Facultad de Informática

PROYECTO DE SISTEMAS INFORMÁTICOS
CURSO 2013/2014

Emulación de rayos cósmicos incidiendo sobre DSPs en FPGAs

Autores:
Sergio Hernández García
Daniel Laseca Chico

Director:
Marcos Sanchez-Élez

Autorización de difusión y utilización

Sergio Hernández García, Daniel Laseca Chico, alumnos matriculados en la asignatura Sistemas Informáticos, autorizan a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria como el código, la documentación y/o el prototipo desarrollado en el proyecto Portal web de gestión de información para estudiantes Erasmus, todo ello realizado durante el curso académico 2013-2014 bajo la dirección de Marcos Sanchez-Élez, profesor del Arquitectura de Computadores y Automática de la Facultad de Informática.

Sergio Hernández García Daniel Laseca Chico

Índice de contenido

Objetivos

En este proyecto se expone una herramienta de programación en alto nivel que permite establecer los parámetros de configuración de forma sencillas de las diversas operaciones aritmético-lógicas que permite realizar un DSP.

Mediante un interfaz sencilla se puede hacer descripciones a alto nivel de abstracción olvidando el nivel estructural de los componentes. Dotando su comportamiento a nivel de “caja negra”, estableciendo los valores de entrada del componente y la operación ha realizar. Y así obtener una instancia en lenguaje vhdl.

Se presenta al usuario una herramienta que permitan a los diseñadores crear, probar y modificar módulos prediseñados basados en DSP sobre una FPGA de manera ágil y sencilla.

Desde esta perspectiva, como segundo objetivo del proyecto, se especifican diferentes aplicaciones utilizando un modelo estructural jerárquico, es decir, creando módulos de mayor nivel que utilicen uno o varios módulos de menor nivel.

Finalmente, se otorga al usuario la posibilidad de generar módulos con un diseño más robusto, que se traduce en un aumento de la fiabilidad del componente.

In this project a tool for high-level programming that allows us to set the configuration parameters in a simple way of the various arithmetic and logical operations that allows a DSP is exposed.

Through a simple interface we will be able to create descriptions from a high abstraction level forgetting about the structural level of the components. Providing behavior-level "black box", setting the component input and the operation has performed. And get an instance in vhdl language. And so get an instance in vhdl language.

Presenting the user with a tool that allows designers to create, test and modify premade DSP modules on a FPGA in an agile and simple way.

From this perspective, as a second objective of the project, we have specified different applications using a hierarchical structural model, ie, creating higher level building blocks that use one or more lower-level modules.

Finally, the user is given the possibility of generating modules with a more robust design, resulting in increased reliability of the component.

Capitulo 1. Introducción

Se sabe que en general, un consumidor promedio interactuar con alrededor de 400 sistemas empotrados por día. Y estos datos tienden a ascender significativamente en los próximos años, si tenemos en cuenta que los procesadores son cada vez más pequeños, consumen menos energía. El número de aplicaciones y de ambientes soportados por los sistema empotrados crece cada vez más, y una de las principales objeciones para la realización de este proyecto.

Antes de entrar en materia, deberemos dar una definición formal de lo que entendemos por sistemas empotrados, un sistema computador (hardware + software) de propósito especial cuyo algoritmo que define su comportamiento no es susceptible de modificación, es decir, estos sistemas emplearán una combinación de recursos hardware (incluyendo elementos como Microcontroladores o Microprocesadores) y software para realizar una función específica.

Por lo tanto, un sistema empotrado es un sistema con un alto grado de heterogeneidad debido a la combinación del hardware a medida y vía software ya que pueden ser programados directamente mediante lenguaje ensamblador o mediante un compilador utilizando lenguajes de alto nivel como puede de ser C, C++ y en algunos casos BASIC.

Dichos sistemas están expuestos a una serie de limitaciones: poseen cantidades pequeñas de memoria de magnitud de KB, los procesadores poseen velocidades que no superan los Mhz y poseen restricciones de consumo de energía. Además hay que tener en cuenta las exigencias en cuanto a disipación de esta potencia, que también crecen, ya que se trata de introducir el mayor número de componentes electrónicos en un área reducida.

Si ademas el sistema empotrado va a utilizarse en el espacio, hay que tener en cuenta los efectos de la radiación. En concreto, los efectos de los llamados Single Effect Upsets (SEUs), ocasionados cuando una particula sub-atómica cargada golpea un flip-flop o una celda SRAM. La partícula deposita suficiente carga para causar que el flip-flop o la celda de memoria cambien de estado, corrompiendo la información contenida. Como esto ocurre sin dañar permanente el elemento de almacenamiento, se suele clasificar como un error leve.

Un buen sistema empotrado debe construirse satisfaciendo una serie de necesidades, respondiendo a las siguientes métricas de desarrollo:

1. Términos económicos.

    - Costo NRE (Nonrecurring Enginnering): El costo monetario de diseñar el sistema por primera vez. Una vez que el sistema está diseñado, cualquier número de unidades puede ser manufacturado sin incurrir en costo adicionales de diseño.
    - Costo unitario: El costo monetario de manufacturar cada copia del sistema, excluyendo el costo NRE.

2. Características de producción.

    - Tiempo para establecer el primer prototipo: El tiempo que se requiere para establecer la primera versión funcional que responda a nuestra necesidades.
    - Tiempo de necesario para una versión estable: El tiempo requerido para desarrollar un sistema al punto en que pueda ser lanzado al mercado.

3. Características del producto.

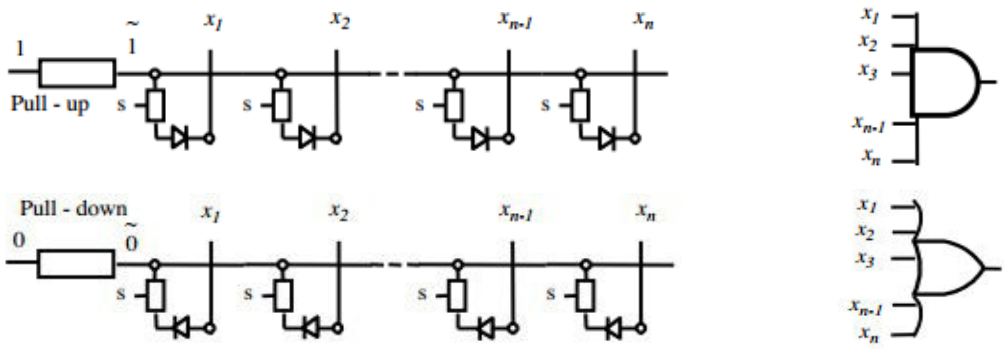
    - El tiempo de ejecución del sistema.
    - Tamaño: El espacio físico requerido para el sistema. Para determinarlo en software se establece en byte y a nivel hardware en transistores o puertas.
    - Energía: La cantidad de energía consumida por el sistema en funcionamiento, que puede determinar el tiempo de vida de la batería.
    - Flexibilidad: El potencial que establece el sistema para incurrir en una nueva funcionalidad sin recurrir a un gran coste NRE.
    - Robustez: El dispositivo tendrá que satisfacer a las especificaciones a pesar de la incertidumbre.

Las características del producto se vuelven más restrictivas en el caso del campo aeroespacial, debido a la necesidad de eficiencia y seguridad nativas del entorno.

A lo largo de la historia, para poder desarrollar sistemas empotrados respondiendo a las expectativas anteriormente mencionadas, la industria de la electrónica a recurrido a un Dispositivo Lógico Programable (PLD) , debido a que la fabricación de un prototipo resultaba muy costoso y un error de diseño hacia que el chip fuera inservible. Los PLD no tenían definida una tarea especifica al salir de la fabrica. Permitiendo ahorrar costo y tiempo en el diseño.

Estos se basan, en que cualquier función lógica se puede expresar como la suma de productos

$$F=m1+m2+m3+ \ldots=xyz+ zyz+xyz$$

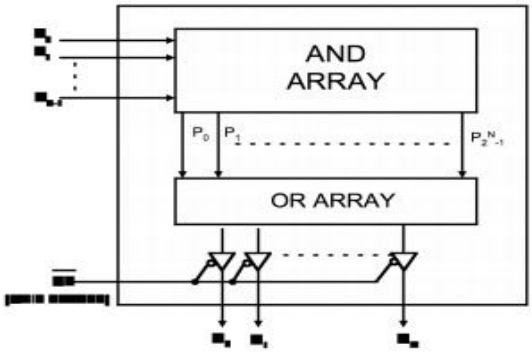


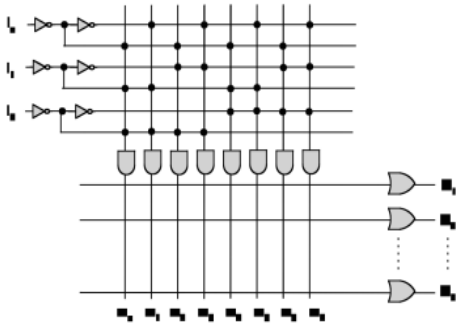
Utilizan diversas tecnología

1. Basada en fusible: programados una sola vez y la información proporcionada no es volátil
2. SRAM: es reconfigurarble pero volátil
3. Tecnología EPROM y EEPROM: reconfigurable y no volátil.

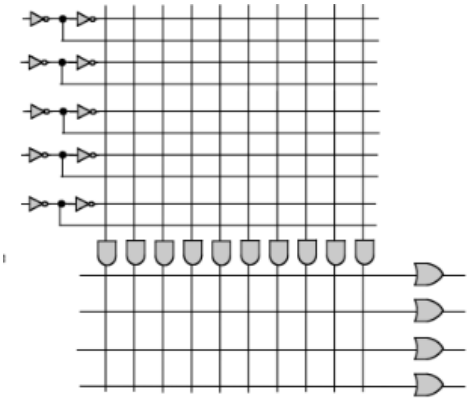
Realizaremos un pequeño repaso por los tipos de dispositivos programables más importantes.

- Read-Only Memory (PROM) son sistemas programables de sólo lectura. Su arquitectura interna esta formada por un número fijo de términos AND que alimenta una matriz programable OR. Se suelen principalmente para decodificar las combinaciones de entrada en funciones de salida.





- Array Lógico Programable (PLA) a diferencia que las PROM son más pequeñas y flexibles de ya que ambos conjuntos de puertas AND y OR son programables, lo que permite implementar cualquier termino producto, no solo mintérminos. Por contrapartida, son dispositivos más lentos al tener una etapa más que ser programada.



- Generic array logic (GAL) mantiene las mismas propiedades lógicas que el PAL, esta formada por una matriz de puertas AND programables cableadas con otra matriz de puertas OR fijas pero puede ser reprogramado y borrado debido a que la celda esta formada por CMOS. Al ser reprogramable resulta muy útil en la fase de prototipado.

- Como evolución de las PLAs surgieron las FPGAs, dispositivos que desarrollaremos con mayor detenimiento en el siguiente capítulo, ya que sera nuestra herramienta de trabajo.

Las herramientas CAD cubren las diferentes etapas de diseño de circuitos integrados, desde síntesis automática a partir de descripciones HDL, hasta la realización de layouts full-custom.

El diseño de un sistema digital usando herramientas CAD es un flujo de tareas.

Se inicia extrayendo las funcionalidades que se desea que realice un circuito hardware dedicado de la caracterización del sistema completo. A continuación, dicho circuito se especifica mediante un lenguaje de descripción de hardware (típicamente VHDL o Verilog) y se simula con un modelo de comportamiento del resto del sistema. Seguidamente, utilizando diferentes herramientas CAD especializadas en diferentes niveles de abstracción, se exploran en un corto espacio de tiempo diferentes diseños alternativos según las ligaduras (área, velocidad, consumo) especificadas por el diseñador para el sistema digital concreto que se está diseñando. Una vez alcanzado el diseño idóneo, éste se proyecta sobre una tecnología programable, de tal manera que en muy poco tiempo disponemos de un prototipo hardware en funcionamiento.

Si el prototipo supera con éxito la evaluación, el diseño puede proyectarse definitivamente sobre un circuito integrado.

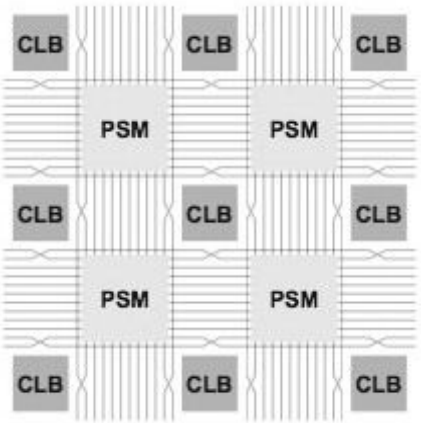
Ventajas:

- Permite a los fabricantes abaratar los costes de producción, reduciendo la mano humana de los operadores y, disminuyendo también, los posibles errores que puedan ocurrir durante el proceso.
- Los sistemas se diseñan más rápido.
- Analizar la viabilidad de un producto.
- Calcular el coste de la fabricación.
- Mejor adaptación a las exigencias del mercado.

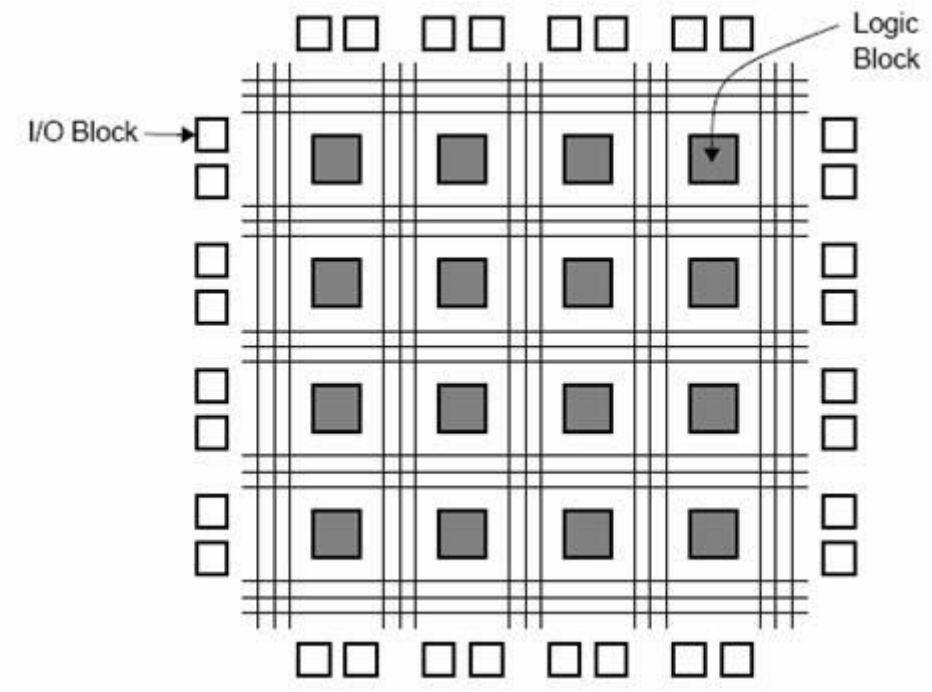
Capitulo 2. Hardware Reconfigurable

Las FPGAs fueron inventadas en el año 1984 por Ross Freeman y Bernard Vonderschmitt, co-fundadores de Xilinx, y surgen como una evolución de los PLAs.

Estas se componen de bloques lógicos configurables (CLB) y una matriz de interconexión. Cada bloque lógico esta formado por: flip-flops, multiplexores a la entrada y salida de bloque, LUTs (Look-up-Tables) tablas de verdad que permiten implementar funciones lógicas sencillas y una memoria que para guardar la configuración de los elementos de este. Los CLB como unidad individual, son capaces de realizar tareas sencillas, necesitan de un entramado de interconexiones organizadas jerárquicamente denominadas PSMs (Programmable Swiching Matrix), que permiten la propagación de señales, algunas de estás son específicas como la encargada de la transmisión del reloj.



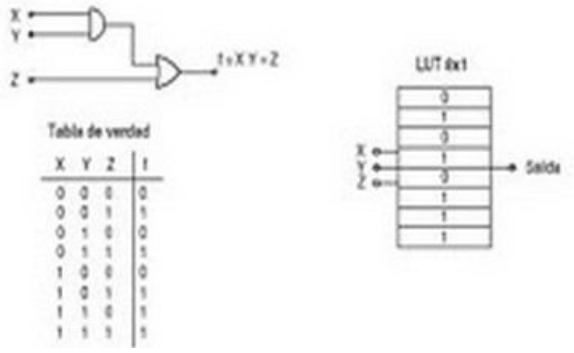
También dispone de otros bloques dedicados para las entradas y salidas del FPGA llamados IOB, bloques de entrada y salida, estos se encuentran en los bordes del dispositivo, son los encargados del paso de señales del hacia el interior y el exterior del dispositivo a través del pin. Estos bloques también son configurables mediante una memoria.



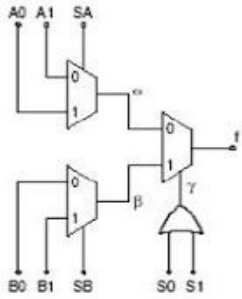
- Bloques Lógicos Configurables (CLB)

  Los bloques lógicos están compuestos por una parte secuencial que permite sincronizar los elemento del bloque con la señal de reloj procedente del exterior y una combinacional encargado de establecer funciones lógicas. Hay que destacar:

  - Bloque lógico basado en LUT (look-up table): Una LUT es una función lógica guardada en una memoria, almacenada en forma de tabla de verdad. Las direcciones son las entradas de la función ha implementar y en la celda se genera el resultado para esa combinación de entradas. Para una LUT de n*1 nos permite implementar cualquier función lógica que contenga como máximo n entradas.



  - Bloque lógico basado en multiplexores: Están basados en multiplexores , estos no permiten implementar cualquier función lógica de n entradas,. Como ventaja, las funciones que implementa requieren menos espacio, permitiendo aprovechar mejor el área de la FPGA



  - Elementos de almacenamiento. Permiten guardar las señales de de control, suelen utilizarse flip-flop o latch.

  - Lógica de acarreo, permitiendo mejorar el rendimiento de sumadores, comparadores, contadores, etc.

- Interconexión entre CLBs

  Para poder establecer modelos de gran complejidad necesitamos recurrir a varios CLBs para ello tenemos que disponer de números canales para poder realizar las comunicaciones entre ambos.
  Atendiendo a las diferentes tecnologías empleadas en interconexión, tenemos diferentes modelos de FPGAs. Las más importantes se basan en:

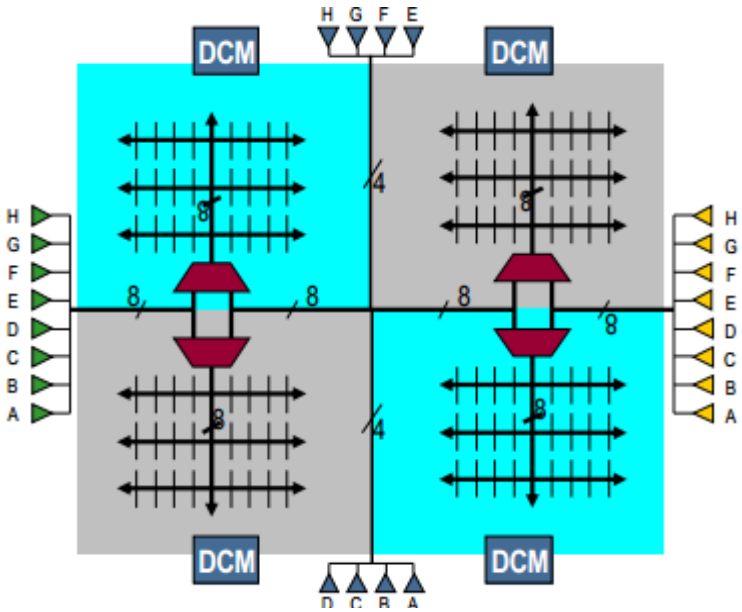
  - SRAM (StaticRAM). Los datos de interconexión se guardan en memoria, almacenando funciones lógicas y datos de control de los multiplexores, Al ser memorias de tipo SRAM, el contenido de la memoria se pierde cuando deja de percibir energía, por lo tanto cada vez que queramos canfigurarla tendremos que escribir en ellas.
    Este tipo de FPGAs también suele disponer de bloques de memoria de tipo EEPROM, para albergar información de las direcciones de control de las memoria SRAM.

  - Flash: Las FPGAs basadas en celdas flash son reprogramables y de carácter no volátil. Problema es que incrementa considerablemente el coste final del dispositivo

- Bloques de Entrada/Salida (Input/Out Block, IOB)
  Para poder transmitir la señales digitales a la FPGA, estos no permiten seleccionar diferentes rangos de tensiones, la frecuencia de los pulsos, dotándola de mayor funcionalidad. Estos pines pueden configurarse de entrada, salida o bidireccionales.
  Las comunicaciones pueden establecerse a través de Buffer de E/S configurares de acuerdo a los estándares fijados o mediante bancos de bloques de E/S.

En términos generales, las FPGAs que utilizan bloques de tamaño pequeño aprovechan mejor los recursos de los que dispone el sistema, pero necesita disponer de una gran red de interconexionado lo que generar retardos, si utilizamos gran cantidad de bloques. Cuando los bloques son de gran tamaño, podemos llegar a implementar funciones lógicas de varios términos dentro de este sin afectar a la frecuencia máxima de trabajo, ya que aumenta paralelismo de procesos con el mismo retardo para todas las funciones, en el caso de que las funciones sean pequeñas, no se necesita utilizar todas los componentes internos, desaprovechando su potencial

Otro de los componentes que integran las FPGAs son los bloques dedicados a que la señales de reloj se transmitan de forma síncrona a través de todos los componentes que forman el sistema, en el caso de Xilinx, los bloques reciben el nombre de Digital Clock Managers. El numero de estos bloques depende del tamaño de la propia FPGAs. Estos bloques se encarga de dividir o multiplicar la frecuencia de reloj y eliminar el sesgo de elementos externos o componentes internos a la FPGA
Un ejemplo representativo seria la siguiente imagen que representa el diseño de la FPGA Spartan, que esta alimentada por cuatro DCMs



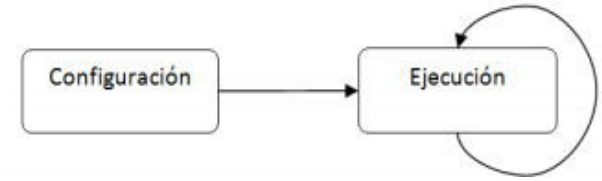
Configuración de una FPGA

Una FPGA a diferencia de otros dispositivos como microcontroladores o microprocesadores que son programables, es decir, reciben un empaquetamiento de ceros y unos que denominamos programa, las FPGAs son dispositivos reprogramables, que reciben una determinada configuración (configuración bitstream) que determina el comportamiento de está.

Reconfiguración de una FPGA

Las FPGAs ofrecen diferente formas de reconfiguración de los bloque e interconexiones:

- Reconfiguración estática. Para poder realizar una configuración total del dispositivo es necesario parar la ejecución. Realizar una nueva configuración que borre los parámetros de configuración previos a la carga



- Reconfiguración dinámica. Nos permite configurar la FPGA mientras está se esta ejecutando.

- Existen diversos modos de configuración:
    - Único contexto. No permite que coexistan dos configuraciones, por lo tanto la configuración nueva sustituye a la anterior aunque los cambios sean mínimos.
    - Multicontexto. Para cada bit de configuración se le asigna varios lo que permite tener diferentes planos o configuraciones. Cada plano debe configurarse de igual que si se tratara de un contexto único. Como inconveniente la latencia es eleva.
    - Reconfiguración parcial. Permite modificar la configuración de parte del dispositivo, mientras este se ejecuta de forma ininterrumpida. El plano de configuración adquiere la metodología de una memoria RAM, determinadas direcciones se establecen para guardar los datos de configuración y durante su ejecución estos se van modificando, para dotar al sistema de una nueva configuración. Esto permite reducir la latencia en los cambios de contexto evitando cargar los bitstream de las configuraciones parciales y la utilización utilizar áreas no utilizadas.

