

**Universidad Politécnica de Madrid**

Escuela Técnica Superior de Ingenieros Industriales

Departamento de Automática, Ingeniería Electrónica e Informática Industrial



TRABAJO FIN DE GRADO

# **Simulación y medida de consumo en FPGAs para arquitecturas de operadores aritméticos**

**Autor: Juan José Montes Salinero**

**Tutores: José Andrés Otero Marnotes**

**Teresa Riesgo Alcaide**



# Índice

<b>Índice de figuras.....</b>	<b>3</b>
<b>Índice de Tablas.....</b>	<b>5</b>
<b>Resumen ejecutivo.....</b>	<b>6</b>
<b>Capítulo 1: Introducción y objetivos.....</b>	<b>8</b>
1.1. Estado del arte .....	8
1.2. Diseño de sistemas electrónicos digitales.....	10
1.2.1. Diseño digital de bajo consumo.....	10
1.3. Circuitos integrados configurables.....	14
1.3.1. FPGAs.....	14
1.3.2. Consumo en FPGAs .....	17
<b>Capítulo 2: Herramientas y Lenguajes para el diseño de sistemas Digitales .....</b>	<b>19</b>
2.1. Lenguajes de descripción de hardware (HDL).....	19
2.1.1. VHDL .....	19
2.2. Xilinx Vivado Design Tools.....	20
2.2.1. Vivado Design Suite .....	20
2.2.2. Vivado HLS .....	21
2.2.3. PYNQ-Z1 .....	21
<b>Capítulo 3: Multiplicadores digitales.....</b>	<b>22</b>
3.1. Multiplicadores sin signo.....	23
3.1.1. Multiplicador Ripplecarry.....	23
3.1.2. Multiplicador Carrysave/de Braun .....	23
3.1.3. Multiplicador de Wallace.....	24
3.2. Multiplicadores con signo(C-2).....	25
3.2.1. Multiplicador Booth paralelo.....	25
3.2.2. Multiplicador Baugh-Wooley.....	26
<b>Capítulo 4: Metodología.....</b>	<b>27</b>
4.1. Metodología de diseño descendente.....	27
4.2. Proceso de diseño de sistemas basados en FPGA .....	27

4.3. Técnicas de diseño .....	29
4.3.1. Paralelización .....	29
4.3.2. Segmentación/pipelining .....	30
4.3.3. Clock-gating .....	30
4.4. Parámetros de diseño/especificaciones .....	31
4.5. Estimación de consumo.....	31
4.5.1. Estimación de consumo con Vivado.....	32
4.6. Archivo SAIF .....	32
4.7. Archivo XDC .....	37
4.8. FPGA in the Loop .....	38
4.9. Vivado HLS .....	45
4.10. Medición de consumo.....	50
<b>Capítulo 5: Resultados y discusión.....</b>	<b>52</b>
5.1. Recursos .....	52
5.2. Timing.....	57
5.3. Pipelining .....	59
5.4. Consumo.....	62
5.4.1. Estimación de consumo.....	62
5.4.2. Consumo Real .....	70
<b>Capítulo 6: Conclusiones .....</b>	<b>71</b>
<b>Capítulo 7: Líneas futuras .....</b>	<b>73</b>
<b>Capítulo 8: Planificación temporal y presupuesto .....</b>	<b>74</b>
8.1. Presupuesto .....	74
8.2. Planificación temporal.....	74
<b>A. Anexo I: Código VHDL .....</b>	<b>75</b>
<b>B. Anexo II: Pines placa PYNQ.....</b>	<b>94</b>
<b>C. Anexo III: Referencias.....</b>	<b>95</b>
C.1. Bibliografía y referencias.....	95
C.2. Abreviaturas, unidades y acrónimos .....	97
C.3. Glosario.....	98

## Índice de figuras

Figura 1.1: Inversor CMOS [13] .....	10
Figura 1.2: Técnica de paralelización .....	13
Figura 1.3: Técnica de segmentación .....	13
Figura 1.4: Arquitectura de un FPGA de Xilinx[2] .....	15
Figura 1.5: Bloque lógico configurable, serie 7 de Xilinx [28] .....	15
Figura 1.6: True Dual-Port Data Flows for a RAMB36, serie 7 de Xilinx [23] .....	16
Figura 1.7: Basic DPS48E1, serie 7 de Xilinx [30] .....	17
Figura 1.8: Transistor de paso[13] .....	18
Figura 2.1: Placa PYNQ [5] .....	21
Figura 3.1: Multiplicador Ripplecarry de 4 bits [14] .....	23
Figura 3.2: Multiplicador Carrysave de 4 bits [14] .....	24
Figura 3.3: Multiplicador de Wallace de 4 bits [14] .....	24
Figura 3.4: Celda CASS [12] .....	25
Figura 3.5: Multiplicador de Booth paralelo de 4 bits [12].....	25
Figura 3.6: Multiplicador Baugh-Wooley de 6 x 4 bits [12].....	26
Figura 4.1: Paralelización.....	30
Figura 4.2: Segmentación.....	30
Figura 4.3: Proceso de estimación de consumo con Vivado [24] .....	32
Figura 4.4: Load Shift register generador de secuencia de máxima longitud [16] .....	33
Figura 4.5: Simulación con LFSR a 100 MHz para generar archivo SAIF .....	33
Figura 4.6: Ventana de Environment Report Power .....	34
Figura 4.7: Resumen de la estimación del Report Power .....	36
Figura 4.8: Gráfica XPE consumo/Voltaje .....	36
Figura 4.9: Gráfica de XPE Consumo/Temperatura .....	36
Figura 4.10: Comunicación Matlab – FPGA [9].....	38
Figura 4.11: Características de la placa.....	39
Figura 4.12: Posición JTAG [4] .....	39
Figura 4.13: Conexión USB-JTAG [4] .....	39
Figura 4.14: Bloque FIL.....	42
Figura 4.15: Modelo Simulink para Multiplicadores con signo.....	42
Figura 4.16: Salida de multiplicadores con signo .....	43
Figura 4.17: Diferencia entre señales .....	44
Figura 4.18: Modelo de Simulink para multiplicadores sin signo .....	44
Figura 4.19: Salida multiplicadores sin signo en FIL .....	45
Figura 4.20: Flujo de diseño en Vivado HLS [21].....	46
Figura 4.21: Función Código C.....	46
Figura 4.22: Resultado de la simulación C .....	47
Figura 4.23: Testbench código C .....	47

Figura 4.24: Resultados de la síntesis en Vivado HLS .....	48
Figura 4.25: Report obtenido al generar bloque IP con Vivado HLS .....	49
Figura 4.26: Bloque IP generado.....	49
Figura 4.27: Configuración Fuente LFSR.....	50
Figura 4.28: Modelo simulink para medir el consumo .....	50
Figura 4.29: Placa utilizada para medir el consumo .....	51
Figura 4.30: Lectura de consumo de la placa.....	51
Figura 5.1: LUTs utilizados por los multiplicadores sin signo .....	56
Figura 5.2: LUTs utilizados por los multiplicadores con signo .....	57
Figura 5.3: Frecuencia máxima de funcionamiento multiplicadores de 4bits.....	57
Figura 5.4: Frecuencia máxima de funcionamiento multiplicadores de 8 bits.....	58
Figura 5.5: Frecuencia máxima de funcionamiento multiplicadores de 16 bits.....	58
Figura 5.6: Descenso de la frecuencia de funcionamiento con el nº de bits .....	59
Figura 5.7: Descenso de la frecuencia de funcionamiento con el nº de bits .....	59
Figura 5.8: Aumento del throughput por pipeline.....	60
Figura 5.9: Aumento del throughput por pipeline del multiplicador Carrysave .....	61
Figura 5.10: Aumento del número de recursos por pipeline .....	61
Figura 5.11: Potencia estimada del multiplicador Ripplecarry por Vivado.....	62
Figura 5.12: Potencia estimada del multiplicador Carrysave por Vivado.....	63
Figura 5.13: Potencia estimada del multiplicador Wallace por Vivado.....	64
Figura 5.14: Potencia estimada del multiplicador de Booth por Vivado .....	65
Figura 5.15: Potencia estimada del multiplicador Baugh-Wooley por Vivado .....	66
Figura 5.16: Potencia estimada del multiplicador Vivado HLS por Vivado: .....	67
Figura 5.18: Potencia total estimada multiplicadores sin signo .....	68
Figura 5.18: Potencia total estimada multiplicadores con signo.....	68
Figura 5.20: Potencia dinámica multiplicadores sin signo.....	68
Figura 5.20: Potencia dinámica estimada multiplicadores con signo .....	68
Figura 5.21: Potencia estática, dinámica y total.....	69
Figura 5.22: Consumo Carrysave de 16bits a 200MHz .....	69
Figura B.1: Pines placa PYNQ [4] .....	94

## Índice de Tablas

Tabla 5.1: Recursos utilizados por el multiplacador Ripplecarry de 4 bits.....	52
Tabla 5.2: Recursos utilizados por el multiplacador Ripplecarry de 8 bits.....	52
Tabla 5.3: Recursos utilizados por el multiplacador Ripplecarry de 16 bits.....	52
Tabla 5.4: Recursos utilizados por el multiplicador Carrysave de 4 bits .....	53
Tabla 5.5: Recursos utilizados por el multiplicador Carrysave de 8 bits .....	53
Tabla 5.6: Recursos utilizados por el multiplicador Carrysave de 16 bits .....	53
Tabla 5.7: Recursos utilizados por el multiplicador Wallace de 4 bits .....	53
Tabla 5.8: Recursos utilizados por el multiplicador Wallace de 8 bits .....	53
Tabla 5.9: Recursos utilizados por el multiplicador Wallace de 16 bits .....	54
Tabla 5.10: Recursos utilizados por el multiplicador Booth de 4 bits .....	54
Tabla 5.11: Recursos utilizados por el multiplicador Booth de 8 bits .....	54
Tabla 5.12: Recursos utilizados por el multiplicador Booth de 16 bits .....	54
Tabla 5.13: Recursos utilizados por el multiplicador Baugh-Wooley de 4 bits.....	54
Tabla 5.14: Recursos utilizados por el multiplicador Baugh-Wooley de 8 bits.....	55
Tabla 5.15: Recursos utilizados por el multiplicador Baugh-Wooley de 16 bits.....	55
Tabla 5.16: Recursos utilizados por el multiplicador de Vivado HLS de 4 bits .....	55
Tabla 5.17: Recursos utilizados por el multiplicador de Vivado HLS de 8 bits .....	55
Tabla 5.18: Recursos utilizados por el multiplicador de Vivado HLS de 16 bits .....	55
Tabla 5.19: Estimación de potencia para el multiplicador Ripplecarry de 4bits.....	62
Tabla 5.20: Estimación de potencia para el multiplicador Ripplecarry de 8bits.....	62
Tabla 5.21: Estimación de potencia para el multiplicador Ripplecarry de 16bits.....	62
Tabla 5.22: Estimación de potencia para el multiplicador Carrysave de 4 bits .....	63
Tabla 5.23: Estimación de potencia para el multiplicador Carrysave de 8bits .....	63
Tabla 5.24: Estimación de potencia para el multiplicador Carrysave de 16 bits .....	63
Tabla 5.25: Estimación de potencia para el multiplicador Wallace de 4bits .....	64
Tabla 5.26: Estimación de potencia para el multiplicador Wallace de 8 bits .....	64
Tabla 5.27: Estimación de potencia para el multiplicador Wallace de 16 bits .....	64
Tabla 5.28: Estimación de potencia para el multiplicador Booth de 4 bits.....	65
Tabla 5.29: Estimación de potencia para el multiplicador Booth de 8 bits.....	65
Tabla 5.30: Estimación de potencia para el multiplicador Booth de 16 bits.....	65
Tabla 5.31: Estimación de potencia para el multiplicador Baugh-Wooley de 4bits .....	66
Tabla 5.32: Estimación de potencia para el multiplicador Baugh-Wooley de 8 bits .....	66
Tabla 5.33: Estimación de potencia para el multiplicador Baugh-Wooley de 16bits .....	66
Tabla 5.34: Estimación de potencia para el multiplicador de Vivado HLS de 4bits .....	67
Tabla 5.35: Estimación de potencia para el multiplicador de Vivado HLS de 8bits .....	67
Tabla 5.36: Estimación de potencia para el multiplicador de Vivado HLS de 16 bits .....	67

## Resumen ejecutivo

Este trabajo de fin de grado tiene como finalidad, la implementación de distintas arquitecturas de **multiplicadores digitales** sobre una **FPGA**. Para ello, utilizamos la herramienta **Vivado** de Xilinx, con el propósito de comparar y analizar los distintos parámetros de diseño (**velocidad, área, consumo**) de cada uno de los multiplicadores, estudiando la relación entre cada uno de estos factores. Además, se utiliza la herramienta de diseño de alto nivel Vivado HLS, para elaborar un multiplicador digital, a partir de código C, que será analizado con el resto de los diseños VHDL. También, se analiza la técnica de diseño **segmentación**, para el aumento del número de muestras procesadas por unidad de tiempo del sistema y como esto afecta a nuestros diseños.

A su vez, se busca medir la precisión de la **estimación de consumo** de la herramienta Vivado de Xilinx, y ver y relacionar los distintos parámetros que afectan a esta estimación, comparándola con la realidad medida en el laboratorio.

Primero, se realiza el estudio de las principales fuentes de consumo en los circuitos digitales electrónicos, poniendo especial interés en las **FPGA**. Además, se estudian distintas técnicas de diseño de sistemas digitales. Posteriormente, se siguen los pasos necesarios para diseñar un sistema electrónico digital utilizando un **lenguaje de descripción de hardware**.

Se ha realizado el diseño de **cinco arquitecturas de multiplicadores**, escritas en lenguaje de descripción **VHDL**. Estos multiplicadores son descritos de forma estructurada, todos ellos son paralelos, tres de ellos sin signo y dos que multiplican en complemento a dos, de 4, 8 y 16 bits. Además, a los cinco diseños, se les implementará la técnica de segmentación, aumentando progresivamente el número de etapas. Por otro lado, se utiliza la herramienta de Xilinx, Vivado HLS, para síntesis de alto nivel, con el objetivo generar un multiplicador en VHDL, a partir de código C, y comparar el resultado con el resto de los multiplicadores.

El primer paso, consiste en simular y verificar el funcionamiento de los multiplicadores, utilizando un testbench adecuado para ello y el simulador integrado en la herramienta Vivado. Una vez comprobado que el fichero de configuración se genera correctamente, para probar que realmente funcionan los multiplicadores en el hardware, se utiliza la herramienta de Matlab, **Simulink**, y dentro de esta, la herramienta **FPGA-in-the-Loop (FIL)**. Esta herramienta nos permitirá tanto introducir datos a la FPGA, como recoger los resultados de la salida y comprobar si las multiplicaciones se realizan de manera correcta.

Una vez observado que el funcionamiento de cada descripción es el requerido, se sintetizan y se implementan estos multiplicadores sobre la placa **PYNQ**, que tiene en su interior la FPGA del fabricante Xilinx, ZYNQ XC7Z020 –CLG400C. Se miden los principales parámetros de diseño de cada uno de los circuitos, los recursos utilizados de la placa, la velocidad máxima de funcionamiento y la estimación de consumo, analizando las variaciones que se producen al



aumentar el número de bits, la frecuencia de funcionamiento o el número de etapas de segmentación.

Para estimar el consumo de los multiplicadores se utilizará la herramienta **Report Power** de Vivado, utilizando un fichero de actividad generado mediante simulación, que almacena la actividad de las señales del diseño. Esta simulación debe ser lo más parecida posible al funcionamiento real del circuito, para así, obtener una estimación precisa del consumo. Para ello, se utiliza un testbench con un Load Feedback Shift Register, que genera una secuencia de números pseudoaleatorios. Con esto, podremos comparar posteriormente las predicciones de esta herramienta, con las mediciones en el laboratorio sobre la FPGA.

Una vez realizados todos estos pasos, se implementarán las distintas arquitecturas sobre la FPGA, utilizando como se menciona anteriormente la herramienta FIL de Matlab. Mediante una fuente de números aleatorios, se introducen datos a las entradas del diseño y medimos el consumo con una placa diseñada previamente en el Centro de Electrónica Industrial, que se conecta entre la alimentación y la FPGA.

Por último, una vez hemos medido todos los datos, los analizamos para ver los resultados de los distintos multiplicadores, y compararlos entre ellos. Para ver si existen diferencias apreciables y cuáles son los que ofrecen mejores resultados, tanto en velocidad, como en área ocupada y consumo. A su vez, se analiza la dependencia entre los distintos parámetros de diseño y la relación entre la estimación de consumo y la medida de consumo real.

A partir del análisis de los datos, podemos ver, que el multiplicador con mejores características es el multiplicador de **Wallace**, ya que, ocupa menos área, alcanza mayor velocidad y consume menos que el resto de los diseños. Por otro lado, el multiplicador diseñado con la herramienta **Vivado HLS**, mejora en varios aspectos al multiplicador de Wallace. Por lo que se puede concluir, que Vivado HLS, es una alternativa útil para el diseño de operadores aritméticos eficientes. La técnica de diseño, **segmentación**, es muy eficiente si queremos aumentar el número de muestras procesadas por unidad de tiempo, con una penalización en el área ocupada y en la latencia. **El consumo** está fuertemente ligado al número de muestras procesadas por unidad de tiempo del sistema, y al número de bits de los operandos. **La estimación de consumo** realizado con Vivado presenta diferencias apreciables con los resultados obtenidos en el laboratorio, por lo que podemos concluir, que la herramienta FPGA-in-the-Loop no es óptima para medir el consumo real de nuestro diseño.

**Palabras clave:** Multiplicadores, FPGA, Vivado, HLS, velocidad, área, consumo, segmentación, VHDL, FPGA-in-the- Loop, PYNQ.

## Capítulo 1: Introducción y objetivos

El increíble desarrollo de la electrónica digital en las últimas décadas, y especialmente de la microelectrónica, ha hecho posible la implementación de sistemas digitales electrónicos cada vez más complejos en circuitos integrados cada vez más económicos. Una de las consecuencias del aumento de la complejidad, es el aumento de componentes integrados, y, por ende, el aumento de potencia que consumen, con el consiguiente aumento de temperatura y todas las consecuencias perjudiciales que ello conlleva en los dispositivos electrónicos. Esto resulta particularmente perjudicial en la electrónica portátil por la reducción de la autonomía de los dispositivos y de su vida útil que implica el aumento de consumo [15]. Así pues, en el mundo de la microelectrónica el tema del **consumo** se ha convertido en fundamental. Por lo que en este trabajo se les dedica especial interés a las técnicas orientadas a la estimación y medición del consumo.

**Los multiplicadores digitales** son uno de los bloques funcionales más utilizados en cualquier sistema electrónico digital, y son especialmente importantes en los sistemas digitales de proceso de datos (DSP). Por tanto, la eficiencia de estos es clave en multitud de sistemas. Una vía para mejorar las características de cualquier sistema electrónico digital, será mejorar las características de estos bloques funcionales.

El objetivo de este trabajo de fin de grado, es comparar y analizar los distintos parámetros de diseño (**velocidad, área, consumo**) de cinco arquitecturas de multiplicadores digitales diseñados en VHDL, estudiando la relación entre cada uno de estos factores y como la variación de cada uno de ellos afecta a los otros dos. Además, también se analiza la técnica de segmentación para el aumento del número de muestras procesadas por unidad de tiempo del sistema y como esto afecta a nuestros diseños.

### 1.1. Estado del arte

La actual **metodología de diseño** de los circuitos integrados consiste en un diseño descendente, que empieza con la descripción del comportamiento de los circuitos mediante lenguajes de descripción de hardware (HDL), simulación a nivel de comportamiento y utilización de herramientas de síntesis lógica [19]. La concentración de esfuerzos se realiza a nivel funcional y arquitectural, evaluando las distintas posibilidades antes de abordar el diseño detallado. Hoy en día, existen herramientas comerciales que permiten describir un sistema electrónico en alto nivel sin un conocimiento previo de su implementación, consiguiendo de forma automatizada y en función las restricciones impuestas (consumo, área, velocidad) una implementación del sistema a nivel lógico [15].

La tecnología actual de circuitos integrados permite realizar e integrar sistemas muy complejos, en muy poco espacio, son los llamados sistemas embebidos (SoC). El desarrollo de los sistemas

de fabricación ha hecho que las **FPGAs** hayan tenido una gran expansión en los últimos años, debido al abaratamiento de su producción y el incremento en sus prestaciones.

Hace años era complicado realizar diseños digitales complejos para volúmenes pequeños de producción, sin embargo, el desarrollo actual de las FPGAs, y de las herramientas de síntesis, permite realizar diseños de bajo coste, totalmente a medida. Esto ha hecho posible el diseño de circuitos digitales complejos para su utilización en pequeñas cantidades, reduciendo el ciclo de diseño, el tiempo al mercado y los costes de desarrollo.

Las FPGAs son empleadas en la industria para el desarrollo de circuitos integrados digitales y centros de investigación para la elaboración de prototipos y pequeñas series de ASIC. Además, las FPGAs, al contrario que los microprocesadores, son capaces de trabajar de forma paralela. Esto hace que sean muy utilizadas para el procesado digital de señal (DPS), y para realizar computación a alta velocidad. Los circuitos digitales configurables, en especial las FPGAs, han comenzado a jugar un papel muy importante en el diseño e implementación de procesadores digitales configurables de elevadas prestaciones. Como hemos contado más arriba, su flexibilidad y elevada potencia de cálculo las está convirtiendo en una pieza muy importante en el campo de la inteligencia artificial.

Como se menciona en la introducción, el consumo de los circuitos integrados es cada vez más importante. La utilización de una metodología de estimación y control de consumo de circuitos integrados resulta indispensable para la realización de un diseño eficiente, para poder aumentar la potencia de computación de estos y para ser competitivo en un mercado marcado por la globalización y una fuerte competencia.

Una nueva revolución, protagonizada por las herramientas de síntesis de alto nivel, es la capacidad de estas herramientas para traducir las funciones que se desean acelerar de los dispositivos SoC en código sintetizable en la FPGA. Estas herramientas de síntesis, como Vivado HLS, permiten generar una implementación para FPGA, a partir de una implementación en lenguaje de alto nivel para un procesador. Se evita así tener que realizar un diseño hardware adicional, en el que el desarrollador volvería a implementar el mismo algoritmo en lenguaje de descripción hardware, debiendo gestionar la lógica de la FPGA. Esto facilita la exploración del espacio de diseño de posibles soluciones.

## 1.2. Diseño de sistemas electrónicos digitales

### 1.2.1. Diseño digital de bajo consumo

Se realiza un análisis de las diferentes fuentes de consumo en circuitos digitales CMOS, y de diversas técnicas para minimizar el consumo. Posteriormente se analiza el consumo específico en las FPGAs.

