABLE
bcd[3..0] => a, b, c, d, e, f, g;
H"0" => 1, 1, 1, 1, 1, 1, 0;
H"1" => 0, 1, 1, 0, 0, 0, 0;
H"2" => 1, 1, 0, 1, 1, 0, 1;
H"3" => 1, 1, 1, 1, 0, 0, 1;
H"4" => 0, 1, 1, 0, 0, 1, 1;
H"5" => 1, 0, 1, 1, 0, 1, 1;
H"6" => 1, 0, 1, 1, 1, 1, 1;
H"7" => 1, 1, 1, 0, 0, 0, 0;
H"8" => 1, 1, 1, 1, 1, 1, 1;
H"9" => 1, 1, 1, 1, 0, 1, 1;
END TABLE;

```

```

%      Generación de la señal w para resetear el contador      %
resetcont.clrn = !base_tiempos;
resetcont.d = VCC;
resetcont.clk = !z1;
w = resetcont.q;

%      Contadores, latches y multiplexor      %
FOR i IN 1 TO N_DIGITOS GENERATE
    contador[i].clock = reloj;
    contador[i].aclr = w;
    latches[i].gate = z;
END GENERATE;
sal_cont[N_DIGITOS..1][3..0] = contador[N_DIGITOS..1].q[];
latches[N_DIGITOS..1].data[] = sal_cont[N_DIGITOS..1][];
qlatch[N_DIGITOS..1][] = latches[N_DIGITOS..1].q[];
multiplexor.data[N_DIGITOS-1..0][] = qlatch[N_DIGITOS..1][];

contador[1].cnt_en = base_tiempos;
nodo[1] = base_tiempos;
FOR n IN 2 TO N_DIGITOS GENERATE
    nodo[n] = nodo[n-1] & contador[n-1].eq[9];
    contador[n].cnt_en = nodo[n];
END GENERATE;

multiplexor.sel[] = selec_decodig[];
sal_mux[] = multiplexor.result[];

bcd[3..0] = sal_mux[];

%      Decodificador      %
decodificador.data[] = selec_decodig[];
selec_digito[N_DIGITOS..1] = !decodificador.eq[N_DIGITOS-1..0];

%      Generación de señal overflow      %
ffoverflow1.clrn = !w;
ffoverflow1.d = VCC;
ffoverflow1.clk = contador[6].cout & contador[5].cout & contador[4].cout
    & contador[3].cout & contador[2].cout & contador[1].cout;
ffoverflow2.clrn = !reset;
ffoverflow2.d = ffoverflow1.q;
ffoverflow2.clk = !base_tiempos;
overflow = ffoverflow2.q;

%      Ubicación del punto (dp) de acuerdo a la base y modo utilizado      %
%      solo para el caso que N_DIGITOS >= 5      %
IF !prescaler_activo THEN
    % Si no se activó prescaler %
    CASE selec_base[] IS
        WHEN B"00" =>
            dp = decodificador.eq[4];%equiv. a !selec_digito[5]%
        WHEN B"01" =>
            dp = decodificador.eq[3];%equiv. a !selec_digito[4]%
        WHEN B"10" =>
            dp = decodificador.eq[2];%equiv. a !selec_digito[3]%
        WHEN B"11" =>
            dp = decodificador.eq[1];%equiv. a !selec_digito[2]%
    END CASE;
ELSE
    % Si se activó prescaler      %
    % en modo medición Frecuencia %
    % corro punto a derecha      %
    IF FoP THEN
        CASE selec_base[] IS
            WHEN B"00" =>
                dp = decodificador.eq [4-1];
            WHEN B"01" =>
                dp = decodificador.eq [3-1];
            WHEN B"10" =>
                dp = decodificador.eq [2-1];
        END CASE;
    END IF;
END IF;

```



```

        WHEN B"11" =>
            dp = decodificador.eq [1-1];
        END CASE;
ELSE
    CASE selec_base[] IS
        WHEN B"00" =>
            dp = decodificador.eq [4+1];
        WHEN B"01" =>
            dp = decodificador.eq [3+1];
        WHEN B"10" =>
            dp = decodificador.eq [2+1];
        WHEN B"11" =>
            dp = decodificador.eq [1+1];
        END CASE;
    END IF;
END IF;

% Salida hacia el puerto de estado (status_port) según la
% entrada por el puerto de datos (data_port)
CASE data_port_F[2..0] IS
    WHEN 0 =>
        status_port_F[3] = overflow;
    WHEN 1 =>
        status_port_F[6..3] = qlatch[1][];
    WHEN 2 =>
        status_port_F[6..3] = qlatch[2][];
    WHEN 3 =>
        status_port_F[6..3] = qlatch[3][];
    WHEN 4 =>
        status_port_F[6..3] = qlatch[4][];
    WHEN 5 =>
        status_port_F[6..3] = qlatch[5][];
    WHEN 6 =>
        status_port_F[6..3] = qlatch[6][];
END CASE;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Memoria FIFO para el adquisidor %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Conexión del puerto con los nodos
clear = data_port_A[0];
cargar_fifo = data_port_A[1];
leer_fifo = data_port_A[2];
HoL_nibble = data_port_A[3];
clk_PC = data_port_A[4];
status_port_A[7] = full;
status_port_A[6..3] = datos[3..0];