Programación

Como hemos explicado anteriormente, el funcionamiento de una FPGA, resulta complejo establecer la funciones que se implementan en los CLB, determinar las interconexiones, configuración del dispositivo, etc. Para facilitar la labor al usuario se han diseñado herramientas basadas en diagramas esquemáticos o mediante lenguajes de programación de alto nivel conocidos como HDL. Los más conocidos son:

- VHDL(Very Hgth Speed Hardware Description Languaje)
  Se caracteriza por ser un lenguaje estándar, portable precisando que es necesario realizar algunos ajuste en función del dispositivo o la tecnología y da soporte a herramientas CAD (Diseño Asistido por Computadora)

- Verilog. Se caracteriza por ser un lenguaje parecido a C, sin embargo ejecución de las sentencias no es estrictamente lineal.

Conclusiones

Tras haber presentado un pequeña introducción sobre los circuitos integrados, podemos establecer algunas características globales sobre las FPGAs, como ventajas establecemos la siguiente enumeración:

- *Rendimiento:*
  Si evaluamos hardware frente a software, la ejecución de un circuito diseñado específicamente para realizar una determinada tarea, esté sera más rápido que un proceso desarrollado mediante software.
  Si nos centramos específicamente en hardware, las FPGAs nos permiten descomponer las tareas en subtareas más sencillas y obtener un alto nivel de paralelismo, ejecutando dichas subtareas de forma simultanea.

- *Tiempo en llegar al mercado:*
  Al tratarse de una tecnología configurable, permite establecer un prototipo, sin pasar por una linea de producción. Además la FPGA nos permite chequear la aplicación y modificarla.
  Al combinar software y hardware nos permite abstraernos del funcionamiento de la circuitería, favoreciendo al usuario su uso.

- *Reducción del consumo:*
  Podemos reducir el consumo mediante el paralelismo, para ello debemos conocer la arquitectura interna del dispositivo con la finalidad de obtener diseños específicos a nuestras necesidades.

- *Precio:*
  Como se ha comentado anteriormente, al poder desarrollar un prototipo con gran celeridad permite ahorrar en coste, si los comparamos con los circuitos integrado para aplicaciones especificas (ASIC).

- *Flexibilidad:*
  Al estar formada por elementos configurables total o parcialmente, permite adaptarse a la necesidades del mercado cambiante como es el de la electrónica.

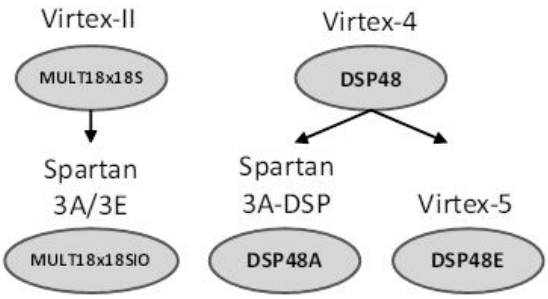
Como inconvenientes hay que destacar:

- *Tiempo de reconfiguración:*
  Supone un cuello de botella si tratamos de establecer un sistema multitarea.

- *Rutado de señales:*
  El encargado por del rutado realizado en la compilación no suele ser optimo y para obtener un diseño optimo es necesario que el usuario disponga de conocimiento avanzados del dispositivo.

Capítulo 3. Introducción al DSP48E

En esta sección estudiaremos el dispositivo DSP48E generado desarrollado para la familia de Virtex5, exponiendo el origen del modulo, y su arquitectura interna.

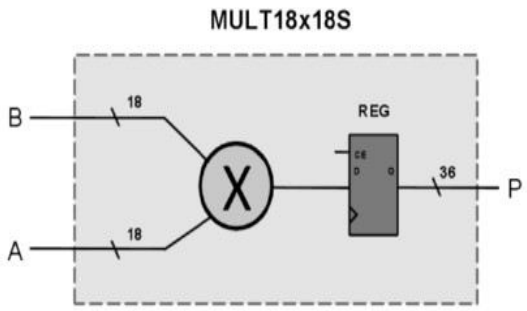


Evolución de las familias FPGAs de Xilinx

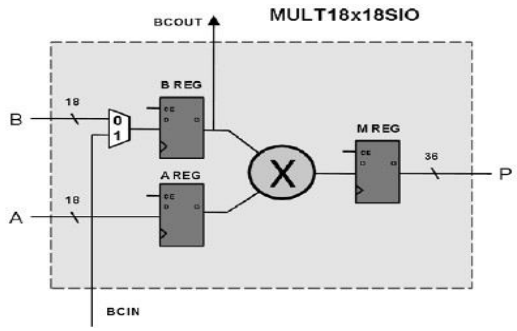
Las primeras sumas o multiplicaciones que tenían lugar dentro de las FPGAs recurrían a algoritmos implementados sobre los LUTs incorporados sobre los CLB.
Para poder mejorar las prestaciones de las FPGAs, la empresa Xilinx, empezó a diseñar componentes basados en funciones especificas, pero que a su vez fueran flexibles.
Con la irrupción al mercado de la familia de FPGAs Virtex, las multiplicaciones ya se realizan sobre componentes dedicado, el MULT18x18S. Un multiplicados de números de 18 bits con salida de 36 bits.
Posteriormente, Xilinx sacó las Spartan 3A y 3E que se caracterizaron por su bajo coste, las cuales, ya introducían un versión mejorada de MULT18x18SIO, logrando una frecuencia de trabajo superior a 400 MHz y respecto a su arquitectura se añadieron registros En referencia a la funcionalidad, ofrecía la posibilidad de sumar, restar y operaciones de rotación.

La Virtex-4 incorporaba un nuevo bloque denominado DSP48, un DSP con salida para 48 bits, este permitía realizar operaciones que incluyen multiplicar, multiplicar y acumular(MACC), multiplicar y sumar, suma de tres componentes, desplazamiento, multiplexores de bus ancho, comparador de magnitud y cuenta amplia en el mismo modulo. Se añadieron registros intercalados para dar una estructura de pipeline. Permitiendo frecuencias de trabajo del orden de la 400 Mhz.
La salida de 48 bit permite la unión de mas componentes DSP con un gran rendimiento y un coste energético bajo. Además este camino dispone de un desplazamiento a la derecha de 17 bits para aritméticas de múltiple precisión. Así un producto parcial de un DSP puede ser desplazado y sumado al siguiente producto parcial que ha sido realizado por otro DSP permitiendo la configuración de operandos de cualquier tamaño.
La entrada del operando C permite el uso de muchas funciones matemáticas de 3 operandos como por ejemplo suma de 3 operandos o multiplicación de dos operandos y suma de un tercero.

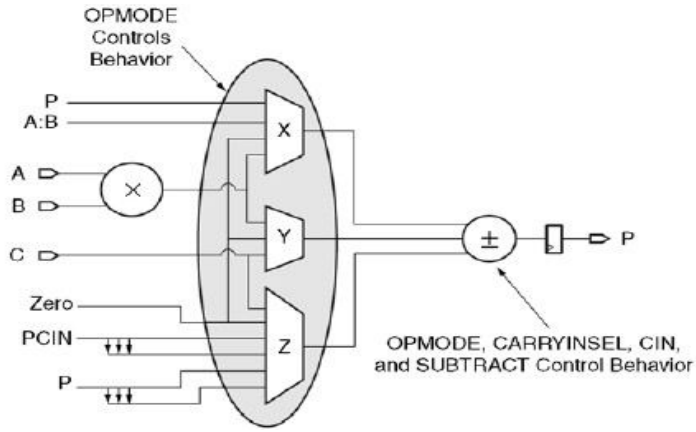
La siguiente de familia de FPGAs, la Virtex5 incorporó un nuevo bloque denominado DSP48E, el bloque sumador se convirtió en una ALU multifunción. A las funciones del DSP48 se le añaden otras como funciones lógicas y la detección de patrones, se añadieron salidas para detectar el overflow y underflow. Introduce mejoras para mejorar la flexibilidad, mejora la eficiencia de las aplicaciones, reduce el consumo energético general, aumenta la frecuencia máxima permitiendo trabajar en frecuencias superiores a 450MHz y reduce los tiempos de set-up y clock-to-out. Se permite a los diseñadores implementar múltiples operaciones lentas en un mismo bloque DSP48E mediante técnicas de multiplexado de tiempo.



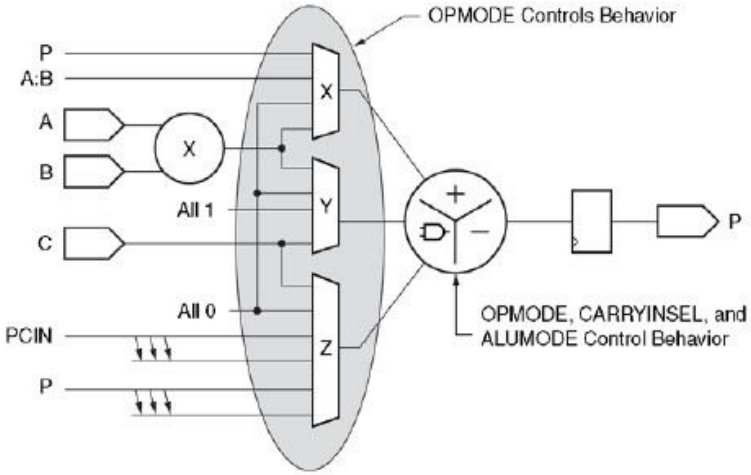
Esquema simplificado del Mult 18x18S



Esquema simplificado del Mult 18x18SIO



Modelo simplificado del DSP48 de la Virtex-4



-

Modelo simplificado del DSP48 de la Virtex-5

Comparativa entre diversos módulos DSP de Xilinx

Función	DSP48	DSP48E	DSP48A	Beneficios
Multiplicador	18x18	25x18	18x18	Se reducen los recursos necesarios para algoritmos de DSP
Pre–sumador	no	no	si	Reducción del camino crítico para filtros FIR
Entradas en cascada	1	2	1	Permite conectar en cascada los módulos para filtros grandes
Salidas en cascada	si	si	si	Permite conectar en cascada los módulos para filtros grandes
Entrada dedicada C	no	si	si	Permite realizar operaciones matemáticas con 3 entradas
Entradas de sumador	3x48 bits	3x48 bits	2x48 bits	Soporte para suma y acumulación
Modo dinámico	si	si	si	Soporte para múltiples operaciones configurables en tiempo de ejecución
Funciones lógicas ALU	no	si	no	Similar a la ALU de un microprocesador
Detección de patrones	no	si	no	Detección de saturación, overflow y otros
Soporte de múltiples ALU	no	si	no	Permite realizar en paralelo múltiples operaciones con la ALU
Señales de carry	Carry in	Carry in–out	Carry in–out	Carry rápido, útil para interconectar DSPs

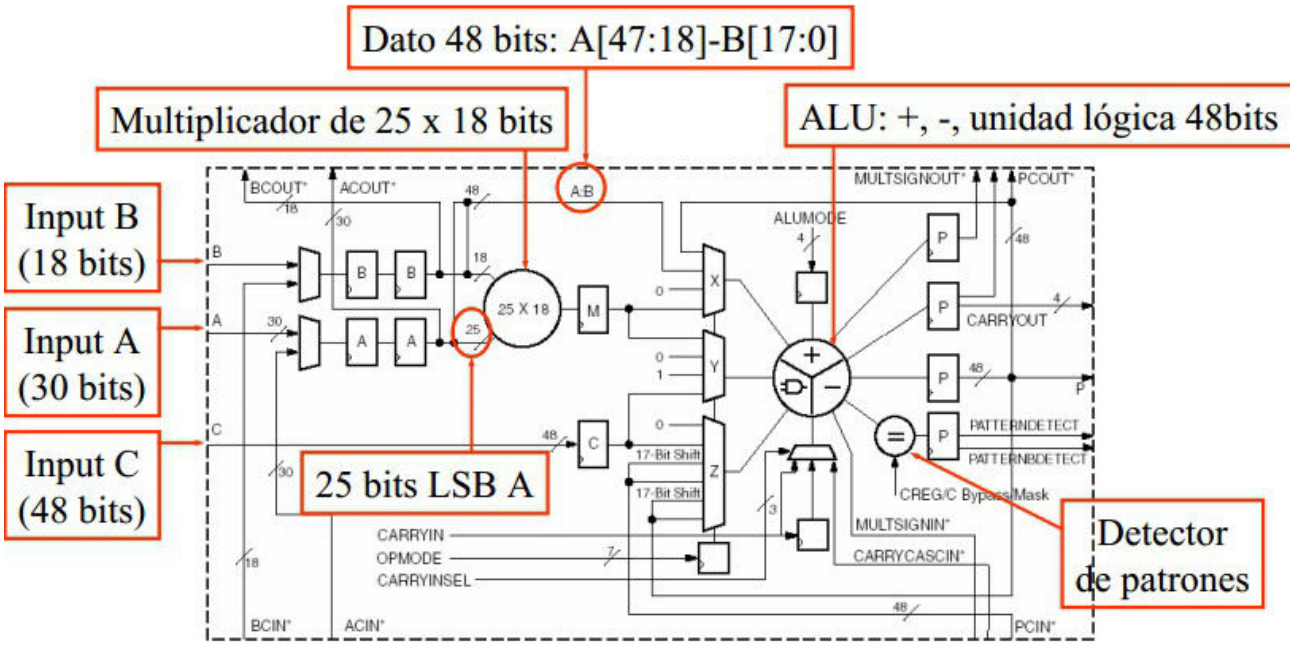
Arquitectura del DSP

La concepción de los DSP, intenta optimizar dos características básicas: precisión y velocidad. Algunos factores que influyen en la precisión son:

- La anchura del bus, que está relacionada con la precisión de los datos y resultados en memoria.
- La utilización de aritmética expandida, para evitar desbordamientos en los cálculos intermedios y finales.
- El uso de escalado previo de los datos, que permite modificar el rango dinámico de la señal a tratar evitando también desbordamientos.

Respecto a la velocidad, el factor más influyente es:

- Cálculo eficiente de productos y sumas.

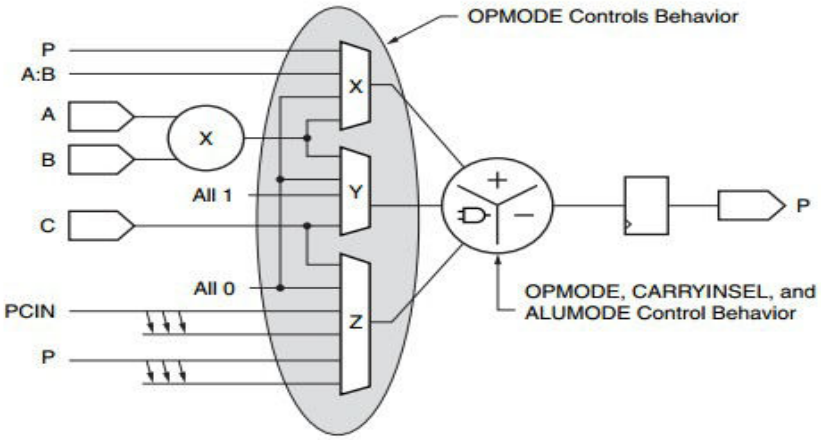


Arquitectura interna del DSP48E

Ahora destacaremos alguna de las características más importantes del modulo DSP48E:

- Multiplicador dedicado de 25x18 bits con salida de 48 bits.
- Arquitectura pipeline para funcionar por encima de los 450 Mhz.
- Elección de operandos por medio de registro OPMODE.
- Selección de la señal de acarreo por medio del registro CARRYINSEL .
- Configurar la operación por medio del registro ALUMODE.
- Concatenación del bus de 18 bits correspondiente a B y el bus de 30 bits correspondiente a A, el cual se conecta al multiplexor A.
- Selección del bus de 48 bits en los multiplexores Z y Y.
- Selección personalizada de la entrada hacia A y B, entre modalidad directa o modalidad en cascada por medio de ACIN y BCIN. En C esta posibilidad no se presenta.
- Habilitación independiente de los registros intermedios de A y B, aunque puede ser utilizado sin ningún registro.
- Funciones SIMD (Múltiples Datos Instrucción Única), de dos operaciones de 24 bits o cuatro de 12.
- Registro para toma de datos P de 48 bits, con posibilidad de retroalimentación
- Selección de la señal de acarreo por medio del registro CARRYINSEL
- Establecer comparaciones mediante el detector de patrones

Uno de los módulos en los que nos hemos centrado para nuestro trabajo, es el encargado de las operaciones aritmético-lógicas por lo tanto, realizaremos un análisis en profundidad.



Operaciones aritméticas:

En función de la lógica de control de cada multiplexor determinamos la entrada de operando.

Bits de control para el multiplexor X				
Z (OPMODE[6:4])	Y (OPMODE[3:2])	X (OPMODE[1:0])	Salida Multiplexor X	Notas
xxx	xx	00	0	Salida por defecto
xxx	01	01	M	Es necesario establecer OPMODE[3-2]=01
xxx	xxx	10	P	
xxx	xxx	11	A:B	
Bits de control para el multiplexor Y				
xxx	00	xx	0	Salida por defecto
xxx	01	01	M	Es necesario establecer OPMODE[1-0]=01
xxx	10	xx	48'fffffffffff	
xxx	11	xx	C	
Bits de control para el multiplexor Z				
000	xx	xx	0	Salida por defecto
001	xx	xx	PCIN	
010	xx	xx	P	
011	xx	xx	C	
100	10	00	P	
101	xx	xx	Shift(PCIN)	
110	xx	xx	Shift(P)	
111	xx	xx	xx	Expresión no reconocida

Configuración del ALUMODE:

Operación	ALUMODE [3:0]
Z + X + Y + CARRYIN	0000
Z – (X + Y + CARRYIN)	0011
-Z + (X + Y + CARRYIN) – 1 = not (Z) + X + Y + CARRYIN	0001
not (Z + X + Y + CARRYIN) = -Z – X – Y – CARRYIN - 1	0001

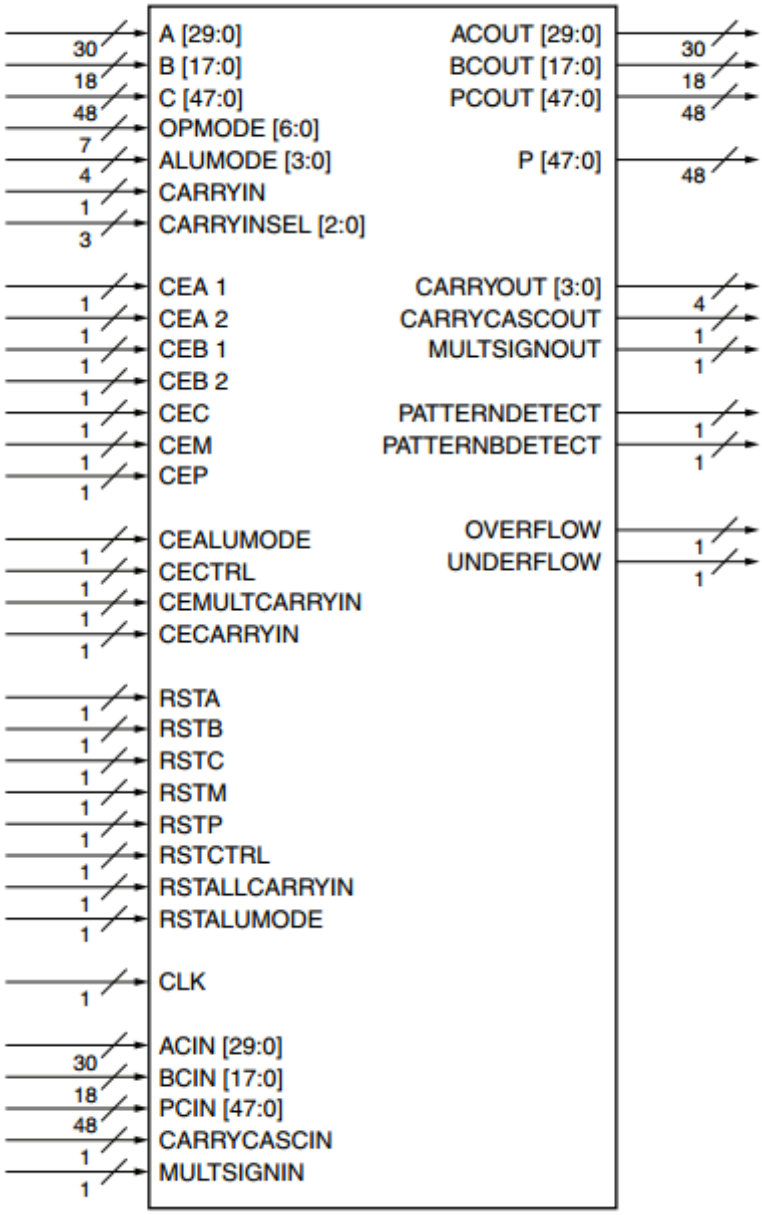
Seleccionar entre las diversas configuraciones de acarreo:

CarryInSel	Operación	Descripción
000	CARRYIN	Como carry toma la entrada CARRYIN
001	~PCIN[47]	Redondeo al infinito de PCIN
010	CARRYCASCIN	Como carry toma la entrada CARRYCASCIN
011	PCIN[47]	Redondeo al cero de PCIN
100	CARRYCASCOUT	Como carry toma la salida CARRYCASCOUT
101	~P[47	Redondeo al infinito de P
110	A[24] XNOR B[17]	Redondeo AxB
111	P[47]	Redondeo a cero de P

Operaciones lógicas

Operación	ALUMODE [3:0]	OPMODE [3:2]
X XOR Z	0100	00
NOT (X XOR Z)	0101	00
X OR Z	1100	10
NOT(X) OR Z	1111	00
X OR NOT(Z)	1101	10
NOT(X OR Z)	1110	10
X AND Z	1100	00
NOT(X) AND Z	1111	10
X AND NOT(Z)	1101	00
NOT(X AND Z)	1110	00

Entradas y salidas del dispositivos



Entradas y salidas del DSP48E

Para poder trabajar adecuadamente con un DSP48E, es necesario instanciarlo de manera adecuada otorgando valor a los diferentes parámetros entrantes del componente. Expondremos a continuación cada uno de ellos, su ancho de palabra característico y su utilidad.

Los parámetros de configuración del DSP48E los podemos clasificar de acuerdo a su funcionalidad en:

- Parámetros de entrada que permiten seleccionar el origen de éste (lógica o DSP adyacente)
- La configuración del multiplicador, dando la posibilidad de deshabilitar elementos, lo que permite ahorrar energía.
- Lógica de control de los multiplexores que controlan la entrada y salida de datos y registro.
- Parámetros de control de la unidad aritmético-lógica, que pueden ser modificados de forma dinámica ciclo a ciclo.

Dado que no nos interesa desarrollar toda la funcionalidad que nos ofrece, nos centraremos en las siguientes entradas y salidas del dispositivo.