#### Análisis de Consumo en circuitos CMOS

El consumo en cualquier puerta lógica puede dividirse en cuatro términos:

$$P_{\text{total}} = P_{\text{estática}} + P_{\text{fugas}} + P_{\text{dinámica}} + P_{\text{cortocircuito}}$$

- **Potencia disipada en estática**

$$P_{\text{estática}} = I_{\text{estática}} \times V_{\text{DD}}$$

Es la potencia consumida debido a la existencia, en condiciones estáticas (es decir, de no conmutación de las señales) de algún camino conductivo de baja impedancia entre  $V_{\text{DD}}$  y GND. En diseño de bajo consumo ha de evitarse la utilización de familias lógicas que tengan consumo en estática [15].

En la lógica CMOS complementaria, la corriente de consumo en estática es despreciable, ya que en condiciones estáticas nunca hay conexión entre la alimentación y el GND.

- **Minimización potencia estática**

El parámetro de diseño que afecta de forma directa al consumo estático es la elección de la tecnología. En diseño de bajo consumo, lo lógico es utilizar familias que en estática no tenga caminos de baja impedancia entre alimentación y tierra, como es el caso de la tecnología CMOS [15].

- **Potencia disipada por corrientes de fuga**

$$P_{\text{fuga}} = I_{\text{fuga}} \times V_{\text{DD}}$$

Es la potencia consumida debido a corrientes de fuga (leakage) en los transistores. La corriente de fuga tiene dos componentes principales:  $I_{\text{fuga}} = I_{\text{pn}} + I_{\text{subumbral}}$

Por un lado,  $I_{\text{pn}}$ , que son las corrientes de las uniones pn polarizadas inversamente. En un CMOS inversor convencional con tecnología de pozo n, se forman uniones pn parásitas. Para que estos diodos parásitos no entren en directa, el pozo n se polariza a  $V_{\text{DD}}$  y el sustrato p a GND, pero, aun así, existe la corriente de saturación en inversa que caracteriza a toda unión pn [15].

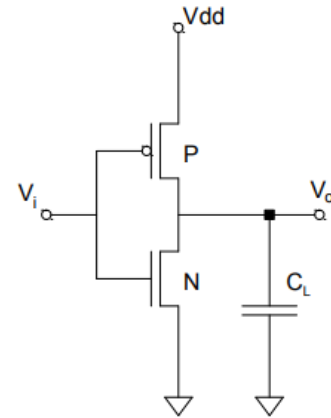


Figura 1.1: Inversor CMOS [13]

**$I_{pn} = I_s \times (e^{V_D/V_T} - 1)$**  donde:

- $I_s$ = corriente de saturación inversa
- $V_D$ = Tensión aplicada en el diodo
- $V_T$ = tensión térmica,  $Kt/q$ , que a temperatura ambiente es aproximadamente 25,6 mV.

Para un circuito integrado con millones de puertas, el consumo total debido a las corrientes inversas de saturación es despreciable, por lo tanto, en tecnologías actuales este consumo es despreciable [15].

En cuanto al consumo debido a la corriente subumbral, un inversor CMOS con un nivel lógico bajo en la entrada, continúa consumiendo una cierta potencia debido a la corriente de conducción subumbral del transistor NMOS que está en corte [15].

**$P_{subumbral} = I_{subumbral} \times V_{DD}$** , el consumo debido a estas corrientes puede llegar ser significativo.

#### ◦ **Minimización de la potencia debido a corrientes de fuga**

En la lógica estática CMOS existen soluciones que pasan por realizar un control dinámico inteligente de la tensión umbral  $V_t$ . Dicho control dinámico se realiza teniendo accesible el sustrato de los transistores (PMOS a  $V_{DD}$  y NMOS a GND), y utilizando el efecto sustrato. Un aumento de la tensión de sustrato provoca un aumento de la tensión umbral, con lo que disminuye la corriente subumbral. El problema es que un aumento de la tensión umbral provoca un aumento del retardo en la puerta. Por ello el control debe ser inteligente, se debe realizar una partición en bloques, y solo aumentar la tensión umbral cuando un determinado bloque no se vaya a utilizar o no requiera gran velocidad [15].

También debe tenerse en cuenta, que las corrientes de fuga tienen una dependencia muy fuerte con la temperatura, de forma que un incremento en la temperatura provoca un aumento considerable del consumo por corrientes de fuga. Una forma de minimizar este consumo puede ser trabajar con sistemas electrónicos con una refrigeración especial [15].

#### • **Potencia disipada por cortocircuito**

Las señales reales requieren de un cierto tiempo para realizar la conmutación, denominado usualmente tiempo de conmutación. Considerando un inversor CMOS y una tensión de alimentación  $V_{DD} > V_{TN} + |V_{TP}|$ , habrá un periodo donde ambos transistores se encuentran conduciendo. Entonces, aparece una corriente que va desde  $V_{DD}$  hasta GND durante la transición de las señales, denominada corriente de cortocircuito. La potencia de cortocircuito es proporcional a la anchura de los transistores, y al tiempo de subida de la entrada. Para

minimizar el consumo de cortocircuito se deben tener unos tiempos de entrada y de salida similares [15].

- **Minimización potencia de cortocircuito**

La lógica CMOS estática tiene corriente de cortocircuito. Así, en puertas convencionales el consumo de cortocircuito representa alrededor del 10% del consumo total. Para disminuir esta potencia se puede controlar separadamente las puertas de transistores NMOS y PMOS y generar adecuadamente retardos entre dichas señales, de forma que se disminuye el tiempo en que ambos transistores están activos de forma simultánea [15].

Otra opción es disminuir la tensión de alimentación. Con esto la corriente de cortocircuito será nula, puesto que nunca se encontrarán activos al mismo tiempo los transistores NMOS y PMOS. Sin embargo, esto se consigue a costa de mucha penalización en la velocidad, esto se podrá hacer en situaciones donde no se requiere una velocidad de proceso importante y se requiera un bajo consumo [15].

- **Potencia dinámica disipada**

El consumo dinámico es debido a las conmutaciones de los nodos del circuito. Para cambiar el valor de la tensión de cualquier nodo se requieren desplazamientos de carga a través de un medio disipativo, (transistores) lo que consume energía. La potencia dinámica consumida en inversor CMOS viene determinada por C que es la capacidad de la carga,  $\alpha$  la actividad del nodo y  $f_{CLK}$ , es la frecuencia de trabajo del circuito [15].

$$P_{dinámica} = \sum_i^{nodos} (\alpha_i \times C_i) \times VDD^2 \times f_{CLK}$$

Este consumo dinámico viene provocado por la transición de las señales. Existen dos tipos de transiciones: Las transiciones de señales funcionales, necesarias para el funcionamiento del sistema, y las transiciones innecesarias (glitching-activity). Este tipo de señales, provocadas por la diferencia de retardo entre los canales de entrada a las puertas lógicas, puede provocar importantes consumos dinámicos [1].

- **Minimización de la potencia dinámica**

Es el término de consumo más importante y al que se dedican más esfuerzos a la hora de minimizar el consumo en circuitos digitales CMOS. Las soluciones consisten en disminuir la tensión de alimentación, la capacidad asociada a los nodos o la actividad del circuito [15].

- a. La tensión de alimentación

La energía necesaria para conmutar un nodo depende de forma cuadrática de la alimentación. Además, esta reducción de la alimentación provoca un menor consumo de fuga y de

cortocircuito, sin embargo, conlleva una disminución del tiempo de respuesta. Para disminuir este aumento de retardo suele disminuirse también las tensiones umbral ( $V_{tn}$  y  $|V_{tp}|$ ), para evitar una disminución excesiva de la tensión de control de puerta de los transistores,  $VDD - V_t$ , lo que provocaría un aumento del consumo por corriente subumbral [15].

El aumento de retardo puede solventarse mediante dos técnicas de diseño. La primera es aumentar la capacidad de procesamiento del sistema mediante **paralelismo**, a expensas de aumentar el área, el precio y el consumo por el aumento de las capacidades parásitas. Sin embargo, el aumento en la capacidad de procesamiento nos permitirá reducir la tensión de alimentación, y, por tanto, obtendremos un consumo final menor [15].

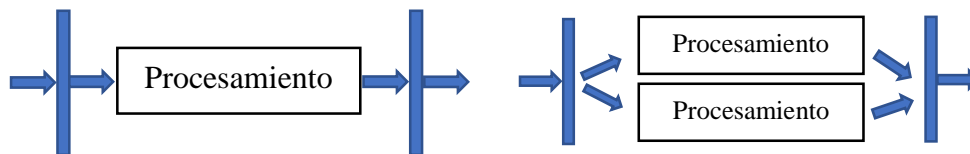


Figura 1.2: Técnica de paralelización

Otra posibilidad para compensar el retardo, es utilizando la técnica de **segmentación**, que consiste en añadir registros en etapas intermedias para aumentar la capacidad de procesamiento del sistema. Esta solución provoca un aumento de área por los registros añadidos, y un aumento en la **latencia** del sistema. Evidentemente existe la posibilidad de combinar ambas técnicas de diseño: paralelismo y pipelining [15].

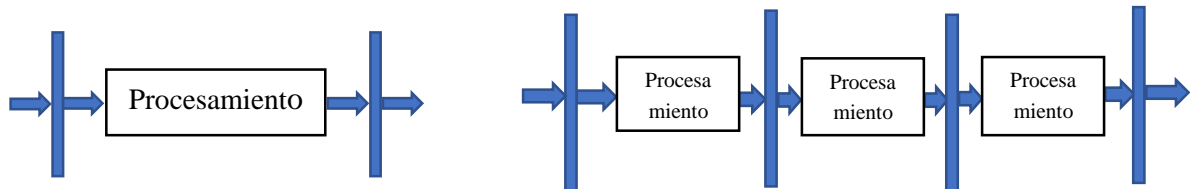


Figura 1.3: Técnica de segmentación

#### b. La actividad del circuito

Si no hay conmutaciones, la potencia dinámica es nula, por lo que una opción para reducir el consumo de un circuito integrado será disminuir la actividad. Un primer factor clave es la **elección del algoritmo**, así como, la codificación a utilizar, para intentar conmutar el menor número de veces [15].

A nivel de arquitectura, existen diversas opciones para reducir la actividad del circuito. El factor más importante consiste en reducir el número de transiciones innecesarias (**glitching activity**). El número de transiciones indeseadas es proporcional al cuadrado de la profundidad del bloque. Para reducir esta profundidad se pueden utilizar las dos técnicas de diseño mencionadas anteriormente, **paralelismo** y **segmentación**. Otra técnica a nivel de arquitectura, consiste en la deshabilitación total o parcial de módulos que en ese momento no vayan a utilizarse. La inhabilitación de la señal de reloj de un determinado bloque se denomina **clock gating**. Como

alternativa a la deshabilitación del reloj, se puede utilizar, sobre todo para bloques dinámicos, la disminución de la frecuencia [15].

c. La capacidad a conmutar

Los niveles afectados son el físico y el de la tecnología. Para reducir esta capacidad deben disminuir los recursos lógicos y los recursos de interconexión. Las herramientas CAD (como VIVADO) realizan la colocación y conexión minimizando la longitud de las interconexiones, para reducir la capacidad de cada línea [15].

Como mejor opción para el diseño de bajo consumo genérico está la lógica CMOS, debido a su gran tolerancia a utilizar tensiones de alimentación muy baja, transistores de dimensiones mínimas y la posibilidad de inhabilitar el reloj [15].

### **1.3. Circuitos integrados configurables**

Son aquellos en los que se puede modificar las conexiones entre los elementos que forman su sistema físico, para constituir una determinada estructura de memoria, por lo que tienen una estructura variable. La modificación de la función que realizan se puede hacer, utilizando una parte de los elementos que los componen, cambiando la interconexión entre ellos o ambas cosas simultáneamente [8].

La configuración del circuito se realiza a través de determinados terminales externos o por medio de los recursos lógicos, denominados recursos de configuración, que actúan sobre los dispositivos electrónicos programables situados en el circuito.

El desarrollo de los circuitos digitales configurables se realizó a partir de dos tecnologías diferentes, lo que dio lugar a dos tipos de circuitos digitales que se distinguen por su organización (forma en que se distribuyen los elementos que lo constituyen y las interconexiones entre los mismo). Estos son los Dispositivos lógicos programables (*PLD*) con recursos de interconexión concentrados o de organización matricial, y los Conjuntos configurables de puertas (*FPGA*) con recursos de interconexión distribuidos [8]. En este trabajo nos centraremos en los dispositivos *FPGAs*, puesto que los *PLDs*, al contrario que las *FPGAs*, se utilizan cada vez menos.

#### **1.3.1. *FPGAs***

Las *FPGAs* están formadas por recursos lógicos denominados bloques lógicos configurables (*CLB*), a través de dispositivos electrónicos cuyo estado de conducción establecen la función que realizan. La interconexión entre bloques se realiza a base de multiplexores que conectan un conjunto de bloques con otro. En función del valor de selección de los multiplexores, se define la conexión del sistema global, y por tanto su definición. El conjunto de valores de selección se guarda en una memoria y cambiando el valor de la memoria, se cambia la funcionalidad [8].



Este tipo de dispositivo está a medio camino entre los circuitos de propósito específico (ASICs) y los procesadores de propósito general en prestaciones y consumo de potencia. Su principal ventaja es que son reprogramables, por lo que proporcionan una gran flexibilidad de diseño, por lo que los costes y el tiempo de desarrollo son considerablemente inferiores. La gran capacidad que tienen para trabajar en paralelo, unido a la gran tolerancia a errores y una fiabilidad alta, lo que les convierte en una gran alternativa frente a los microprocesadores para algunas tareas específicas [8].

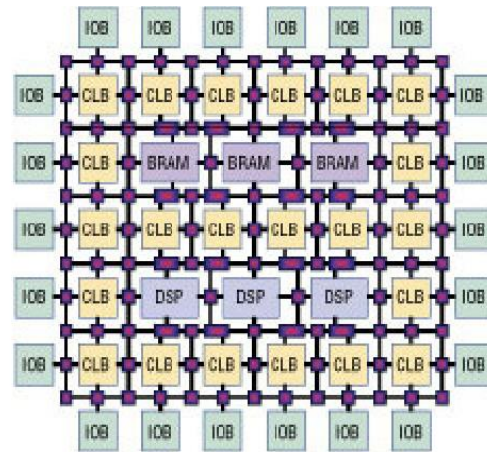


Figura 1.4: Arquitectura de un FPGA de Xilinx [2]

Los principales conceptos a tener en cuenta a la hora de analizar un circuito FPGA son:

### 1. Bloques lógicos configurables (CLB)

Los CLB de una FPGA son los principales recursos lógicos para implementar diferentes circuitos combinacionales, que ejecutan operaciones aritméticas y lógicas, así como sistemas secuenciales síncronos que constituyen unidades de control. En la serie 7 de FPGAs de Xilinx los CLBs están dispuestos en columnas. Cada CLB está constituido por varios Slices, dos en el caso de la serie 7 de Xilinx, y cada Slice está formado por cuatro LUTs (Look-up tables) de seis entradas. Cada LUT puede ser configurado con seis entradas y una salida, o dos LUTs de cinco entradas con salidas separadas, pero que comparten dirección o entradas lógicas. Además, aproximadamente dos tercios de los slices son slices lógicos, y el resto, pueden utilizar sus LUTs para funcionar como RAM distribuida de 64 bits o registros de desplazamiento de 32 o 16 bits [28]. En la imagen inferior podemos ver un ejemplo de un CLB de la serie 7 de Xilinx:

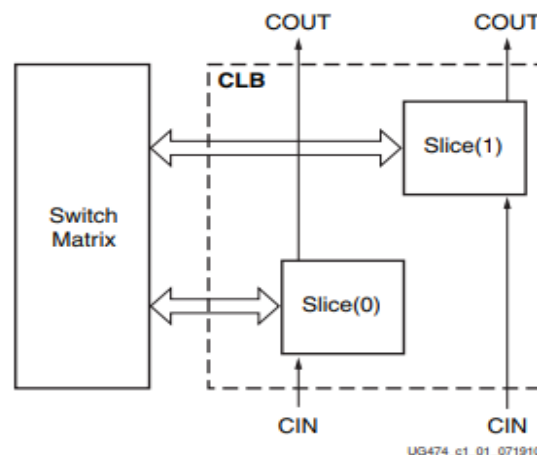


Figura 1.5: Bloque lógico configurable, serie 7 de Xilinx [28]

## 2. Canales de interconexión

Se denomina canales de interconexión (routing channels) a las líneas que permiten transmitir las señales entre los diferentes bloques lógicos. La flexibilidad de una FPGA depende en gran medida de la capacidad de configurar esta red de conexión. Esta red consiste en una serie de canales dispuestos vertical y horizontalmente, conectados entre sí mediante switch boxes (SB) y conectados a los bloques de configuración a través de los bloques de conexión [8].

## 3. Bloques lógicos de entrada/salida

Son bloques lógicos configurables que tiene como objetivo adaptar las señales internas del circuito FPGA a las exigencias de los sistemas externos conectados a sus terminales. Controlan el flujo de datos entre los pines de entrada/salida y la lógica interna del dispositivo [8].

## 4. Block RAM (BRAM)

Son bloques dedicados de memoria utilizados para almacenar datos, para aplicaciones que requieren acceso a memoria, ya sea para escritura y lectura (tipo RAM) o sólo lectura (tipo ROM). En la serie 7 de Xilinx cada bloque de RAM puede almacenar 36 Kbits de datos, que pueden ser utilizados como bloques independientes de 18 Kbits. Las BRAM son bloques configurables de acuerdo a las necesidades del diseño, es decir el mismo bloque puede ser configurado para que funcione como RAM, ROM, FIFO (First Input First Output), convertidor de ancho de palabra, buffers circulares y registros de desplazamientos. A su vez, cada una de estas configuraciones soporta diferentes anchos de la palabra de datos y diferentes tamaños del bus de direcciones [23].

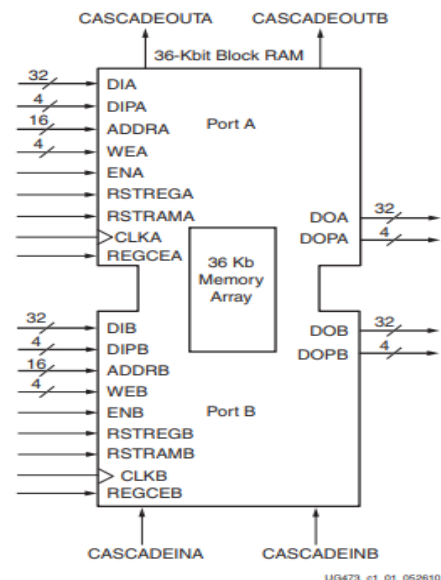


Figura 1.6: True Dual-Port Data Flows for a RAMB36, serie 7 de Xilinx [23]

## 5. Digital Clock Manager (DCM)

Las FPGA tienen unos bloques de lógica dedicados exclusivamente a funciones de control y generación de señales de reloj. En las FPGAs de Xilinx, a estos bloques genéricamente se los llama Digital Clock Managers (DCMs). Se trata de un componente electrónico capaz de manipular la señal de reloj de la FPGA, y que tiene diversas funciones como multiplicar o dividir la frecuencia de reloj a partir de un oscilador con una frecuencia determinada, generar clocks totalmente independientes funcionando al mismo tiempo o mantener la señal de reloj estable de posibles perturbaciones [21].

## 6. Bloques DSP (Procesado Digital de señal)

Como se menciona en el capítulo 1, las FPGAs son muy eficientes a la hora de realizar tareas de DSP. Esto es consecuencia, en gran medida, de los bloques DSP, que son elementos lógicos configurables dedicados a la multiplicación y a la suma. La serie 7 de FPGAs de Xilinx está compuesta por varios de estos DSP slices, que combinan gran velocidad de funcionamiento, con un tamaño reducido. Además, los DSP slices mejoran la velocidad y eficiencia de muchas tareas, más allá del procesamiento digital de señal, como, por ejemplo, generar direcciones de memoria [30].

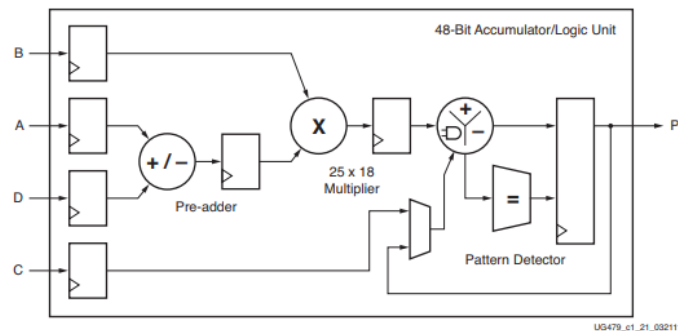


Figura 1.7: Basic DPS48E1, serie 7 de Xilinx [30]

### 1.3.2. Consumo en FPGAs

Ya se ha explicado anteriormente el consumo de los circuitos formados por transistores CMOS. El consumo en una FPGA se produce de manera similar, si bien, se deben tener algunos factores en cuenta.

El consumo de las FPGAs viene dado por dos componentes principales:

**1. Potencia estática:** Una FPGA puede ser considerada como una red de millones de transistores modelados como pequeños condensadores, estos condensadores consumen una pequeña cantidad de corriente de fuga cuando trabajan. Este consumo se puede medir programando un Bitstream vacío sobre el dispositivo. Depende del proceso, el voltaje y la temperatura y es difícil de controlar por el usuario [25]. Debe tenerse en cuenta que, aunque se denomine consumo estático, se corresponde con la potencia de fuga consumida por los circuitos CMOS mencionada previamente. A medida que la tecnología avanza, y el tamaño de los transistores CMOS se hace más pequeño, este consumo se vuelve más relevante, pudiendo llegar a ser mayor que el consumo dinámico [1].

Las FPGAs de RAM estática almacenan su programación en SRAM (Static RAM). Esto trae como consecuencia dos consumos que se deben tener en cuenta. Uno es el consumo transitorio inicial debido a la programación del chip, y el otro, es el consumo de todas esas celdas SRAM [13]. Estas celdas consumen energía en las FPGAs SRAM por el refresco de la memoria.

## 2.Potencia dinámica:

Es la potencia consumida por el diseño del usuario, debido al modelo de entrada de datos y la actividad interna de los nodos. Esta potencia es instantánea y varía cada ciclo de reloj. Depende de los niveles de voltaje, la lógica utilizada y la cantidad de recursos de interconexión utilizados, por lo que lógicamente dependerá del circuito digital implementado. Desde el punto de vista de los multiplicadores, dependerá de la arquitectura utilizada. Incluye la corriente estática de las terminaciones I/O, clock managers y otros circuitos que necesitan potencia mientras son utilizados [25]. Esta potencia dinámica, a su vez, puede separarse en la potencia consumida por la actividad del circuito y la potencia de cortocircuito.

**Consumo dinámico** =  $(\alpha \times C) \times VDD \times f_{CLK}$ , donde  $C$  es la capacidad de la carga,  $\alpha$  la actividad del circuito y  $f_{CLK}$ , es la frecuencia de trabajo del circuito [25]. Esta es la fórmula de consumo dinámico que se presentó en el capítulo de consumo en circuitos CMOS y que sigue siendo válida para las FPGAs.