% Obtención de las salidas /RD y /WR
cuenta2.aclr = clear;
cuenta2.clock = reloj_xtal;
salidawr.clrn = !clear;
salidard.clrn = !clear;
salidawr.clk = reloj_xtal;
salidard.clk = reloj_xtal;

IF cuenta2.q[] < 7 THEN
    salidawr.d = GND;
    salidard.d = VCC;
ELSE IF cuenta2.q[] < 14 THEN
    salidawr.d = VCC;
    salidard.d = VCC;
ELSE IF cuenta2.q[] < 18 THEN
    salidawr.d = VCC;
    salidard.d = GND;
ELSE IF cuenta2.q[] < 20 THEN

```

```

        salidawr.d = VCC;
        salidard.d = VCC;
    ELSE
        salidawr.d = GND;
        salidard.d = VCC;
    END IF;
END IF;
END IF;
END IF;

/WR = salidawr.q;
/RD = salidard.q;

%   Los pines de entrada           %
fifordreq = leer_fifo;
fifoaclr = clear;
fifowrreq = !/INT;
fifodata[] = DB[];

%   Los pines de salida           %
/CS = !cargar_fifo;
full = fifo.full;

%   De acuerdo a el modo en que esté trabajando la FIFO           %
IF cargar_fifo THEN           % Si está cargando datos del ADC           %
    fifoclock = salidard.q;   % Reloj generado por el contador           %
ELSE                           % Si va a cargar datos en PC           %
    fifoclock = clk_PC;      % Reloj generado por la PC           %
END IF;

%   El multiplexor divide las datos de salida para cargar por           %
%   puerto "datos[3..0]"           %
%   cuando HoL_nibble = 0 => datos[3..0] = q[7..4] High Nibble           %
%   HoL_nibble = 1 => datos[3..0] = q[3..0] Low Nibble           %
mux_HoL_nibble.data[0][] = fifo.q[7..4];
mux_HoL_nibble.data[1][] = fifo.q[3..0];
mux_HoL_nibble.sel[] = HoL_nibble;
datos[] = mux_HoL_nibble.result[];