Nombre	Tipo	Ancho de la palabra	Descripción
Puertos de datos			
A	Entrada	30	A[24:0] en el caso de ser utilizado por el multiplicador, A[29: 0] bits más significativos si concatenamos A:B
B	Entrada	18	B[17:0] como entrada del multiplicador o concatenación A:B
C	Entrada	48	Datos de entrada para el modulo pre-sumador/restador, comparador o modulo de operaciones lógicas
CARRYIN	Entrada	1	Entrada de Carry
CARRYINSEL	Entrada	3	Selecciona entre los diferentes formas de carry
P	Salida	48	Salida de datos del sumador/restador o del modulo de operaciones lógicas
CARRYOUT	Salida	4	
Puertos de control			
OPMODE	Entrada	7	Señales de control de los multiplexores X, Y y Z
ALUMODE	Entrada	4	Selecciona la función lógica ha realizar
CLK	Entrada	1	Permite sincronizar los registros y flip-flops internos
Puertos en cascada			
ACOUNT	Salida	30	Salida de datos en cascada hacia la entrada ACIN de otro DSP48E
BCOUNT	Salida	18	Salida de datos en cascada hacia la entrada BCIN de otro DSP48E
PCIN	Entrada	48	Entrada que procede de la salida P de otro DSP48E conectado en cascada
PCOUT	Salida	48	Dato de salida en cascada hacia la entrada PCIN de otro DSP48
Puertos de Reset y Habilitación			
CE{x}	Entrada	1	Habilitación de registros A1,A2, B1,B2, C, M, P, ALUMODE, CTRL, CARRYIN,MULTICARRYIN y OPMODE
RS{x}	Entrada	1	Reset de registros A, B, C, M, P, CTRL, ALLCARRYIN y ALUMODE

Instancia del módulo DSP48E de Xilinx para la Virtex5 en VHDL

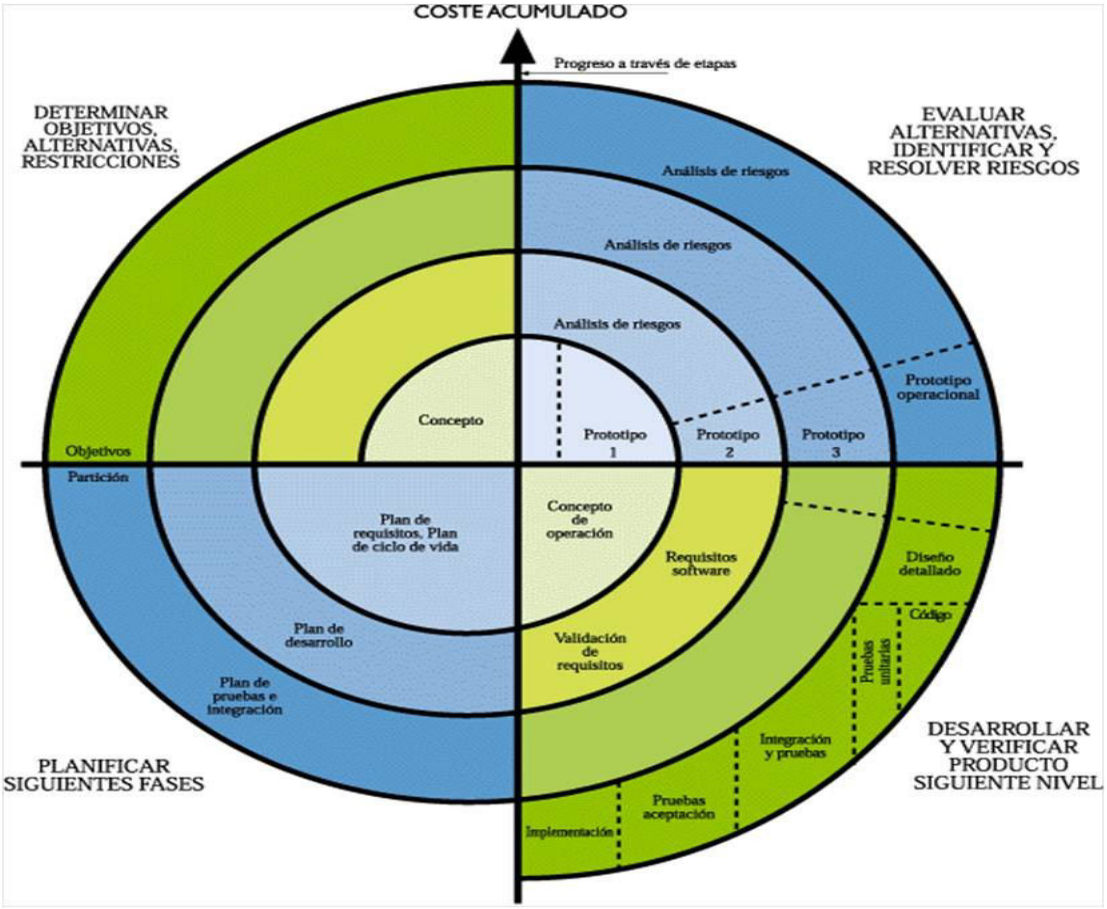
A continuación presentamos un ejemplo VHDL de la instanciación de un módulo DSP48E

```
Library UNISIM;
use UNISIM.vcomponents.all;
-- DSP48E: DSP Function Block
--         Virtex-5
-- Xilinx HDL Libraries Guide, version 10.1.2

DSP48E_inst : DSP48E
generic map (
ACASCREG => 1,           --  Número de registros en el pipeline entre
                         -- la entrada A/ACIN y salida ACOUT, 0, 1, o 2
ALUMODEREG => 1,     -- Número de registros en el pipeline de la entrada ALUMODE,  0 o 1
AREG => 1,               -- Número de registros en el pipeline de la entrada  A , 0, 1 o 2
AUTORESET_PATTERN_DETECT => FALSE,                  -- Auto-reset  si detecta un patrón, TRUE o FALSE
AUTORESET_PATTERN_DETECT_OPTINV => "MATCH", -- Reset si "MATCH" o "NOMATCH"
A_INPUT => "DIRECT", -- Selecciona la entrada A usada, "DIRECT" (puerto A) o "CASCADE" (puerto ACIN)
BCASCREG => 1,           -- Número de registros en el pipeline entre la entrada  B/BCIN y la salida BCOUT, 0, 1, o 2
BREG => 1,                    -- Número de registros en el pipeline de la entrada  B, 0, 1 o 2
B_INPUT => "DIRECT",       -- Selecciona la entrada B usada, "DIRECT" (puerto B) o "CASCADE" (puerto BCIN)
CARRYINREG => 1,              -- Número de registros en el pipeline de la entrada  CARRYIN, 0 o 1
CARRYINSELREG => 1,           -- Número de registros en el pipeline de la entrada  CARRYINSEL, 0 o 1
CREG => 1,                    -- Número de registros en el pipeline de la entrada  C, 0 o 1
MASK => X"3FFFFFFFFFFF",   -- Valor de la máscara de 48-bit para la detección de patrón
MREG => 1,                    -- Número de registros en el pipeline del multiplicador, 0 o 1
MULTCARRYINREG => 1,       -- Número de registros en el pipeline para el bit de acarreo de la multiplicación, 0 o 1
OPMODEREG => 1,               -- Número de registros en el pipeline de la entrada OPMODE, 0 o 1
PATTERN => X"000000000000", -- Patrón de 48-bit a comparar para la detección de patrón
PREG => 1,                    -- Número de registros en el pipeline de la salida P, 0 o 1
SIM_MODE => "SAFE",           -- Simulación: "SAFE" vs "FAST", ver "Synthesis and Simulation
                              -- Design Guide" para más detalles
SEL_MASK => "MASK",           -- Selección del valor de la máscara entre el valor "MASK" o el valor del puerto"C"
SEL_PATTERN => "PATTERN", -- Selección del valor del patrón entre el valor "PATTERN" o el valor del puerto "C"
SEL_ROUNDING_MASK => "SEL_MASK",    -- "SEL_MASK", "MODE1", "MODE2"
USE_MULT => "MULT_S",                -- Selección del uso del multiplicador, "MULT" (MREG => 0),
                                     -- "MULT_S" (MREG => 1), "NONE" (no se usa el multiplicador)
USE_PATTERN_DETECT => "NO_PATDET",  -- Activa la detección de patrón, "PATDET", "NO_PATDET"
USE_SIMD => "ONE48")                 – Selección de SIMD, "ONE48", "TWO24", "FOUR12"
port map (
ACOUT => ACOUT,                      -- Puerto de salida en cascada A de 30-bit
BCOUT => BCOUT,                      -- Puerto de salida en cascada B de 18-bit
CARRYCASCOUT => CARRYCASCOUT,       -- Acarreo de salida en cascada de 1-bit
CARRYOUT => CARRYOUT,                -- Acarreo de salida de 4-bit
MULTSIGNOUT => MULTSIGNOUT,          -- Salida en cascada del signo del multiplicador de 1-bit
OVERFLOW => OVERFLOW,                -- Salida del overflow en suma/acumulación de 1-bit
P => P,                              -- Salida de 48-bit
PATTERNBDETECT => PATTERNBDETECT, --  Activación de la salida de 1-bit
                                     -- de detección del complemento del patrón
PATTERNDETECT => PATTERNDETECT,     --  Activación de la salida de 1-bit de detección del patrón
PCOUT => PCOUT,                      -- Salida en cascada de 48-bit
UNDERFLOW => UNDERFLOW,              -- Activación de 1-bit de underflow en la salida en suma/acumulación
A => A,                              -- Entrada de datos A de 30-bit
ACIN => ACIN,                        -- Entrada de datos en cascada A de 30-bit
ALUMODE => ALUMODE,                  -- Entrada de control de la ALU de 4-bit
B => B,                              --  Entrada de datos B de 18-bit
BCIN => BCIN,                        --  Entrada de datos en cascada B de18-bit
C => C,                              --  Entrada de datos C de 48-bit
CARRYCASCIN => CARRYCASCIN,          -- Entrada de acarreo en cascada de 1-bit
```

```
CARRYIN => CARRYIN,                     -- Entrada de señal de acarreo de 1-bit
CARRYINSEL => CARRYINSEL,               -- Entrada de selección de acarreo de 3-bit
CEA1 => CEA1,                   -- Entrada de activación de 1-bit del reloj para la 1ª fase de registros de A
CEA2 => CEA2,                   -- Entrada de activación de 1-bit del reloj para la 2ª fase de registros de A
CEALUMODE => CEALUMODE, -- Entrada de activación de 1-bit del reloj del registro de ALUMODE
CEB1 => CEB1,                   -- Entrada de activación de1-bit del reloj para la 1ª fase de registros de B
CEB2 => CEB2,                   -- Entrada de activación de 1-bit del reloj para la 2ª fase de registros de B
CEC => CEC,                     -- Entrada de activación de 1-bit del reloj para el registro de C
CECARRYIN => CECARRYIN,     -- Entrada de activación de 1-bit del reloj para el registro de CARRYIN
CECTRL => CECTRL,               -- Entrada de activación de 1-bit del reloj para los registros de  OPMODE y acarreo
CEM => CEM,                     -- Entrada de activación de 1-bit del reloj para los registros del multiplicador
CEMULTCARRYIN => CEMULTCARRYIN,    -- Entrada de activación de 1-bit del reloj para múltiples registros de
                                   -- acarreo entrante
CEP => CEP,                             -- Entrada de activación de 1-bit del reloj para el registro de P
CLK => CLK,                             -- Entrada de reloj
MULTSIGNIN => MULTSIGNIN,               -- Entrada de 1-bit de signo de la multiplicación
OPMODE => OPMODE,                       -- Entrada de 7-bit de modo de operación
PCIN => PCIN,                           -- Entrada en cascada P de 48-bit
RSTA => RSTA,                           -- Entrada de 1-bit de reset para los registros del pipeline de A
RSTALLCARRYIN => RSTALLCARRYIN, -- Entrada de 1-bit de reset para los registros del pipeline de acarreo
RSTALUMODE => RSTALUMODE,       -- Entrada de 1-bit de reset para los registros del pipeline de ALUMODE
RSTB => RSTB,                           -- Entrada de 1-bit de reset para los registros del pipeline de B
RSTC => RSTC,                           -- Entrada de 1-bit de reset para los registros del pipeline de C
RSTCTRL => RSTCTRL,                     -- Entrada de 1-bit de reset para los registros del pipeline de OPMODE
RSTM => RSTM,                           -- Entrada de 1-bit de reset para los registros del multiplicador
RSTP => RSTP                            -- Entrada de 1-bit de reset para los registros del pipeline de P
);

-- Fin de la instanciación de DSP48E_inst
```

Metodología empleada

Uno de los apartados más importantes cuando se trabaja en grupo, es fijar un modelo o metodología de trabajo con vistas a los objetivos finales del trabajo. Inicialmente fijamos con el tutor establecer reuniones de carácter periódico cada quince días y establecimos seguir un modelo de espiral, es decir, ceñirnos a desarrollar en módulos pequeños y mediante iteraciones obtener los objetivos finales fijados, para cada iteraciones establecemos los siguientes procesos:



1.*Análisis de requerimientos*: Mediante reuniones con el tutor evaluamos la situación actual y del proyecto y fijamos los detalles de la nueva funcionalidad deseada de nuestra aplicación de cara a la nueva iteración

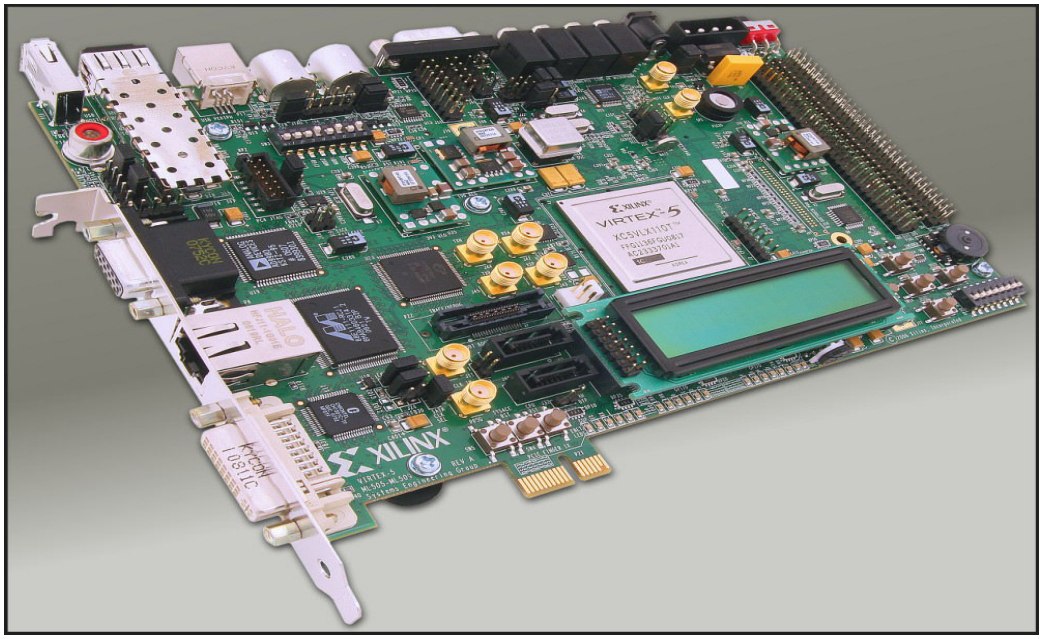
2.*Planificación y análisis del riesgo*: En función de las especificaciones técnicas evaluamos las diferentes formas de llegar a los objetivos fijados y en el caso de que no sean viables establecemos alternativas.

3.*Ingenieria*: Desarrollo de prototipos, generamos nueva funcionalidad y test que nos permitan verificar que lo realizado, sigue cumpliéndose para las nuevas iteraciones.

Capítulo 4.
Entorno de trabajo y descripción de la herramienta

Hardware:

A nivel hardware hemos trabajado con sobre una FPGA XC5VLX110T de la familia Virtex 5, la cual está empotrada en una placa de prototipado XUPV505-LX110T



Placa XUPV505-LX110T

La placa XUPV505-LX110T dispone de los siguientes componentes empotrados:

- Memoria de 64 MB hasta 2GB de DDR2 RAM
- Puerto Ethernet 10/100
- Códecs de audio y vídeo
- Puerto PS/2
- Switches
- Leds
- Puerto USB, PS/2, XSGA de video, SATA, entre otros.

FPGA XC5VLX110T

Como indicamos previamente las FPGAs están compuestas por elementos programables (CLBs, IOBs), matriz de interconexionado, gestores de reloj (CMTs) y bloques de memoria (BRAM).

La FPGA XC5VLX110T posee una matriz de 160 x 54 de CLBs proporcionando un total de 17280 Slice y ya que estos pueden ser utilizados como memoria RAM otorgándole 1,09MB entre todos los CLBs, 64 módulos de DSP48E, para controlar la señal de reloj distribuye 6 CMTs compuestos cada uno por 2 DCMs y 1 PLL y con respecto a la memoria la FPGA cuenta con un total de 5328KB de memoria RAM distribuida en 148 módulos de 36KB.

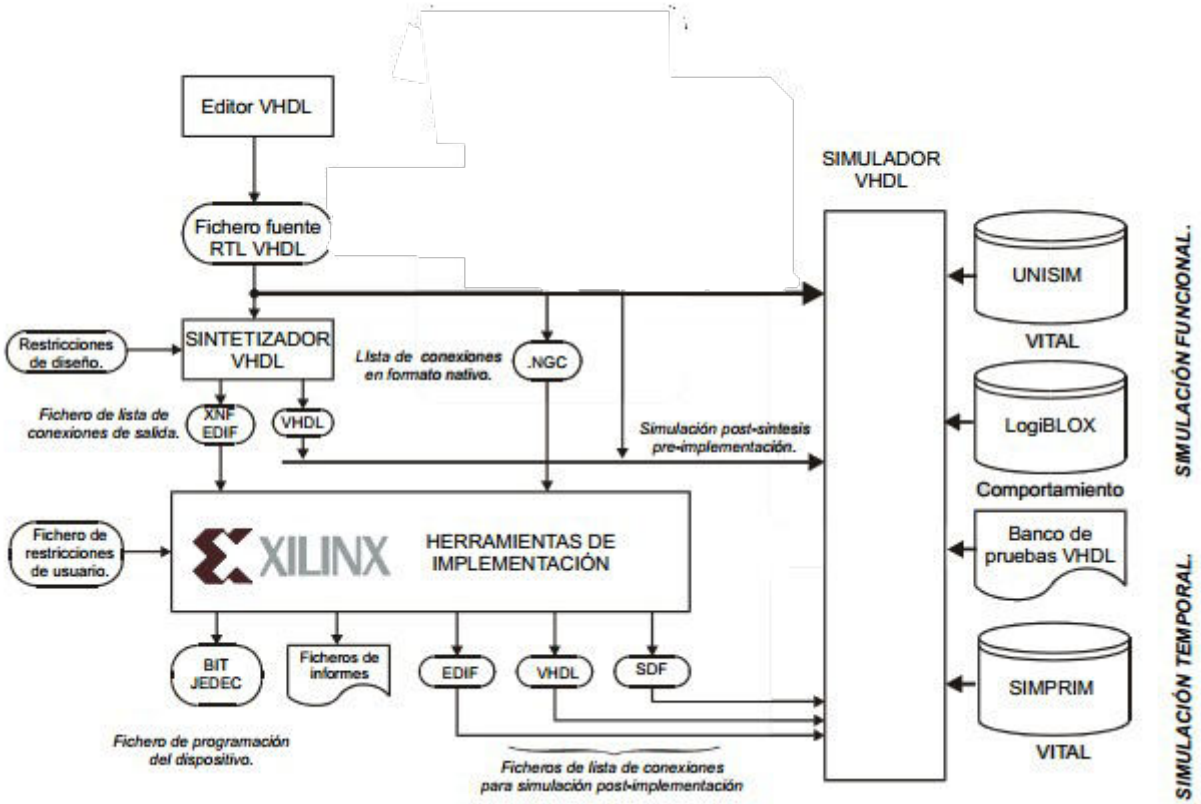
Encontrará más información de la arquitectura interna de la FPGA XC5VLX110T en el anexo1.

Entorno Software

A la hora de programar la FPGA hemos utilizado Xilinx ISE para desarrollar el código VHDL y para desarrollar la aplicación en java hemos empleado NetBeans. A continuación realizaremos una explicación de ambos programas.

Xilinx ISE (Integrated Software Environment) versión 12.1

Xilinx ISE proporciona las herramientas para poder diseñar, simular e implementar mediante Circuitos Programables como FPGA's para resolver y optimizar diferentes tareas. Permitiendo un mejor aprovechamiento y optimización de costes con una mayor productividad del diseño.



En nuestro caso el flujo de trabajo con Xilinx ISE, responde a las siguientes fases:

- *Fase de diseño, restricciones y síntesis*

  A través del editor utilizamos un lenguaje de descripción de hardware en nuestro caso VHDL, para generar diversos ficheros fuentes.

  Para poder relacionarse con el exterior a través de los pines del encapsulado del dispositivo configurable. Estos pines están, a su vez, unidos a una serie de elementos de la placa de desarrollo (puertos de entrada/salida,). Es necesario, por tanto, establecer las asignaciones de entradas y salidas a los pines del dispositivo físico concreto. Esto forma parte de las restricciones de usuario que además incluyen las limitaciones temporales. Para realizar las asignaciones de pines del encapsulado generamos un fichero de extensión .ucf.

  Finalmente hemos de realizar la traducción del lenguaje descripción a hardware, mediante la síntesis con la herramienta XST: Xilinx Synthesis Technology. Es el sintetizador nativo de Xilinx y se materializa en el ejecutable xst.exe. Genera (entre otros) un fichero de extensión .ngc.

- *Fase de implementación*

  La fase de implementación consiste en la creación del fichero de configuración que, una vez cargado en el dispositivo programable, le hará trabajar conforme a las especificaciones establecidas. Consta de dos partes: la creación de una base de datos o netlist que recopile tanto el circuito diseñado como sus restricciones (fichero .ngd) y la creación del fichero de programación propiamente dicho (.jed).

- *Simulación con lSim*

  Es una forma para verificar el comportamiento del hardware. En un proceso interactivo, el usuario especifica qué entradas y salidas mediante señales denominadas estímulos del modelo y así realizar una simulación del sistema.

- *Carga del programa de configuración*

  La última fase en el flujo de diseño consiste en generar un archivo de bits (biststream .bit) y configurar nuestro dispositivo mediante la herramienta IMPACT, y lo transferimos a través JTAG de la FPGA.

NetBeans

Como entorno de trabajo en Java nos hemos decantado por utilizar Netbeans ya que nos permite generar interfaces gráficos con gran simplicidad.

Antes de introducirnos de lleno en la herramienta especificaremos el brevemente el patrón utilizado en la generación del software. Al tratase de un proyecto que estaba en constante crecimiento al generar el primer interfaz gráfico, se nos presento la problemática de sí dicho diseño seria el definitivo.

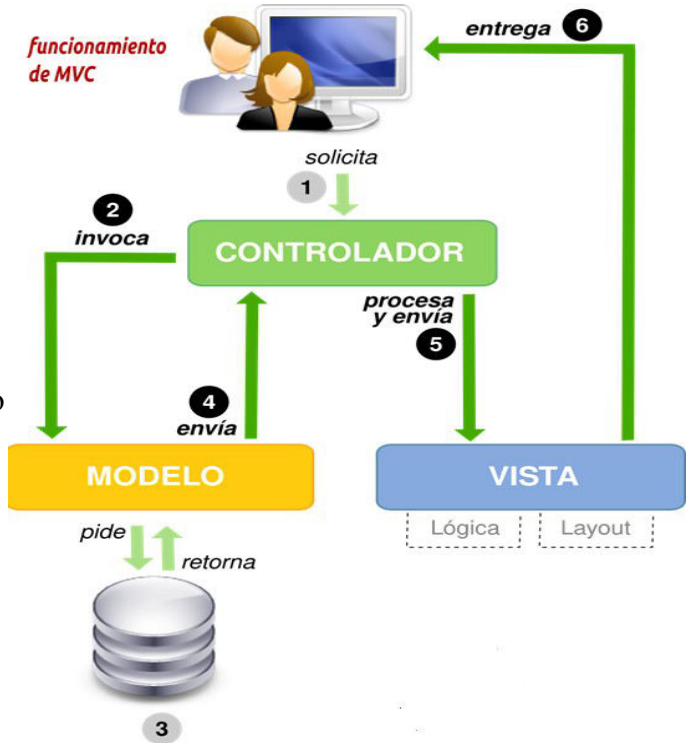
Al añadir nueva funcionalidad y la parte visual requería de nuevos componentes lo cual suponía un esfuerzo añadido, para ello, decidimos realizar una arquitectura software en la que separamos los datos y la lógica de negocia de la parte visual fundamentándonos en el modelo vista controlador (MVC).

Las componentes de MVC se pueden definir de la siguiente manera:

- Modelo: encapsula la funcionalidad de la aplicación y es independiente de la vista y del controlador
- Controlador: Responde a los eventos solicitados por el usuario e invoca dicha peticiones al modelo. También se encarga de comunicarse los resultados a la vista.
- La Vista: Presenta el 'modelo' y es la forma de interactuar con el usuario.