Una particularidad de las FPGAs es el uso intensivo de los transistores de paso, tanto en las interconexiones programables como en la lógica. Las ecuaciones de consumo de potencia en un transistor de paso difieren ligeramente de las del inversor CMOS [13].

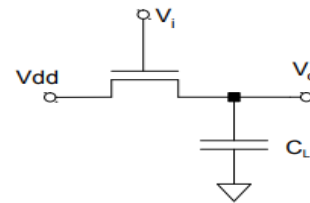


Figura 1.8: Transistor de paso [13]


Otro aspecto a tener en cuenta en cuanto al consumo de las FPGAs, es la distribución de relojes dentro del chip, y los bloques para generar relojes. Estos bloques, que incluyen PLLs o DLLs, a veces cuentan con fuentes de alimentación separadas. En cuanto a la distribución de reloj, se hace mediante líneas especiales dedicadas, y puede ser una parte muy importante del consumo total. Entre el 4 y el 22% [13].

Los bloques hardware internos de las FPGAs, como los bloques de memorias RAM o los bloques DSP, consumen potencia del mismo modo que un circuito ASIC. Las celdas de entrada-salida de una FPGA se ajustan a la ecuación de potencia dinámica dependiente de la capacidad externa de carga [13].

## Capítulo 2: Herramientas y Lenguajes para el diseño de sistemas Digitales

### 2.1. Lenguajes de descripción de hardware (HDL)

Los lenguajes de descripción de hardware son lenguajes de alto nivel, con sintaxis y semántica definida para facilitar el modelado y descripción de circuitos electrónicos, pudiéndose realizar estas descripciones a distintos niveles de abstracción, precisión y estilos. Nacen debido a la creciente complejidad de los diseños. Pueden emplearse para modelar el comportamiento de un componente de cara a su simulación o para describir el diseño de un circuito para su posterior implementación. Los lenguajes de descripción de hardware (HDL) proporcionan soporte a las tareas de diseño, comunicación, documentación y mantenimiento. Proporcionan independencia de metodologías, de herramientas y tecnología, y tiene la ventaja de que los diseños se pueden reutilizar [19].

- Niveles de abstracción: - Funcional - Arquitectural - Lógico
  - Estilos de descripción: - Algorítmico – Flujo de datos - Estructural
- 

Aunque se puede ver la correspondencia de los estilos de descripción con los distintos niveles de abstracción, en VHDL conviven descripciones mixtas-multinivel.

#### 2.1.1. VHDL

El VHDL (Very high speed integrated circuits Hardware Description Language) es un lenguaje de descripción de hardware, por lo que posee las características fundamentales comentadas anteriormente.

VHDL fue diseñado originariamente por el Departamento de defensa de los Estados Unidos como una forma de documentar las diversas especificaciones y el comportamiento de los dispositivos ASIC. Con la posterior posibilidad de simular dichos dispositivos, comenzaron a crearse simuladores que pudieran llevar a cabo esta tarea leyendo archivos VHDL. El paso siguiente fue el desarrollo de software capaz de sintetizar las descripciones generadas y obtener una salida apta para su posterior implementación, ya sea en ASIC como en dispositivos CPLD y FPGA. VHDL fue estandarizado por primera vez por la IEEE en 1987 [19].

#### Tres características fundamentales que incorpora VHDL para facilitar la descripción de hardware

1. **Modelo de estructura:** Cualquier sistema electrónico puede dividirse en subsistemas. Por ello VHDL incorpora el concepto de estructura, esto nos permite realizar un sistema digital a partir de la referencia a las distintas partes que lo forman y las conexiones entre dichas partes. Para poder utilizar elementos ya definidos en VHDL en descripciones

estructurales, VHDL incorpora el concepto de componente. Cualquier elemento puede ser usado como un componente de otro diseño, para ello solo es necesario hacer referencia al elemento a utilizar y conectar los puertos de su interfaz a los puertos necesarios para realizar el nuevo diseño [19].

2. **Modelo de concurrencia:** El hardware es concurrente, el elemento básico que ofrece VHDL para modelar el paralelismo es el proceso. Un proceso describe un comportamiento y el código que contiene se ejecuta de forma secuencial. Todos los procesos en VHDL se ejecutan de forma paralela [19].
3. **Modelo de tiempo:** Una de las utilidades del modelado en VHDL del hardware es poder simular su comportamiento a lo largo del tiempo. La simulación de un modelo VHDL es una simulación dirigida por eventos. La simulación VHDL abstrae el comportamiento real del hardware, implementando el mecanismo de estímulo-respuesta (componentes funcionales reaccionan a la actividad en sus entradas produciendo un cambio en su salida). En la primera etapa las señales actualizan su valor. En la segunda etapa los procesos que se activan se ejecutan hasta que se suspenden, entonces el tiempo de simulación avanza hacia el siguiente instante de tiempo en el que haya un evento programado y se repiten los dos pasos. Este mecanismo se implementa para permitir la simulación de hardware usando máquinas secuenciales [19].

Estas características, junto a las propiedades antes mencionadas, hacen del VHDL un lenguaje potente y flexible.

## 2.2. Xilinx Vivado Design Tools

Xilinx es una empresa americana, líder en el mercado mundial de las FPGAs. Es conocida, además, por ser la creadora de las FPGAs. Fue fundada en 1984, y es el fabricante de la FPGA que vamos a utilizar para la implementación de los circuitos digitales. También desarrolla el entorno de desarrollo Vivado que vamos a utilizar para el diseño de los multiplicadores.

### 2.2.1. Vivado Design Suite

Vivado Design Suite es un software producido por Xilinx para la síntesis de alto nivel, la implementación y simulación de diseños HDL. Este es el software que vamos a utilizar para el diseño de los multiplicadores, para la simulación, síntesis e implementación. Además, Vivado es una herramienta potentísima que nos permitirá a partir de especificar unos parámetros de diseño, estimar la potencia que consumirán nuestros diseños.

### 2.2.2. Vivado HLS

Como se menciona en el capítulo 1, el desarrollo de herramientas de síntesis de alto nivel está produciendo una revolución en el mundo de las FPGAs. Una de estas herramientas, es Vivado HLS (High level Synthesis), es una herramienta de Xilinx, que transforma código C en un diseño HDL, que puede ser implementado en una FPGA de Xilinx.

### 2.2.3. PYNQ-Z1

El dispositivo utilizado a lo largo de todo el trabajo, para la implementación de los distintos diseños, es la placa PYNQ-Z1, de DIGILENT, que tiene en su interior el FPGA, del fabricante Xilinx, ZYNQ XC7Z020 –CLG400C. Tiene como característica particular, que, a diferencia de otras FPGA de Xilinx, este dispositivo está diseñado alrededor de un microprocesador (integrado en el mismo chip), que actúa como maestro de la lógica programable y todos los periféricos del sistema.

#### Características

- Procesador 650Mhz dual-core Cortex-A9
- 512 MB DDR3
- Conectividad USB, Ethernet, de vídeo y de audio
- Arduino Shield and PMOD connectors para añadir dispositivos hardware
- Programable desde JTAG, QUAD-SPI flash y tarjeta microSD

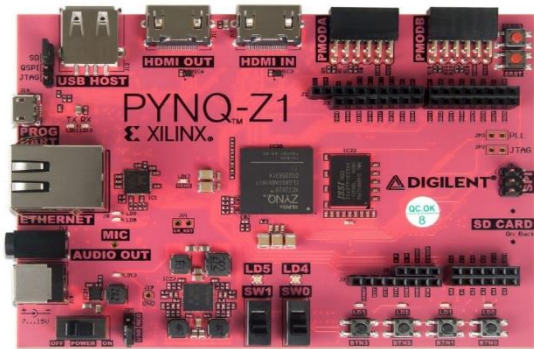


Figura 2.1: Placa PYNQ [5]

#### Especificaciones FPGA

<b>Logic slices</b> 13.300	<b>6-input LUTS</b> 53.200	<b>DSP slices</b> 220
<b>Flip-Flops</b> 106.400	<b>Block RAM</b> 630 KB	

#### Power supply

Supply	Circuits	Current(max/typical)
<b>3.3V</b>	FPGA I/O, USB ports, Clocks, Ethernet, flash, HDMI	1.6A/0.1A to 1.5A
<b>1.0V</b>	FPGA, Ethernet core	2.6A/0.2A to 2.1A
<b>1.5V</b>	DDR3	1.8A/0.1A to 1.2A
<b>1.8V</b>	FPGA auxiliary, Ethernet I/O, USB controller	1.8A/0.1A to 0.6A

## Capítulo 3: Multiplicadores digitales

Uno de los principales bloques funcionales de cualquier sistema digital complejo, son **los multiplicadores digitales**. Es un circuito digital capaz de multiplicar dos números  $M$  y  $n$  obteniendo un producto  $P$ . Los micros de altas prestaciones disponen de multiplicadores implementados por hardware para aumentar la velocidad de las operaciones aritméticas, además, son especialmente importantes en los sistemas digitales de proceso de señales (DSP). Una vía para mejorar cualquier sistema electrónico digital será elegir la arquitectura con mejores características y que mejor se adecúe a los requisitos establecidos.

La elección del algoritmo es un aspecto clave en las prestaciones de un circuito. En los siguientes capítulos se presentarán las distintas arquitecturas de multiplicadores, explicando su funcionamiento, los bloques funcionales que lo forman y los resultados obtenidos en función de las distintas especificaciones de diseño. El código VHDL de cada diseño se encuentra en el anexo A.

### Algoritmo desplazamiento y acumulación

1. Se obtienen los productos parciales recorriendo el multiplicador bit a bit de derecha a izquierda:

- Si el bit es un 0, el producto parcial es 0
- Si el bit es un 1, el producto parcial tiene el valor del multiplicando

2. Cada producto parcial debe estar desplazado una posición a la izquierda respecto al producto parcial anterior

3. Una vez calculados todos los productos parciales, se suman para obtener el producto

$$\begin{array}{r} M3 \ M2 \ M1 \ M0 \\ N3 \ N2 \ N1 \ N0 \\ \hline M3*n0 \ M2*n0 \ M1*n0 \ M0*n0 \\ M3*n1 \ M2*n1 \ M1*n1 \ M0*n1 \\ M3*n2 \ M2*n2 \ M1*n2 \ M0*n2 \\ M3*n3 \ M2*n3 \ M1*n3 \ M0*n3 \\ \hline P7 \ P6 \ P5 \ P4 \ P3 \ P2 \ P1 \end{array}$$

La multiplicación está formada por la suma de tantas filas como bits tenga el multiplicador. El retardo, en el peor de los casos, será el de la suma de  $m$  bits más la suma final de los dos productos parciales de  $n$  bits.



Los multiplicadores paralelos son arreglos iterativos, los cuales utilizan estructuras formadas por sumadores Ripplecarry o Carrysave. El funcionamiento de estos multiplicadores se basa en el algoritmo de desplazamiento y acumulación. Son más rápidos que los multiplicadores secuenciales y están formado por una matriz de lógica combinatoria que, a partir de todas las combinaciones posibles de las entradas, genera sus productos a la salida. También ocupan un área mayor, es decir necesitan más lógica y tienen un consumo mayor.

### 3.1. Multiplicadores sin signo

#### 3.1.1. Multiplicador Ripplecarry

El multiplicador Ripplecarry está basado en un arreglo de sumadores de acarreo propagado. Es una primera aproximación para implementar el algoritmo de sumas sucesivas y desplazamientos. Este tipo de multiplicador tiene la característica de transferir la propagación del acarreo a la siguiente suma parcial, hasta que terminen los productos parciales correspondientes a esta fila. En ese instante, el acarreo generado se propaga al último producto de la siguiente fila de productos parciales, y así sucesivamente [14]. El bloque estructural básico es un sumador completo (Full-adder), que se utiliza para construir un sumador Ripplecarry y éste, a su vez, para construir el multiplicador Ripplecarry. El código VHDL correspondiente a esta arquitectura se encuentra en el anexo A.3.

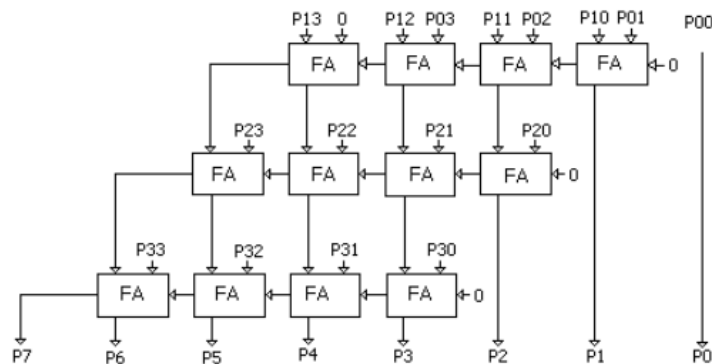


Figura 3.1: Multiplicador Ripplecarry de 4 bits [14]

#### 3.1.2. Multiplicador Carrysave/de Braun

El multiplicador Carrysave está basado en el arreglo de sumadores con acarreo salvado. Consiste en una implementación distinta del algoritmo de sumas sucesivas y desplazamientos. La idea es romper la cadena de acarreo propagado del sumador Ripplecarry, para disminuir el retardo de cada suma, lo cual permite acelerar la multiplicación. El multiplicador tiene la característica de permitir salvar el acarreo generado en las sumas parciales y transferirlo como acarreo de entrada a la siguiente suma parcial. Este multiplicador es también conocido como multiplicador de Braun. Su elemento básico también es un sumador completo, pero estos se utilizan para formar un sumador Carrysave. En la última fila del multiplicador deben sumarse

los productos parciales con un sumador Ripplecarry [14]. El código VHDL correspondiente a esta arquitectura se encuentra en el anexo A.4.

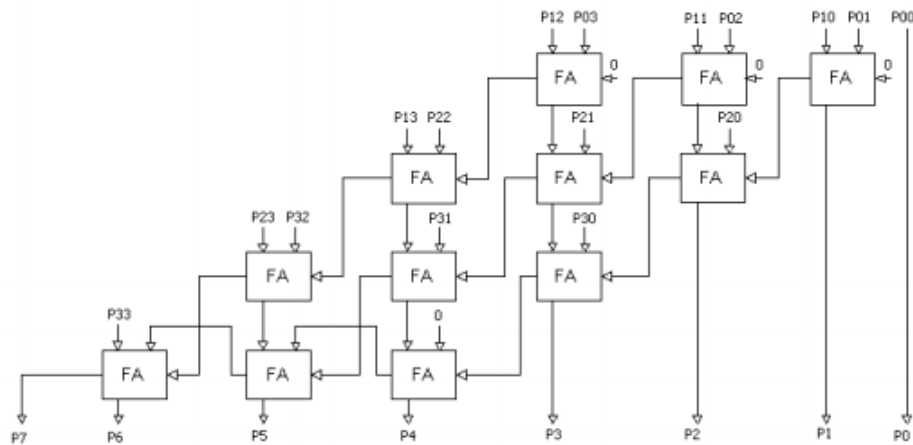


Figura 3.2: Multiplicador Carrysave de 4 bits [14]

### 3.1.3. Multiplicador de Wallace

El multiplicador de Wallace es una variante del algoritmo de sumas sucesivas y desplazamientos. Esta variante consiste en agrupar los productos parciales de cada columna, con el objetivo de reducir el número de recursos utilizados y aumentar la velocidad de procesamiento. Se utilizan sumadores Carrysave, en los que cada bloque sumador completo con sus tres entradas, recibiendo términos productos y genera un término suma que se adiciona con otro término suma [14]. El código VHDL correspondiente a esta arquitectura se encuentra en el anexo A.5.

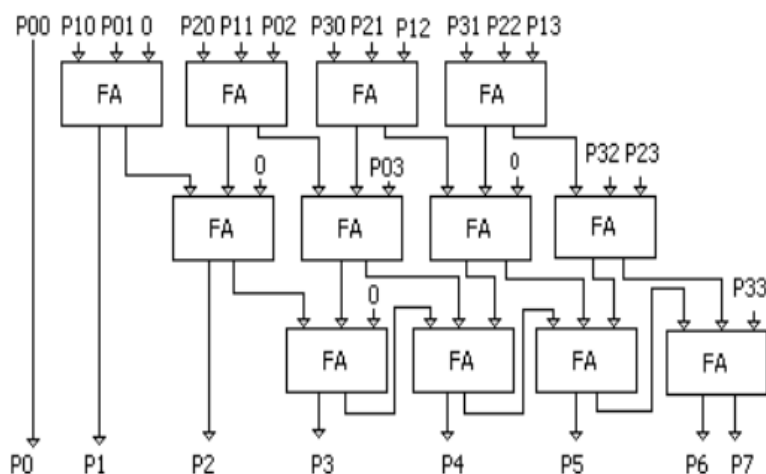


Figura 3.3: Multiplicador de Wallace de 4 bits [14]

## 3.2. Multiplicadores con signo(C-2)

### 3.2.1. Multiplicador Booth paralelo

El algoritmo de Booth representa dos ventajas importantes: primero, unifica los multiplicadores tanto para números positivos como negativos de n bits. Segundo, logra eficiencia respecto al número de productos parciales generados cuando el multiplicador tiene bloques grandes de unos [12]. El código VHDL correspondiente a esta arquitectura se encuentra en el anexo A.8.

El algoritmo de Booth está basado en la observación de que podemos calcular un producto a partir de sumas y restas. Por tanto, su estructura está formada por una celda de control para adicción/sustracción/desplazamiento (CASS) [12].

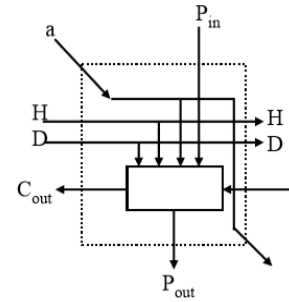


Figura 3.4: Celda CASS [12]

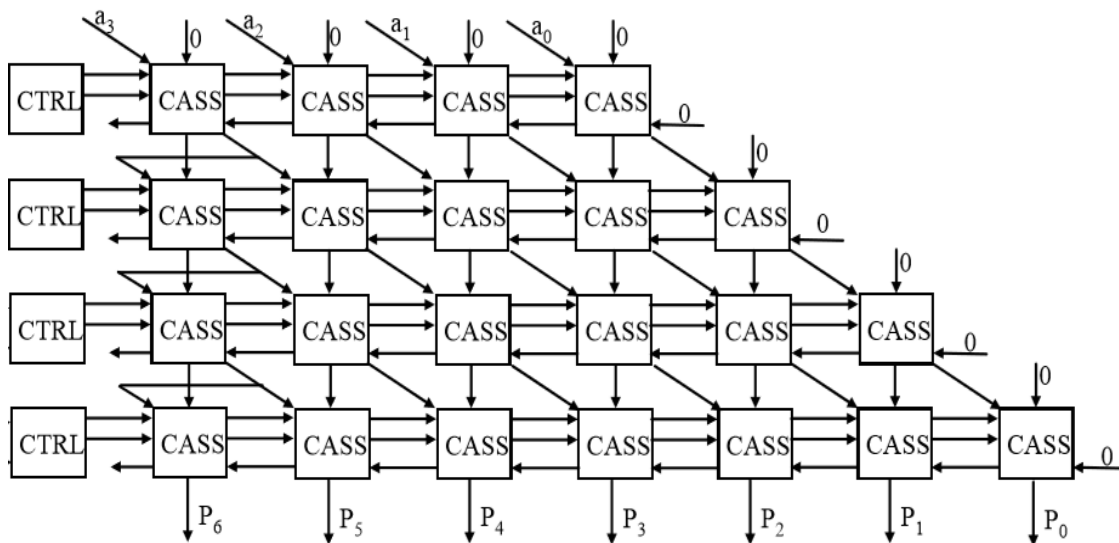
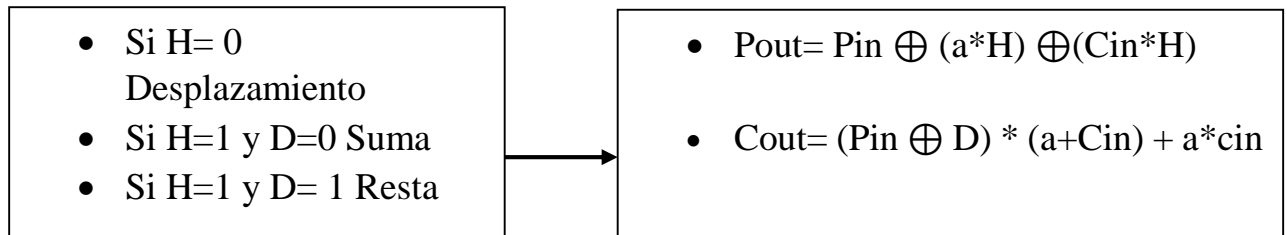


Figura 3.5: Multiplicador de Booth paralelo de 4 bits [12]

### 3.2.2. Multiplicador Baugh-Wooley

El producto de dos números expresados en complemento a dos viene dado por:

$$\begin{aligned}
 X &= x_{n-1} x_{n-2} x_{n-3} \dots x_0 = -2^{n-1} x_{n-1} + \sum_{i=0}^{n-2} x_i 2^i \\
 Y &= y_{n-1} y_{n-2} y_{n-3} \dots y_0 = -2^{n-1} y_{n-1} + \sum_{j=0}^{n-2} y_j 2^j \\
 XY &= \left( -2^{n-1} x_{n-1} + \sum_{i=0}^{n-2} x_i 2^i \right) \cdot \left( -2^{n-1} y_{n-1} + \sum_{j=0}^{n-2} y_j 2^j \right) = \\
 &= \sum_{i=0}^{n-2} \sum_{j=0}^{n-2} 2^{i+j} x_i y_j + 2^{2n-2} x_{n-1} y_{n-1} - 2^{n-1} x_{n-1} \sum_{j=0}^{n-2} y_j 2^j - 2^{n-1} y_{n-1} \sum_{i=0}^{n-2} x_i 2^i
 \end{aligned}$$

El algoritmo de Baugh-Wooley es una modificación de la expresión de arriba, que permite multiplicar números en C-2 sin utilizar restadores, solo utilizando sumadores completos [12]. En la figura inferior se muestra un ejemplo de este tipo de multiplicador. El código VHDL correspondiente a esta arquitectura se encuentra en el anexo A.7.

$$\begin{aligned}
 AB &= \left( -2^{n-1} a_{n-1} + \sum_{i=0}^{n-2} a_i 2^i \right) \cdot \left( -2^{n-1} b_{n-1} + \sum_{j=0}^{n-2} b_j 2^j \right) = \\
 &= \sum_{i=0}^{n-2} \sum_{j=0}^{n-2} 2^{i+j} a_i b_j + 2^{2n-2} a_{n-1} b_{n-1} - 2^{n-1} a_{n-1} \sum_{j=0}^{n-2} b_j 2^j - 2^{n-1} b_{n-1} \sum_{i=0}^{n-2} a_i 2^i = \\
 &= \sum_{i=0}^{n-2} \sum_{j=0}^{n-2} 2^{i+j} a_i b_j + 2^{2n-2} a_{n-1} b_{n-1} + 2^{n-1} \sum_{j=0}^{n-2} a_{n-1} b_j 2^j + 2^{n-1} \sum_{i=0}^{n-2} b_{n-1} a_i 2^i + 2^n - 2^{2n-1}
 \end{aligned}$$