END;

```

Apéndice A

A.1 Decodificadores

Permiten seleccionar una de **N** salidas de un circuito, el cual tiene **n** entradas. En decodificadores binarios se cumple la relación: $N = 2^n$, como el de la siguiente figura:

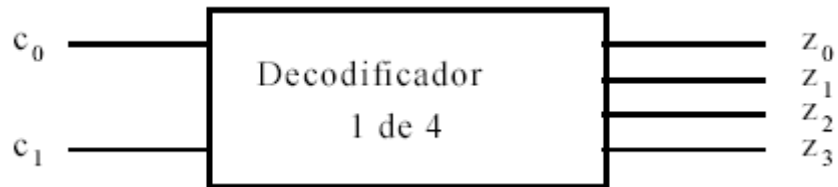


Figura A.1: Decodificador

Dependiendo de la combinación presente en la entrada, se tendrá sólo una de las salidas en alto; el resto de las salidas serán bajas. En caso de tener : $C_0 = 1, C_1 = 0$; se tendrá que $Z_1 = 1, Z_0 = Z_2 = Z_3 = 0$. El siguiente circuito implementa un decodificador binario en base a compuertas:

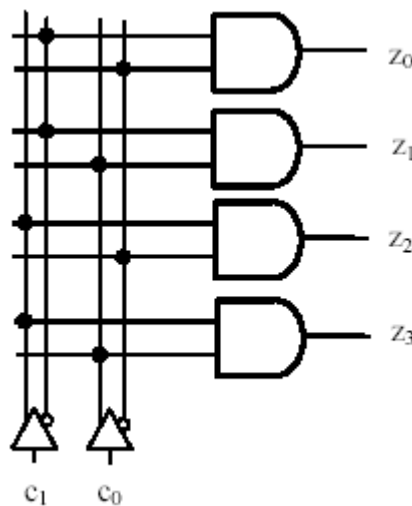


Figura A.2: Diseño de un decodificador en base de compuertas

La tabla de verdad:

$$\begin{aligned} Z_0 &= \bar{C}_1 \bar{C}_0 \\ Z_1 &= \bar{C}_1 C_0 \\ Z_2 &= C_1 \bar{C}_0 \\ Z_3 &= C_1 C_0 \end{aligned}$$

C ₁	C ₀	Z ₃	Z ₂	Z ₁	Z ₀
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Tabla A.1: Tabla de verdad

A.2 Demultiplexers

Una variante de los circuitos decodificadores es el demultiplexer (DEMUX), del cual si hacemos un diagrama funcional podemos hacer una analogía eléctrica con una llave distribuidora que esquemáticamente puede representarse por:

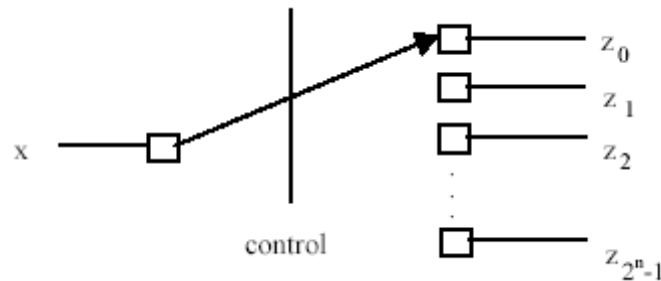


Figura A.3: Esquema funcional de un DEMUX

Es decir, permite dirigir la información que fluye de x , por una (y sólo una) de las 2^n salidas, de acuerdo a una entrada de control codificada, en n líneas; ésta selecciona la vía por la que fluirá la única información de entrada. Si diseñamos un multiplexor en base a compuertas:

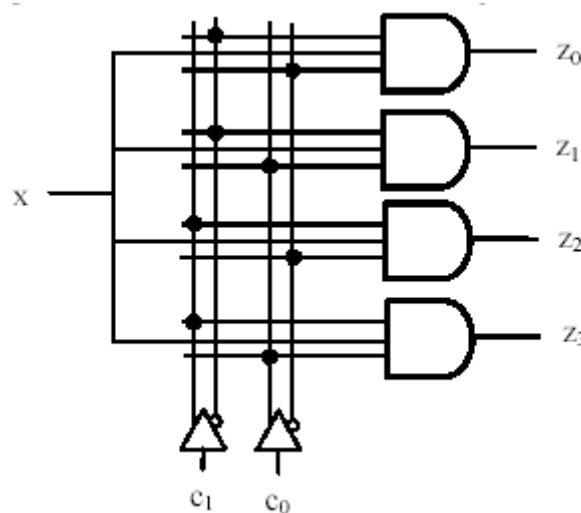


Figura A.4: multiplexor en base a compuertas

donde:

$$\begin{aligned} Z_0 &= \bar{C}_1 \bar{C}_0 X \\ Z_1 &= \bar{C}_1 C_0 X \\ Z_2 &= C_1 \bar{C}_0 X \\ Z_3 &= C_1 C_0 X \end{aligned}$$

Como ejemplo la señal X fluye hacia el receptor, conectado en la salida Z_3 , si las señales de control toman valores: $C_0 = 1$, $C_1 = 1$. El resto de las salidas: Z_0 , Z_1 y Z_2 toman valor cero.

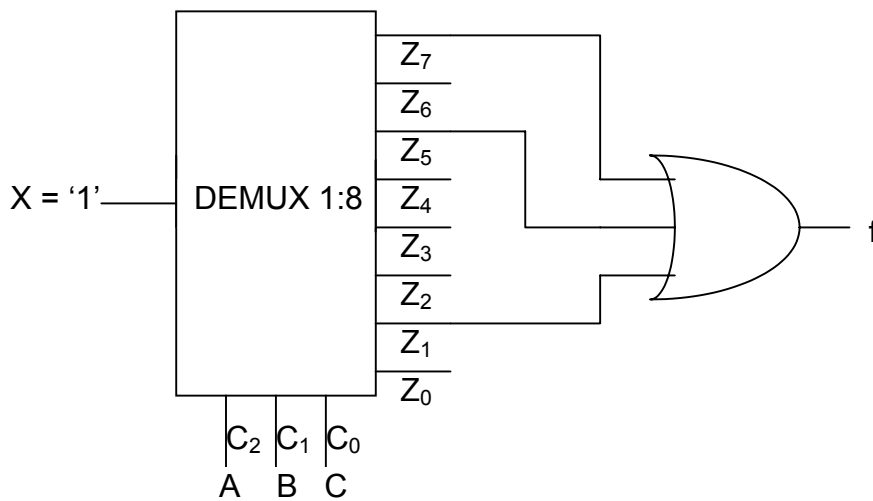