La aplicación sigue la siguiente estructura:

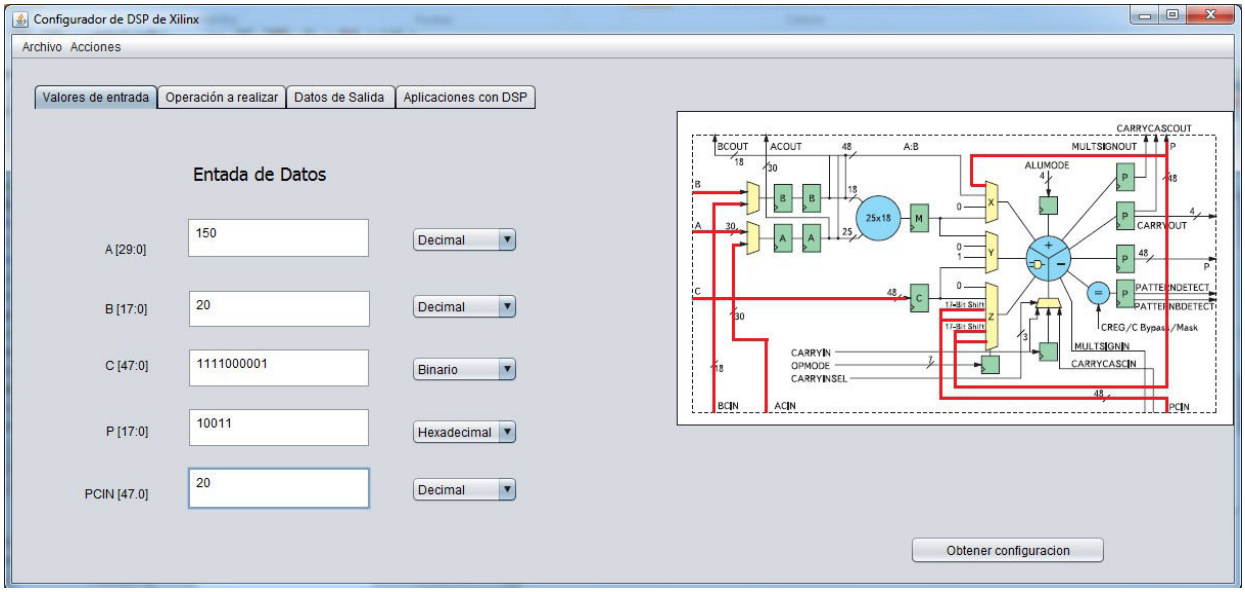
1. El usuario realiza una petición, por ejemplo desea obtener los parámetros de configuración para una determinada operación
2. El controlador captura el evento (Pulsar el botón de obtener configuración)
3. Hace la llamada al modelo/modelos, que determinara si el formato de los datos es valido y retornará esta información al controlador de los datos de salida
4. El controlador recibe la información y la envía a la vista
5. La vista, los entregará al usuario de forma legible.



De esta manera, podemos encargarnos de forma independiente de la lógica de negocio de aplicación software y de la vista de la aplicación y en el caso de querer actualizar la vista, el modelo ni el controlador se ven afectados

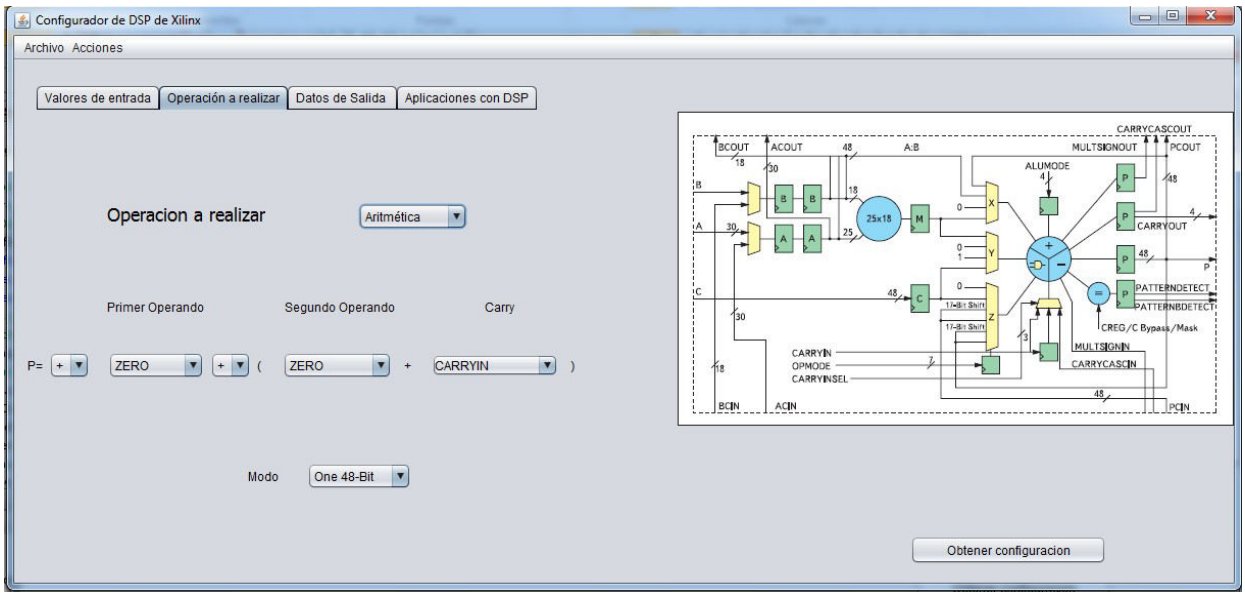
Una vez presentado la metodología de trabajo e indicado el modo de proceder para generar la aplicación, nos disponemos a presentar la aplicación. Como indicamos en el inicio de la memoria el primer objetivo es proporcionar una herramienta que nos permita configurar un DPS, en nuestro caso el DSP48E que equipan la familia de FPGA Virtex5, tras estudiar detenidamente la documentación y con el objetivo de facilitar al usuario la labor de configuración, la realizaremos en varios pasos que hemos diferenciado: la entrada de datos al DSP, la operaciones a realizar y las señales de salida.

- *La entrada de datos al DSP*



El usuario establece los valores de la entradas del DSP, eligiendo entre los valores estén en binario, decimal o hexadecimal. La arquitectura del DPS nos determina el tamaño de las entradas, por lo tanto hemos de controlar mediante sistema de mensajes que los datos introducido son validos de acuerdo al formato.

- *Operaciones a realizar*



El DSP se encarga de realizar generalmente operaciones basadas en la aritmética (suma, resta, multiplicación, desplazamiento) y operaciones de lógica (AND, OR, NOT, etc). Para poder simplificar al usuario la gestión de DSP , inicialmente pedimos al usuario que especifique el tipo de operación que desea realizar:

- Aritmética
  El DSP48E permite realizar la combinación de las siguientes operaciones aritméticas.

Adder/Sub Out = (Z ± (X + Y + CIN)) ó (-Z + (X + Y + CIN) –1)

Correspondiéndose el valor de Z con la salida del multiplexor Z y así respectivamente con los valores de X e Y y CIN con la entrada CIN.
Para poder facilitar todas las combinaciones posibles al usuario sin tener que conocer la arquitectura interna, es decir, la interconexión de los diversos multiplexores. Hemos establecido que el usuario establezca la operación que desea realizar siguiendo este formato:

P= ± Primer Operando ± (Primer Operando + Carry)

De esta manera el Primer operando podrá adquirir los siguientes valores correspondientes a las entradas del DSP:

- ZERO
- PCIN
- P
- C
- Shirft(PCIN)
- Shirft(P)

En el caso del segundo operando, hay que tener en cuanta las restricciones, es decir, que no todos los valores del primer operando se pueden combinar con el segundo:

- ZERO
- -1
- C
- A x B
- P
- P - 1
- P + C
- A : B
- A : B - 1
- A : B + C

Respecto a los valores de CARRY, posee restricciones en función de los anteriores valores, correspondiendo con los valores de la tabla de la pagina (por determinar)

- Lógica
  En el caso de las operaciones lógicas, lo único que tenemos que establecer es la salida del multiplexor X, la operación y el valor del multiplexor Z.
  Las entradas del multiplexor X son las siguientes:

  - ZERO
  - P
  - A : B

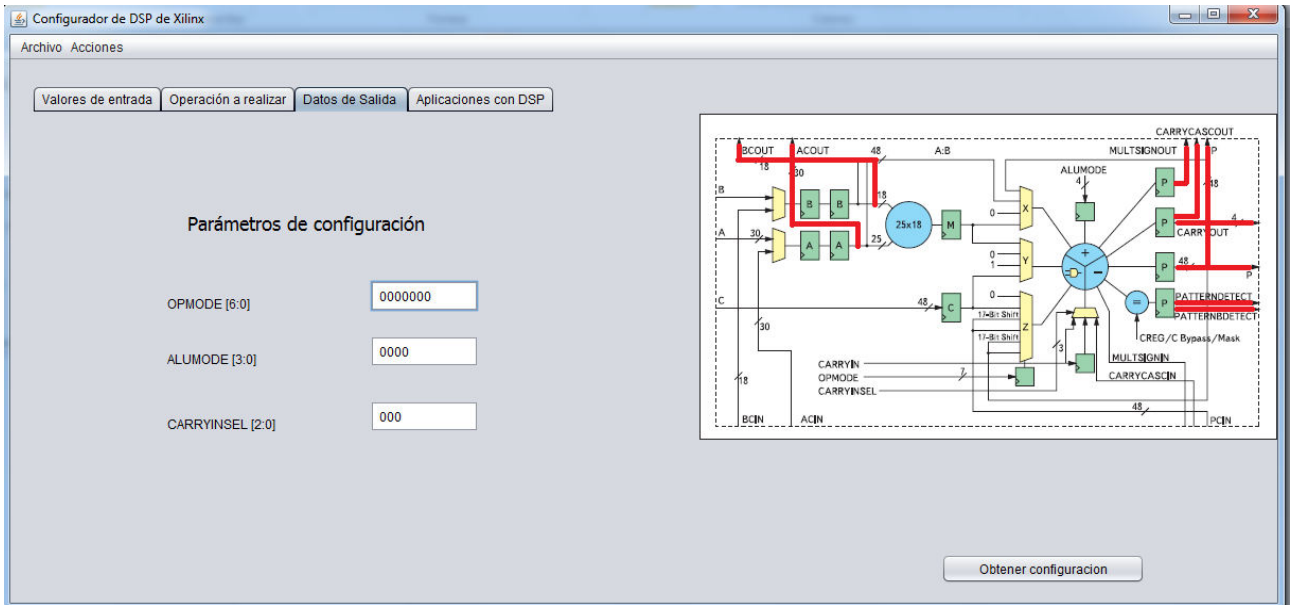
Para el multiplexor Z:

- ZERO
- C
- P
- PCIN
- Shirtf(P)
- Shirtf(PCIN)

Y las operaciones lógicas definidas en función de los multiplexores:

- X XOR Z
- NOT (X XOR Z)
- X OR Z
- (NOT X) OR Z
- X OR (NOT Z)
- NOT (X OR Z)
- X AND Z
- (NOT X) AND Z
- X AND (NOT Z)
- NOT(X AND Z)

- *Señales de salida*



Según el tipo de operación que queremos realizar obtenemos los parámetros par OPMODE, ALUMODE y CARRYINSEL.

De esta manera podríamos modificar de forma rápida una instancia en el caso de que poseamos una instancia del DSP48E. En el caso, de que nos dispongamos de una instancia de este componente la aplicación nos generara una en formato vhdl. Así podremos importarla aun proyecto de Xilinx ISE.

Como se indico en la descripción del módulo DSP, para poder ahorrar energía dentro del componente se puede deshabilitar el multiplicador. Con la finalidad de optimizar el funcionamiento de este, en el caso de que no necesitamos recurrir al multiplicador debemos modificar el parámetro MREG => 0, y USE_MULT => "NONE", en caso contrario, MREG => 1, y USE_MULT => "MULT_S".

A la hora de generar las instancias de DSP pudimos observar que los parámetros anteriores son valores GENERIC MAP y no podemos establecer como variables, lo cual supone un problema. Para resolverlo, generamos dos instancia diferentes del componente, una de ellas mantiene deshabilitado el multiplicador de 25x18 bits y el registro M la cual hemos denominado MyDSP48E y otra que lo permanece activo que hemos llamado MyDSPMult.

Técnicas de control:

Como indicamos la metodología se basa en espiral y en cada iteración establecemos una fases de control, para ello hemos generado dos formas de verificar que la nueva funcionalidad que añadimos con cada iteración no afecte sobre el trabajo anterior. La primera forma de control recae sobre la aplicación java para ello hemos recurrido a la librería JUnit (versión 4,10) y en Xilinx ISE mediante simulaciones.

JUnit se trata de un Framework Open Source para la automatización en nuestro caso de pruebas unitarias. Esto nos permite aislar partes del código y comprobar el correcto funcionamiento del código por separado.
En nuestra aplicación hemos generado dos subsistemas de pruebas, uno para las operaciones aritméticas compuesto por 84 test y otro para las operaciones lógicas formado por 120 test.

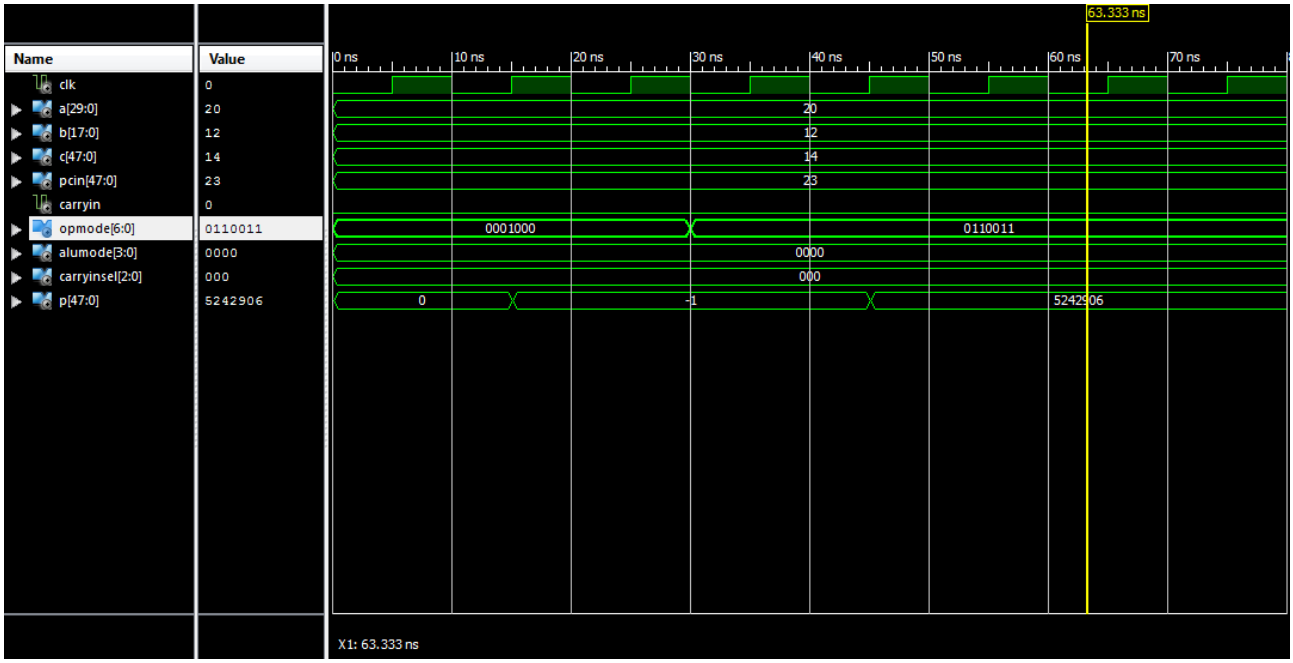
```
40      public void testXOR1()
41      {
42          //según labview
43          /*
44          OPMODE => "0000000",
45          ALUMODE => "0100",
46          CARRYINSEL => "000",
47          */
48          expresion= "ZERO XOR ZERO";
49          resultado="000 00 00 0100 000"; //OPMODE ALUMODE CARRYINSEL
50          System.out.println(expresion);
51          String operandoX="ZERO";
52          String operandoZ="ZERO";
53          String operacion="x XOR z";
54          String carryInSel="";
55          prueba.tipoOperacion("Logica");
56          myOPMODE=prueba.comprobarOPEMODE(operandoX, operandoZ,operacion);
57          myALUMODE=prueba.comprobarALUMODE();
58          myCarryInSel=prueba.comprobarCarryInSel(carryInSel);
59          System.out.println("OPMODE: " +myOPMODE+", ALUMODE: " +myALUMODE+", CARRYINSEL:"+ myCarryInSel);
60          assertEquals(resultado, myOPMODE+" "+myALUMODE+" "+myCarryInSel);
61      }
62
```

El funcionamiento de cada prueba unitaria es muy sencillo, se establece una serie de valores de entrada, en este caso concreto, comprobamos la operación ZERO XOR ZERO, determinamos que la entrada del multiplexor X y Z es XERO y la operación lógica es XOR, tras ejecutar la aplicación comprobamos que los parámetros de OPMODE, ALUMODE y CARRYINSEL son los esperados.

Y como modo de control para la aplicación Xilinx ISE, recurrimos a los testbench para comprobar el funcionamiento del circuito, mediante la herramienta ISIM Simulator.
En cuanto al hardware, una vez sintetizado el diseño es necesario verificar si cumple con las especificaciones del problema. Para ello realizaremos un banco de prueba (testbench) que prueba lo más exhaustivamente posible el diseño.
Para ello proporcionamos valores a las entradas de la instancia, lo cual determinamos estímulos y comprobamos que los valores de salida son los deseados. De esta manera podemos observar el valor interno de las señales sin tener que volcar los diseños sobre la FPGA.
Por ejemplo, para la instancia de DSP hemos proporcionado las siguientes valores para las entradas:

A <= transport "00000000000000000000000010100"; -- A=20
B <= transport "0000000000001100"; --B=12
C <= transport "00000000000000000000000000000000000000000001110"; -- C=14
PCIN <= transport "00000000000000000000000000000000000000010111"; -- PCIN=23
OPMODE <= transport "0001000";
ALUMODE <= transport "0000";
CARRYINSEL<= transport "000";
CARRYIN<= transport '0';

Dichos valores de configuración corresponde con la operación C+(A : B+ CARRYIN) y tras un ciclo de reloj obtenemos el valor en P



Aplicaciones utilizando un DSP48A

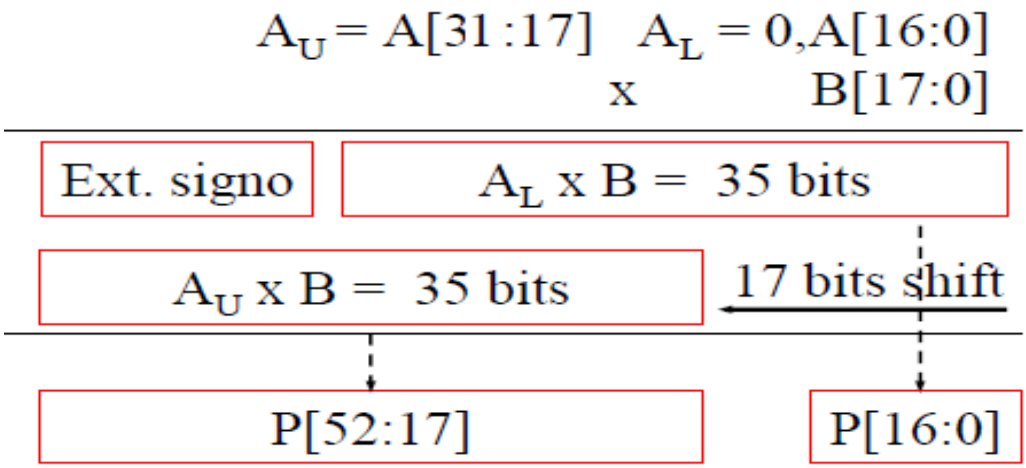
Tras obtener una aplicación que nos permite obtener una instancia del DSP, nos interesa desarrollarlo para obtener otras instancias que utilicen varios DSP. Para ello hemos pensado en la implementación de las siguientes aplicaciones:

- Operaciones lógicas de 16 bits
- Multiplicador 16 x 16 bits
- Multiplicador 32 x 16 bits
- Multiplicador 32 x 32 bits
- Multiplicador de números complejos
- Filtro FIR

Multiplicador 32 x 16 bits

Mediante la utilización de múltiples instancias del componente, podemos implementar un multiplicador para números con signo formados por muchos bits. Recurriendo al sistema multi-granular, podemos implementar complejos multiplicados mediante pequeños bloques multiplicadores empotrados. La compleja operación se dividirá en sucesivas operaciones más simples y mediante un correcto interconexionado de los DSP.

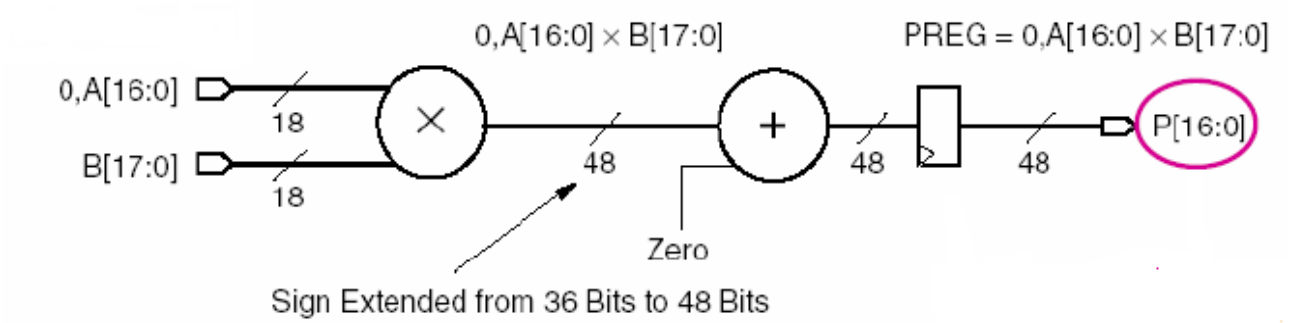
Para ello establecemos el siguiente algoritmo.



Como se aprecia en la imagen, necesitamos realizar dos multiplicaciones, para ello necesitamos recurrir al módulo de 25x18 bits integrado en el DSP y por lo tanto, necesitamos dos DSP y adaptar los operandos a las entrada A de 30 bits y la entrada B de 18 bits.

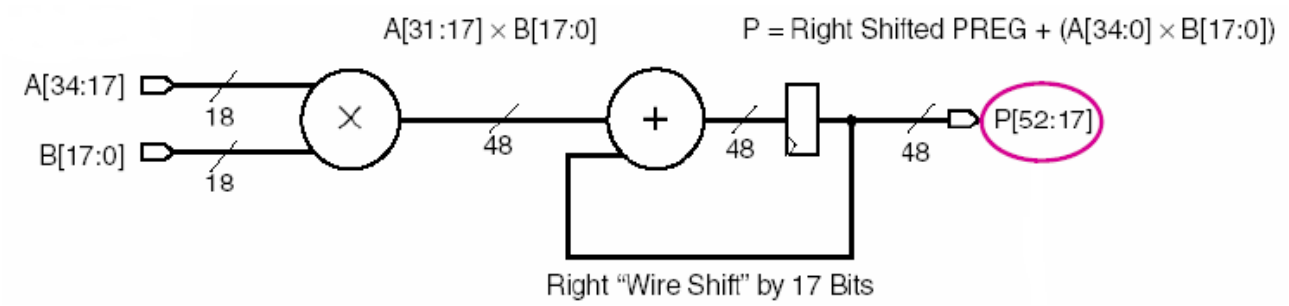
Configuración del primer DSP:

El primer DSP tiene que realizar la siguiente acción.