Ejemplo de multiplicador Baugh-Wooley de 6 x 4 bits:

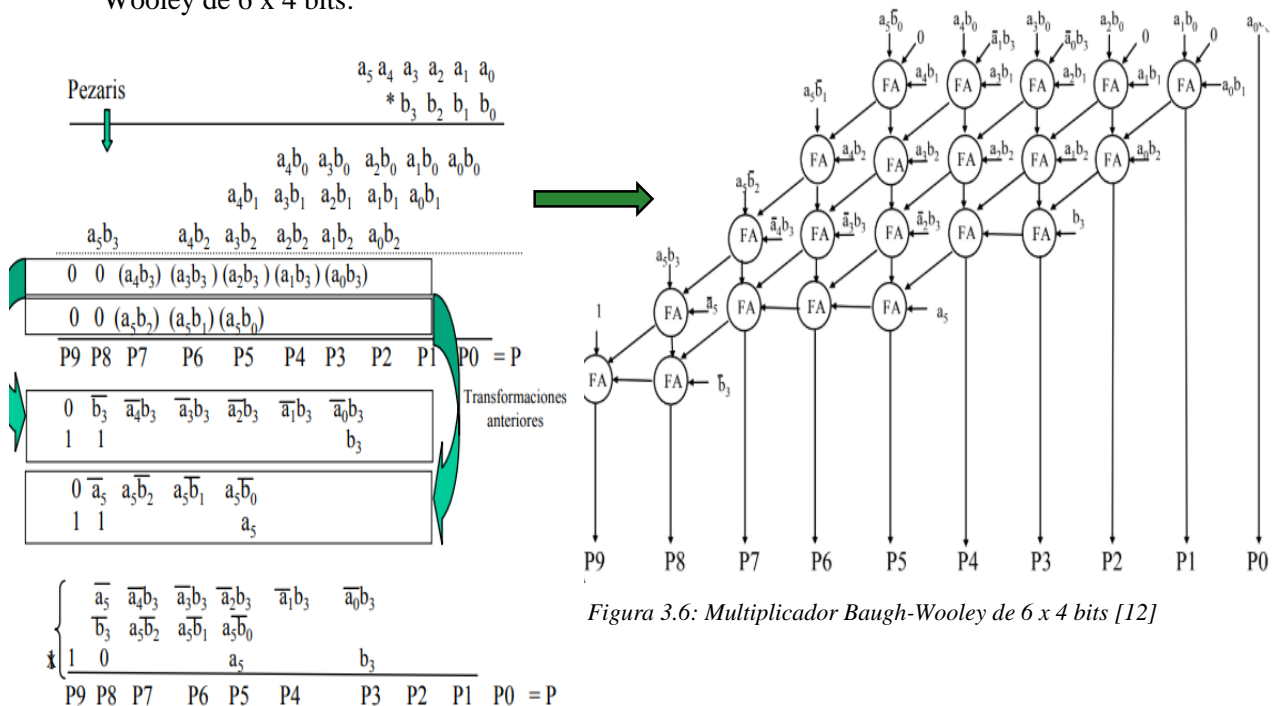


Figura 3.6: Multiplicador Baugh-Wooley de 6 x 4 bits [12]

## Capítulo 4: Metodología

El diseño de los circuitos implementados sobre la FPGA está limitado por varios factores externos, como son la FPGA utilizada, la frecuencia de reloj máxima o el voltaje de alimentación. Aun así, gran parte de las prestaciones del circuito vienen determinadas por la metodología de diseño, por la elección de la arquitectura del multiplicador y por la optimización del código VHDL. Por lo tanto, en los siguientes puntos se explica los pasos seguidos en el trabajo, las técnicas de diseño implementadas y la configuración de cada herramienta utilizada a lo largo del proyecto.

### 4.1. Metodología de diseño descendente

Establecer una metodología de diseño significa definir las distintas etapas que recorrerán los diferentes niveles de abstracción y como evolucionaremos a través de ellos. Las técnicas de diseño descendente apuestan por la formalización de las tareas de diseño desde las primeras etapas de concepción. Estas técnicas se centran en promover el diseño a nivel comportamental, facilitando la evaluación de soluciones alternativas, dando lugar al diseño de alto nivel [19].

La actual metodología de diseño de los circuitos integrados consiste en un diseño descendente. El primer paso consiste en la descripción del comportamiento de los circuitos mediante lenguajes de descripción de hardware (HDL), simulación a nivel de comportamiento y utilización de herramientas de síntesis lógica [19].

### 4.2. Proceso de diseño de sistemas basados en FPGA

#### 1. Especificación

El primer paso para la realización de un diseño digital basado en FPGA, es tener claro el funcionamiento que queremos conseguir con el diseño del circuito. A partir de esta idea, se determinan los parámetros de diseño. En nuestro caso, se diseñará un circuito multiplicador, que con dos entradas  $M$  y  $n$ , nos de su producto  $P$ . Los parámetros de diseño son área, velocidad y consumo, que son explicados posteriormente con mayor detalle.

#### 2. Diseño digital

Una vez que tenemos clara la función del sistema digital y los parámetros requeridos, habrá que realizar primero el diseño a nivel algorítmico, estudiando las posibles alternativas que nos darán el comportamiento deseado. Para el diseño de multiplicadores digitales existen multitud de arquitecturas que nos pueden dar el funcionamiento requerido. El objetivo de este trabajo es comparar estas posibles alternativas, por lo que se estudiarán y diseñarán las distintas posibilidades. Una vez comprendido el algoritmo de funcionamiento, debe pasarse al diseño lógico, es decir, definir los distintos bloques lógicos que integrarán el circuito. Este paso es

crucial para el posterior diseño utilizando VHDL, ya que, a diferencia a los lenguajes de programación “software”, al estar describiendo hardware, debemos ser conscientes de la arquitectura que queremos implementar en todo momento, ya que esto influirá enormemente en las prestaciones del diseño.

### **3. Descripción del HDL**

Una vez tenemos los bloques lógicos que formarán el circuito, pasaremos a describirlo mediante un lenguaje HDL, en nuestro caso VHDL. Aunque existen distintos estilos y niveles de descripción como se menciona anteriormente. Se procurará realizar las descripciones de los multiplicadores a un nivel estructural, para que se estén claro los componentes de cada uno de ellos y evitar que la herramienta de síntesis realice su propia interpretación del hardware que queremos implementar. Así nos aseguraremos que las implementaciones se realicen de acuerdo con las arquitecturas de interés.

### **4. Simulación**

Antes de proceder a la síntesis del circuito, se realizará una simulación funcional de todos los multiplicadores descritos, comprobando el correcto funcionamiento en todos los posibles casos que se puedan dar. Para ello se diseñarán unos ficheros de test y se simularán con el simulador de la herramienta Vivado.

### **5. Síntesis**

Cuando se ha comprobado que la descripción del circuito que queremos implementar funciona correctamente, se pasa a la síntesis utilizando la herramienta Vivado. Antes, deben establecerse las restricciones físicas (physical constraints), por ejemplo, la asignación de puertos I/O a las distintas señales, y las restricciones de tiempo (timing constraints).

La síntesis consiste en la formación de los distintos bloques lógicos y los recursos de interconexión que compondrán el diseño, según la interpretación de la descripción VHDL realizada que haga la herramienta de síntesis. Por esta razón, cuanto mejor definido esté nuestro diseño, más nos acercaremos al circuito que realmente queremos implementar. En esta fase de diseño se puede volver a realizar una simulación de síntesis, para comprobar el correcto funcionamiento del circuito una vez los elementos lógicos han sido establecidos.

### **6. Implementación**

En esta fase se obtiene el esquema físico de implementación y conexionado. La implementación es la asignación de los recursos lógicos de la síntesis en la FPGA utilizada, se puede dividir en dos etapas:

**Etapas de Mapeado (Mapping):** El mapeado es la asignación de recursos, es la fase en la que a cada elemento lógico de la netlist se le asigna un recurso lógico de la FPGA.

**Ubicación (placement) y enrutado (Routing):** Una vez realizada la asignación de los módulos que forman parte del sistema, es necesario llevar a cabo la ubicación, es decir su colocación en una parte determinada de la FPGA, y proceder a su interconexión.

Estas fases dependen del dispositivo FPGA utilizado y cada fabricante ha desarrollado sus propias herramientas de diseño. En esta fase de diseño, se realiza una estimación de retardos incluyendo las interconexiones. Se puede volver a realizar una simulación temporal de implementación para comprobar el correcto funcionamiento del circuito, una vez los elementos lógicos han sido distribuidos en el hardware de la FPGA.

En esta fase se obtienen los resultados de los parámetros de diseño, como el área ocupada y la velocidad del circuito, así como una estimación del consumo. Con estos resultados podremos comparar los resultados de los distintos multiplicadores realizados. Para que la estimación de consumo sea lo más precisa posible, en la simulación de implementación se genera un fichero de actividad que contiene la actividad del circuito y que se le proporciona como entrada a la herramienta Report Power de Vivado.

Una vez finalizada la compilación de los resultados, se procede a la generación del **bitstream**, que es un archivo de configuración que determina la configuración del circuito que se implementa sobre la FPGA. Si se encontrara algún error antes de esta fase, habría que cambiar alguna de las opciones de implementación o modificar la descripción del circuito.

## **7. Laboratorio**

Con el bitstream generado, ya tenemos todos los archivos para implementar los diseños sobre la FPGA, y poder medir el consumo en el laboratorio, para compararlo con la estimación inicial.

### **4.3. Técnicas de diseño**

Estas son algunas técnicas de diseño para modificar las prestaciones de un circuito de procesamiento, para cumplir con los requisitos y especificaciones. Algunas de estas técnicas se utilizan posteriormente en el diseño de los circuitos multiplicadores en VHDL. Se debe tener en cuenta que los parámetros del circuito no son independientes y que las técnicas utilizadas sacrifican un parámetro para intentar optimizar otro.

#### **4.3.1. Paralelización**

Esta técnica consiste en realizar varias operaciones en paralelo para aumentar el número de elementos procesados por unidad de tiempo. Para esto, debemos aumentar los módulos de procesamiento. Esta técnica consigue un aumento de la velocidad, a coste de un aumento del área y el consumo [7]. La capacidad de trabajar en paralelo es una de las grandes ventajas de las FPGAs frente a los microprocesadores.

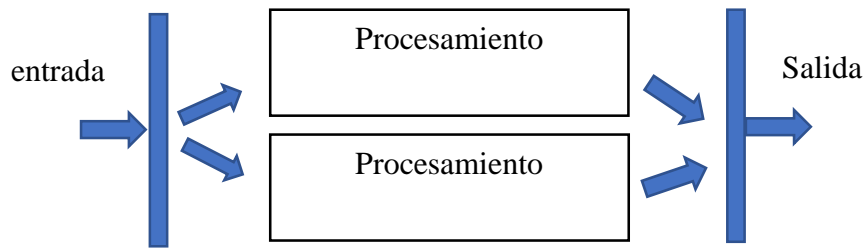


Figura 4.1: Paralelización

### 4.3.2. Segmentación/pipelining

La segmentación consiste en dividir un procesamiento en tareas más sencillas y separarlas por elementos de memoria.

Con la segmentación se pueden conseguir dos objetivos:

- Aumentar la frecuencia reloj
- Aumentar la cantidad de elementos procesados por unidad de tiempo

Puede ocurrir que el tiempo de procesamiento de un módulo limite la frecuencia máxima del circuito. En cambio, si segmentamos ese módulo, cada una de las divisiones podrá trabajar por separado y así, aumentar la frecuencia del circuito. Con esto conseguiremos aumentar la cantidad de elementos procesados (throughput). Por otro lado, hay que tener en cuenta, que la segmentación implica un aumento de la latencia del circuito. Además, implica un aumento de área por los elementos de memoria añadidos, y suele necesitar alguna lógica extra de control [7].

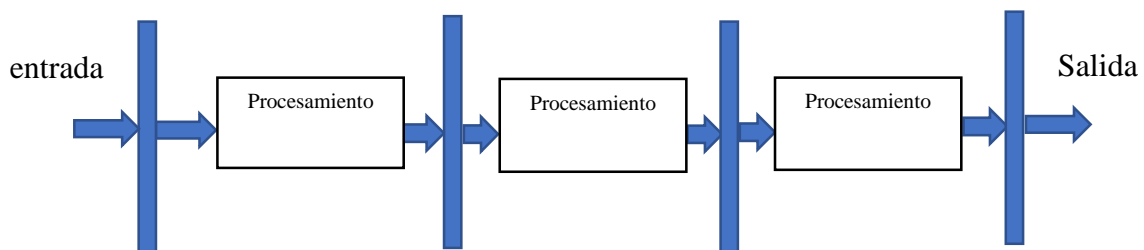


Figura 4.2: Segmentación

### 4.3.3. Clock-gating

Vivado ofrece diversas opciones para reducir el consumo. Entre ellas está la técnica de clock gating, que consiste en deshabilitar el reloj en el circuito síncrono o en ciertas partes de él, cuando no se vayan a utilizar. Con esto, conseguimos reducir la actividad del circuito, y, por ende, el consumo dinámico global. Esta técnica de reducción de consumo puede afectar al timing de nuestro diseño [25].



#### 4.4. Parámetros de diseño/especificaciones

Se describen los tres parámetros fundamentales que marcan los diseños electrónicos, las consideraciones de cada uno y como serán cuantificados a la hora de recoger los datos de cada diseño.

**Área:** Viene determinada por el número de dispositivos lógicos, el tamaño de estos y por la cantidad y la longitud de las interconexiones que se necesitan. El área se medirá como el número de LUTs y flip-flops que ocupa el circuito multiplicador de la FPGA. Las LUTs son unas unidades elementales de las FPGAs de XILINX, y contienen los elementos básicos para implementar la lógica de los circuitos.

**Velocidad:** La velocidad de procesamiento depende de muchos factores, como son la frecuencia de reloj a la que funciona el circuito, la velocidad de propagación de las señales y la tecnología empleada. En un circuito multiplicador la velocidad está determinada por el número de multiplicaciones por unidad de tiempo y la latencia, que es el tiempo que tarda en obtenerse el primer resultado de una multiplicación completa.

**Consumo:** Parámetro fundamental, que se mide en vatios y que se divide en potencia estática y potencia dinámica, tal y como se ha explicado en el capítulo 2 de este trabajo.

#### 4.5. Estimación de consumo

La estimación y control de consumo a la hora de diseñar un circuito sobre una FPGA es un proceso sumamente complejo, en el que influyen múltiples factores que están interrelacionados entre ellos. Una buena estimación del consumo en las fases previas del diseño nos permite conocer las especificaciones que requerirá nuestra FPGA [25]. En este punto se describen todas las consideraciones que se deben tener en cuenta a la hora de estimar la potencia del circuito, y el proceso de estimación que hemos realizado en este trabajo.

Para una correcta estimación del consumo se debe:

1. Proveer información adecuada del **dispositivo** que se utilizará, así como el **ambiente** en el que trabajará el circuito.
2. Siempre que sea posible determinar **la actividad** de todos los nodos del circuito
3. Restringir correctamente **el reloj**
  - Un reloj sin restricciones se le considerará que tiene una frecuencia de 0 MHz.
  - Un reloj al que se le fija una frecuencia de funcionamiento demasiado alta provocará un aumento del consumo dinámico.

4. **Controlar las señales**, especificando la *static probability* y la actividad de las señales de control, tales como el Global set, el reset o los clock enables.
5. **Definir correctamente los inputs**, ya que la actividad de los puertos de entrada determina la actividad del circuito.

#### 4.5.1. Estimación de consumo con Vivado

La estimación de consumo es necesaria a lo largo de todas las fases del diseño. Vivado es capaz de estimar el consumo a lo largo de todas estas fases, para ellos utilizaremos su herramienta **Report Power**, que nos da una estimación del consumo de cualquier diseño una vez implementado. La estimación del consumo será realizada en la etapa post-route, una vez que todos los recursos de interconexión y la información temporal de cada canal ha sido definida. De esta forma, el simulador es capaz de conocer la actividad interna de todos los nodos.

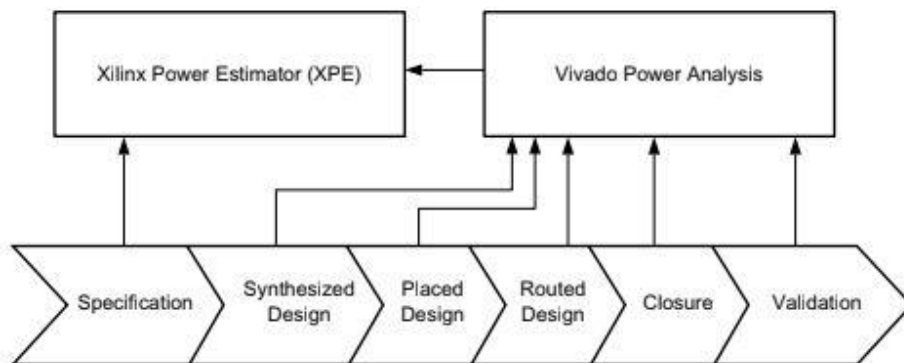


Figura 4.3: Proceso de estimación de consumo con Vivado [24]

Existen dos métodos de estimación de consumo:

**Vectorless:** Cuando la actividad del circuito no se conoce, la herramienta se encarga de estimar la actividad (*switch rate* y *static probability*). La estimación es poco precisa comparada con la estimación vector-based [25].

**Vector-based:** La actividad del circuito se obtiene a partir de una simulación post-implementación, basándonos en el funcionamiento real del circuito, en la que generamos un archivo SAIF [25].

#### 4.6. Archivo SAIF

Es un ASCII report que almacena la información de la actividad del circuito generado por la herramienta de simulación de Vivado. Durante la simulación podemos especificar de qué nodos queremos extraer la actividad. Este archivo ayuda a estimar el consumo de forma mucho más precisa.

Un importante factor para una estimación precisa del consumo es que la actividad del diseño debe ser realista. Debe representar el caso típico de entrada de datos del diseño [25]. Por lo tanto, es evidente que la simulación de la que obtenemos el archivo SAIF debe parecerse lo máximo posible a cómo funcionará el circuito en la realidad.

En este trabajo analizamos multiplicadores digitales, muy utilizados en el procesamiento digital de señales (DSP), que requiere el tratamiento de señales a alta velocidad. Por lo tanto, para la generación del archivo SAIF, se ha decidido que una representación realista sería realizar una multiplicación cada ciclo de reloj. Se ha realizado la simulación durante 0,04 milisegundos, para las frecuencias de 25, 50, 100, 150 y 200 MHz.

Para simular la actividad real de los circuitos necesitamos una fuente de números aleatorios. En este trabajo hemos realizado un **Load Feedback Shift Register (LFSR)** de tipo Fibonacci, que consiste en la utilización de un registro realimentado y la utilización de varias puertas XNOR para generar una secuencia pseudoaleatoria. Para conseguir secuencias de máxima longitud, de  $2^n - 1$  números distintos, siendo  $n$  el número de etapas o bits del LFSR y así aumentar la aleatoriedad de la simulación, se debe utilizar una realimentación determinada. Los LFSR se pueden expresar como polinomios, por ejemplo:

$P(x) = 1 + x^4 + x^5 + x^6 + x^8$  donde el 1 representa la salida del LFSR,  $x^4$  la realimentación del registro 4,  $x^5$  la realimentación del registro 5,  $x^6$  la realimentación del registro 6 y  $x^8$  la realimentación del registro 8. Este polinomio debe ser primitivo, con esto conseguiremos un LFSR que genera una secuencia de máxima longitud. En la imagen inferior se puede ver el LFSR que representa el polinomio anterior.

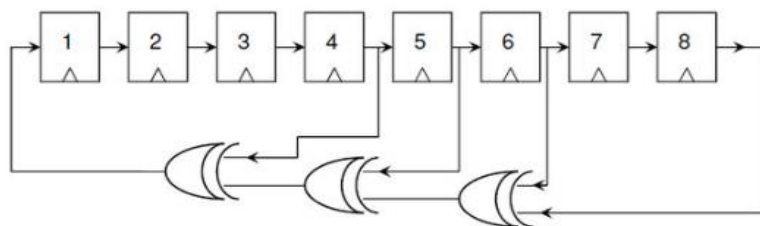


Figura 4.4: Load Shift register generador de secuencia de máxima longitud [16]

El código de este LFSR se encuentra en el anexo A.13, en la imagen inferior se puede ver la simulación realizada, con la secuencia pseudoaleatoria de números.

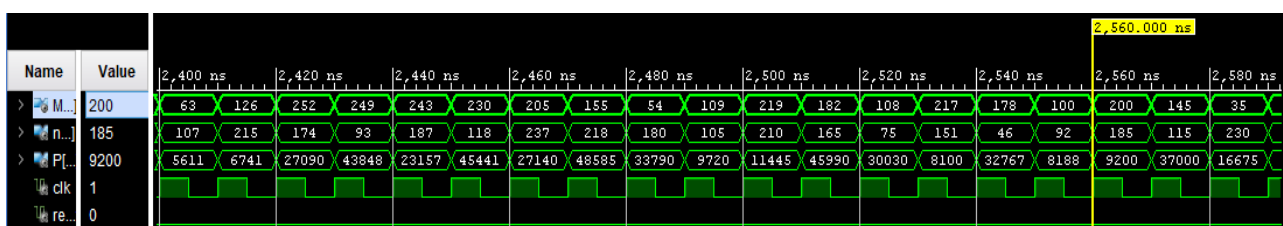
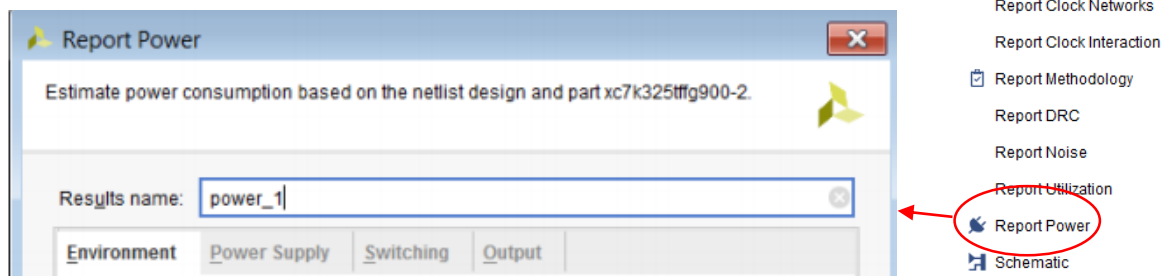


Figura 4.5: Simulación con LFSR a 100 MHz para generar archivo SAIF

Para generar el archivo SAIF, se deben escribir las siguientes instrucciones durante la simulación:

1. `open_saif`
2. `log_saif [get_object /<toplevel_testbench/uut/*>]`
3. `run *ns`
4. `close_saif`

Con estas estas instrucciones primero se abre el archivo y se le indica los objetos que queremos que analice. En este caso le hemos indicado que analice todas las señales de entrada, salida e internas. Se hace correr la simulación un tiempo determinado y se cierra el archivo SAIF. Con este generaremos el archivo que utilizaremos en la posterior estimación.