Por lo tanto, en general para un DEMUX $1:2^n$ conectando la entrada X a un nivel '1' lógico, cada salida puede verse como asociada a cada uno de los posibles minterminos de una función de n variables.

A.2.1 Resolución de ecuaciones lógicas mediante el uso de demultiplexers

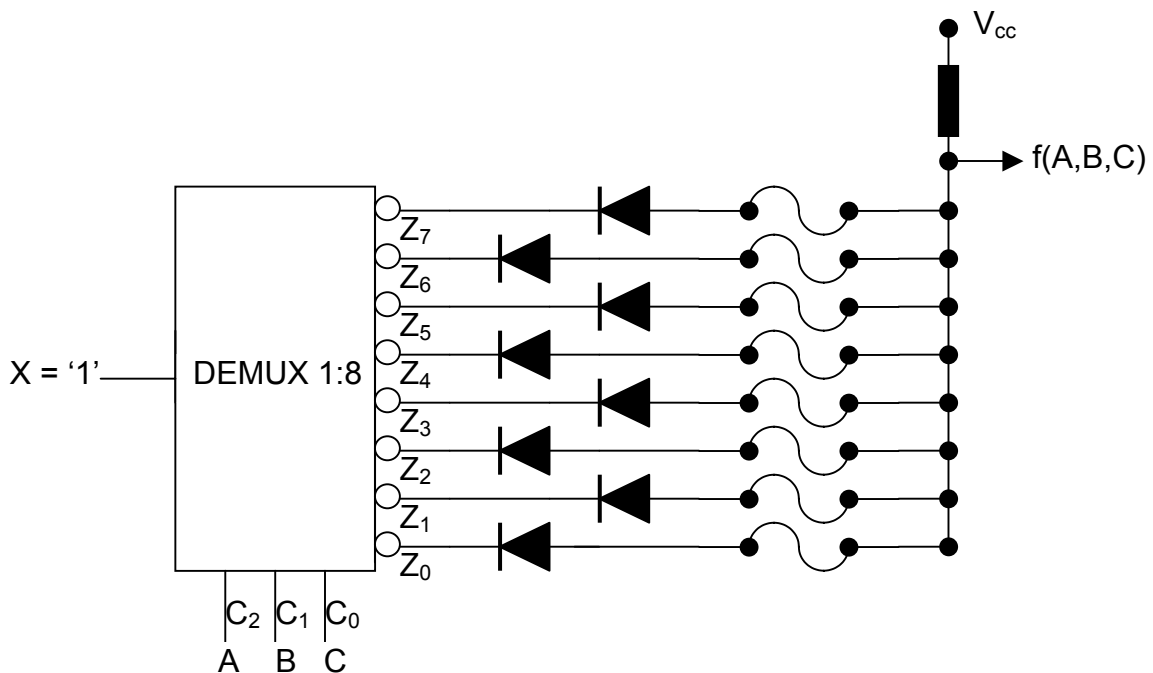
Si para cualquier función f de n variables se generan mediante un DEMUX los 2^n minterminos, y se realiza mediante una compuerta OR de un máximo de 2^n entradas la unión lógica de aquellos minterminos que la componen, a la salida de la compuerta OR se tendrá la función con un máximo de tres niveles de compuertas.

Ejemplo:

Sintetizar la función $f = A B C + A /B C + /A /B C$ utilizando un DEMUX 1:8 y una compuerta OR.



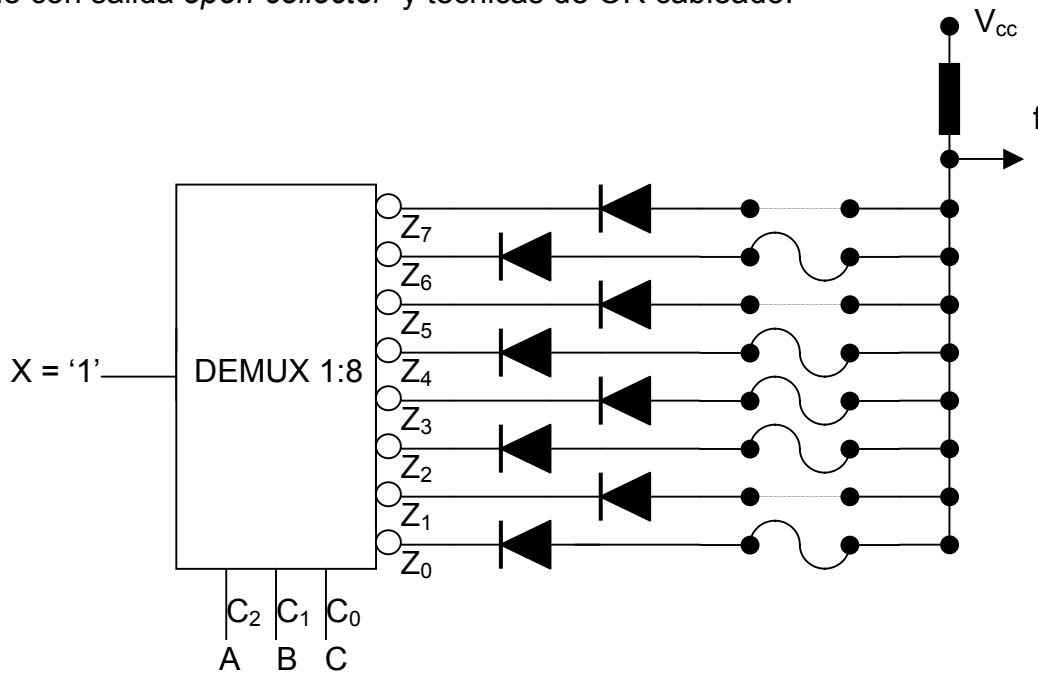
Ésta complejidad de una compuerta OR de hasta 2^n entradas (también llamada matriz de unión) es resuelta mediante la utilización de demultiplexers con salidas *open-colector* y técnicas de OR/AND cableado



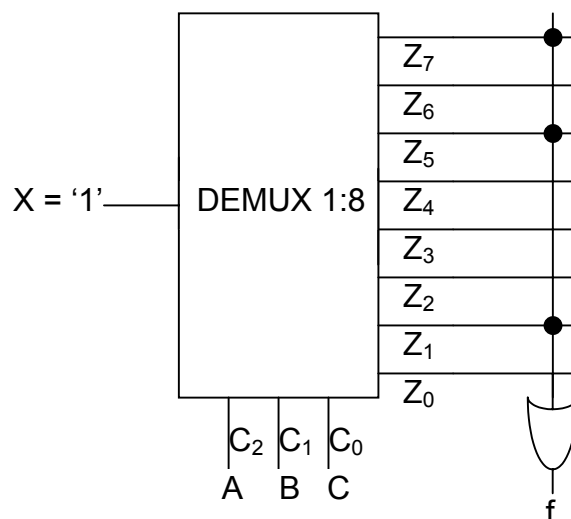
La función OR/AND cableado se realiza mediante lógica de diodos y todos los mintérminos se conectan a través de diodos con un fusible en serie a la línea de salida. Éste circuito podrá configurarse eléctricamente (o programarse) para cumplir con una dada función mediante el quemado selectivo (programación) de aquellos fusibles que corresponden a mintérminos que no componen (caso OR) o que componen (caso AND) la función).

Ejemplo:

Sintetizar la función $f = A B C + A /B C + /A /B C$ utilizando un DEMUX 1:8 con salida *open-collector* y técnicas de OR cableado.



O bien utilizando otra nomenclatura circuital:



Aunque esta solución con DEMUX parece mas compleja que la solución con MUX, en el caso de necesitar generar muchas funciones lógicas distintas, con las mismas variables de entrada, basta con utilizar un único DEMUX el cual genera los mintérminos y utilizar una línea OR cableada distinta para cada función de salida.

Apéndice B

B.1 Multiplexers

El multiplexer (MUX) es un dispositivo electrónico que dispone de 2^n entradas y una sola salida.

A través de las n líneas de control, denominadas líneas de selección, podemos seleccionar cual de las 2^n entradas tendrá conexión directa con la salida.