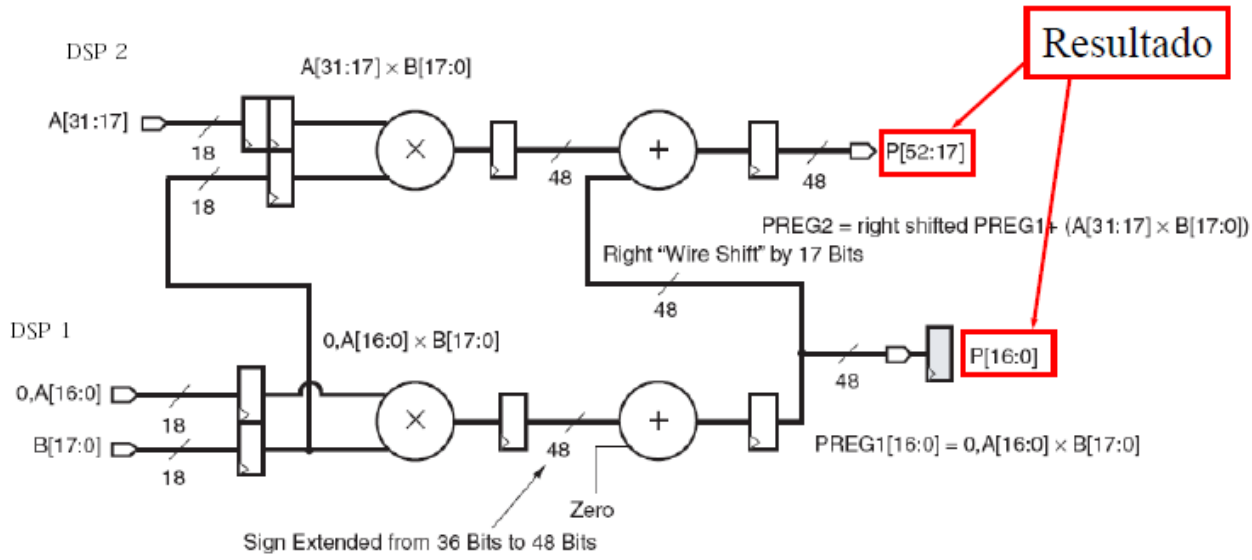
Utilizaremos las entradas A y B del DSP, para la entrada A formada por 13 ceros concatenado con los 17 bits menos significativos del operando A y en el caso de la entrada B, extenderemos en signo el operando B hasta que forme una palabra de 18 bits. Y establecemos que el valor de OPEMODE a “0000101” , ALUMODE a “0000”, CARRYINSEL “000” y CARRYIN '0'. Mediante esta configuración el DSP realiza la siguiente operación "(AxB + CIN)".

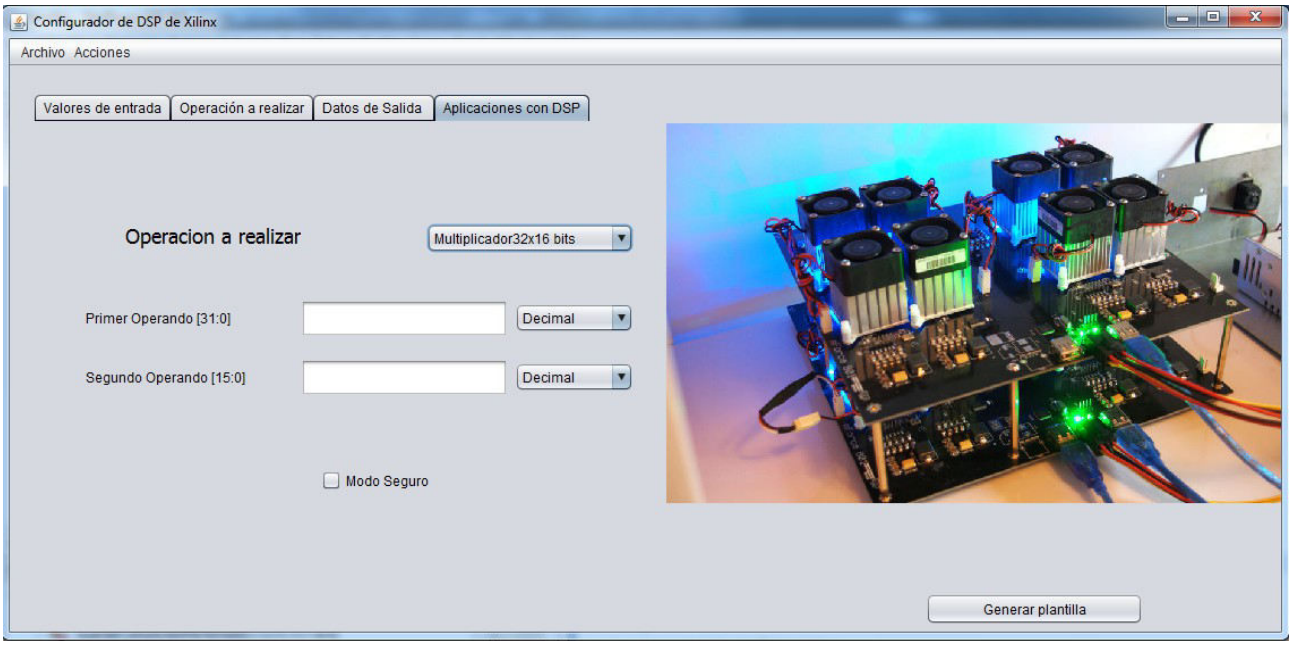
En el caso del segundo DSP necesitamos utilizar el valor del primer DSP, por lo tanto, para completar la acción necesitamos disponer de dos ciclos de reloj.



Al igual que en caso anterior, tendremos que transformar las entrada de datos del DSP, ya que utilizaremos las mismas entradas. En el caso de B realizaremos lo mismo que en el anterior DSP y para A cogeremos los 14 bits más significativos del operando A y los extendemos en signo hasta obtener una palabra de 30 bits.
Una vez obtenido los 48 bits de la multiplicación desplazamos el resultado 17 bits a la izquierda y lo sumamos con el resultado parcial del anterior DSP. Para ello tenemos codificar las señales OPEMODE a "1010101" , ALUMODE a "0000", CARRYINSEL "000", CARRYIN '0' y PCIN con el valor de P del anterior DSP, dando como resultado Shift(PCIN) + (AxB + CIN)

Quedando los dos DSP interconexionados de la siguiente manera:





A nivel de la aplicación java, al tratarse de una multiplicación solo hemos de indicar los valores de los diversos operandos. Para facilitar la labor al usuario las entradas de datos se puede realizar en base binaria, decimal o hexadecimal. Como resultado obtendremos esta plantilla, es decir, un fichero vhdl que podremos incorporar a nuestro proyecto en VHDL.

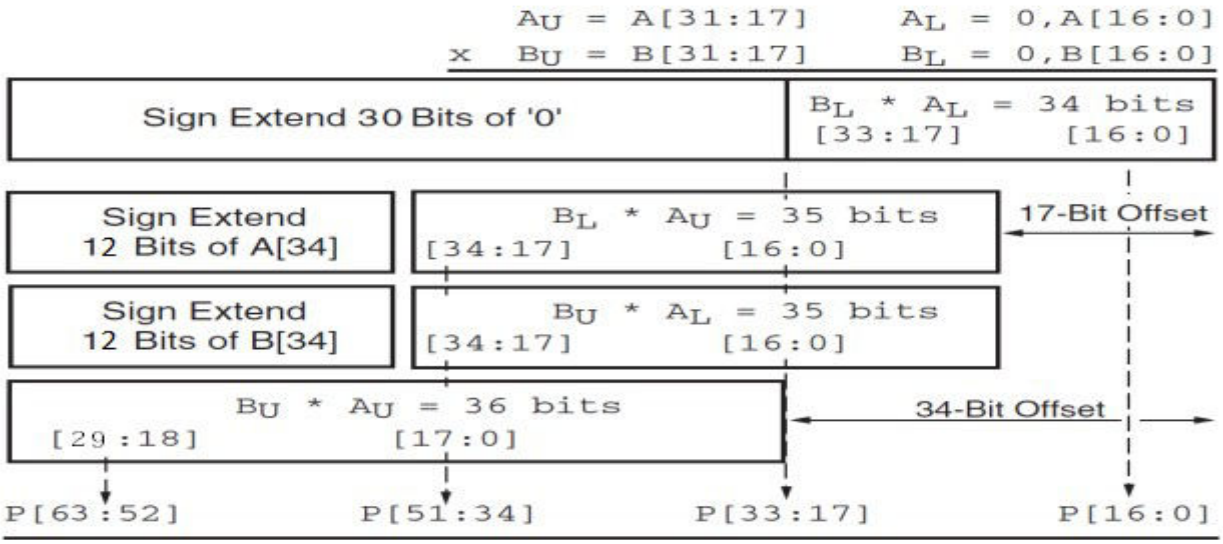
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Mult32x16 is
	PORT (
			CLK :			in  STD_LOGIC;
			operandoA:		in  STD_LOGIC_VECTOR (31 DOWNTO 0);
			operandoB:		in  STD_LOGIC_VECTOR (15 DOWNTO 0);
			Resultado:		out STD_LOGIC_VECTOR (50 DOWNTO 0)
		);
end Mult32x16;

architecture Behavioral of Mult32x16 is
	component SuperiorMult is
		PORT (
				CLK :			in  STD_LOGIC;
				A :			in  STD_LOGIC_VECTOR (29 DOWNTO 0);
				B :			in STD_LOGIC_VECTOR (17 DOWNTO 0);
				C :			in STD_LOGIC_VECTOR (47 DOWNTO 0);
				OPMODE :		in STD_LOGIC_VECTOR (6 DOWNTO 0);
				ALUMODE :		in STD_LOGIC_VECTOR (3 DOWNTO 0);
				CARRYINSEL :	in STD_LOGIC_VECTOR (2 DOWNTO 0);
				CARRYIN :		in STD_LOGIC;
				PCIN :		in STD_LOGIC_VECTOR (47 DOWNTO 0);
				P :			out  STD_LOGIC_VECTOR (47 DOWNTO 0);
				PCOUT :		out STD_LOGIC_VECTOR (47 DOWNTO 0));
	end component;

	SIGNAL Zero: STD_LOGIC_VECTOR(12 DOWNTO 0) := (OTHERS=>'0');
	SIGNAL A1: STD_LOGIC_VECTOR(29 DOWNTO 0);
	SIGNAL A2: STD_LOGIC_VECTOR(29 DOWNTO 0);
	SIGNAL B: STD_LOGIC_VECTOR(17 DOWNTO 0);
	SIGNAL parcial1: STD_LOGIC_VECTOR(47 DOWNTO 0);
	SIGNAL parcial2: STD_LOGIC_VECTOR(47 DOWNTO 0);
```

```
begin
	transformarSeñales: process (operandoA,operandoB)

		begin
		B   <= operandoB(15)&operandoB(15)&operandoB(15 downto 0);
		A1 <= Zero & operandoA(16 downto 0);
		if(operandoA(31)='0') then
			A2 <= "000000000000000" & operandoA(31 downto 17);
		else
			A2 <= "111111111111111" & operandoA(31 downto 17);
		end if;
	end process;

	utt1: SuperiorMult PORT MAP (Clk => Clk, A => A1, B =>B, C =>(OTHERS=>'0'), OPMODE=> "0000101",
ALUMODE =>  "0000", CARRYINSEL => "000", CARRYIN => '0', PCIN => (others=>'0'), P => parcial1, PCOUT
=> OPEN);

	uut2: SuperiorMult PORT MAP (Clk => Clk, A => A2, B =>B, C =>(OTHERS=>'0'), OPMODE=>
"1010101", ALUMODE =>  "0000", CARRYINSEL => "000", CARRYIN => '0', PCIN =>parcial1 , P => parcial2,
PCOUT => OPEN);

	resultado <= parcial2(33 downto 0)&parcial1(16 downto 0);
end Behavioral;
```

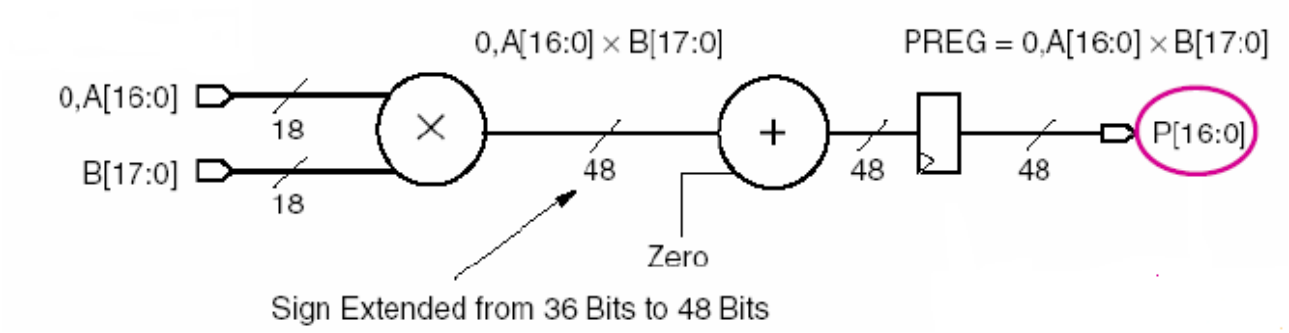
Como realizamos para los anteriores caso del DSP, hemos construido un banco de prueba en Xilinx ISE para ejecutarlo en ISIM Simulator.

Multiplicador 32 x 32 bits

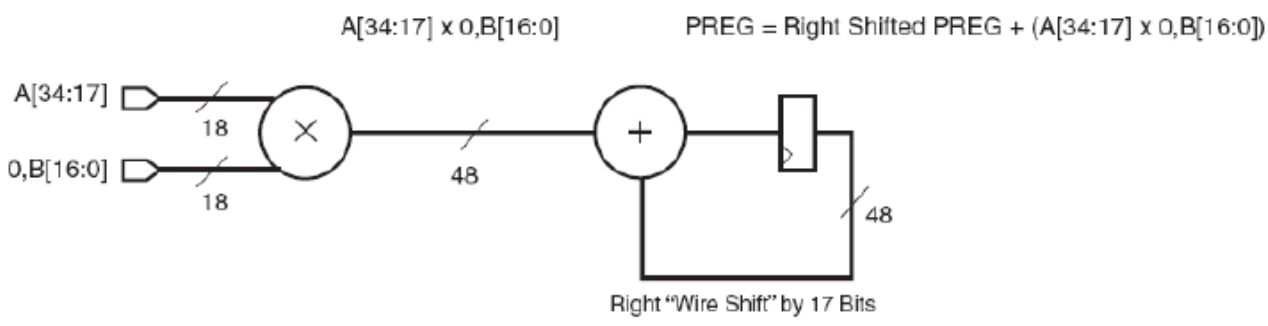
En el caso de la multiplicación para operandos de 32 bits el procedimiento a seguir es muy parecido al caso anterior como podemos observar.



Como podemos apreciar necesitamos utilizar cuatro DSP para poder realizar las diferentes multiplicaciones.
La configuración del primer DSP se realiza de la misma manera que en caso de la multiplicación 32x16 bits

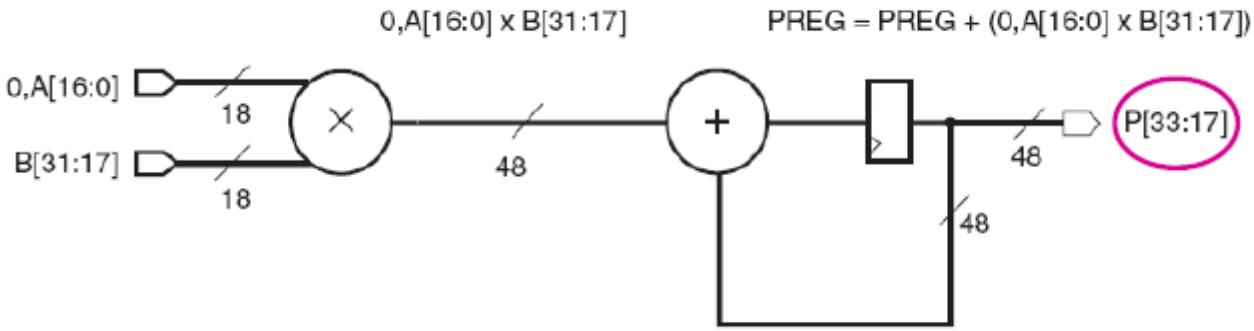


El segundo DSP se encarga de realizar la siguiente operación.

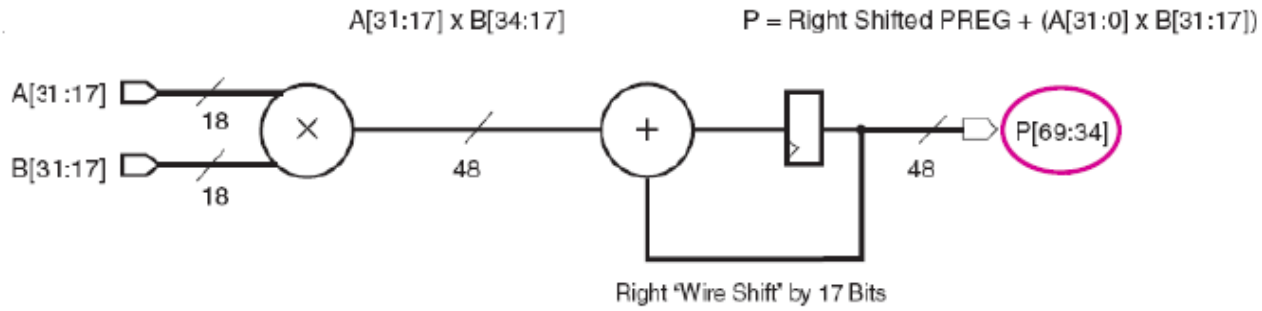


Como establecimos en el caso anterior, utilizaremos las entradas A y B del DSP, para ello tenemos que adaptar las entradas, como valor de B utilizaremos el mismo vector que hemos utilizado en DSP anterior y en el caso de valor de A cogeremos de los bist 31 al 17 y extenderemos con el bit 31 hasta formar una palabra de 30 bits. La pración que estamos buscando es Shift(PCIN) + (AxB + CIN), para ello daremos el valor “1010101” para OPMODE, "0000” a ALUMODE, “000” a CARRYINSEL, '0' para CARRYIN y como PCIN el valor obtenido de la salida P del primer DSP.

Tercer DSP nos permite calcular una subsecuencia de bits de resultado final.

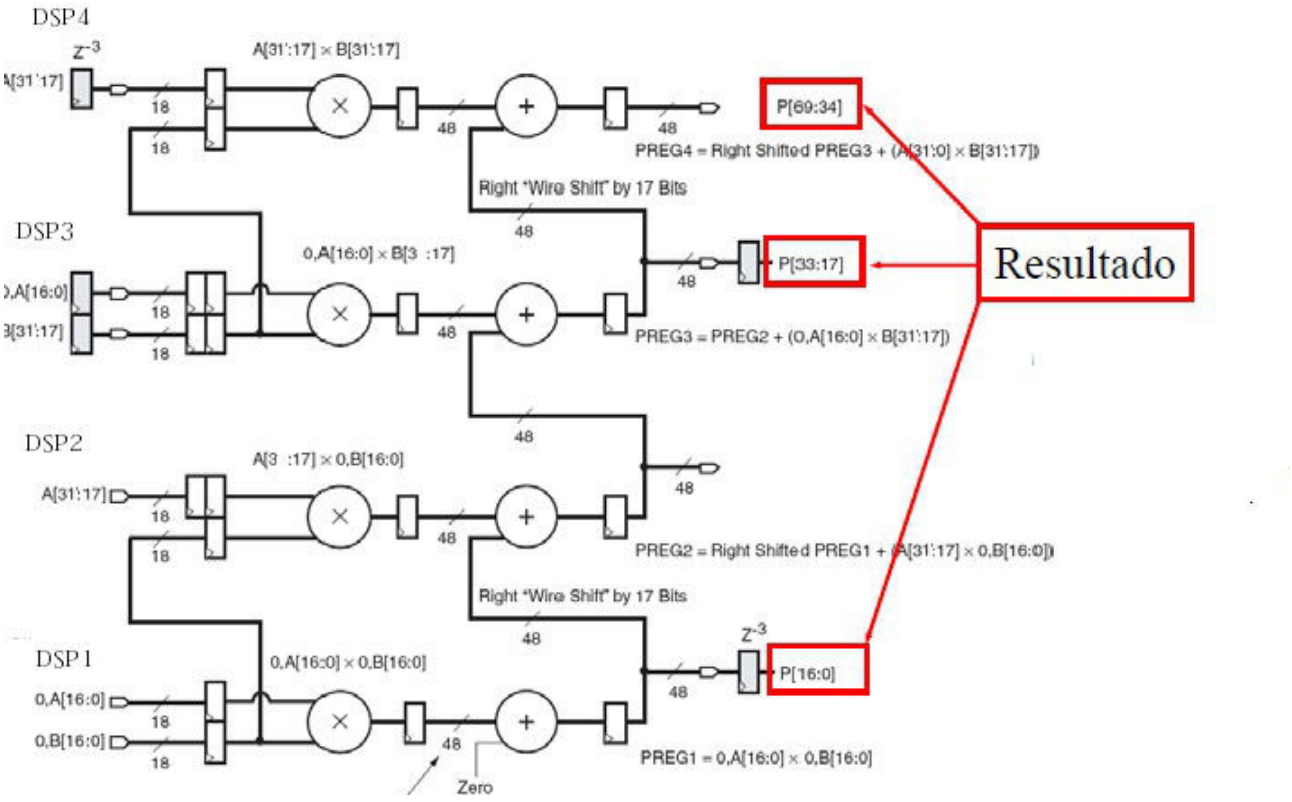


Como entradas al DSP establecemos que la entada A estará formada los 17 bits menos significativos del primer operando y los concatenaremos a la izquierda con ceros hasta formar una palabra de 30 bits y en el caso de la entrada B es 15 bits más significativos y extendemos el vector con tres bits más que corresponden con el bit más significativo del segundo operando. Y respecto a la configuración, buscamos que el DSP realice PCIN + (AxB + CIN), para ello damos como valor de OPMODE “0010101”, ALUMODE "0000", CARRYINSEL"000", CARRYIN '0' y como entrada de PCIN el resultado del segundo DSP.

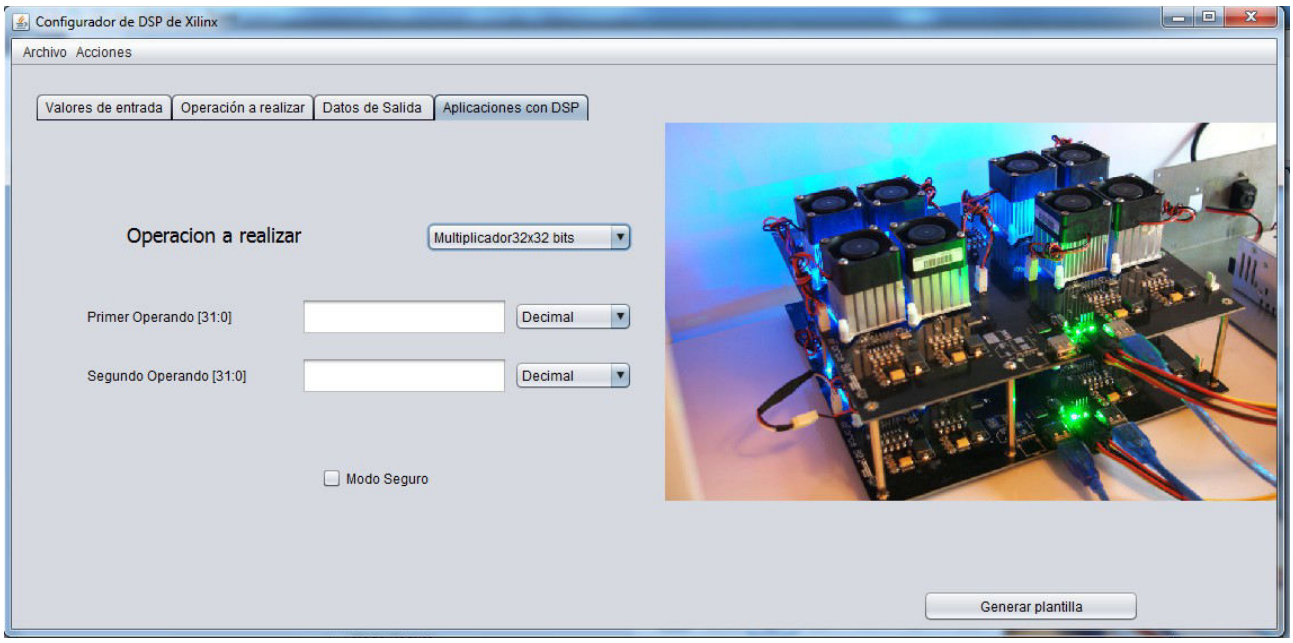


Y para finalizar el cuarto DSP que calcula los bits más significativo de la multiplicación. Para ello, establecemos que la entrada A son los 15 bits más significativos del vector del primer operando y lo extendemos en signo hasta formar una palabra de 30 bits y en el caso de B, realizamos la misma acción establecemos los 15 bits más significativos y completamos hasta formar una palabra de 18 bits extendiéndolo en signo. Con respecto a los valores de configuración establecemos: OPMODE "1010101", ALUMODE "0000", CARRYINSEL "000", CARRYIN '0' y PCIN recibe la entrada del tercer DSP. Dichos parámetros de configuración nos permite calcular Shift(PCIN) + (AxB + CIN)

Dando como resultado la siguiente esquema:



Respecto al entorno de la aplicación Java, no varia con respecto al anterior aplicación, salvo por el hecho de que el segundo operando consta de 32 bits.

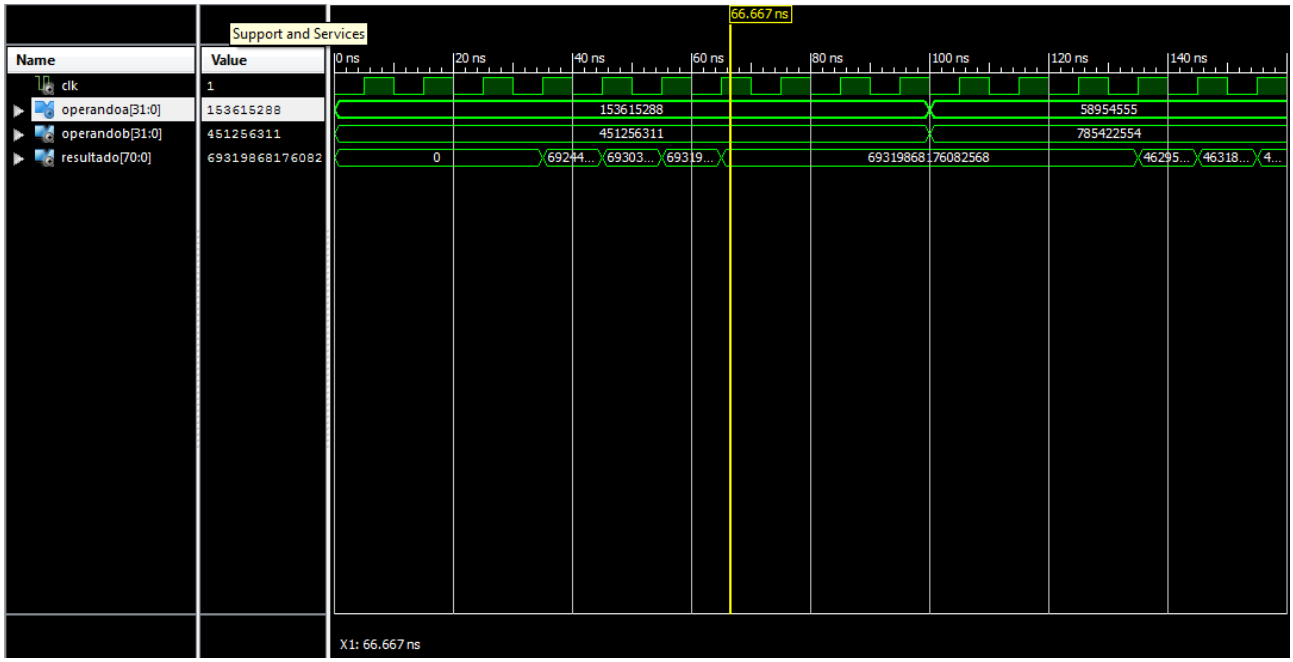


Como resultado obtenemos la siguiente fichero vhdl.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Mult32x32 is
        PORT (
                        CLK :           in  STD_LOGIC;
                        operandoA:      in  STD_LOGIC_VECTOR (31 DOWNTO 0);
                        operandoB:      in  STD_LOGIC_VECTOR (31 DOWNTO 0);
```