Una vez hemos generado el archivo que contiene la actividad del diseño con la simulación post-implementación, vamos a la herramienta Report Power en Vivado. Al acceder a la herramienta, nos aparecerá una ventana para introducir los distintos parámetros del diseño que afectan al consumo. En la primera ventana se muestran los parámetros de **Environment**, que son aquellos factores externos que afectan a la temperatura de la placa y que tienen gran influencia en el consumo estático. Se explican a continuación los factores más relevantes que se han tenido en cuenta.

Environment	Power Supply	Switching	Output
<b>Device Settings</b>			
Temp grade:	commercial		
Process:	typical		
<b>Environment Settings</b>			
Output Load:	0	pF	[0 - 10000]
<input type="checkbox"/> Junction temperature:	26.701 °C		
Ambient temperature:	25	°C	
<input type="checkbox"/> Effective $\theta_{JA}$ :	14.38	°C/W	[0 - 100]
Airflow:	0	LFM	
Heat sink:	none		
$\theta_{SA}$ :	0	°C/W	[0 - 100]
Board selection:	medium (10"x10")		
Number of board layers:	8to11 (8 to 11 Layers)		
$\theta_{JB}$ :	7.4	°C/W	[0 - 100]
Board temperature:	25	°C	[-55 - 85]

Figura 4.6: Ventana de Environment Report Power

**Tem-grade:** Puede ser comercial o industrial, según la configuración afectará a la potencia estática de la placa.

**Process:** Sirve para elegir entre el caso típico de funcionamiento o el peor caso del diseño a analizar.

**Junction temperature:** Indica la temperatura máxima esperada dentro de la placa. La corriente de fuga aumenta exponencialmente con la temperatura, por lo que es crucial una buena estimación. Es determinada por el programa a partir de los demás factores.

**Airflow (LFM):** El airflow en el chip es medido en Linear Feet per Minute (LFM). Por defecto viene dado con un valor de 250, como nosotros no vamos a proporcionar un flujo de aire activo le damos valor 0.

En la siguiente ventana de **Power Supply**, se indica los voltajes de alimentación para los distintos recursos y canales de la FPGA. En general no se suelen modificar estos parámetros.

En la ventana de **Switching** podemos modificar las actividades de las principales señales de nuestro diseño o proporcionar el archivo SAIF generado anteriormente.

Por último, en **Output** indicamos los archivos que queremos generar. El *Output text file* es un archivo de texto que almacena los resultados de la estimación. El *output XPE file* es un archivo que podremos utilizar posteriormente en la herramienta Xilinx Power Estimator.

Cuando finalizamos el *Report Power* lo primero que podemos ver, es que se nos genera un resumen con los resultados obtenidos. Donde se indica la potencia total consumida separándola en dinámica y estática.

## Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

**Total On-Chip Power:** 0.141 W  
**Junction Temperature:** 27.0 °C  
 Thermal Margin: 58.0 °C (3.9 W)  
 Effective  $\theta_{JA}$ : 14.4 °C/W  
 Power supplied to off-chip devices: 0 W  
 Confidence level: High

[Launch Power Constraint Advisor](#) to find and fix invalid switching activity

## On-Chip Power

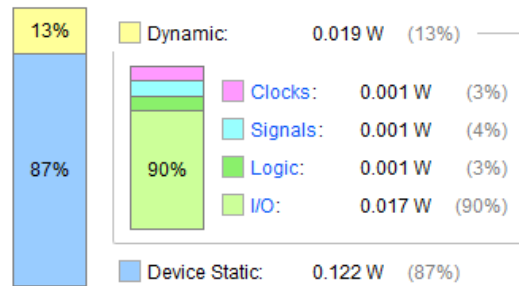


Figura 4.7: Resumen de la estimación del Report Power

En el resumen vemos un apartado de **Confidence level**, que nos indica el nivel de confianza de nuestra estimación, que en nuestro caso se puede observar que es alto. Si hacemos clic se puede ver los factores que influyen en una estimación precisa.

Confidence Level Details	
Design State:	High Design is routed
Clock Activity:	High User specified more than 95% of clocks
I/O Activity:	High User specified more than 95% of inputs
Internal Activity:	High User specified more than 25% of internal nodes
Characterization Data:	High Device models are Production

## Xilinx Power Estimator (XPE)

Es una herramienta de estimación de potencia típicamente utilizada en la fase de pre-implementación. XPE ayuda con la evaluación de la arquitectura, la selección del dispositivo y a elegir la fuente de alimentación. XPE es también utilizado en el ciclo de diseño durante la implementación, para evaluar las implicaciones en la potencia de las engineering change orders (ECO) [25]. Además, nos da la posibilidad de ver información extra sobre el consumo de nuestro diseño. Por ejemplo, en las dos gráficas inferiores se puede observar como varía el consumo de nuestro diseño, con la temperatura de la placa y con el voltaje de alimentación, que son parámetros que podemos modificar si queremos reducir el consumo de nuestro diseño.

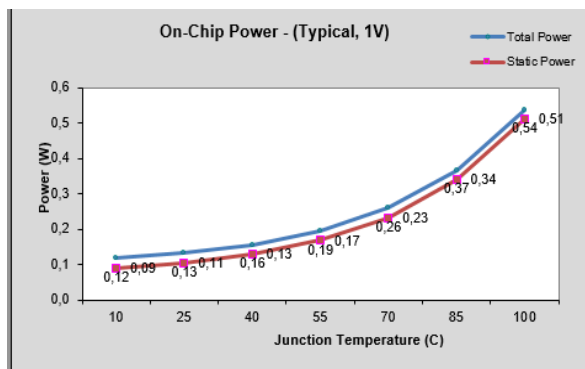


Figura 4.8: Gráfica de XPE Consumo/Temperatura

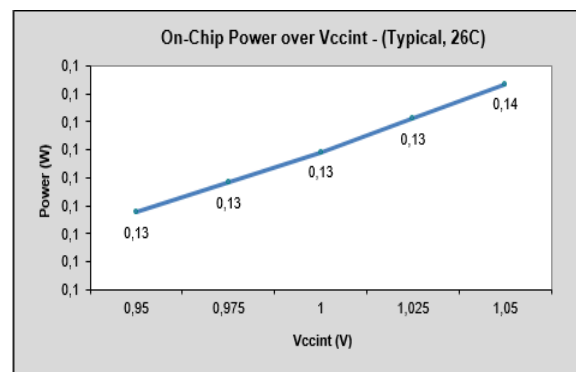


Figura 4.9: Gráfica XPE consumo/Voltaje

## 4.7. Archivo XDC

Como se explica en el punto 4.2, es necesario establecer las restricciones físicas de nuestro diseño. Para ellos se utiliza un archivo XDC (Xilinx Design Constraints). Para todos los multiplicadores se han utilizado las mismas restricciones, ya que éstas pueden afectar considerablemente a las especificaciones de nuestro diseño.

### Restricciones físicas (Physical constraints)

Las restricciones físicas permiten asociar las señales de entrada y salida del diseño a los distintos puertos de nuestra placa. Se deben tener dos aspectos en cuenta, lo primero es asociar nuestras señales a los puertos correctos de la placa, ya que mientras algunos puertos son multifunción, el reloj, por ejemplo, está situado en un puerto específico. Para realizar esta asociación correctamente debemos mirar el datasheet de nuestra placa que se encuentra en el anexo B. También debemos tener en cuenta la configuración que tendrá el puerto asociado a cada señal, ya que ésta afectará al voltaje y por tanto al consumo de nuestro diseño. Para determinar correctamente la configuración de cada puerto debemos mirar los niveles de tensión que necesitan los distintos recursos de la placa. Este paso es crucial, ya que un voltaje incorrecto podría hacer que nuestro diseño no funcione o aún peor, podríamos dañar la placa.

En la imagen inferior podemos ver la asociación del reloj, al puerto H16, y el reset al puerto V17. Además, como se ha comentado anteriormente, se le asigna, al reloj, el IOSTANDARD LVCMOS33 que otorga un voltaje de 3.3 V y al reset, el IOSTANDARD LVCMOS18 que otorga un voltaje de 1.8 V. La asociación de las señales de entrada y salida se encuentra en el anexo A.12.

```
set_property PACKAGE_PIN H16 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports clk]
set_property IOSTANDARD LVCMOS18 [get_ports reset]
set_property PACKAGE_PIN V17 [get_ports reset]
```

### Restricciones temporales (Timing constraints)

Las restricciones temporales son las instrucciones que el diseñador da a la herramienta Vivado sobre las limitaciones o imposiciones de velocidad que tiene el diseño. Estas restricciones temporales se especifican en el archivo de restricciones XDC. Una restricción muy importante para una estimación precisa del consumo, como se menciona en el capítulo anterior, es la restricción que afecta a la frecuencia de reloj. En la imagen inferior podemos ver la restricción del reloj en el que se establece una frecuencia de 50MHz.

```
create_clock -period 5.000 -name clk -waveform {0.000 2.500} [get_ports clk]
```

## 4.8. FPGA in the Loop

Para probar que realmente funcionan los multiplicadores en el hardware se utilizará la herramienta de Matlab, Simulink, y dentro de esta, la herramienta FPGA-in-the-Loop (FIL), que provee la capacidad probar los diseños en hardware real para código HDL. Además, nos permitirá tanto introducir datos a la FPGA, como recoger los valores de la salida. Cabe destacar que este proceso se debe realizar para todos los multiplicadores que se han diseñado, a continuación, se describen los pasos seguidos para testear un solo diseño.

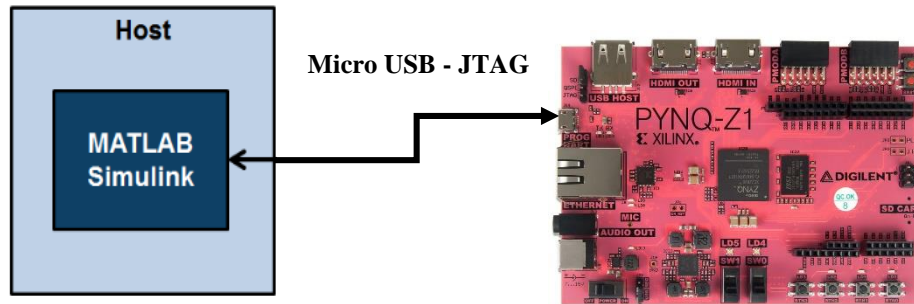


Figura 4.10: Comunicación Matlab – FPGA [9]

FIL realiza las siguientes funciones:

- Genera un bloque en Simulink el cual representa el código HDL
- Realiza la síntesis e implementación del diseño
- Carga el fichero de configuración en la FPGA
- Transmite datos desde Simulink o Matlab a la FPGA y recibe datos de la FPGA
- Ejecuta el diseño en un ambiente real

Los pasos a seguir para poder realizar esto son:

**El primer paso** es definir la placa que vamos a utilizar en nuestro diseño. Aunque Matlab provee un paquete de FPGAs de Xilinx, la FPGA utilizada en este trabajo no se encontraba entre ellas. Matlab también otorga la posibilidad de crear un archivo con nuestra propia placa, en este caso la hemos denominado PYNQ. La configuración de nuestra placa se guarda en un archivo de extensión .XML, de manera que puede ser reutilizado en pruebas sucesivas.



Enter the basic information about your FPGA board such as board name, FPGA specification, and clock and reset pin numbers.

Board Name:

File Location:

Device Information

Vendor:  Family:  Device:

Package:  Speed:  JTAG Chain Position:

FPGA Input Clock

Clock Frequency:  MHz Clock Type:

Clock Pin Number:

Clock IO Standard:

Reset (Optional)

Reset Pin Number:  Active Level:

Reset IO Standard:

Figura 4.11: Características de la placa

En el proceso de customizar nuestra propia placa debemos especificar varias características, como son la FPGA utilizada o la frecuencia del reloj, su posición y la salida estándar.

La conexión entre la placa y Simulink se puede hacer a través de diversos métodos, como puede ser JTAG, Ethernet o PCI Express. Nuestra placa solo dispone de la opción de hacerlo a través de JTAG. Cabe mencionar que no fue necesaria la utilización de ningún cable JTAG, ya que en el datasheet de nuestra placa se puede observar que viene con un chip que realiza de puente entre las conexiones JTAG-USB. Por lo que solo fue necesario utilizar el cable micro-USB, que además, alimenta la placa.

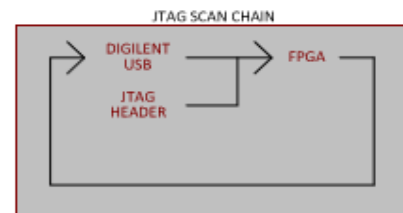


Figura 4.12: Conexión USB-JTAG [4]

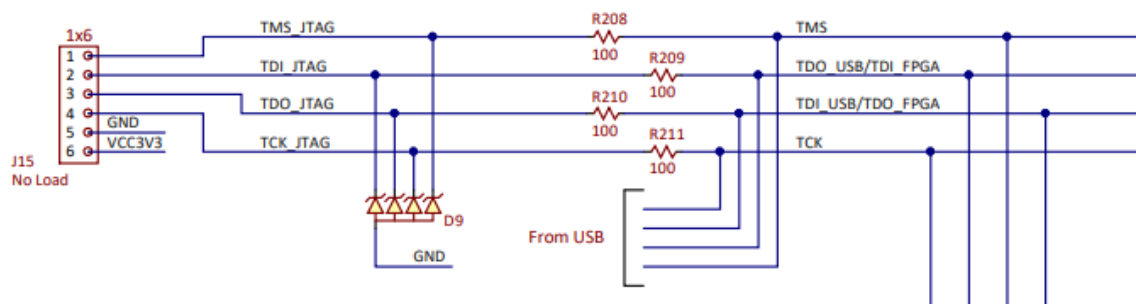



Figura 4.13: Posición JTAG [4]

Por último, debe modificarse la conexión JTAG, para indicar que hay un dispositivo anterior a la FPGA con un IR (instruction register) de longitud 4.

 JTAG (via Digilent cable)

Action

Specify the instruction register (IR) lengths for

(a) The sum of IR lengths for devices before FPGA in the chain

(b) The sum of IR lengths for devices after FPGA in the chain

---

Signal/Parameter List

Sum of IR length before:

Sum of IR lengths after:

**El segundo paso** consiste en establecer en Matlab el software necesario para realizar la implementación del diseño sobre la FPGA, que en este caso es Vivado. Para ello se utiliza una instrucción de Matlab en la que en primer lugar debemos señalar el nombre de la herramienta y segundo su localización en el ordenador. La instrucción utilizada es la siguiente:

```
>>hdlsetuptoolpath('toolname','XilinxVivado','toolpath','F:\Xilinx\Vivado\2017.1\bin\vivado.bat')
```

Prepending following Xilinx Vivado path(s) to the system path:

F:\Xilinx\Vivado\2017.1\bin

Una vez tenemos la placa que vamos a utilizar y el entorno de desarrollo, tenemos que generar el bloque de Simulink que representa el código VHDL que queremos probar. Matlab realiza la síntesis, implementación y generación del bitstream con el software que le hemos proporcionado en el paso dos, de manera automática. En primer lugar, elegimos la placa que hemos configurado al comienzo.

Steps

- > FIL Options
- Source Files
- DUT I/O Ports
- Output Types
- Build Options

Actions

Specify options for FPGA-in-the-Loop.

FIL simulation with

☐ MATLAB System Object

☒ Simulink

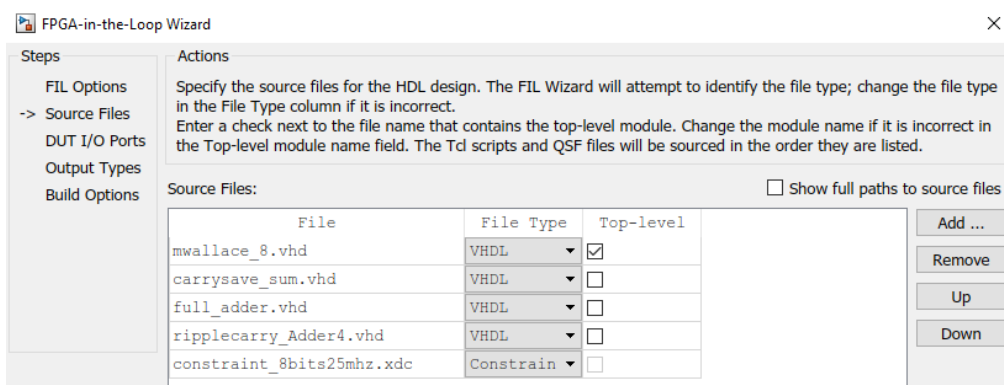
Board Options

Board Name:  Launch Board Manager

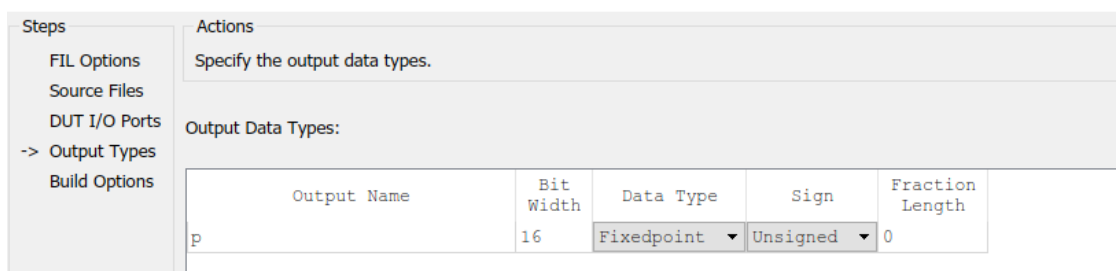
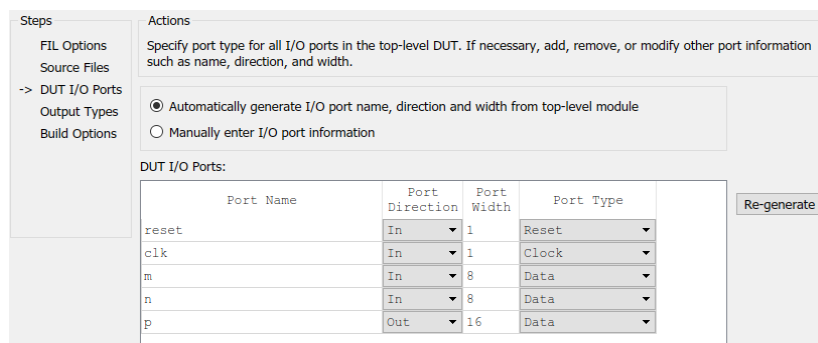
FPGA Device: Zynq XC7Z020-1-CLG400

FPGA-in-the-Loop Connection:

Después, debemos proporcionar los archivos que componen la descripción VHDL. Debemos especificar cuál es el top-module, y también añadimos el archivo XDC que contiene las restricciones del diseño.



El siguiente paso consiste en especificar los puertos de entrada/salida, si bien, la herramienta los detecta automáticamente. Después nos pide especificar el tipo de dato de las salidas del diseño.



Una vez realizados todos estos pasos, especificamos en que carpeta queremos que se cree el bloque de Simulink que contendrá el código HDL y le damos a build. Aparece una nueva ventana y empieza el proceso de generación del bitstream.

Con esto, se genera el bloque FIL, que es el bloque de Simulink que representa nuestro código VHDL. En la imagen inferior podemos ver el bloque que se genera:

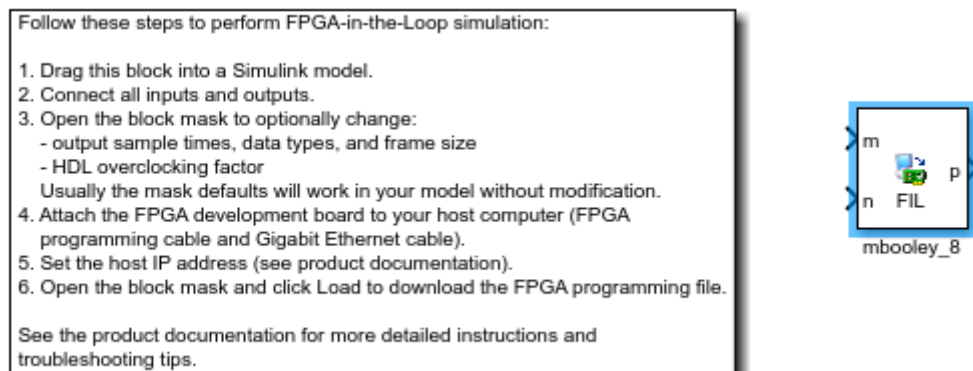
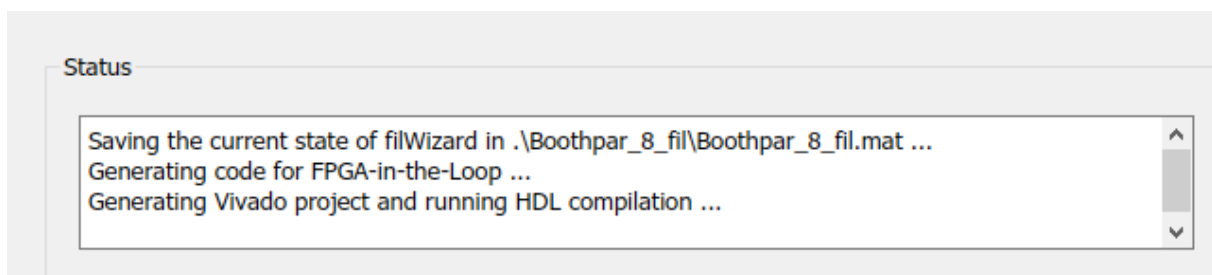


Figura 4.14: Bloque FIL

Comienza el proceso de síntesis, implementación y la generación del bistream. Este proceso tarda varios minutos, al final, se genera, el proyecto VHDL de nuestro diseño.



**En cuarto lugar**, debemos desarrollar el modelo de Simulink en el que se incluirá nuestro bloque FIL, y que nos servirá para enviar los datos a la FPGA, así como para recibirlos.