Tal como muestra la figura, si hacemos un diagrama funcional de un MUX, podemos hacer una analogía eléctrica con una llave selectora.

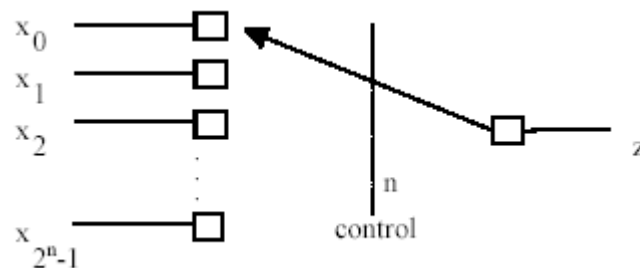


Figura B.1: Esquema funcional de un multiplexor

Los multiplexers se pueden clasificar de acuerdo al tipo de señal que manejan, en *digitales* y *analógicos*, aunque en nuestro caso solo haremos hincapié en los MUX digitales.

B.1.1 Multiplexers digitales

El multiplexor digital está compuesto por un circuito combinatorio que generalmente incluye compuertas AND ó NAND, negadores y una compuerta OR ó NOR a la salida.

El siguiente esquema lógico, muestra un MUX (multiplexor) de 4 vías a una, implementado con compuertas, y también se ilustra un símbolo para el MUX.

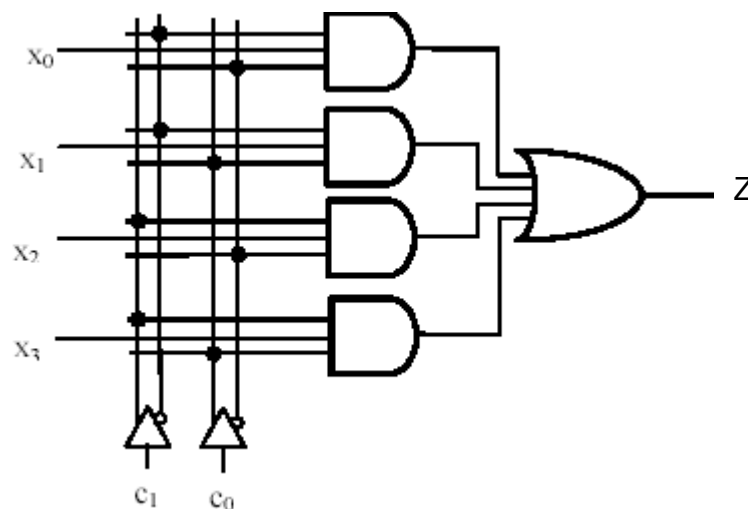


Tabla B.1: Figura B.2: Diseño combinacional de multiplexor

donde:

$$Z = C_1C_0X_3 + C_1/C_0X_2 + /C_1C_0X_1 + /C_1/C_0X_0$$

Suele existir una señal denominada enable (habilitación) que habilita el control del multiplexor. Cuando se autoriza el ingreso del control en un breve lapso de tiempo, la señal de habilitación toma la forma de un pulso angosto. Este pulso de control, se denomina **STROBE**, en inglés. Puede emplearse el siguiente símbolo, para un mux de 4 vías a 1. Un ejemplo de este mux es el 74151.

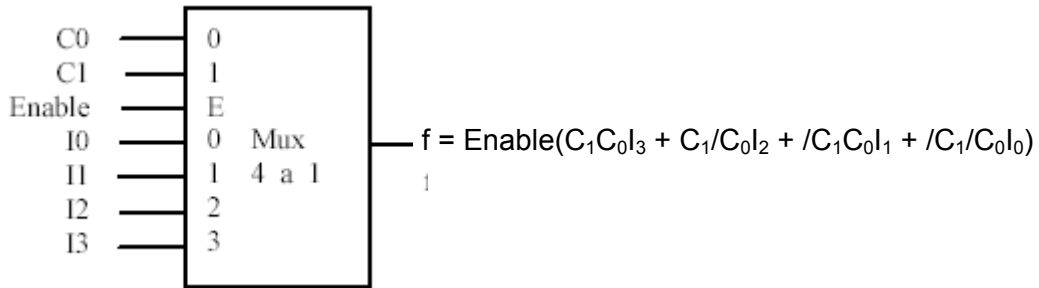


Figura B.3: MUX de 4 vías a una (4:1)

Un multiplexor de 8 vías a una, ejemplos: 74151, 74152:

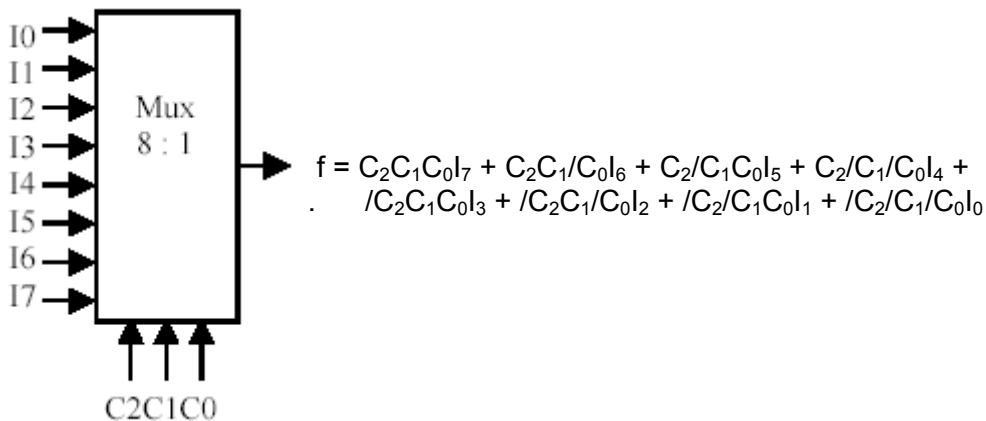


Figura B.4: MUX de 8 vías a una (8:1)

Una alternativa electrónica es dotar a estos multiplexores de una salida de tercer estado (TRI-STATE). En este caso la compuerta de salida además de los valores lógico 1 y 0, puede quedar en un estado de alta impedancia. Este control adicional, permite conectar las salidas de dos multiplexores a un mismo alambre; solo uno de los multiplexores impone valores lógicos en la salida; el otro, está en tercer estado.

Ejemplos de estas configuraciones de tercer estado son: 74251, 74253, 74257

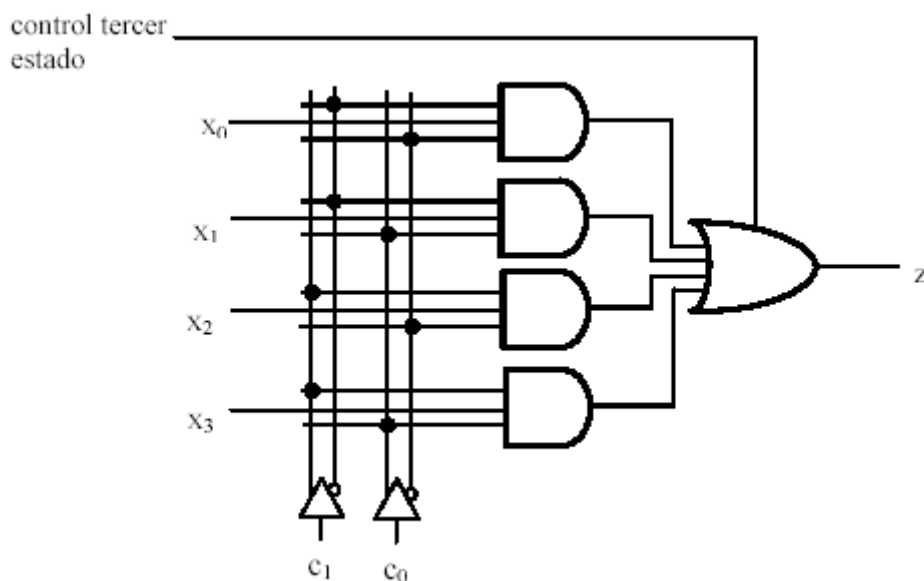


Tabla B.2: Figura B.5: Multiplexor con salida de tercer estado.