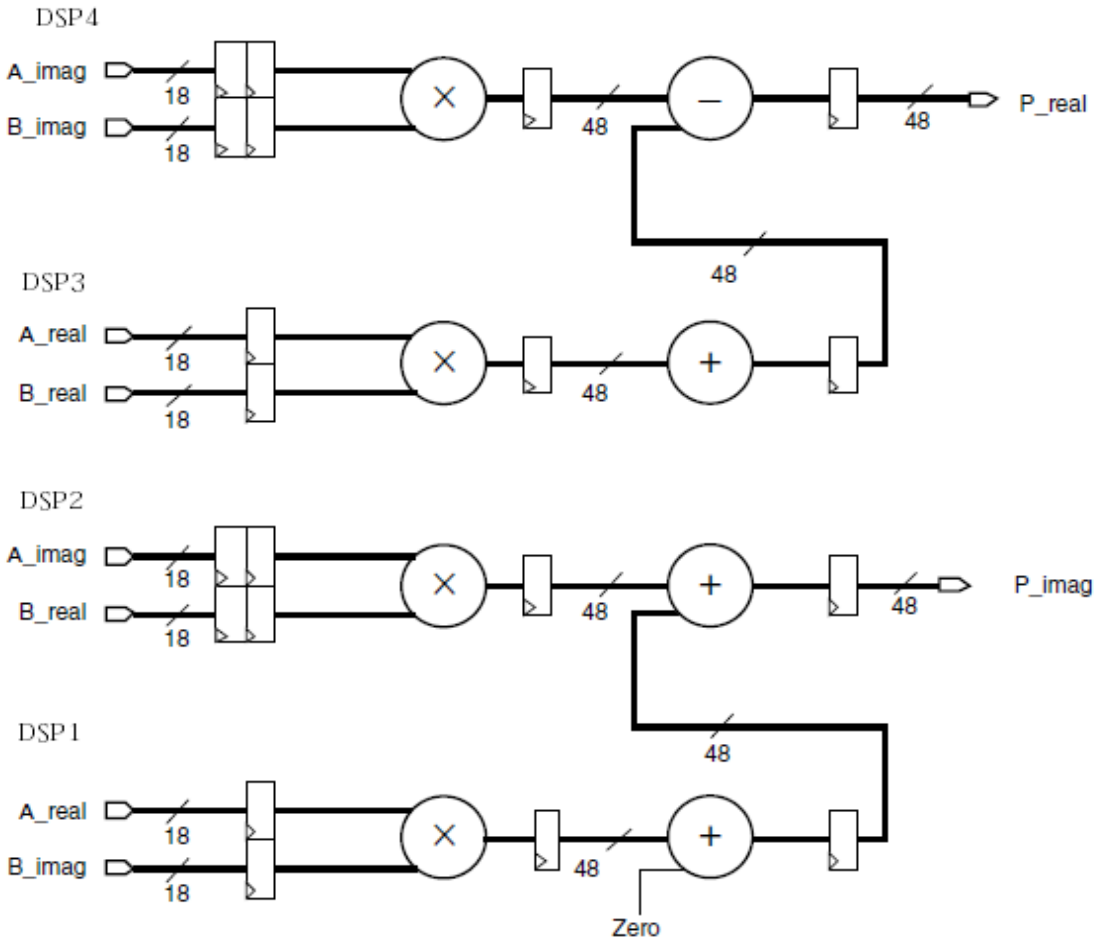
```
                        Resultado:          out STD_LOGIC_VECTOR (70 DOWNTO 0)
                    );
end Mult32x32;

architecture Behavioral of Mult32x32 is

component SuperiorMult is
        PORT (
                        CLK :               in  STD_LOGIC;
                        A :                 in  STD_LOGIC_VECTOR (29 DOWNTO 0);
                        B :                 in STD_LOGIC_VECTOR (17 DOWNTO 0);
                        C :                 in STD_LOGIC_VECTOR (47 DOWNTO 0);
                        OPMODE :            in STD_LOGIC_VECTOR (6 DOWNTO 0);
                        ALUMODE :           in STD_LOGIC_VECTOR (3 DOWNTO 0);
                        CARRYINSEL :  in STD_LOGIC_VECTOR (2 DOWNTO 0);
                        CARRYIN :           in STD_LOGIC;
                        PCIN :          in STD_LOGIC_VECTOR (47 DOWNTO 0);
                        P :                 out  STD_LOGIC_VECTOR (47 DOWNTO 0);
                        PCOUT :             out STD_LOGIC_VECTOR (47 DOWNTO 0));
end component;

SIGNAL Zero: STD_LOGIC_VECTOR(12 DOWNTO 0) := (OTHERS=>'0');
SIGNAL Au: STD_LOGIC_VECTOR(29 DOWNTO 0);
SIGNAL Al: STD_LOGIC_VECTOR(29 DOWNTO 0);
SIGNAL Bu: STD_LOGIC_VECTOR(17 DOWNTO 0);
SIGNAL Bl: STD_LOGIC_VECTOR(17 DOWNTO 0);
SIGNAL parcial1: STD_LOGIC_VECTOR(47 DOWNTO 0);
SIGNAL parcial2: STD_LOGIC_VECTOR(47 DOWNTO 0);
SIGNAL parcial3: STD_LOGIC_VECTOR(47 DOWNTO 0);
SIGNAL parcial4: STD_LOGIC_VECTOR(47 DOWNTO 0);

begin
        transformarSeñales: process (operandoA,operandoB)
                begin
                Bl<= operandoB(31)&operandoB(31)&operandoB(31)&operandoB(31 downto 17);
                Bl<= '0'&operandoB(16 downto 0);
                Al <= Zero & operandoA(16 downto 0);
                Bu <= operandoB(31)&operandoB(31)&operandoB(31)&operandoB(31 downto 17);
                if(operandoA(31)='0') then
                        Au <= "000000000000000" & operandoA(31 downto 17);
                else
                        Au <= "111111111111111" & operandoA(31 downto 17);
                end if;
        end process;

  Al_Bl: SuperiorMult PORT MAP (Clk => Clk, A => Al, B =>Bl, C =>(OTHERS=>'0'), OPMODE=> "0000101",
ALUMODE =>  "0000", CARRYINSEL => "000", CARRYIN => '0', PCIN => (others=>'0'), P => parcial1, PCOUT
=> OPEN);

Au_Bl: SuperiorMult PORT MAP (Clk => Clk, A => Au, B =>Bl, C =>(OTHERS=>'0'), OPMODE=> "1010101",
ALUMODE =>  "0000", CARRYINSEL => "000", CARRYIN => '0', PCIN =>parcial1 , P => parcial2, PCOUT =>
OPEN);

Al_Bu: SuperiorMult PORT MAP (Clk => Clk, A => Al, B =>Bu, C =>(OTHERS=>'0'), OPMODE=> "0010101",
ALUMODE =>  "0000", CARRYINSEL => "000", CARRYIN => '0', PCIN => parcial2, P => parcial3, PCOUT =>
OPEN);

Au_Bu: SuperiorMult PORT MAP (Clk => Clk, A => Au, B =>Bu, C =>(OTHERS=>'0'), OPMODE=> "1010101",
ALUMODE =>  "0000", CARRYINSEL => "000", CARRYIN => '0', PCIN => parcial3, P => parcial4, PCOUT =>
OPEN);
        Resultado<=parcial4(36 downto 0)&parcial3(16 downto 0)&parcial1(16 downto 0);

end Behavioral;
```

Como medio de control en Xilinx ISE, hemos generado un testbench



Multiplicador de números complejos

Utilizando el multiplicador del DSP también podemos generar una aplicación que nos permita realizar multiplicaciones de números complejos de 18 bits con signo mediante el siguiente interconexionado.



$$a = a.real + a.imag \cdot i$$

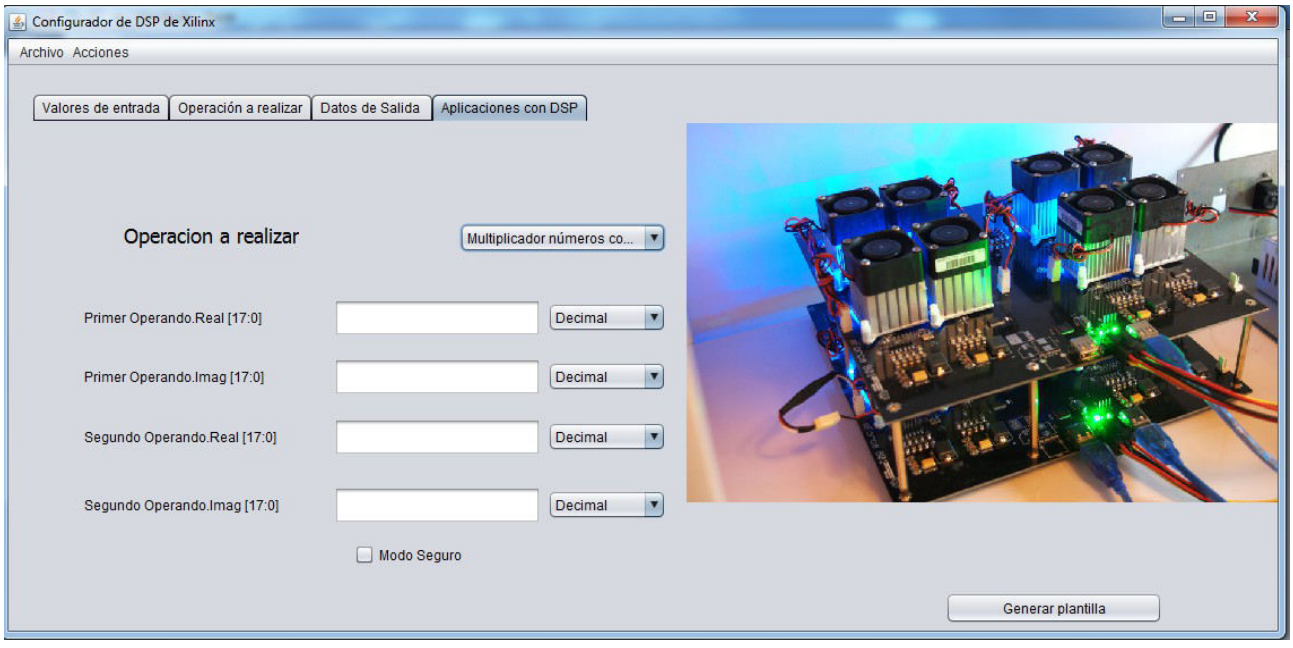
$$b = b.real + b.imag \cdot i$$

$$p = a \times b = p.real + p.imag \cdot i$$

$$p.real = a.real \times b.real - a.imag \times b.imag$$

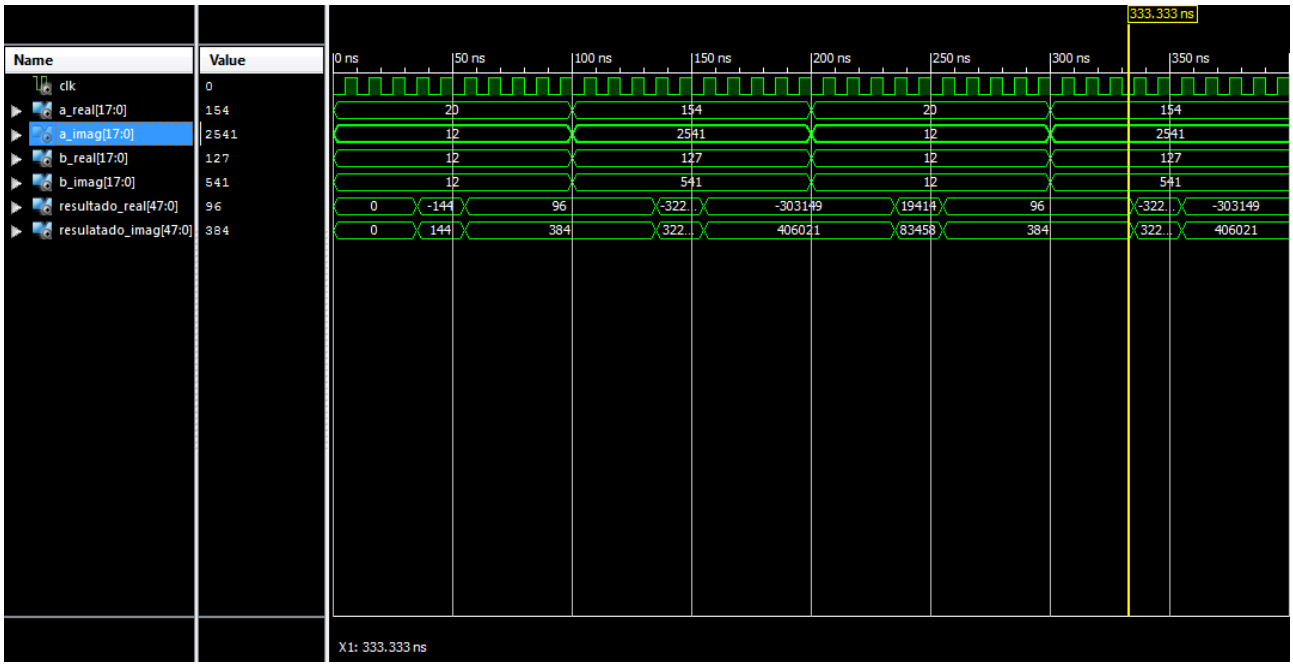
$$p.imag = a.real \times b.imag + a.imag \times b.real$$

Necesitamos dos DSP para calcular la parte real y otros dos para obtener la parte imaginaria, por lo tanto, al tratarse de operaciones independientes se pueden calcular en paralelo.
Como podemos apreciar el DSP3 y DSP1 realizan la misma operación (AxB + CIN), por lo tanto, tienen la mima configuración: OPMODE "0000101", ALUMODE "0000", CARRYINSEL"000" y CARRYIN '0'.
El DSP 4 nos da el resultado la parte real mediante la operación C - (AxB + CIN), dando los siguientes valores: OPMODE "0110101", ALUMODE "0011", CARRYINSEL "000", CARRYIN '0' y la entrada C recibe la salida P del DSP 3.
Y por ultimo el DSP 2 que difiere muy poco del DSP anterior, ya que la operación es C - (AxB + CIN) para ello modificamos el valor de ALUMODE "0000" y C la salida P del DSP 1. Esté nos proporciona el resultado de la parte imaginaria



Respecto a la aplicación unicamente tenemos que indicar la cuatro entradas de datos, para establecer el fichero vhdl contenido en el apéndice.

Y al igual que en el resto de los casos se ha realizado un banco de pruebas para Xilinx



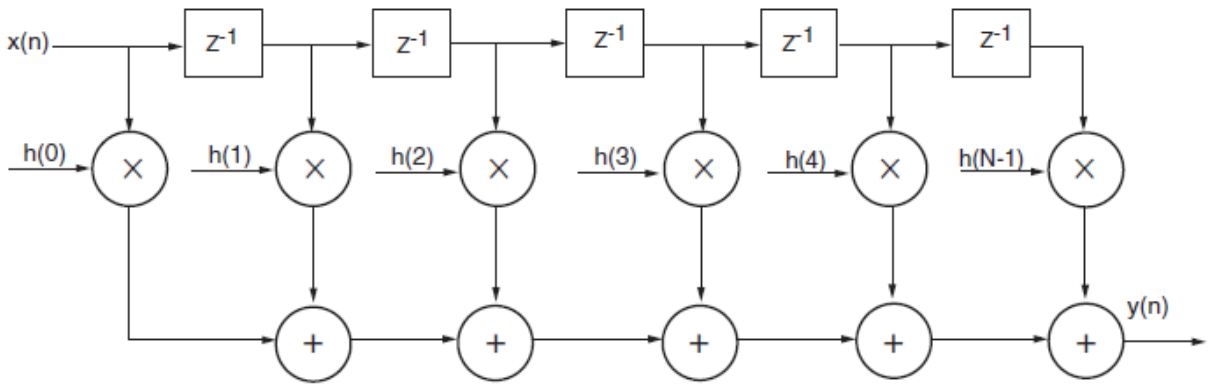
Filtro FIR

El filtro FIR (Finite Impulse Response) se basan en obtener la salida a partir, exclusivamente, de las entradas actuales y anteriores. Así, para un filtro de longitud N:

$$y(n) = \sum_{k=0}^{k=N-1} h(k)x(n-k)$$

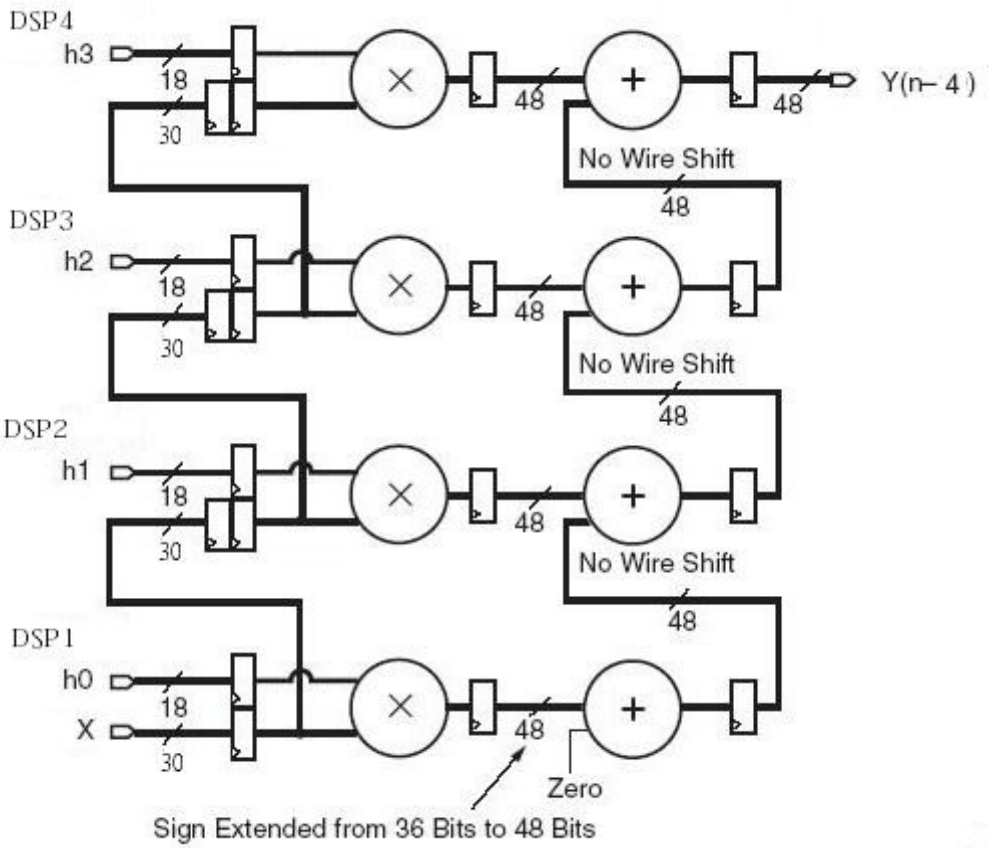
Los términos de la ecuación definen las muestras de entrada (x), las muestras de salida (y) y sus coeficientes (h). El valor de n corresponde con un instante de tiempo de tiempo de las muestras de entrada y salida.