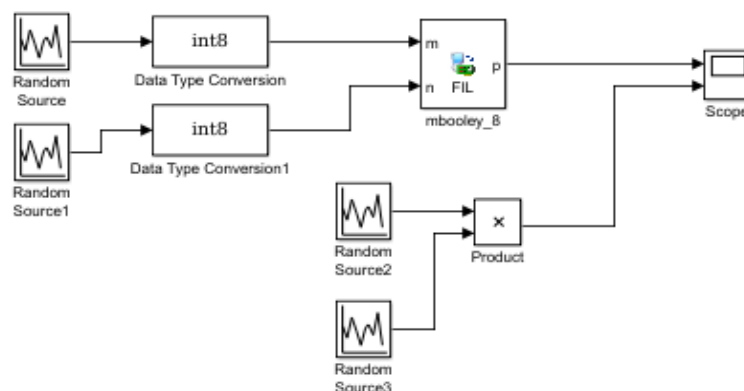
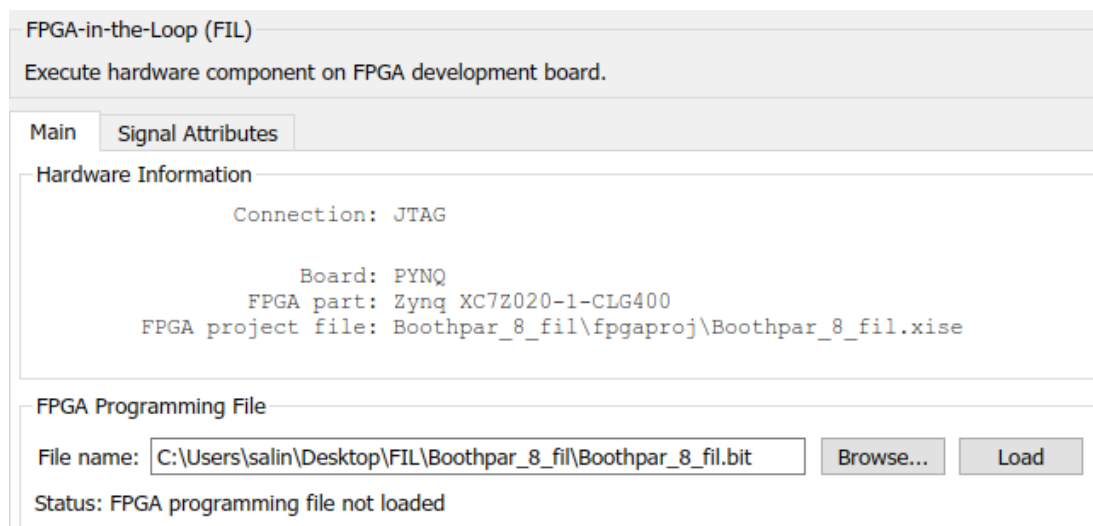


Figura 4.15: Modelo Simulink para Multiplicadores con signo

Como se puede ver en la figura, el modelo de Simulink está compuesto de un Scope, para ver la salida de nuestro diseño, un bloque producto, para comparar la salida de éste con la de nuestro multiplicador y cuatro fuentes que generan una secuencia de números aleatorios. Las señales de cada bloque deben ajustarse para que coincidan con las señales de nuestra descripción VHDL, por esa razón se utilizan los convertidores a int8. Con esto podremos comprobar si efectivamente nuestro multiplicador realiza correctamente las multiplicaciones. Se realizaron diversos modelos de Simulink, con diversas fuentes para ver que los multiplicadores funcionaban correctamente en todas las situaciones.

Por último, ya solo nos queda cargar el diseño sobre la FPGA, para ellos hacemos doble clic en el bloque FIL y le damos a Load. Si no cometimos ningún error al generar el bloque FIL y la placa está conectada correctamente, nos aparecerá que el programa se ha cargado correctamente. Por lo que ya podremos simular nuestro modelo de Simulink y ver los resultados.



La figura siguiente muestra la salida tanto del bloque FIL, como del bloque producto:

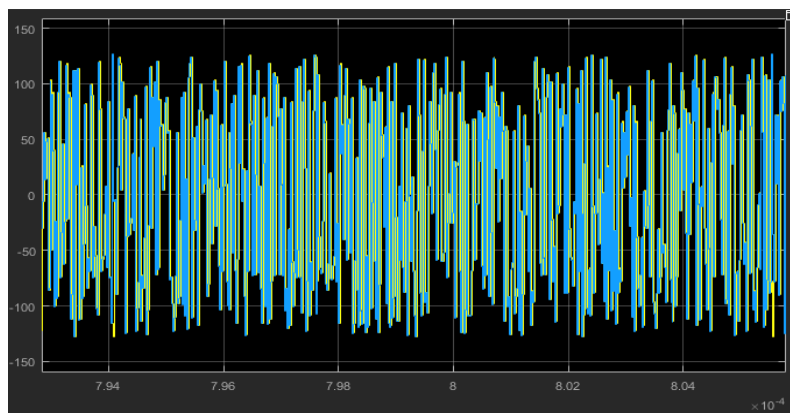


Figura 4.16: Salida de multiplicadores con signo

La imagen superior muestra la salida de los multiplicadores que multiplican números con signo. La imagen se corresponde con una simulación con un sample time de 20 nanosegundos. Como se puede ver, la salida producida por el bloque FIL y la generada como producto en Simulink siguen el mismo recorrido y por lo tanto se puede observar que los circuitos realizan bien las multiplicaciones. Sin embargo, si se hace Zoom se puede ver que las dos señales no son coincidentes. Esto se debe a los registros, tanto de entrada como de salida que tienen los diseños VHDL, que producen un delay entre ambas señales de 40 nanosegundos.

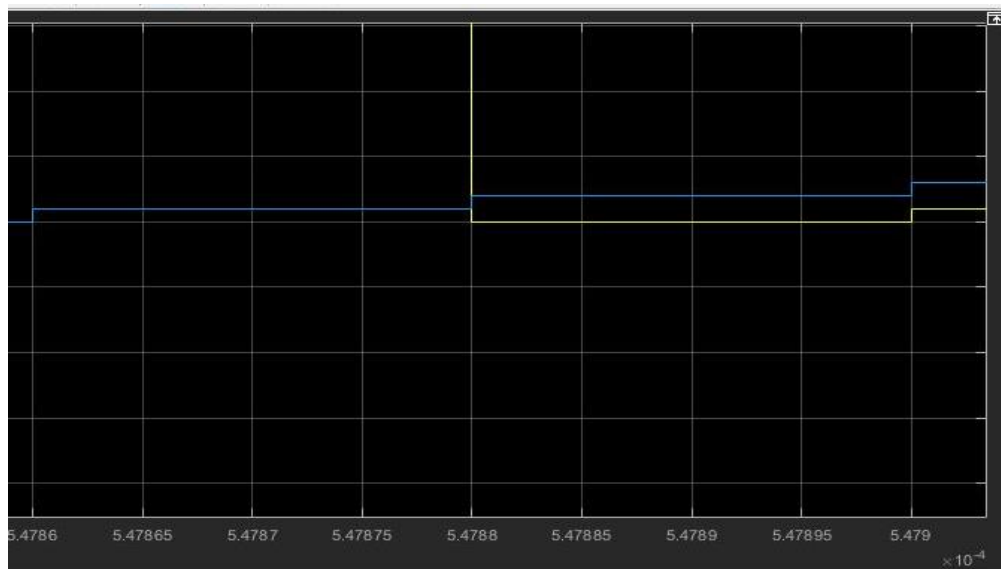


Figura 4.17: Diferencia entre señales

Para comprobar el funcionamiento de los multiplicadores sin signo se utilizó otro modelo de Simulink, que se muestra en la imagen inferior. En este caso, las fuentes son dos contadores libres, que tienen siempre signo positivo.

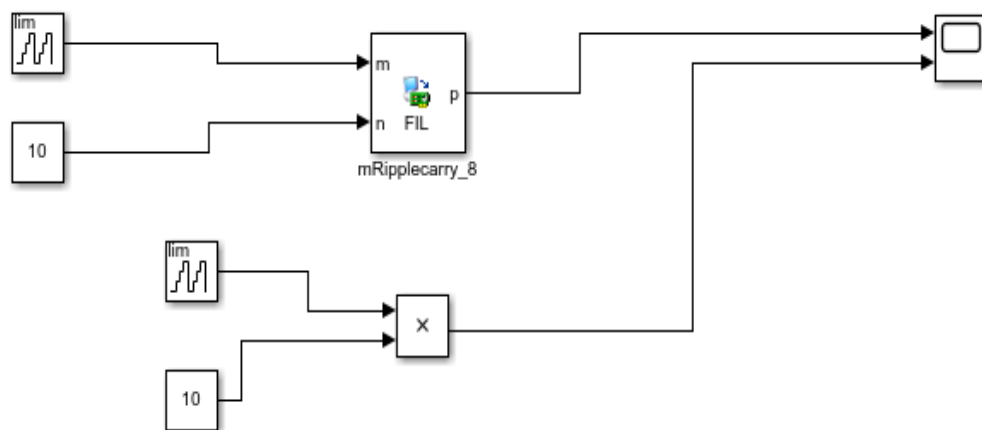
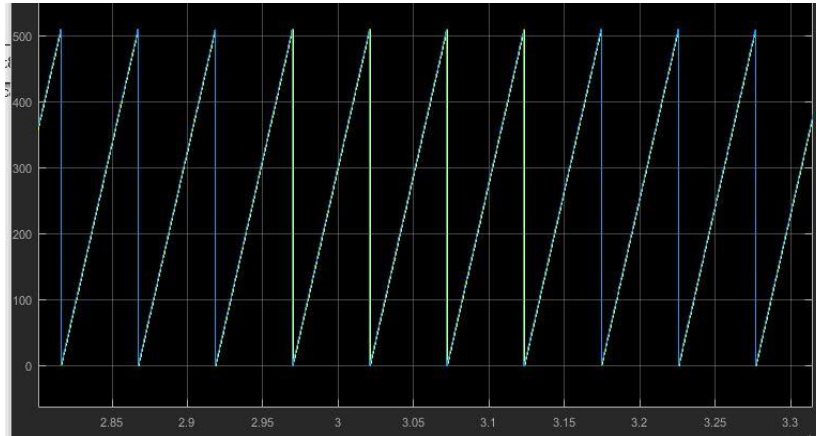


Figura 4.18: Modelo de Simulink para multiplicadores sin signo

La simulación de la imagen inferior también se hizo con un sample time de 20 nanosegundos. Se puede observar como las salidas de los multiplicadores y del bloque producto son coincidentes, si bien como hemos mencionado anteriormente si hacemos zoom podemos ver un pequeño retraso de la señal de salida P.



*Figura 4.19: Salida multiplicadores sin signo en FIL*

## 4.9. Vivado HLS

En puntos anteriores hemos mencionado la metodología que debemos seguir para realizar un diseño HDL para ser implementado en una FPGA. Este proceso puede implicar largos tiempos de desarrollo en comparación con la utilización de otros lenguajes de más alto nivel como podría ser C. Este es uno de los factores que limita la utilización de FPGAs en el mercado electrónico. Una nueva revolución se está produciendo en la actualidad, protagonizada por las herramientas de síntesis de alto nivel. Estas herramientas nos permiten generar un diseño RTL (Register Transfer Level) a partir de códigos escritos en lenguajes de programación de alto nivel (como C o C++), y así mejorar la productividad de los diseñadores de hardware.

**Vivado HLS** (High level Synthesis) es una herramienta que transforma código C en un diseño HDL que puede ser implementado en una FPGA de Xilinx [22]. En este trabajo realizaremos una primera aproximación, transformando el código C de una función que multiplica dos números y nos devuelve su resultado, en un diseño VHDL. Mediremos sus parámetros para compararlos con el resto de multiplicadores descritos en VHDL, de acuerdo a las descripciones arriba entregadas. En la figura 4.20 podemos ver el flujo de diseño de Vivado HLS. A continuación, se explican los pasos seguidos.

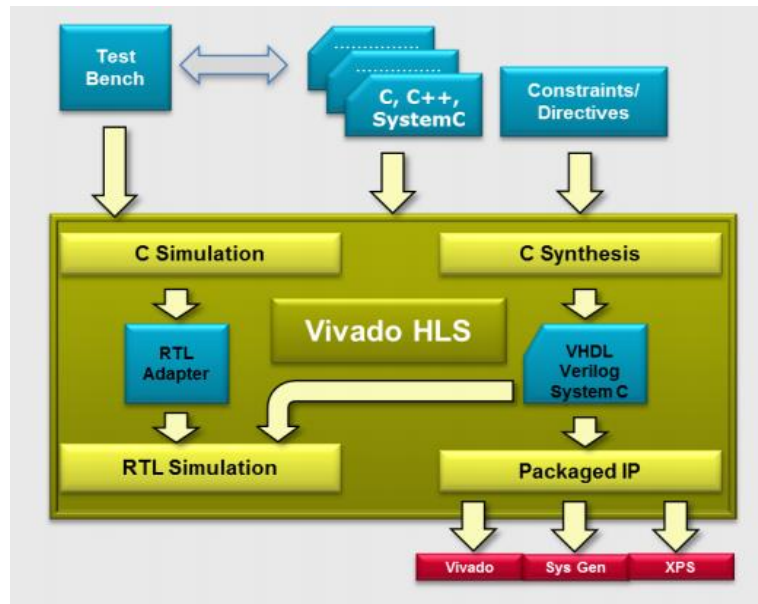


Figura 4.20: Flujo de diseño en Vivado HLS [21]

**1.Código C:** El primer paso en el desarrollo en HLS consiste en elaborar el código C que queremos sintetizar en VHDL.

Este proceso explicado a continuación lo hacemos también para los multiplicadores de 16 y 4 bits.

Vivado HLS tiene una serie de herramientas adicionales para ayudarnos a diseñar hardware mediante lenguaje C. En este caso utilizamos una librería específica de Vivado HLS, que nos permite declarar señales del número de bits que queramos, cosa que en C no se puede hacer.

```
#include <ap_cint.h>
```

Podemos ver como el código C es realmente sencillo en comparación cualquier diseño VHDL.

```

1. #include <ap_cint.h>
2. void multi (int8 M, int8 n, int16 *P) {
3. #pragma HLS PIPELINE
4.
5. *P=M*n;
6.
7. }
  
```

Figura 4.21: Función código C

**2.Directivas/constraints:** En segundo lugar, las directivas y restricciones son las que indican a la herramienta como queremos que sintetice el código C en hardware. Dos procesos son clave en HLS:

**Scheduling:** determina en que ciclo de reloj se realiza cada operación, es decir distribuye las distintas operaciones en el tiempo.



Binding: determina que recursos hardware se asociarán a las distintas operaciones.

En este trabajo hemos realizado multiplicadores que realizan las operaciones en paralelo, pero el código C y los micros trabajan intrínsecamente en serie. Por lo tanto, la función que hemos realizado anteriormente se sintetizará en un multiplicador serie. Si queremos que el multiplicador sea paralelo debemos utilizar una directiva que indique a la herramienta esto.

Por lo que dentro de la función realizada debemos escribir: `#pragma HLS PIPELINE`

**3. Tesbench Código C:** El tercer paso consiste en ver que el código C tiene la funcionalidad correcta. Para esto debemos realizar un testbench en C que compruebe que el valor que devuelve el diseño es el correcto. Si la función principal main devuelve 0 se considerará que la funcionalidad es correcta, si devuelve 1 la herramienta considera que ha habido un error.



```
16 Resultado hardware 70 Resultado software 70
17 INFO: [SIM 1] CSim done with 0 errors.
18 INFO: [SIM 3] ***** CSIM finish *****
```

Figura 4.22: Resultado de la simulación C

```
1. #include<stdio.h>
2. #include<ap_cint.h>
3. int8 M = 10;
4. int8 n = 7;
5. int main() {
6.     int16 result;
7.     int16 P;
8.
9.     result=M*n;
10.    P=multi(M,n);
11.
12.    printf ("Resultado hardware %d",P);
13.    printf ("Resultado software %d",result);
14.
15.    if(P==result) {
16.        return 0;
17.    }
18.    else{
19.        return 1;
20.    }
21. }
```

Figura 4.23: Testbench código C

**4.Elaboración RTL:** Una vez hemos comprobado que el código C funciona correctamente debemos transformar éste en código HDL. Como resultado obtendremos un archivo con el código tanto en VHDL, como en Verilog y un resumen de las características de nuestro diseño.

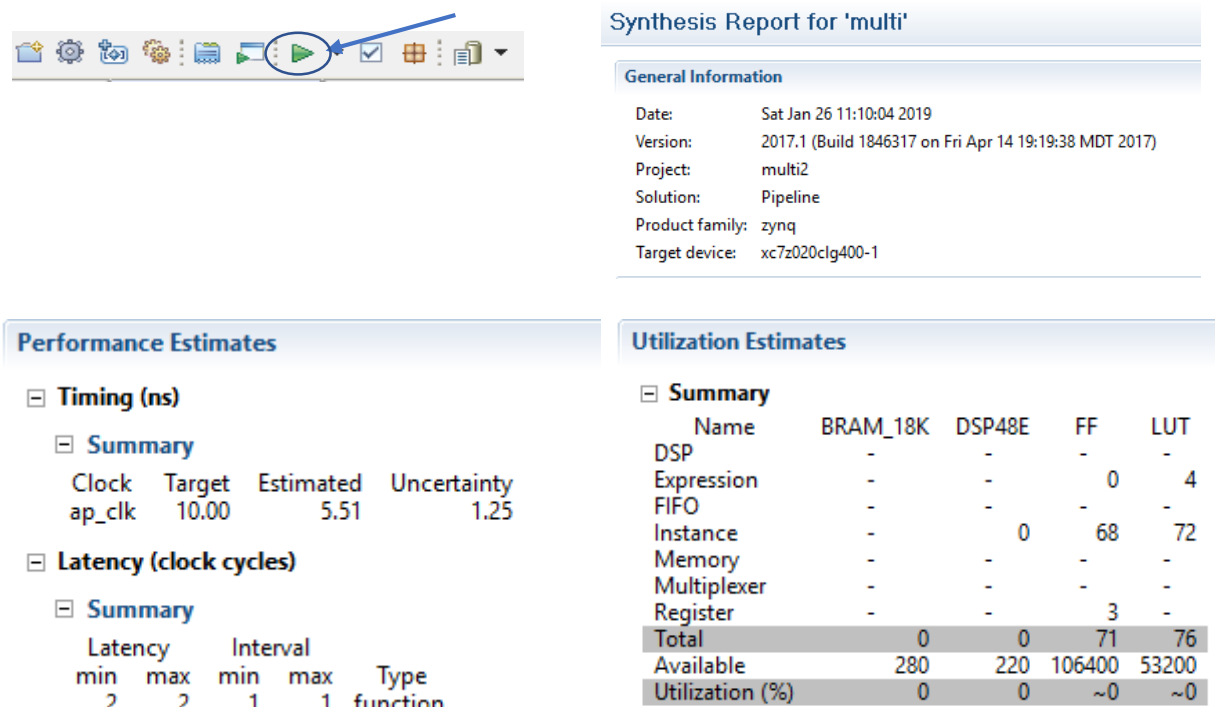


Figura 4.24: Resultados de la síntesis en Vivado HLS

**5.Testbench RTL:** Esta es una de las grandes ventajas de Vivado HLS, y es que nos permite testear el funcionamiento de la transformación HDL con el testbench utilizado previamente para el código C. Así, se evita así tener que realizar testbench en VHDL, que es muchas veces un proceso tedioso. El funcionamiento es el mismo, se compara el resultado que devuelve el código hardware con el resultado que queremos obtener y si coinciden la función main devolverá un 0.



**6.Exportar el diseño:** El último paso consiste en exportar el diseño realizado. En este caso vamos a exportarlo como un bloque IP (intellectual property) que podemos utilizar posteriormente en un sistema en Vivado y que podemos añadir a nuestros diseños. Al mismo tiempo se elabora un resumen de implementación con los parámetros del diseño obtenido.



```

Implementation tool: Xilinx Vivado v.2017.1
Project:          multi2
Solution:         Pipeline
Device target:    xc7z020clg400-1
Report date:      Sat Jan 26 11:21:17 +0100 2019

==== Post-Implementation Resource usage ====
SLICE:           20
LUT:             63
FF:             34
DSP:            0
BRAM:           0
SRL:            0
==== Final timing ====
CP required:     10.000
CP achieved post-synthesis: 5.479
CP achieved post-implementation: 6.454
Timing met
    
```

Figura 4.25: Report obtenido al generar bloque IP con Vivado HLS

**7. Añadir IP a Vivado:** Por último, solo nos queda ir a Vivado y añadir el IP generado con Vivado HLS, al catálogo de IP, y así, posteriormente podremos utilizar este bloque para los diseños que nosotros queramos. En nuestro caso lo utilizaremos para realizar un multiplicador y mediremos sus parámetros en comparación con los diseñados en VHDL.

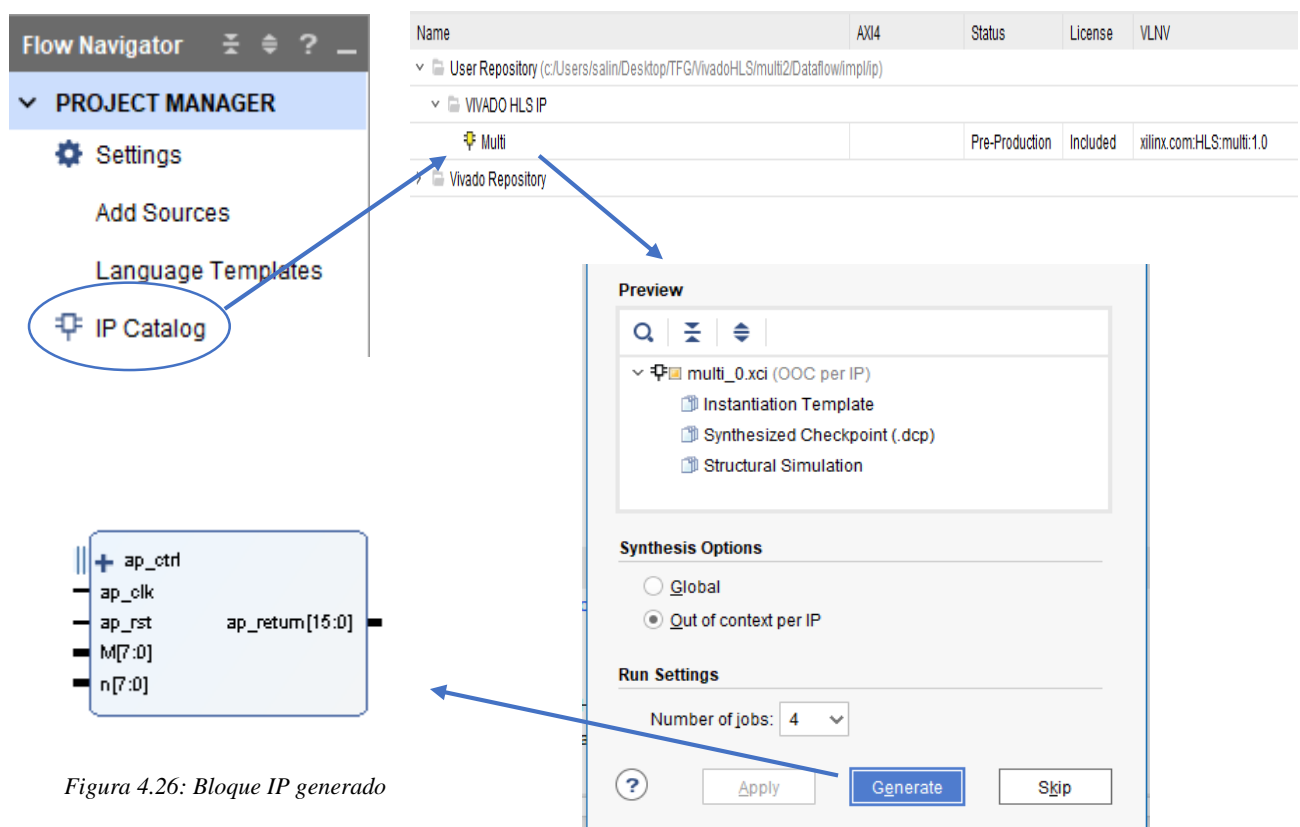


Figura 4.26: Bloque IP generado

Para el multiplicador de 16 bits es necesario añadir unas directivas adicionales que indiquen a la herramienta el tipo de recursos de la FPGA queremos utilizar. Si no utilizamos estas directivas adicionales, Vivado HLS utiliza los recursos DSP de la placa, que en este trabajo no estamos utilizando. De este modo, garantizamos una justa comparativa entre diseños.

```

1. #include<ap_cint.h>
2. int32 multi (int16 M, int16 n){
3. #pragma HLS RESOURCE variable=return core=Mul_LUT
4. #pragma HLS RESOURCE variable=n core=Mul_LUT
5. #pragma HLS RESOURCE variable=M core=Mul_LUT
6. #pragma HLS PIPELINE
7. int32 P;
8. P=M*n;
9. return P;
10. }

```

#### Resource Usage

	VHDL
SLICE	81
LUT	272
FF	205
DSP	0
BRAM	0
SRL	0

## 4.10. Medición de consumo

En esta sección se explica que herramientas vamos a utilizar y que proceso se ha seguido para medir el consumo real de los multiplicadores.