Las dos principales aplicaciones del MUX son:

- Selector de fuentes de señal.
- Generador de funciones booleanas.

B.1.2 Generación de funciones booleanas a partir de multiplexers.

Mediante un multiplexor, pueden implementarse funciones lógicas. Consideremos el mux de 4:1, con **enable** =1, resulta:

$$f = X_3C_1C_0 + X_2C_1/C_0 + X_1/C_1C_0 + X_0/C_1/C_0$$

Se observa que podemos seleccionar un término producto (un mintérmino) C_1C_0 de las de las 2^n posibles con n entradas (con $n = 2$ para éste caso). La salida será '1' ó '0' dependiendo si la entrada correspondiente está '1' ó '0'.

Por lo tanto podemos inicialmente implementar cualquier función en primera forma canónica (unión de mintérminos) de n (2 para el caso de un MUX 4:1) variables (con tres niveles de compuertas), donde las variables se conectan a las entradas de selección del MUX y los términos producto que intervengan en la función se habilitarán poniendo a '1' y los demás a '0' en las entradas del multiplexor.

Si bien parece que un MUX de n entradas de selección está limitado a generar funciones de hasta n variables, se puede extender éste número hasta $n+1$ variables, con el agregado en general de un inversor.

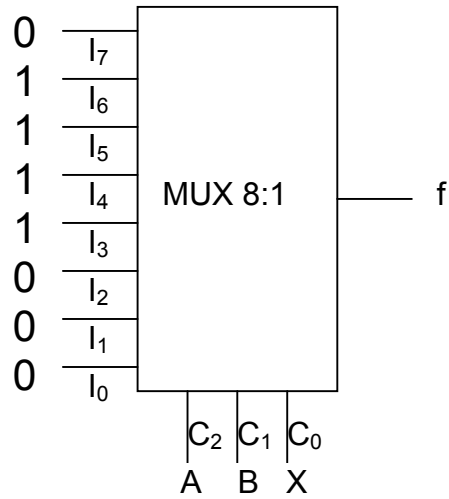
Como vimos anteriormente, para generar la función, poníamos en '1' las entradas X_i ($i = 0, 1, \dots, 2^n$) que corresponden a mintérminos que aparecen en la función, mientras que los demás a '0'. Si en cambio en algunas de éstas entradas conectamos una nueva variable (ya sea negada o sin negar) podemos generar una función de $n+1$ variables con en MUX de n entradas de selección.

Ejemplo:

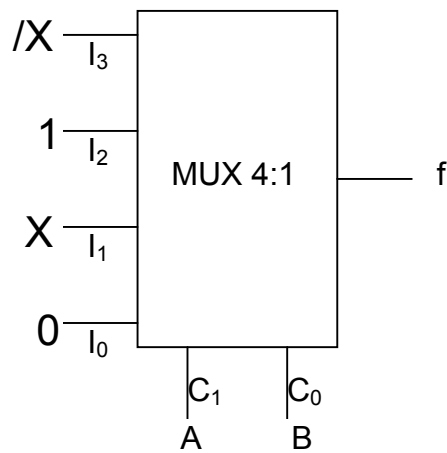
Sintetizar la función $f = A / B + /A B X + A B /X$ utilizando un MUX:

- a) 8:1.
- b) 4:1.

a)



b)



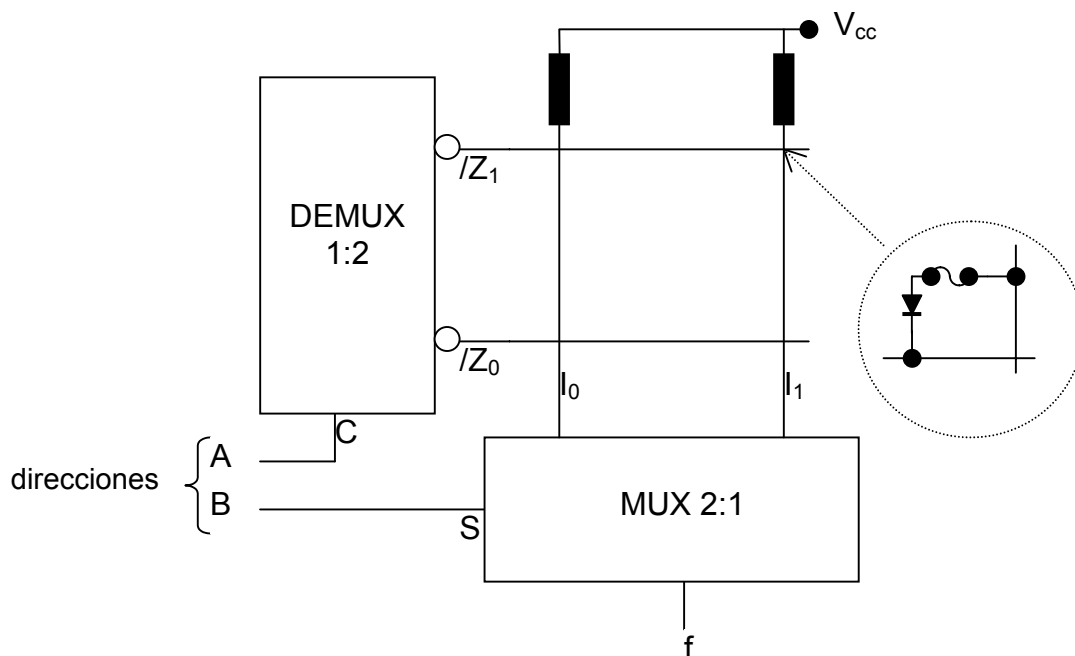
Apéndice C

C.1 Generación de funciones booleanas con ROMs

Tal como se muestra en el Apéndice B un multiplexer con n variables de entradas puede ser usado para generar funciones de orden m (m mayor que n), dividiendo la función original en 2^n subfunciones de $k=(m-n)$ variables cada una. Por otra parte, como se ve en el Apéndice A, un demultiplexer con k líneas de selección y 2^k líneas de salida permite generar las 2^n funciones mencionadas, usando para cada una de ellas una línea de OR/AND cableado.