Gráficamente podemos definirlo de la siguiente manera

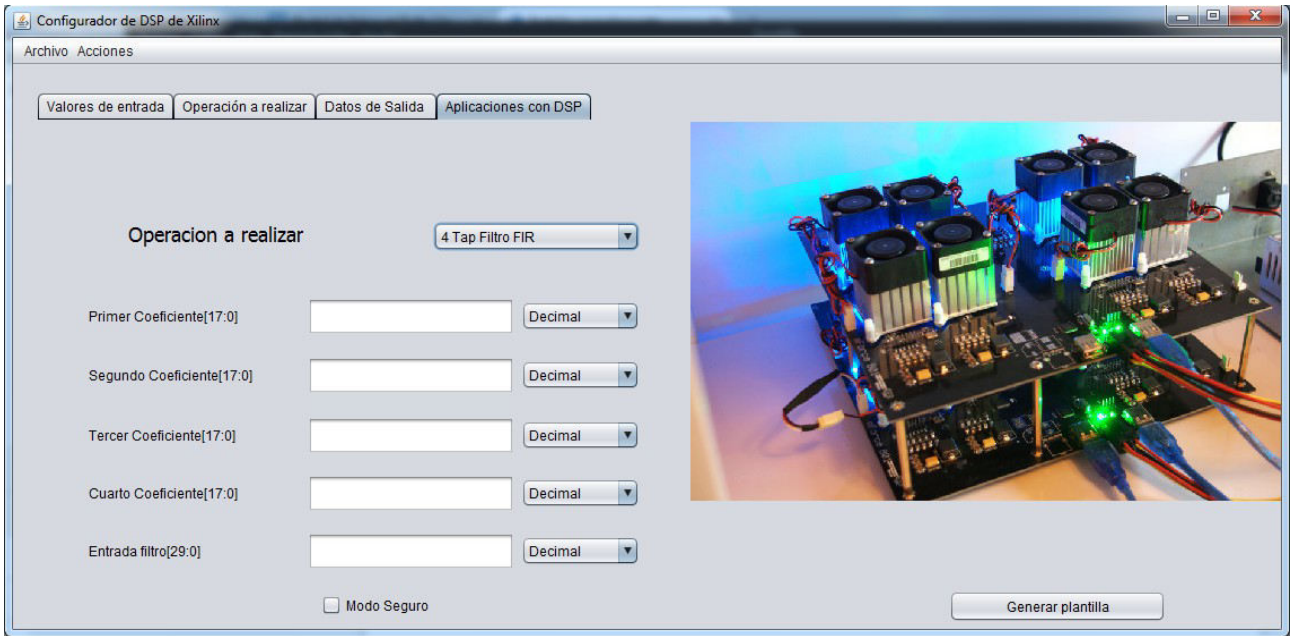


En nuestra aplicación hemos generado una filtro FIR de longitud 4. Para cada uno de los términos vamos a necesitar un DSP, es decir, utilizaremos un total de 4 DSP.

En cada DSP vamos a utilizar la entrada A que otorgaremos el valor de X, la entrada B por donde introduciremos el valor de los coeficientes y por PCIN el resultado obtenido del DSP anterior. Por lo tanto, todos los DSP mantendrán la misma configuración para realizar la operación (AxB + CIN). Para ello, establecemos los siguientes parámetros OPMODE "0000101", ALUMODE "0000", CARRYINSEL "000", CARRYIN '0' y PCIN tomara como entrada la sailda P del anterior DSP.

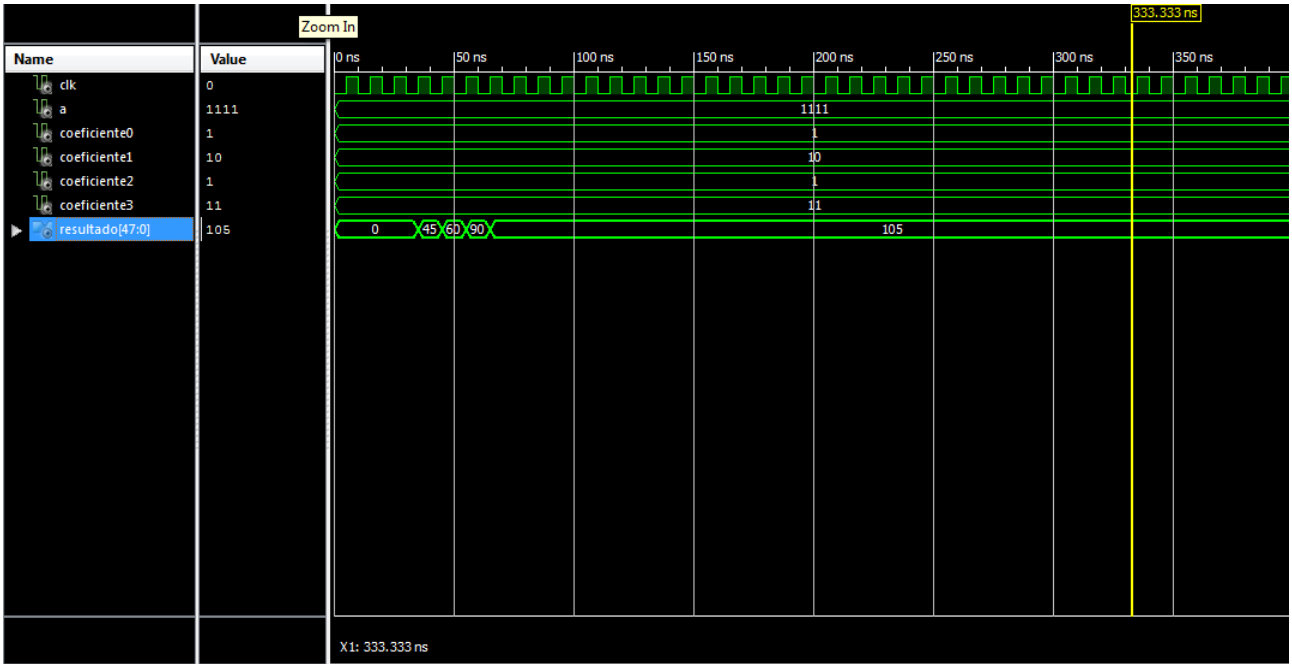


Para la aplicación Java tendremos que determinar los valores de X y los coeficientes.



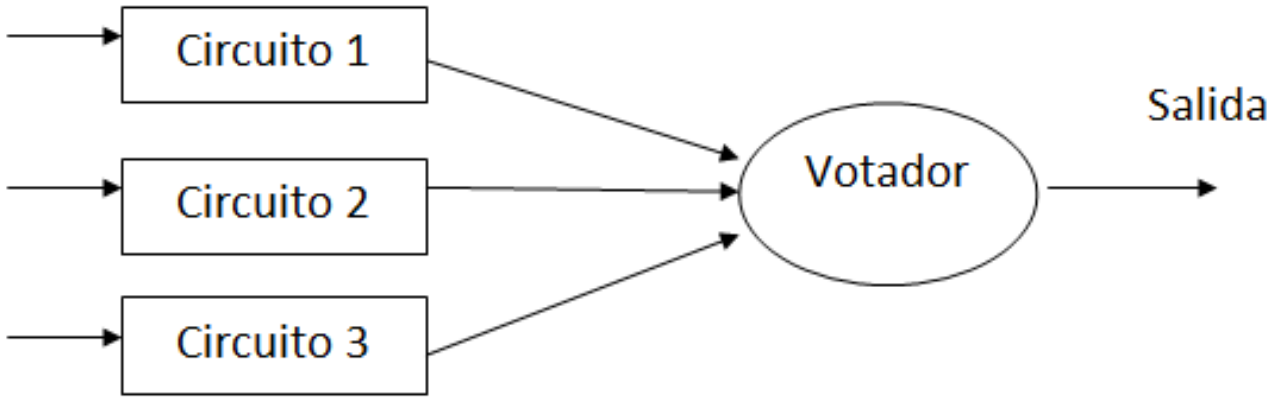
Para generar la plantilla incluida en el apéndice.

Y al igual que en el resto de los casos se ha realizado un banco de pruebas para comprobar el comportamiento del circuito en Xilinx.



Capitulo 5. Modo seguro

Para dotar de mayor robustez hemos generado un modo seguro para cada una de las cuatro aplicaciones anteriores basándonos en la técnica pasiva de redundancia triple modular, es decir, triplicamos el circuito y comparamos el resultado mediante un votador.



La implementación del votador se realiza mediante la opción de detección de patrones del módulo DSP que permite comparar dos señales entrantes. En el caso de que al menos dos de ellas sean iguales, la generación de resultados por parte de los módulos es válida y se difunde hacia la salida del votador, en otro caso, el votador resuelve una salida nula.

La ocupación del DSP comparador en la FPGA se detalla a continuación:

- Número de LUTs: 48 de 69120 0%
- Número de IOBs: 147 de 640 22%
- Número de DSP48E: 1 de 64 1%

Éste método se ha implementado mediante un sistema de plantillas debido a ser fácilmente extensible a futuras aplicaciones del DSP y para facilitar su generación a usuarios no familiarizados con el lenguaje VHDL.

Para emplear este método se han generado plantillas predefinidas para las cuatro aplicaciones anteriores. En las cuales existen los caracteres especiales '@' que la aplicación reconoce y sustituye por los valores elegidos por el usuario, generando un archivo VHDL funcional con tres instancias del modulo deseado y cuyas salidas se encuentran conectadas a un votador.

La diferencia de ocupación en la FPGA entre emplear este modo seguro o no es simplemente triplicar el uso de DSPs y la ocupación del módulo votador.

Modo Normal	*Mult 16x16*	*Mult 32x16*	*Mult 32x32*	*Filtro FIR 4-Tap*
Número DSP	1 de 64 (1%)	2 de 64 (3%)	4 de 64 (6%)	4 de 64 (6%)
Número LUTs	0 de 69120 (0%)	0 de 69120 (0%)	0 de 69120 (0%)	0 de 69120 (0%)
Número IOBs	0 de 640 (0%)	0 de 640 (0%)	0 de 640 (0%)	0 de 640 (0%)
Frecuencia Máx	-	607.903MHz	607.903MHz	591.017MHz

Modo Seguro	*Mult 16x16*	*Mult 32x16*	*Mult 32x32*	*Filtro FIR 4-Tap*
Número DSP	6 de 64 (3%)	9 de 64 (14%)	18 de 64 (28%)	15 de 64 (23%)
Número LUTs	0 de 69120 (0%)	0 de 69120 (0%)	0 de 69120 (0%)	0 de 69120 (0%)
Número IOBs	0 de 640 (0%)	0 de 640 (0%)	0 de 640 (0%)	0 de 640 (0%)
Frecuencia Máx	-	607.903MHz	607.903MHz	591.017MHz

Datos obtenidos trabajando sobre una placa Virtex-5 de modelo XC5VLX110T

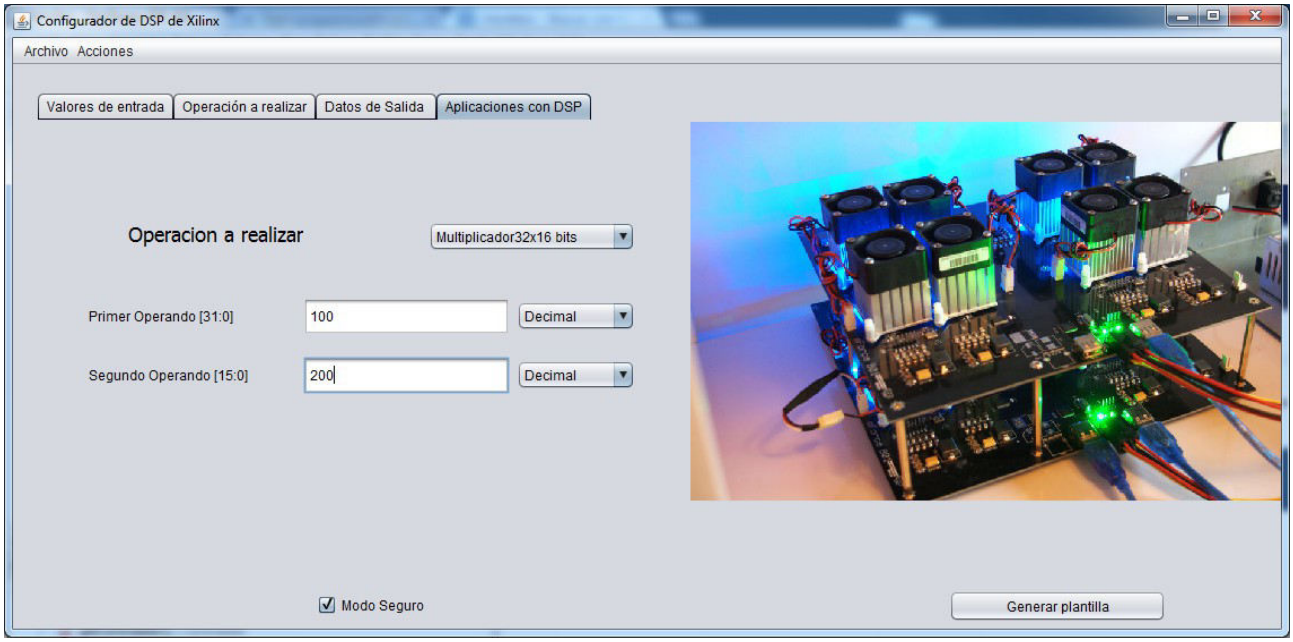
A continuación, mostramos el código generado en el caso de seleccionar el multiplicador 32x16.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
use ieee.numeric_std.ALL;

ENTITY plantillaMult32x16Seguro IS
END plantillaMult32x16Seguro;

ARCHITECTURE behavior OF plantillaMult32x16Seguro IS

   COMPONENT Mult32x16
   PORT(
      CLK :                  IN  std_logic;
      operandoA : IN  std_logic_vector(31 downto 0);
      operandoB : IN  std_logic_vector(15 downto 0);
      Resultado : OUT  std_logic_vector(50 downto 0)
     );
```

```
 END COMPONENT;
 component ComparadorFir is
        GENERIC (  size: INTEGER := 50);
        PORT (
                        CLK :                   in  STD_LOGIC;
                        Resultado1:     in STD_LOGIC_VECTOR (size DOWNTO 0);
                        Resultado2:     in STD_LOGIC_VECTOR (size DOWNTO 0);
                        Resultado3:     in STD_LOGIC_VECTOR (size DOWNTO 0);
                        Resultado:              out STD_LOGIC_VECTOR (size DOWNTO 0)
                );
        end component;

 --Inputs
 signal CLK :                   STD_LOGIC := '0';
 signal operandoA :             STD_LOGIC_VECTOR (31 DOWNTO 0);
 signal operandoB :             STD_LOGIC_VECTOR (15 DOWNTO 0);

        signal Resultado1:              STD_LOGIC_VECTOR (50 DOWNTO 0);
        signal Resultado2:              STD_LOGIC_VECTOR (50 DOWNTO 0);
        signal Resultado3:              STD_LOGIC_VECTOR (50 DOWNTO 0);
        --Outputs
 signal Resultado : std_logic_vector(50 downto 0);

BEGIN
 uut1: Mult32x16 PORT MAP (
      CLK => CLK,
      operandoA => operandoA,
      operandoB => operandoB,
      Resultado => Resultado1
    );
        uut2: Mult32x16 PORT MAP (
      CLK => CLK,
      operandoA => operandoA,
      operandoB => operandoB,
      Resultado => Resultado2
    );
        uut3: Mult32x16 PORT MAP (
      CLK => CLK,
      operandoA => operandoA,
      operandoB => operandoB,
      Resultado => Resultado3
    );
        uut4:ComparadorFIR GENERIC MAP (size=>50)
                        PORT MAP (
                        CLK => CLK,
                        Resultado1 => Resultado1,
                        Resultado2 => Resultado2,
                        Resultado3 => Resultado3,
                        Resultado  => Resultado
                        );
 process -- clock process for clk
 begin
                clock_loop: loop
                        CLK <= transport '0';
                        wait for 5 ns;
                        CLK <= transport '1';
                        wait for 5 ns;
                end loop clock_loop;
 end process;


 -- Stimulus process
 stim_proc: process
```

```
  begin
                operandoA <= transport std_logic_vector(to_signed(10, 32));
                operandoB <= transport std_logic_vector(to_signed(10, 16));
                wait for 100 ns;

        end process;

END;
```

Otra forma de generar un diseño mas robusto sin la necesidad de triplicar el módulo que estamos generando sería triplicar las señales de entrada mediante un votador mas sencillo, implementado mediante clausulas condicionales.
Para poder generar la plantilla en este modo seguro en el interfaz debemos seleccionar el checkbox "Modo Seguro", como se aprecia en la imagen.



Comparando la ocupación en la FPGA de un multiplicador 32x16 vemos que se obtiene un módulo más ligero que empleando la técnica de TMR.

Multiplicador 32x16	*Sin modo seguro*	*Modo seguro (interfaz)*	*Modo Seguro (TMR)*
Número DSP	2 de 64 (3%)	2 de 64 (3%)	9 de 64 (14%)
Número LUTs	0 de 69120 (0%)	0 de 69120 (0%)	0 de 69120 (0%)
Número IOBs	0 de 640 (0%)	0 de 640 (0%)	0 de 640 (0%)

Un ejemplo del código resultante en el caso del multiplicador 32x16 bits es el mostrado a continuación:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.ALL;


entity Mult32x16MonoSeguro is
end Mult32x16MonoSeguro;

architecture Behavioral of Mult32x16MonoSeguro is
```

```
	component Mult32x16 is
		PORT (
				CLK :			in  STD_LOGIC;
				operandoA:		in  STD_LOGIC_VECTOR (31 DOWNTO 0);
				operandoB:		in  STD_LOGIC_VECTOR (15 DOWNTO 0);
				Resultado:		out STD_LOGIC_VECTOR (47 DOWNTO 0)
			);
	end component;

	component ComparadorV2 is
	PORT (
			CLK :				in  STD_LOGIC;
			valorA :		in STD_LOGIC_VECTOR (47 DOWNTO 0);
			valorB :		in STD_LOGIC_VECTOR (47 DOWNTO 0);
			PATTERNBDETECT:	out STD_LOGIC;
		PATTERNDETECT:		out STD_LOGIC;
			RESULTADO:			out STD_LOGIC_VECTOR (47 DOWNTO 0)
			 );
	end component;

	signal CLK :				STD_LOGIC;
	signal Resultado1:		STD_LOGIC_VECTOR (47 DOWNTO 0);
	signal Resultado2:		STD_LOGIC_VECTOR (47 DOWNTO 0);
	signal Resultado3:		STD_LOGIC_VECTOR (47 DOWNTO 0);
	signal Resultado4:		STD_LOGIC_VECTOR (47 DOWNTO 0);
	signal Resultado5:		STD_LOGIC_VECTOR (47 DOWNTO 0);
	signal Resultado6:		STD_LOGIC_VECTOR (47 DOWNTO 0);
	signal operandoA:			STD_LOGIC_VECTOR (47 DOWNTO 0);
	signal operandoB:		STD_LOGIC_VECTOR (47 DOWNTO 0);

	signal Resultado:		STD_LOGIC_VECTOR (47 DOWNTO 0);

begin
	comp1:ComparadorV2
			PORT MAP (
			CLK => CLK,
			valorA => operandoA,
			valorB => operandoA,
			PATTERNBDETECT => OPEN,
			PATTERNDETECT => OPEN,
			RESULTADO => Resultado3
			);
	comp2:ComparadorV2
			PORT MAP (
			CLK => CLK,
			valorA => operandoA,
			valorB => operandoA,
			PATTERNBDETECT => OPEN,
			PATTERNDETECT => OPEN,
			RESULTADO => Resultado4
			);
	comp3:ComparadorV2
			PORT MAP (
			CLK => CLK,
			valorA => Resultado3,
			valorB => Resultado4,
			PATTERNBDETECT => OPEN,
			PATTERNDETECT => OPEN,
			RESULTADO => Resultado1
			);
	comp4:ComparadorV2
			PORT MAP (
```

```
                        CLK => CLK,
                        valorA => operandoB,
                        valorB => operandoB,
                        PATTERNBDETECT => OPEN,
                        PATTERNDETECT => OPEN,
                        RESULTADO => Resultado5
                        );

        comp5:ComparadorV2
                        PORT MAP (
                        CLK => CLK,
                        valorA => operandoB,
                        valorB => operandoB,
                        PATTERNBDETECT => OPEN,
                        PATTERNDETECT => OPEN,
                        RESULTADO => Resultado6
                        );

        comp6:ComparadorV2
                        PORT MAP (
                        CLK => CLK,
                        valorA => Resultado5,
                        valorB => Resultado6,
                        PATTERNBDETECT => OPEN,
                        PATTERNDETECT => OPEN,
                        RESULTADO => Resultado2
                        );

        utt:Mult32x16 PORT MAP (
                CLK => CLK,
                operandoA => Resultado1 (31 downto 0),
                operandoB => Resultado2 (15 downto 0),
                Resultado => Resultado
                  );

   process -- clock process for clk
   begin
                clock_loop: loop
                        CLK <= transport '0';
                        wait for 5 ns;
                        CLK <= transport '1';
                        wait for 5 ns;
                end loop clock_loop;
   end process;

   -- Stimulus process
   stim_proc: process
   begin
                operandoA <= transport std_logic_vector(to_signed(15, 48));
                operandoB <= transport std_logic_vector(to_signed(15, 48));
                wait for 100 ns;
        end process;

end Behavioral;
```

Conclusiones y trabajo futuro

El proyecto que hemos realizado ha contribuido de manera muy importante para reforzar nuestro conocimiento sobre las FPGA y desarrollar nuestra experiencia en los componentes DSP que incluyen. Nos deja muchas cosas importantes que reflexionar y muchas otras las ha reforzado como puntos angulares para llevar a cabo una buena implementación sobre dispositivos reconfigurables.

Dentro de los puntos que consideramos que tienen mas importancia dentro de un proyecto de esta naturaleza se encuentra la necesidad de rapidez y fiabilidad en las bases de los elementos que incluyen sistemas empotrados, los cuales asientan unas bases firmes y fiables para el control de dispositivos o el desarrollo de sistemas mas avanzados.

Como trabajo futuro a realizar en este proyecto, la creación de elementos mas complejos implementados sobre DSP para la aplicación a campos mas concretos. Podrían emplearse en sistemas aeronáuticos, industriales o energéticos empleando las bases expuestas anteriormente, conocimientos de un experto y documentación localizada del entorno.

Otro campo de expansión del trabajo consistiría en la ampliación del sistema de configuración para asociar un gran número de DSP entre varias placas FPGA del mismo o diferentes modelos.

Bibliografía

Xilinx. Virtex-5 FPGAXtremeDSP DesignConsiderations.User Guide.UG193 (v3.5) January 26, 2012.