En primer lugar, para la entrada de datos en la FPGA vamos a utilizar la herramienta comentada anteriormente FPGA-in-the-Loop, utilizando fuentes de números aleatorios. Para que el modelo de entrada de datos sea lo más parecido posible al que utilizamos para estimar el consumo, la fuente consistirá en un LFSR con la misma estructura que utilizamos para generar el archivo SAIF. La simulación es realizada con un sample time de 10 nanosegundos. El sample time en Simulink es un parámetro que indica cuando cada bloque produce una salida y cambia su estado.

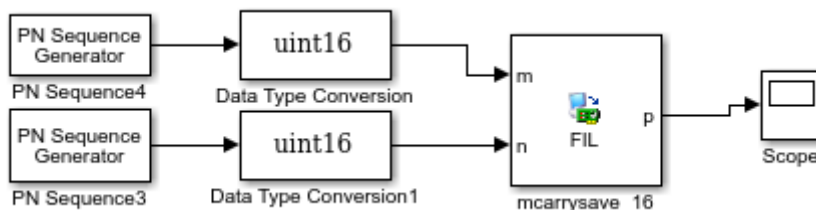


Figura 4.28: Modelo Simulink para medir el consumo

Generator polynomial:  
 $z^{16} + z^{15} + z^{13} + z^4 + 1$

Initial states:  
 [ 0 0 1 1 1 1 0 1 0 0 0 1 0 0 1 1 ]

Output mask source: Dialog parameter

Output mask vector (or scalar shift value):  
 0

☐ Output variable-size signals

Sample time:  
 0.00000001

Samples per frame:  
 1

☐ Reset on nonzero input

☒ Enable bit-packed outputs

Number of packed bits:  
 16

Figura 4.27: Configuración Fuente LFSR

Para medir el consumo vamos a utilizar una placa diseñada por un compañero para su Trabajo de fin de grado. Esta placa se conecta entre la alimentación y la FPGA, y mide la potencia

consumida por la misma, mediante un conjunto de resistencias de shunt, amplificadores de instrumentación y un conversor Analógico/Digital (ADC).

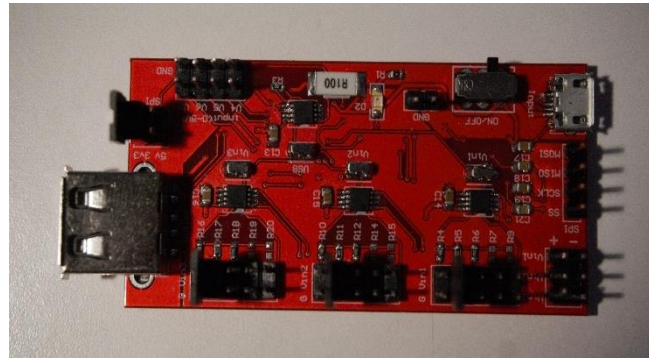


Figura 4.29: Placa utilizada para medir el consumo

Primero, cargamos el programa sobre la FPGA y le damos a simular durante un tiempo suficientemente largo, la placa recoge muestras durante 1 microsegundo, en este tiempo recoge la potencia instantánea consumida cada ciclo de reloj. Cabe destacar que las lecturas que hacemos no son en vatios, para pasarlas a vatios debemos aplicar la fórmula indicada posteriormente:

$$\text{Potencia(W)} = 6,055 \cdot 10^{-4} \cdot \text{Lectura}$$



Figura 4.30: Lectura de consumo de la placa

Un aspecto importante sobre la herramienta FIL, que está integrada en Simulink es que la velocidad a la que se producen las multiplicaciones no se produce a la velocidad de reloj, si no que el programa es capaz de introducir datos a una velocidad determinada en la simulación. Esto complica la medición del consumo porque hay que sincronizar la medición de potencia, con el momento en que se producen las multiplicaciones, hemos considerado recoger el consumo máximo a lo largo de la simulación para cada diseño. Los datos recogidos y su análisis se encuentran en el capítulo 5 de resultados.

## Capítulo 5: Resultados y discusión

### 5.1. Recursos

En esta sección se exponen los recursos de la FPGA utilizados por cada multiplicador para 4, 8 y 16 bits, estos recursos se dividen en LUT, LUTRAM, FF, IO y BUFG.

LUT: Look-up tables (LUTs) son utilizados para implementar generadores de función en CLBs. Estos generadores de función pueden implementar cualquier función Booleana de 5 o 6 entradas.

FF: Flip-flop, son elementos de memoria de un bit.

IO: Entradas y salidas de los multiplicadores. Las entradas son el reloj, el reset y dos operandos de  $n$  bits. La salida es el producto, que siempre es de  $2n$  bits.

BUFG: Global Clock Buffer, distribuye high fan-out clock signals a través de una FPGA.

#### **Recursos utilizados por multiplicador Ripplecarry de 4 bits, 8bits, 16 bits**

Resource	Utilization	Available	Utilization %
LUT	16	53200	0.03
FF	16	106400	0.02
IO	18	125	14.40
BUFG	1	32	3.13

Tabla 5.1: Recursos utilizados por el multiplacador Ripplecarry de 4 bits

Resource	Utilization	Available	Utilization %
LUT	95	53200	0.18
FF	32	106400	0.03
IO	34	125	27.20
BUFG	1	32	3.13

Tabla 5.2: Recursos utilizados por el multiplacador Ripplecarry de 8 bits

Resource	Utilization	Available	Utilization %
LUT	555	53200	1.04
FF	64	106400	0.06
IO	66	125	52.80
BUFG	1	32	3.13

Tabla 5.3: Recursos utilizados por el multiplacador Ripplecarry de 16 bits

### **Recursos utilizados por multiplicador Carrysave de 4 bits, 8bits, 16 bits**

Resource	Utilization	Available	Utilization %
LUT	15	53200	0.03
FF	16	106400	0.02
IO	18	125	14.40
BUFG	1	32	3.13

Tabla 5.4: Recursos utilizados por el multiplicador Carrysave de 4 bits

Resource	Utilization	Available	Utilization %
LUT	79	53200	0.15
FF	32	106400	0.03
IO	34	125	27.20
BUFG	1	32	3.13

Tabla 5.5: Recursos utilizados por el multiplicador Carrysave de 8 bits

Resource	Utilization	Available	Utilization %
LUT	397	53200	0.75
FF	64	106400	0.06
IO	66	125	52.80
BUFG	1	32	3.13

Tabla 5.6: Recursos utilizados por el multiplicador Carrysave de 16 bits

### **Recursos utilizados por multiplicador Wallace de 4 bits, 8bits, 16 bits**

Resource	Utilization	Available	Utilization %
LUT	18	53200	0.03
FF	16	106400	0.02
IO	18	125	14.40
BUFG	1	32	3.13

Tabla 5.7: Recursos utilizados por el multiplicador Wallace de 4 bits

Resource	Utilization	Available	Utilization %
LUT	85	53200	0.16
FF	32	106400	0.03
IO	34	125	27.20
BUFG	1	32	3.13

Tabla 5.8: Recursos utilizados por el multiplicador Wallace de 8 bits

Resource	Utilization	Available	Utilization %
LUT	370	53200	0.70
FF	64	106400	0.06
IO	66	125	52.80
BUFG	1	32	3.13

Tabla 5.9: Recursos utilizados por el multiplicador Wallace de 16 bits

### **Recursos utilizados por multiplicador Booth de 4 bits, 8bits, 16 bits**

Resource	Utilization	Available	Utilization %
LUT	17	53200	0.03
FF	15	106400	0.01
IO	17	125	13.60
BUFG	1	32	3.13

Tabla 5.10: Recursos utilizados por el multiplicador Booth de 4 bits

Resource	Utilization	Available	Utilization %
LUT	126	53200	0.24
FF	31	106400	0.03
IO	33	125	26.40
BUFG	1	32	3.13

Tabla 5.11: Recursos utilizados por el multiplicador Booth de 8 bits

Resource	Utilization	Available	Utilization %
LUT	569	53200	1.07
FF	64	106400	0.06
IO	65	125	52.00
BUFG	1	32	3.13

Tabla 5.12: Recursos utilizados por el multiplicador Booth de 16 bits

### **Recursos utilizados por multiplicador Baugh-Wooley de 4 bits, 8bits, 16 bits**

Resource	Utilization	Available	Utilization %
LUT	18	53200	0.03
FF	16	106400	0.02
IO	18	125	14.40
BUFG	1	32	3.13

Tabla 5.13: Recursos utilizados por el multiplicador Baugh-Wooley de 4 bits



Resource	Utilization	Available	Utilization %
LUT	75	53200	0.14
FF	32	106400	0.03
IO	34	125	27.20
BUFG	1	32	3.13

Tabla 5.14: Recursos utilizados por el multiplicador Baugh-Wooley de 8 bits

Resource	Utilization	Available	Utilization %
LUT	393	53200	0.74
FF	64	106400	0.06
IO	66	125	52.80
BUFG	1	32	3.13

Tabla 5.15: Recursos utilizados por el multiplicador Baugh-Wooley de 16 bits

### **Recursos utilizados por multiplicador de Vivado HLS 4 bits, 8bits, 16 bits**

Resource	Utilization	Available	Utilization %
LUT	23	53200	0.04
FF	16	106400	0.02
IO	23	125	18.40
BUFG	1	32	3.13

Tabla 5.16: Recursos utilizados por el multiplicador de Vivado HLS de 4 bits

Resource	Utilization	Available	Utilization %
LUT	63	53200	0.12
FF	36	106400	0.03
IO	39	125	31.20
BUFG	1	32	3.13

Tabla 5.17: Recursos utilizados por el multiplicador de Vivado HLS de 8 bits

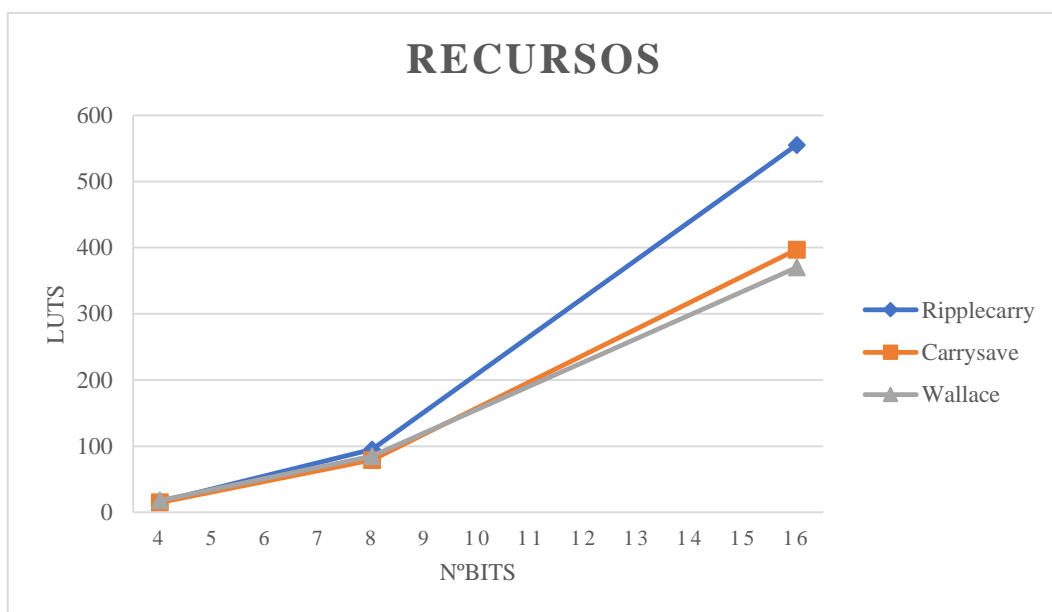
Resource	Utilization	Available	Utilization %
LUT	269	53200	0.51
FF	67	106400	0.06
IO	70	125	56.00
BUFG	1	32	3.13

Tabla 5.18: Recursos utilizados por el multiplicador de Vivado HLS de 16 bits

## Análisis Área ocupada

Las tablas anteriores muestran los recursos de todos los multiplicadores para 4, 8 y 16 bits. Para ponerlos en contexto y comparar las distintas arquitecturas, se muestran las dos gráficas inferiores que muestran el número LUTs que ocupa cada solución a medida que aumentamos el número de bits. Solo se representa el número de LUTs, ya que tanto puertos de entrada/salida, como el número de flip-flops son idénticos para todos los multiplicadores, que en este caso corresponden a los registros que se ponen a la entrada y la salida.

Como podemos observar, el Ripplecarry es el diseño menos eficiente en cuanto al número de recursos, luego el Carrysave y por el último, el Wallace. Otra observación, es que el número de LUTs no crece linealmente, si no que se aproxima a una función exponencial.



*Figura 5.1: LUTs utilizados por los multiplicadores sin signo*

Con los multiplicadores con signo ocurre exactamente lo mismo en cuanto al crecimiento de LUTs a medida que aumenta el número de bits. En este caso se observa que el multiplicador de Baugh-Wooley utiliza menos LUTs que el multiplicador Booth con el mismo número de bits, y que el multiplicador más eficiente en cuanto al número de recursos es el generado por Vivado HLS.

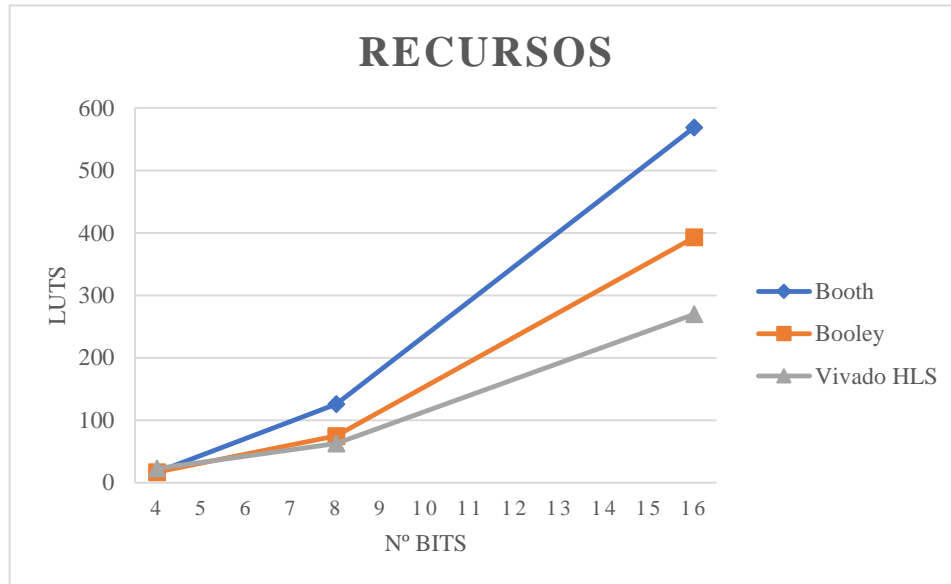


Figura 5.2: LUTs utilizados por los multiplicadores con signo

## 5.2. Timing

En este capítulo se exponen las frecuencias máximas de funcionamiento que alcanzan los multiplicadores para 4,8 y 16 bits.

Se puede observar que para 4 bits, las diferencias en el máximo throughput alcanzado por las distintas arquitecturas no es significativa, si bien los multiplicadores con signo alcanza mayor velocidad de funcionamiento. Además, el diseño en Vivado HLS es más lento que los diseñados en VHDL.

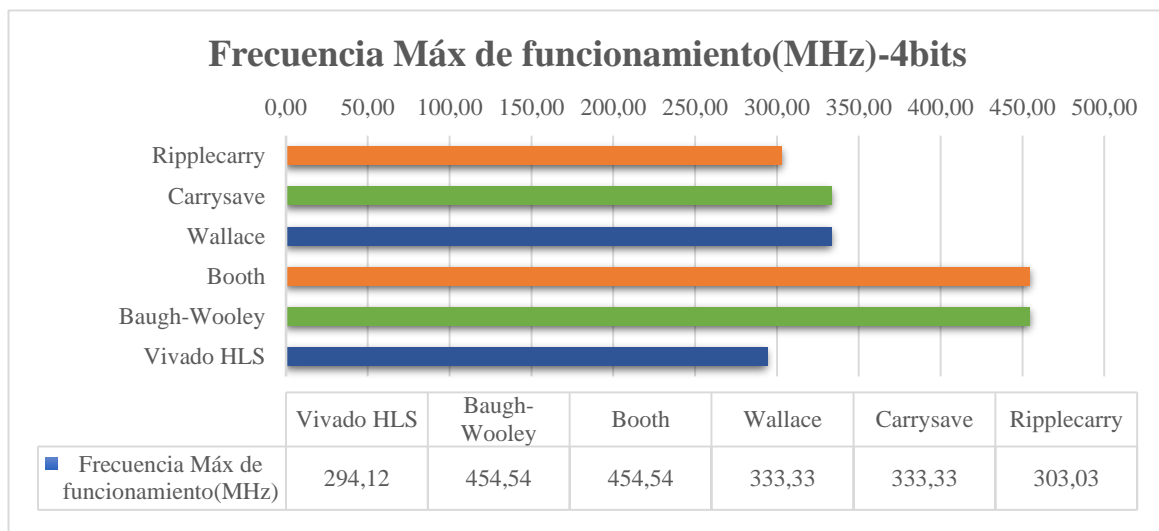
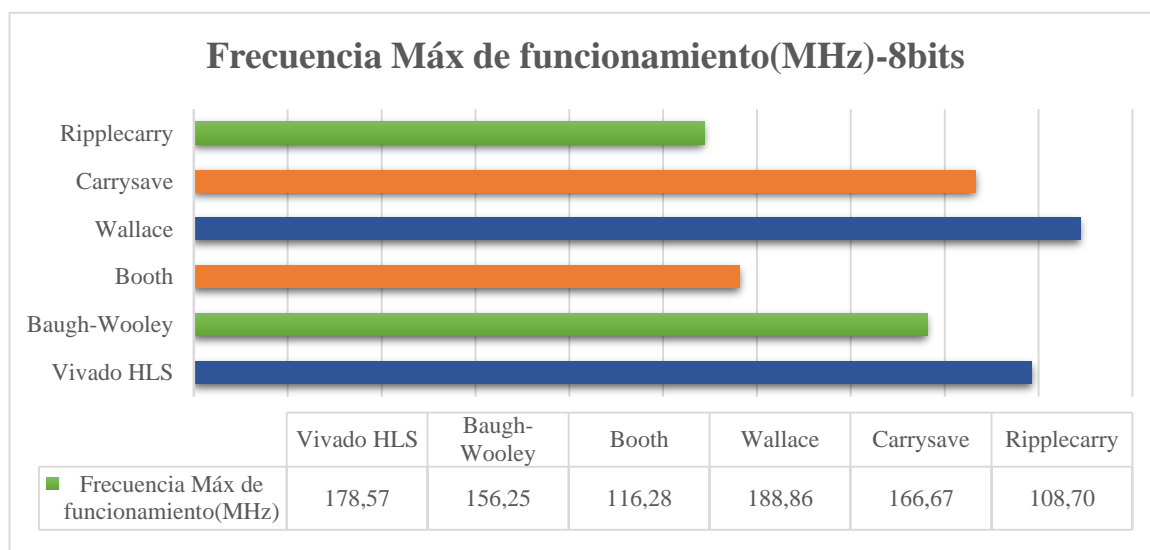


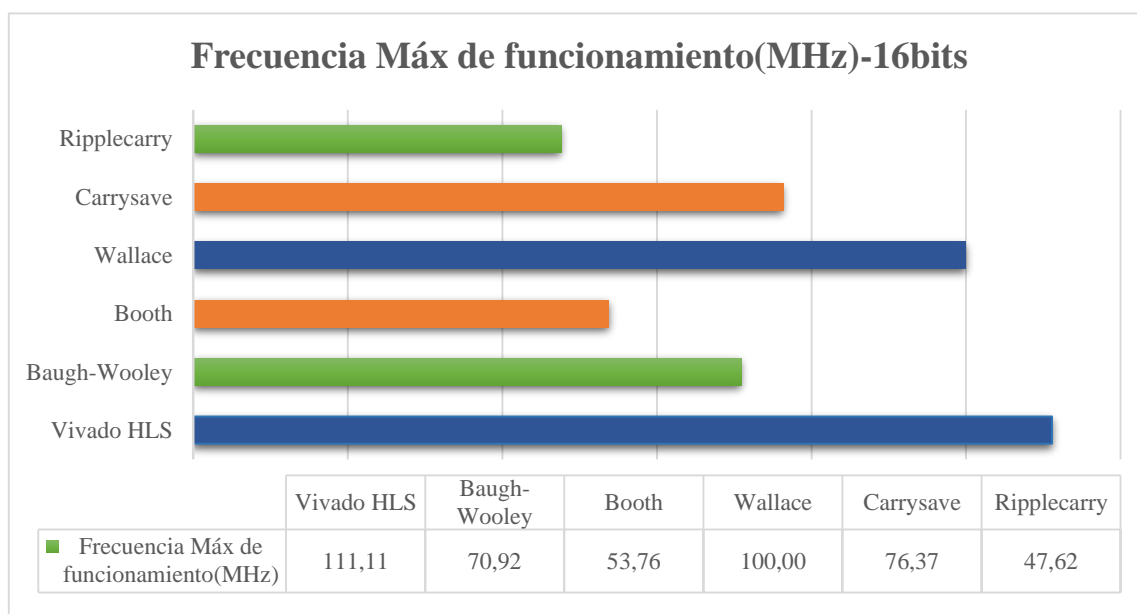
Figura 5.3: Frecuencia máxima de funcionamiento multiplicadores de 4bits

En la gráfica para 8 bits, los resultados obtenidos son distintos que, para 4 bits, y las diferencias arquitecturales de cada diseño empiezan a plasmarse. En este caso vemos como, el multiplicador de Wallace alcanza la mayor velocidad, con una diferencia considerable. En los multiplicadores con signo el multiplicador Vivado HLS alcanza una frecuencia de funcionamiento ostensiblemente mayor, siendo el que menos velocidad alcanza el multiplicador de Booth



*Figura 5.4: Frecuencia máxima de funcionamiento multiplicadores de 8 bits*

En el caso de 16 bits, ocurre algo similar que, en el caso de 8 bits, es decir, de nuevo se aprecian notables diferencias de velocidad entre los distintos multiplicadores. Sin embargo, estas diferencias en la velocidad máxima alcanzada se vuelven todavía más notorias.