La memoria ROM surge como combinación de las dos técnicas mencionadas anteriormente.

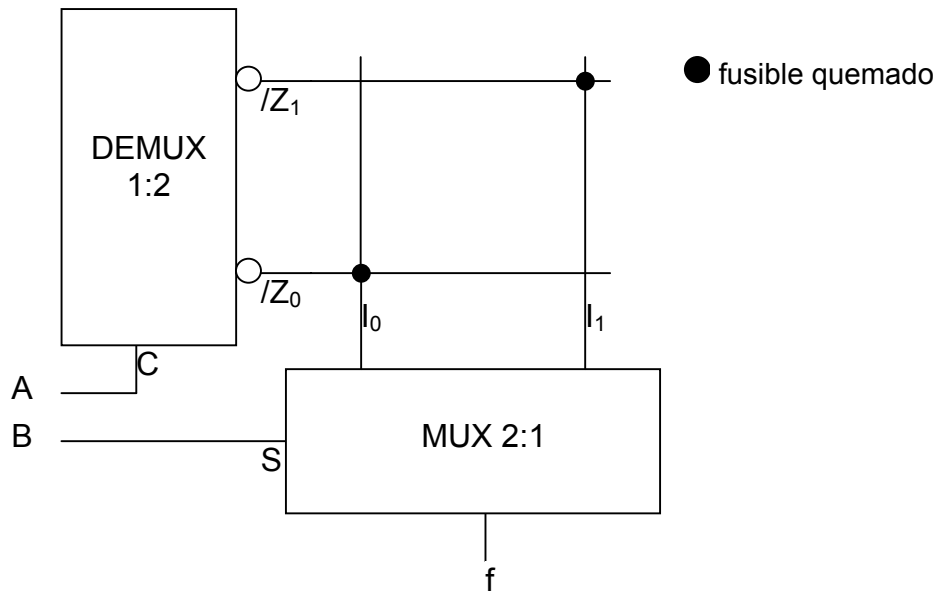
Vamos a tomar un ejemplo sencillo como el que se muestra la siguiente Figura para entender como podemos generar una función booleana con una ROM.



Las variable A de entrada selecciona una dada fila (Z_0 ó Z_1), a través del demultiplexador también llamado matriz de intersección, y en base a lo valores de la matriz de fusibles (matriz unión) de esa fila generan en las distintas columnas las funciones mediante OR cableado (véase Apéndice A, ítem A.2.1) que ingresan al multiplexador (I_0 ó I_1), que en base a la variables B selecciona el valor correspondiente a la función para el mintermino elegido, el que aparece en la salida de la ROM.

Mediante la quema de fusibles podemos (al igual que para decodificadores como muestra el Apéndice A) podemos elegir aquellas filas que mediante OR cableado ingresarán en las entradas I_0 e I_1 del multiplexador.

Supongamos ahora a modo de ejemplo que para programar quemamos los fusibles que marcamos como puntos en la siguiente figura:



Con lo cual:

$$f = B \cdot I_1 + /B \cdot I_0$$

$$I_1 = /C$$

$$I_0 = C$$

$$f = B \cdot /C + /B \cdot C$$

Conclusiones

El desarrollo de éste proyecto me sirvió para adquirir un conocimiento bastante completo sobre las distintas familias de lógica programable, y en especial sobre las FPGA. Además nunca había hecho una implementación práctica de un circuito lógico y creo que ésta es una forma muy moderna de comenzar, ya que la utilización de FPGAs se encuentra relacionada íntimamente con el diseño de circuitos lógicos de alta velocidad desde complejidad media hasta alta, de la ingeniería actual.

La elección del dispositivo programable empleado en la implementación como así también la complejidad de los circuitos, se debe a que la cátedra de 'Introducción a los Sistemas Lógicos y Digitales' ya poseía la plaqueta experimental utilizada y nuestro proyecto se limitaba a realizar un kit didáctico de enseñanza mediante ejemplos de aplicación utilizando la Upx10K10.