Víctor Alaminos Benéitez. *Inyección de errores sobre fpgas tipo virtex-5*. 2012

Silvia Alcázar Andrés, Almudena Alonso de la Iglesia, Beatriz Álvarez-Buylla Fernández. *Técnicas de protección de circuitos para aplicaciones aeroespaciales sobre hardware reconfigurable.* 2013

Marcelo Salazar Arcucci. *Diseño de circuitos y sistemas electrónicos estudio del módulo dsp48a de la familia spartan-3a dsp de xilinx.* 2009

Enrique de Lucas Casamayor. *Metodología de síntesis para uso de bloques dsp con hdl sobre fpgas*. 2011

Carlos Diego Moreno Moreno. *Optimización de recursos hardware para la operación de convolución utilizada en el procesamiento digital de señales*. 2013

Miguel Fernández del Barrio. *Diseño, implementación y simulación del procesador externo de sonido de un implante coclear.* 2013

Godoy Garcés Mencía. *Técnicas de low-power design en FPGAs*. 2013

Sara Román Navarro. *Entorno para multitarea hardware en dispositivos reconfigurables con gestión dinámica de particiones y complejidad constante*, 2010

Hortensia Mecha López. *Diseño con hardware reconfigurable. 2001.Disponible en: http://www.dacya.ucm.es/horten/dhr/introduccion.pdf*

Iván Aldea López, María Bertrán de Lis Martín-Artajo *Estudio y diseño con hardware reconfigurable de un sistema de análisis citológico. 2006*

Miguel Ángel Sánchez Marcos. *Implementación de algoritmos de procesado de señal sobre FPGA: especificación, reutilización y exploración del espacio del diseño*. 2012

Angulo Montenegro Diego Marcelo, Calle Vélez Pedro Xavier, Ochoa Rodríguez David Leonardo. *Sistemas FPGA enfocados a DSP*. 2011. Disponible en: http://www.buenastareas.com/ensayos/Sistemas-Fpga-Enfocados-a-Dsp/3265612.html

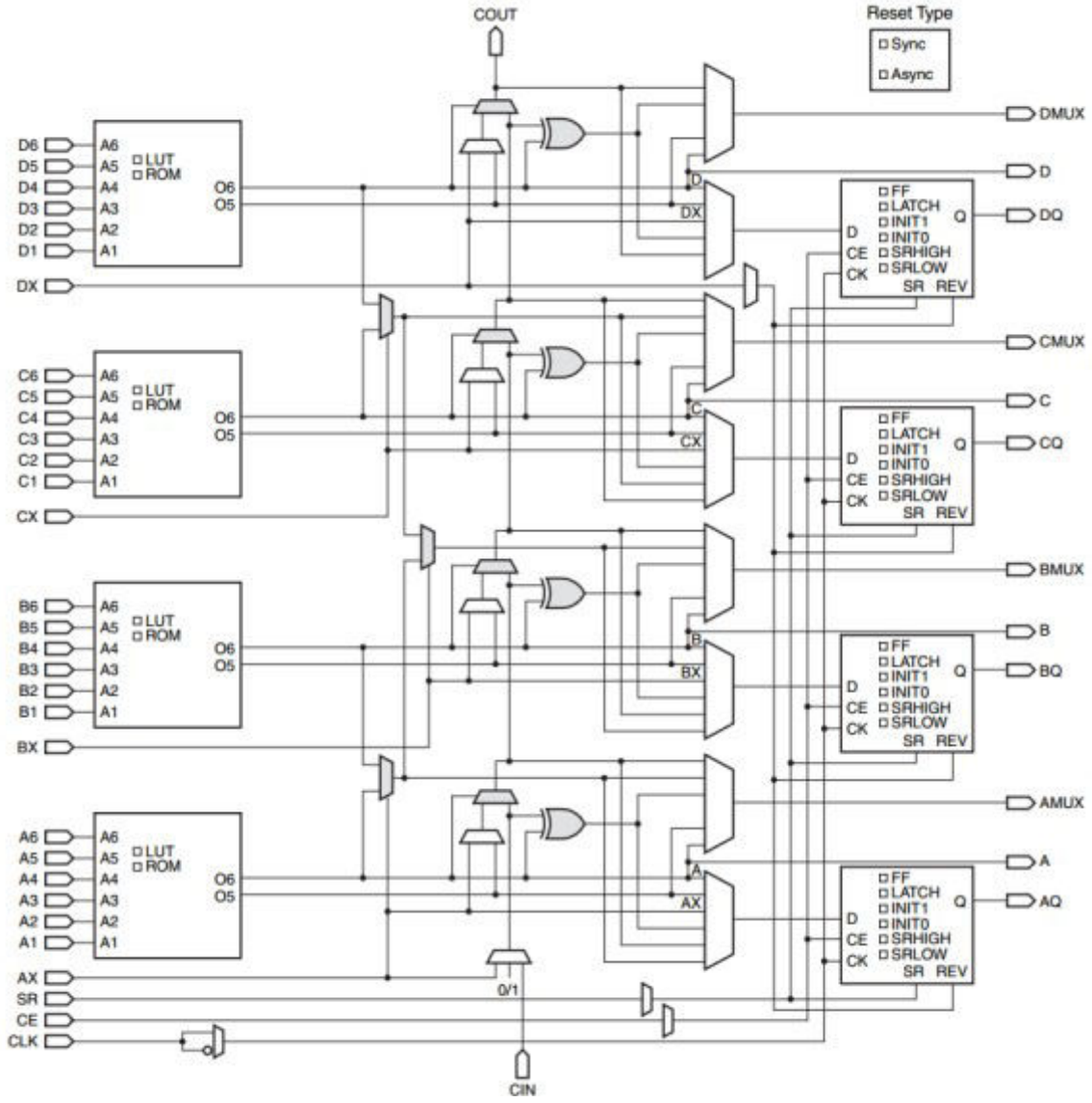
José Ramón Sendra Sendra. *Lógica Programable. Disponible en: http://www.iuma.ulpgc.es/~jrsendra/Docencia/Electronica_Basica/download/transparencias/logica_programable.pdf*

Abreviatura y acrónimos:

ASIC	Application Specific Integrated Circuit
BASIC	Beginner's All-purpose Symbolic Instruction Code
BRAM	Bloc RAM
CAD	Computer-aided design
CLB	Configurable logic blocks
CMOS	Complementary metal-oxide-semiconductor
DCM	Digital Clock Managers
DSP	Digital Signal Processing
E/S	Entrada – Salida
EEPROM	Electrically Erasable Programmable Read-Only Memory
EPROM	Erasable Programmable Read-Only Memory
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
GAL	Generic array logic
HDL	Hardware Description Languaje
I/O IO	Input-Output
IOB	Input Output Block
ISE	Integrated Software Environment
JTAG	Joing Test Action Group
KB	Kilobyte
LUT	Look-up-Table
MB	Megabyte
MHz	Megahercio
NRE	Nonrecurring Enginnering
PAL	Programmable Array Logic
PLA	Array Lógico Programable
PLD	Programmable Logic Device
PLL	phase-locked loop
PROM	programmable read-only memory
PSM	Programmable Swiching Matrix
RAM	random-access memory
ROM	Read Only Memory
SATA	Serial Advanced Technology Attachment
SRAM	Static random-access memory
USB	Universal Serial Bus
VHDL	Very Hgth Speed Hardware Description Languaje

Anexo:

Estructura de un CLB de la FPGA virtex-5



Código de MyDSP:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

library UNISIM;
use UNISIM.VComponents.all;

entity MyDSPMult is
	Port (
		CLK :		in  STD_LOGIC;
		A :		in  STD_LOGIC_VECTOR (29 DOWNTO 0);
		B :		in STD_LOGIC_VECTOR (17 DOWNTO 0);
		C :		in STD_LOGIC_VECTOR (47 DOWNTO 0);
		OPMODE :	in STD_LOGIC_VECTOR (6 DOWNTO 0);
		ALUMODE :	in STD_LOGIC_VECTOR (3 DOWNTO 0);
		CARRYINSEL :	in STD_LOGIC_VECTOR (2 DOWNTO 0);
```

```
                CARRYIN :       in STD_LOGIC;
                PCIN :          in STD_LOGIC_VECTOR (47 DOWNTO 0);
                P :             out  STD_LOGIC_VECTOR (47 DOWNTO 0);
                PCOUT :         out STD_LOGIC_VECTOR (47 DOWNTO 0)
                );
end MyDSPMult;

architecture Behavioral of MyDSPMult is

begin

DSP48E_inst : DSP48E
GENERIC MAP(
        ACASCREG => 1,
        ALUMODEREG => 1,
        AREG => 2,
        AUTORESET_PATTERN_DETECT => FALSE,
        AUTORESET_PATTERN_DETECT_OPTINV => "MATCH",
        A_INPUT => "DIRECT",
        BCASCREG => 1,
        BREG => 2,
        B_INPUT => "DIRECT",
        CARRYINREG => 1,
        CARRYINSELREG => 1,
        CREG => 1,
        MASK => X"3FFFFFFFFFFF",
        MREG => 1,
        MULTCARRYINREG => 1,
        OPMODEREG => 1,
        PATTERN => X"000000000000",
        PREG => 1,
        SEL_ROUNDING_MASK => "SEL_MASK",
        SEL_MASK => "MASK",
        SEL_PATTERN => "PATTERN",
        USE_MULT => "MULT_S",
        USE_PATTERN_DETECT => "NO_PATDET",
        USE_SIMD => "ONE48")
PORT MAP(
        CLK => CLK,
        A => A,
        B => B,
        C => C,
        CARRYIN => CARRYIN,
        ACIN => (OTHERS => '0'),
        BCIN => (OTHERS => '0'),
        PCIN => PCIN,
        CARRYCASCIN => '0',
        MULTSIGNIN => '0',
        ACOUT => OPEN,
        BCOUT => OPEN,
        CARRYCASCOUT => OPEN,
        MULTSIGNOUT => OPEN,
        P => P,
        PATTERNBDETECT => OPEN,
        PATTERNDETECT => OPEN,
        OVERFLOW => OPEN,
        UNDERFLOW => OPEN,
        CARRYOUT => OPEN,
        PCOUT => PCOUT,
        OPMODE => OPMODE,
        ALUMODE => ALUMODE,
        CARRYINSEL => CARRYINSEL,
        CEA1 => '1',
```

```
        CEA2 => '1',
        CEALUMODE => '1',
        CEB1 => '1',
        CEB2 => '1',
        CEC => '1',
        CECARRYIN => '1',
        CEM => '1',
        CECTRL => '1',
        CEP => '1',
        CEMULTCARRYIN => '1',
        RSTA => '0',
        RSTALUMODE => '0',
        RSTB => '0',
        RSTC => '0',
        RSTALLCARRYIN => '0',
        RSTM => '0',
        RSTCTRL => '0',
        RSTP => '0');
end Behavioral;
```

Código SuperiorMult:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
--USE opers.ALL;
entity SuperiorMult is
        PORT (
                        CLK :           in  STD_LOGIC;
                        A :             in  STD_LOGIC_VECTOR (29 DOWNTO 0);
                        B :             in STD_LOGIC_VECTOR (17 DOWNTO 0);
                        C :             in STD_LOGIC_VECTOR (47 DOWNTO 0);
                        OPMODE :        in STD_LOGIC_VECTOR (6 DOWNTO 0);--=> "0001111",
                        ALUMODE :       in STD_LOGIC_VECTOR (3 DOWNTO 0);--=> "0000",
                        CARRYINSEL :  in STD_LOGIC_VECTOR (2 DOWNTO 0);--=> "000",
                        CARRYIN :       in STD_LOGIC;
                        PCIN :          in STD_LOGIC_VECTOR (47 DOWNTO 0);
                        P :             out  STD_LOGIC_VECTOR (47 DOWNTO 0);
                        PCOUT :         out STD_LOGIC_VECTOR (47 DOWNTO 0)
                          );
end SuperiorMult;

architecture Behavioral of SuperiorMult is

COMPONENT MyDSPMult is
         Port (
                        CLK :           in  STD_LOGIC;
                        A :             in  STD_LOGIC_VECTOR (29 DOWNTO 0);
                        B :             in STD_LOGIC_VECTOR (17 DOWNTO 0);
                        C :             in STD_LOGIC_VECTOR (47 DOWNTO 0);
                        OPMODE :        in STD_LOGIC_VECTOR (6 DOWNTO 0);--=> "0001111",
                        ALUMODE :       in STD_LOGIC_VECTOR (3 DOWNTO 0);--=> "0000",
                        CARRYINSEL :  in STD_LOGIC_VECTOR (2 DOWNTO 0);--=> "000",
                        CARRYIN :       in STD_LOGIC;
                        PCIN :          in STD_LOGIC_VECTOR (47 DOWNTO 0);
                        P :             out  STD_LOGIC_VECTOR (47 DOWNTO 0);
                        PCOUT :         out STD_LOGIC_VECTOR (47 DOWNTO 0)
                          );
end COMPONENT;

begin
```

```
u1: MyDSPMult PORT MAP (CLK,A,B,C,OPMODE,ALUMODE,CARRYINSEL,CARRYIN,PCIN,P,PCOUT);

end Behavioral;
```

Código del multiplicador de números complejos:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity MultNumCompl is
	PORT (
			CLK :			in STD_LOGIC;
			A_real :		in STD_LOGIC_VECTOR (17 DOWNTO 0);
			B_real :		in STD_LOGIC_VECTOR (17 DOWNTO 0);
			A_imag :		in STD_LOGIC_VECTOR (17 DOWNTO 0);
			B_imag :		in STD_LOGIC_VECTOR (17 DOWNTO 0);
			Resultado_real :	out STD_LOGIC_VECTOR (47 DOWNTO 0);
			Resulatado_imag :	out STD_LOGIC_VECTOR (47 DOWNTO 0));

end MultNumCompl;

architecture Behavioral of MultNumCompl is


component SuperiorMult is
	PORT (
			CLK :		in STD_LOGIC;
			A :		in STD_LOGIC_VECTOR (29 DOWNTO 0);
			B :		in STD_LOGIC_VECTOR (17 DOWNTO 0);
			C :		in STD_LOGIC_VECTOR (47 DOWNTO 0);
			OPMODE :	in STD_LOGIC_VECTOR (6 DOWNTO 0);
			ALUMODE :	in STD_LOGIC_VECTOR (3 DOWNTO 0);
			CARRYINSEL:	in STD_LOGIC_VECTOR (2 DOWNTO 0);
			CARRYIN :	in STD_LOGIC;
			PCIN :		in STD_LOGIC_VECTOR (47 DOWNTO 0);
			P :		out STD_LOGIC_VECTOR (47 DOWNTO 0);
			PCOUT :		out STD_LOGIC_VECTOR (47 DOWNTO 0));
end component;

SIGNAL AZero: STD_LOGIC_VECTOR(11 DOWNTO 0) := (OTHERS=>'0');
SIGNAL Areal: STD_LOGIC_VECTOR(29 DOWNTO 0);
SIGNAL Aimag: STD_LOGIC_VECTOR(29 DOWNTO 0);
SIGNAL Caux: STD_LOGIC_VECTOR(29 DOWNTO 0):= (OTHERS=>'0');
SIGNAL ArxBr: STD_LOGIC_VECTOR(47 DOWNTO 0);
SIGNAL ArxBi: STD_LOGIC_VECTOR(47 DOWNTO 0);
begin

	multiplicarNumerosComplejos: process (clk, A_real, B_real,A_imag,B_imag)
	begin
		Areal<=AZero&A_real;
		Aimag<=AZero&A_imag;
	end process;

--dos DSP48 para calcular la parte real
ArealxBreal: SuperiorMult PORT MAP (Clk => Clk, A => Areal, B =>B_real, C =>(OTHERS=>'0'), OPMODE=>
"0000101", ALUMODE =>  "0000", CARRYINSEL => "000", CARRYIN => '0', PCIN => (others=>'0'), P => ArxBr,
PCOUT => OPEN);

AimagxBimag: SuperiorMult PORT MAP (Clk => Clk, A => Aimag,B =>B_real,C =>ArxBr,OPMODE=>
"0110101",ALUMODE => "0011",CARRYINSEL => "000",CARRYIN => '0',PCIN => (others=>'0'),P =>
Resultado_real, PCOUT => OPEN);
```

```
--dos DSP48 para calcular la parte imaginaria
ArealxBimag: SuperiorMult PORT MAP (Clk => Clk, A => Areal, B =>B_imag, C =>(OTHERS=>'0'), OPMODE=>
"0000101", ALUMODE =>  "0000", CARRYINSEL => "000", CARRYIN => '0', PCIN => (others=>'0'), P => ArxBi,
PCOUT => OPEN);

AimaglxBreal: SuperiorMult PORT MAP (Clk => Clk,A => Aimag,B =>B_real,C =>ArxBi,OPMODE=>
"0110101",ALUMODE =>  "0000",CARRYINSEL => "000",CARRYIN => '0',PCIN => (others=>'0'),P =>
Resulatado_imag, PCOUT => OPEN);

end Behavioral;
```

Código del filtro 4-Tap FIR:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Tap4FirFilter is
        PORT (
                        CLK :                   in  STD_LOGIC;
                        A:                      in  STD_LOGIC_VECTOR (29 DOWNTO 0);
                        COEFICIENTE0:           in  STD_LOGIC_VECTOR (17 DOWNTO 0);
                        COEFICIENTE1:           in  STD_LOGIC_VECTOR (17 DOWNTO 0);
                        COEFICIENTE2:           in  STD_LOGIC_VECTOR (17 DOWNTO 0);
                        COEFICIENTE3:           in  STD_LOGIC_VECTOR (17 DOWNTO 0);
                        Resultado:              out STD_LOGIC_VECTOR (47 DOWNTO 0));
end Tap4FirFilter;

architecture Behavioral of Tap4FirFilter is
        component SuperiorMult is
                PORT (
                                CLK :                   in  STD_LOGIC;
                                A :                     in  STD_LOGIC_VECTOR (29 DOWNTO 0);
                                B :                     in STD_LOGIC_VECTOR (17 DOWNTO 0);
                                C :                     in STD_LOGIC_VECTOR (47 DOWNTO 0);
                                OPMODE :                in STD_LOGIC_VECTOR (6 DOWNTO 0);
                                ALUMODE :               in STD_LOGIC_VECTOR (3 DOWNTO 0);
                                CARRYINSEL :            in STD_LOGIC_VECTOR (2 DOWNTO 0);
                                CARRYIN :               in STD_LOGIC;
                                PCIN :                  in STD_LOGIC_VECTOR (47 DOWNTO 0);
                                P :                     out  STD_LOGIC_VECTOR (47 DOWNTO 0);
                                PCOUT :                 out STD_LOGIC_VECTOR (47 DOWNTO 0));
        end component;

        SIGNAL P1: STD_LOGIC_VECTOR(47 DOWNTO 0);
        SIGNAL P2: STD_LOGIC_VECTOR(47 DOWNTO 0);
        SIGNAL P3: STD_LOGIC_VECTOR(47 DOWNTO 0);


begin

utt0: SuperiorMult PORT MAP (Clk => Clk, A => A, B =>COEFICIENTE0, C =>(OTHERS=>'0'), OPMODE=>
"0000101", ALUMODE =>  "0000", CARRYINSEL => "000", CARRYIN => '0', PCIN => (others=>'0'),P => OPEN,
PCOUT => P1);

uut1: SuperiorMult PORT MAP (Clk => Clk, A => A, B =>COEFICIENTE1, C =>(OTHERS=>'0'), OPMODE=>
"0010101", ALUMODE =>  "0000", CARRYINSEL => "000", CARRYIN => '0', PCIN =>P1 , P => OPEN, PCOUT
=> P2);

uut2: SuperiorMult PORT MAP (Clk => Clk, A => A, B =>COEFICIENTE2, C =>(OTHERS=>'0'), OPMODE=>
"0010101", ALUMODE =>  "0000", CARRYINSEL => "000", CARRYIN => '0', PCIN =>P2 , P => OPEN, PCOUT
=> P3);
```

```
uut3: SuperiorMult PORT MAP (Clk => Clk, A => A, B =>COEFICIENTE3, C =>(OTHERS=>'0'), OPMODE=>
"0010101", ALUMODE =>  "0000", CARRYINSEL => "000", CARRYIN => '0', PCIN =>P3 ,P => Resultado, PCOUT
=> OPEN);

end Behavioral;
```

Código del módulo comparador:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


entity ComparadorV2 is
	PORT (
			CLK :			in  STD_LOGIC;
			valorA :		in STD_LOGIC_VECTOR (47 DOWNTO 0);
			valorB :		in STD_LOGIC_VECTOR (47 DOWNTO 0);
			PATTERNBDETECT:	out STD_LOGIC;
			PATTERNDETECT:	out STD_LOGIC;
			RESULTADO:		out STD_LOGIC_VECTOR (47 DOWNTO 0)
			  );
end ComparadorV2;

architecture Behavioral of ComparadorV2 is

component MyDSPCompV2 is
	 Port (	CLK :			in  STD_LOGIC;
		A :			in  STD_LOGIC_VECTOR (29 DOWNTO 0);
		B :			in STD_LOGIC_VECTOR (17 DOWNTO 0);
		C :			in STD_LOGIC_VECTOR (47 DOWNTO 0);
		OPMODE :		in STD_LOGIC_VECTOR (6 DOWNTO 0);
		ALUMODE :		in STD_LOGIC_VECTOR (3 DOWNTO 0);
		CARRYINSEL :		in STD_LOGIC_VECTOR (2 DOWNTO 0);
		CARRYIN :		in STD_LOGIC;
		PCIN :			in STD_LOGIC_VECTOR (47 DOWNTO 0);
		P :			out  STD_LOGIC_VECTOR (47 DOWNTO 0);
		PCOUT :			out STD_LOGIC_VECTOR (47 DOWNTO 0);
		PATTERNBDETECT:		out STD_LOGIC;
		PATTERNDETECT:		out STD_LOGIC
			  );
end component;

	SIGNAL PCOUT :		 STD_LOGIC_VECTOR (47 DOWNTO 0);
	SIGNAL P :		STD_LOGIC_VECTOR (47 DOWNTO 0);
	SIGNAL COMP :		STD_LOGIC := '0';

begin

	transformarSeñales: process (COMP)

	begin
		if(COMP='1') then
			RESULTADO <= valorA;
		else
			RESULTADO <= (others=>'0');
		end if;
	end process;

	utt1: MyDSPCompV2 PORT MAP (Clk,(others=>'0'),(others=>'0'),valorA, "0011100", "0011" , "000" ,
CARRYIN => '0', PCIN => valorB, P => P, PCOUT => PCOUT, PATTERNBDETECT => COMP, PATTERNDETECT
=> PATTERNDETECT);
```

```
	PATTERNBDETECT <= COMP;

end Behavioral;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


entity MyDSPComp2 is
	Port (	CLK :			in  STD_LOGIC;
		A :			in  STD_LOGIC_VECTOR (29 DOWNTO 0);
		B :			in STD_LOGIC_VECTOR (17 DOWNTO 0);
		C :			in STD_LOGIC_VECTOR (47 DOWNTO 0);
		OPMODE :		in STD_LOGIC_VECTOR (6 DOWNTO 0);
		ALUMODE :		in STD_LOGIC_VECTOR (3 DOWNTO 0);
		CARRYINSEL :		in STD_LOGIC_VECTOR (2 DOWNTO 0);
		CARRYIN :		in STD_LOGIC;
		PCIN :			in STD_LOGIC_VECTOR (47 DOWNTO 0);
		P :			out  STD_LOGIC_VECTOR (47 DOWNTO 0);
		PCOUT :			out STD_LOGIC_VECTOR (47 DOWNTO 0);
		PATTERNBDETECT:		out STD_LOGIC;
		PATTERNDETECT:		out STD_LOGIC
			);
end MyDSPComp2;

architecture Behavioral of MyDSPComp2 is

begin


DSP48E_inst : DSP48E
GENERIC MAP(
	ACASCREG => 1,
	ALUMODEREG => 1,
	AREG => 2,
	AUTORESET_PATTERN_DETECT => FALSE,
	AUTORESET_PATTERN_DETECT_OPTINV => "MATCH",
	A_INPUT => "DIRECT",
	BCASCREG => 1,
	BREG => 2,
	B_INPUT => "DIRECT",
	CARRYINREG => 1,
	CARRYINSELREG => 1,
	CREG => 1,
	MASK => X"3FFFFFFFFFFF",
	MREG => 1,
	MULTCARRYINREG => 1,
	OPMODEREG => 1,
	PATTERN => X"000000000000",
	PREG => 1,
	SEL_ROUNDING_MASK => "SEL_MASK",
	SEL_MASK => "MASK",
	SEL_PATTERN => "PATTERN",
	USE_MULT => "MULT_S",
	USE_PATTERN_DETECT => "PATDET",
	USE_SIMD => "ONE48")
PORT MAP(
	CLK => CLK,
	A => A,
	B => B,
	C => C,
	CARRYIN => CARRYIN,
```

```
            ACIN => (OTHERS => '0'),
            BCIN => (OTHERS => '0'),
            PCIN => PCIN,
            CARRYCASCIN => '0',
            MULTSIGNIN => '0',
            ACOUT => OPEN,
            BCOUT => OPEN,
            CARRYCASCOUT => OPEN,
            MULTSIGNOUT => OPEN,
            P => P,
            PATTERNBDETECT => PATTERNBDETECT,
            PATTERNDETECT => PATTERNDETECT,
            OVERFLOW => OPEN,
            UNDERFLOW => OPEN,
            CARRYOUT => OPEN,
            PCOUT => PCOUT,
            OPMODE => OPMODE,
            ALUMODE => ALUMODE,
            CARRYINSEL => CARRYINSEL,
            CEA1 => '1',
            CEA2 => '1',
            CEALUMODE => '1',
            CEB1 => '1',
            CEB2 => '1',
            CEC => '1',
            CECARRYIN => '1',
            CEM => '1',
            CECTRL => '1',
            CEP => '1',
            CEMULTCARRYIN => '1',
            RSTA => '0',
            RSTALUMODE => '0',
            RSTB => '0',
            RSTC => '0',
            RSTALLCARRYIN => '0',
            RSTM => '0',
            RSTCTRL => '0',
            RSTP => '0');
end Behavioral;
```