En la actualidad existen una gran cantidad de dispositivos FPGA con mejores prestaciones y menor precio, pero lamentablemente en nuestro país los mas complejos que se comercializan son los FLEX10K10, mientras que los superiores solo pueden adquirirse a través del contacto con las empresas fabricantes, las cuales se ubican en otros países.

Bibliografía

- “Circuitos digitales y procesadores”. Herbert Taub. McGraw Hill.
- “Síntesis y Descripción de Circuitos Digitales Utilizando VHDL”. Francisco Javier Torres Valle. Capítulo II, Dispositivos Lógicos Programables.
- “Sistemas Digitales. Principios y aplicaciones”. Ronald J Tocci.
- “Introducción a la lógica programada”. Ing Guillermo Adolfo Jaquenod. Facultad de Ingeniería. UNLP.
- “MAX+PLUS II. Getting Started”. Altera.
- “MAX+PLUS II. AHDL”. Altera.
- “Visual Basic 6.0. Edición Profesional en un solo libro”. Editorial GYR.
- Hoja de Características de dispositivo PAL/GAL 22V10.
- Hoja de Características del FPGA Xilinx XC 4000.
- Hoja de Características del FPGA Xilinx Virtex 2.5V.
- Hoja de Características del FPGA Xilinx Spartan IIE 1.8V.
- Hoja de Características del FPGA Altera FLEX 10K.
- Hoja de Características del FPGA Altera ACEX 1K.
- Hoja de Características del FPGA Altera Cyclone.
- Hoja de Características del FPGA Altera Stratix.
- Hoja de Características de FPGAs ACTEL.
- Hoja de datos del conversor ADC0820 de National.
- Hojas de datos de los componentes utilizados en las plaquetas.

Nota: Todas las hojas de características, hojas de datos, plaquetas diseñadas, programas desarrollados en MAX+plus II y Visual Basic, y demás están disponibles en un CD que se entregará a la Cátedra de Trabajo Final.

Páginas Web visitadas.

- Página web propia del fabricante Xilinx: <http://www.xilinx.com/>
- Página web propia del fabricante Altera: <http://www.altera.com/>
- Pagina web propia del fabricante Actel: <http://www.actel.com/>
- Página web propia del fabricante Cypress: <http://www.Cypress.com/>