*Figura 5.5: Frecuencia máxima de funcionamiento multiplicadores de 16 bits*

Como cabía esperar, la velocidad alcanzada va reduciéndose considerablemente a medida que aumenta el número de bits para el mismo diseño, en las gráficas inferiores se puede observar más claramente este efecto.

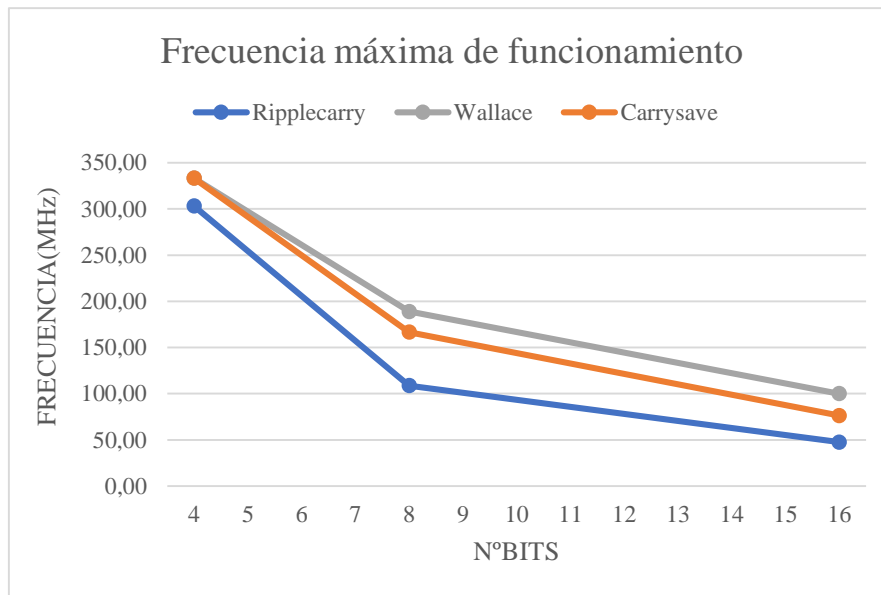


Figura 5.6: Descenso de la frecuencia de funcionamiento con el n° de bits

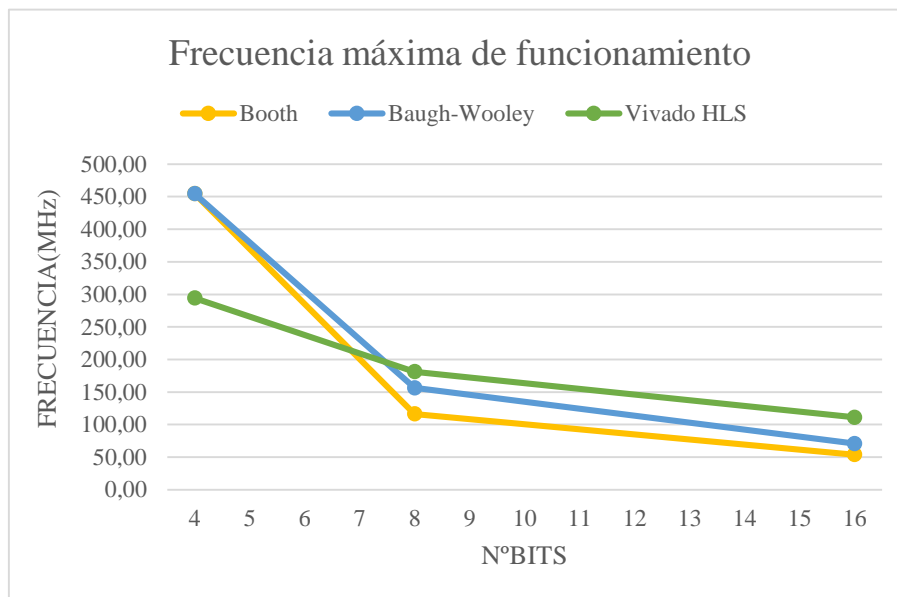


Figura 5.7: Descenso de la frecuencia de funcionamiento con el n° de bits

### 5.3. Pipelining

La frecuencia máxima que alcanza el reloj de la FPGA utilizada es de 200MHz. Como se puede observar en las gráficas mostradas anteriormente, no todos los multiplicadores son capaces de alcanzar esta velocidad, sobre todo cuando crece el número de bits. De hecho, ninguno de los multiplicadores con 8 bits es capaz de alcanzar los 200MHz, y ninguno de los multiplicadores de 16 bits es capaz de alcanzar los 150 MHz. En consecuencia, se ha utilizado la técnica

mencionada anteriormente **Pipelining** para conseguir aumentar el throughput de los multiplicadores, viendo cuantas etapas son necesarias para alcanzar la frecuencia objetivo y como afecta esta técnica al aumento de recursos y consumo.

Número de etapas pipeline hasta alcanzar los 200MHz			
4	8	16	N.º bits
0	2	15	Ripplecarry
0	1	5	Carrysave
0	1	3	Wallace
0	1	4	Booley
0	2	No alcanza	Booth

En la gráfica inferior se observa el aumento de la frecuencia máxima de funcionamiento haciendo pipelining, se representa la frecuencia normal, sin ninguna etapa, añadiendo una etapa de pipelining y añadiendo registros en todas las etapas. Se puede observar que puede conseguirse un gran aumento del throughput del sistema con esta técnica, llegando a doblar o triplicar la capacidad de procesamiento de los multiplicadores.

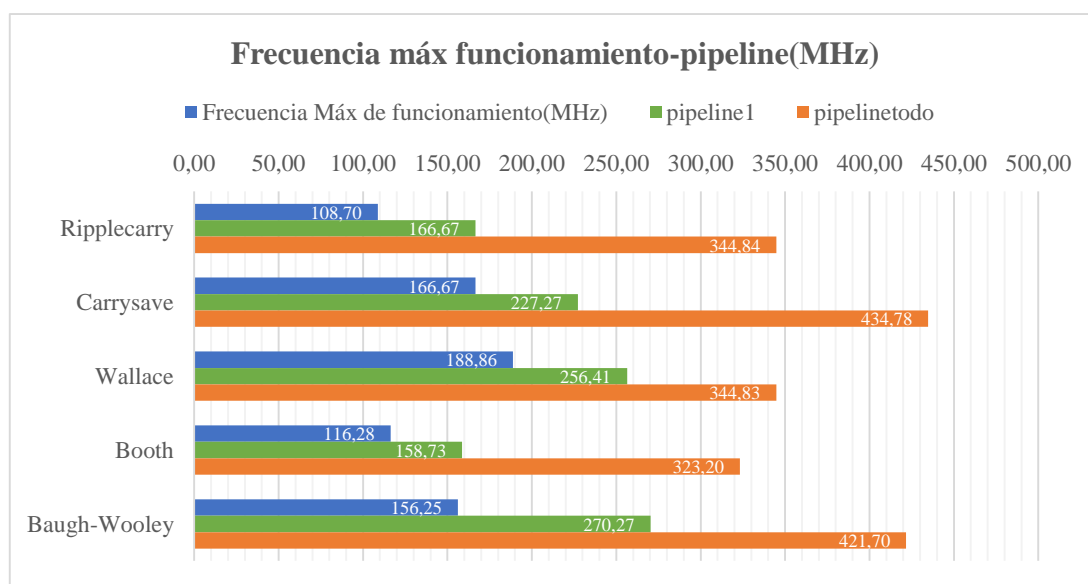


Figura 5.8: Aumento del throughput por pipeline

En la gráfica 5.9 se puede observar el crecimiento del throughput del sistema para el multiplicador Carrysave. A medida que vamos aumentando el número de etapas, va aumentando el throughput del sistema. Pero se puede observar que este aumento no ocurre linealmente, sino que hay algunas etapas en las que se produce un salto considerable, mientras en otras se mantiene casi constante.

Esto se debe a que todos los multiplicadores necesitan un sumador ripple-carry al final de su estructura para terminar la multiplicación. Este sumador es el que limita la velocidad de todos

los diseños en última instancia, es decir, determina el camino crítico. Por esta razón, en las etapas en las que añadimos un registro que reduce el camino que incluye el ripple-carry se observa un aumento sustancial del throughput. A partir de la cuarta etapa se observa que la frecuencia de funcionamiento se mantiene prácticamente constante, ya que hemos alcanzado el límite de velocidad delimitado por el sumador ripple-carry.

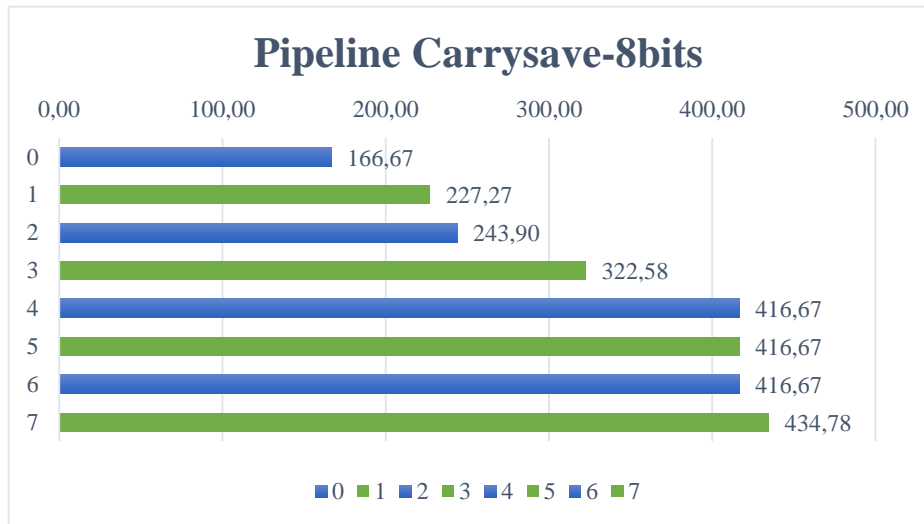


Figura 5.9: Aumento del throughput por pipeline del multiplicador Carrysave

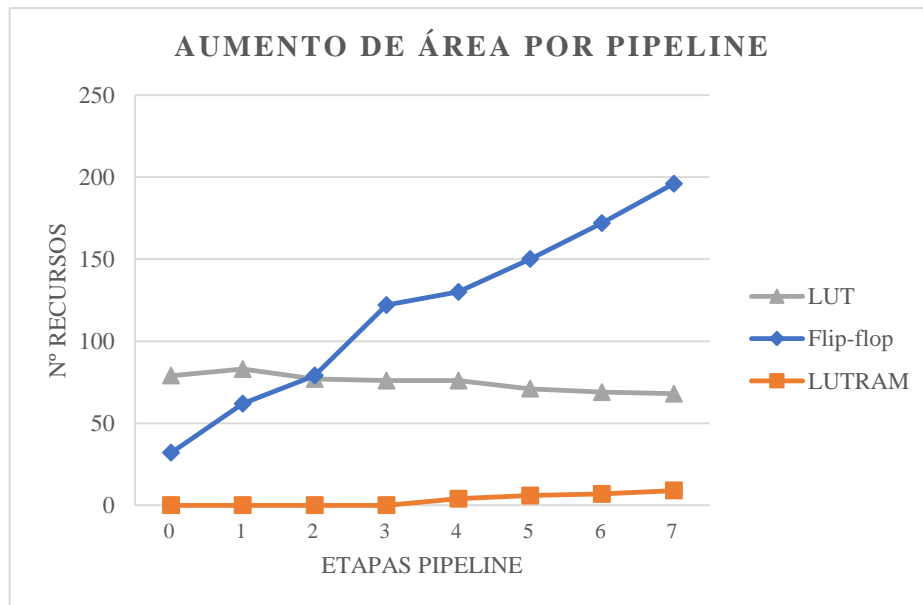


Figura 5.10: Aumento del número de recursos por pipeline

Vemos como, además, al añadir etapas de pipelining, se produce un aumento del área. Como se menciona en la descripción de esta técnica, el número de flip-flops aumenta linealmente y el número de LUTs se podría suponer constante ya en principio las unidades de procesamiento necesarias son las mismas. Sin embargo, en la gráfica vemos que se reduce con el número de etapas, aunque es prácticamente inapreciable

## 5.4. Consumo

### 5.4.1. Estimación de consumo

A continuación, se exponen los resultados obtenidos de la estimación de consumo con la herramienta Report Power de Vivado para los seis multiplicadores con 4, 8 y 16 bits a 25,50,100,150 y 200MHz. Las gráficas de potencia estimada, compara los resultados obtenidos según el número de bits. Además, se añade una última tabla donde se muestra la correlación y la covarianza, entre la frecuencia de funcionamiento y la estimación de consumo.

#### Estimación de potencia multiplicador Ripplecarry 4 bits, 8bits, 16 bits

Tabla 5.18: Estimación de potencia para el multiplicador Ripplecarry de 4bits

Ripplecarry-4bits						
Frecuencias (MHz)		25	50	100	150	200
Estimación(W)	Potencia estática	0,12	0,12	0,12	0,12	0,12
	Potencia dinámica	0,002	0,004	0,008	0,011	0,015
	Potencia total	0,122	0,124	0,128	0,131	0,135

Tabla 5.19: Estimación de potencia para el multiplicador Ripplecarry de 8bits

Ripplecarry-8bits						
Frecuencias (MHz)		25	50	100	150	200
Estimación(W)	Potencia estática	0,122	0,122	0,122	0,122	0,122
	Potencia dinámica	0,003	0,008	0,018	0,026	0,037
	Potencia total	0,125	0,13	0,14	0,148	0,159

Tabla 5.20: Estimación de potencia para el multiplicador Ripplecarry de 16bits

Ripplecarry-16bits						
Frecuencias (MHz)		25	50	100	150	200
Estimación (W)	Potencia estática	0,122	0,122	0,123	0,124	0,125
	Potencia dinámica	0,017	0,034	0,067	0,095	0,132
	Potencia total	0,139	0,156	0,19	0,219	0,257

Bits	Correlación	Covarianza
4bits	0,99859569	0,37625
8bits	0,99876565	0,95875
16bits	0,99859498	3,07625

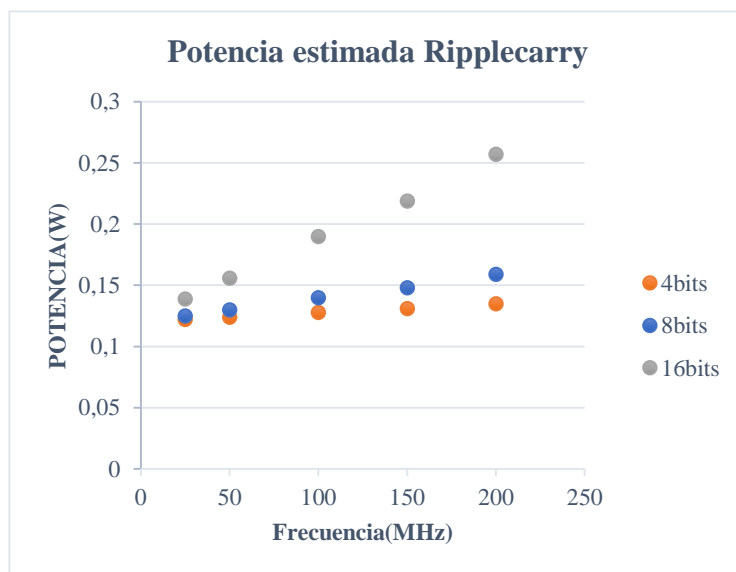


Figura 5.11: Potencia estimada del multiplicador Ripplecarry por Vivado



### Estimación de potencia multiplicador Carrysave 4 bits, 8bits, 16 bits

Tabla 5.21: Estimación de potencia para el multiplicador Carrysave de 4 bits

Carrysave-4bits						
Frecuencias (MHz)		25	50	100	150	200
Estimación(W)	Potencia estática	0,12	0,12	0,12	0,12	0,12
	Potencia dinámica	0,002	0,004	0,007	0,011	0,015
	Potencia total	0,122	0,124	0,127	0,131	0,135

Tabla 5.22: Estimación de potencia para el multiplicador Carrysave de 8bits

Carrysave-8bits						
Frecuencias (MHz)		25	50	100	150	200
Estimación(W)	Potencia estática	0,122	0,122	0,122	0,122	0,122
	Potencia dinámica	0,003	0,007	0,017	0,025	0,036
	Potencia total	0,125	0,129	0,139	0,147	0,158

Tabla 5.13: Estimación de potencia para el multiplicador Carrysave de 16 bits

Carrysave-16bits						
Frecuencias (MHz)		25	50	100	150	200
Estimación(W)	Potencia estática	0,122	0,122	0,123	0,124	0,125
	Potencia dinámica	0,015	0,03	0,06	0,084	0,119
	Potencia total	0,137	0,152	0,183	0,208	0,244

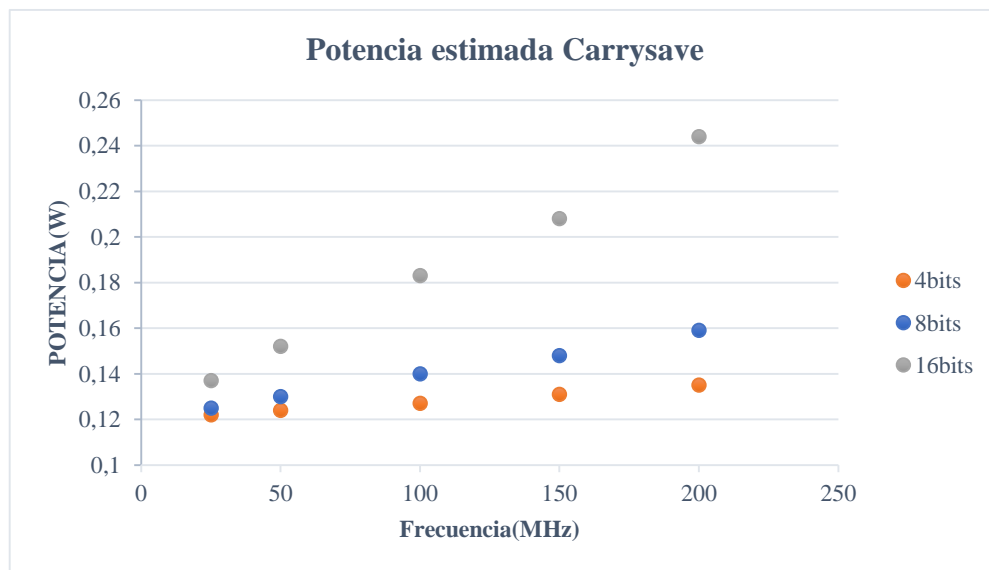


Figura 5.12: Potencia estimada del multiplicador Carrysave por Vivado

Bits	Correlación	Covarianza
4bits	0,998890737	0,375
8bits	0,998978063	0,97875
16bits	0,999238087	3,405

## Estimación de potencia multiplicador Wallace 4 bits, 8bits, 16 bits

Tabla 5.24: Estimación de potencia para el multiplicador Wallace de 4bits

Wallace-4bits						
Frecuencias (MHz)		25	50	100	150	200
Estimación(W)	Potencia estática	0,12	0,12	0,12	0,12	0,12
	Potencia dinámica	0,002	0,004	0,008	0,011	0,015
	Potencia total	0,122	0,124	0,128	0,131	0,135

Tabla 5.25: Estimación de potencia para el multiplicador Wallace de 8 bits

Wallace-8bits						
Frecuencias (MHz)		25	50	100	150	200
Estimación(W)	Potencia estática	0,122	0,122	0,122	0,122	0,122
	Potencia dinámica	0,003	0,007	0,017	0,025	0,035
	Potencia total	0,125	0,129	0,139	0,147	0,157

Tabla 5.26: Estimación de potencia para el multiplicador Wallace de 16 bits

Wallace-16bits						
Frecuencias (MHz)		25	50	100	150	200
Estimación(W)	Potencia estática	0,122	0,122	0,123	0,124	0,125
	Potencia dinámica	0,013	0,026	0,057	0,074	0,107
	Potencia total	0,135	0,148	0,18	0,198	0,232

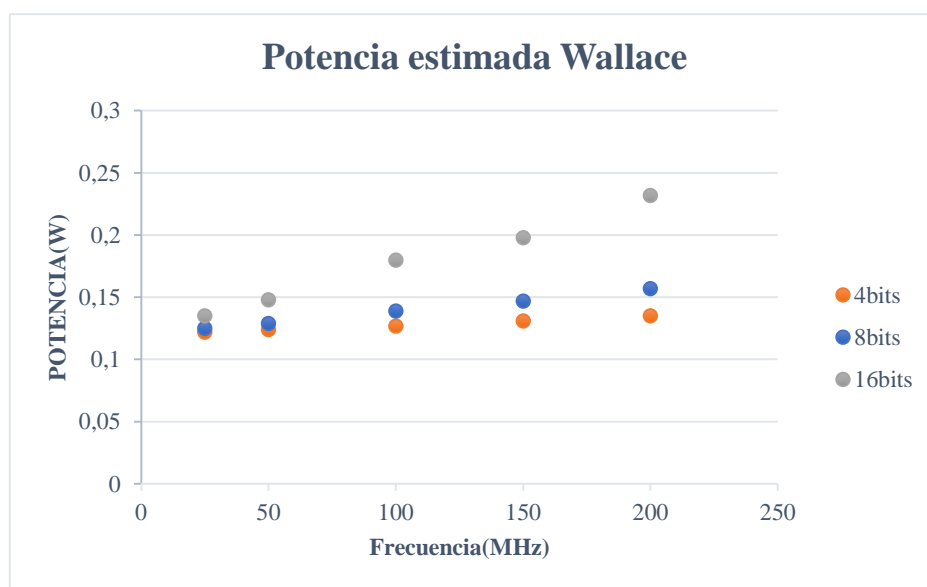


Figura 5.13: Potencia estimada del multiplicador Wallace por Vivado

Estimación	Correlación	Covarianza
4bits	0,998890737	0,375
8bits	0,999357193	0,935
16bits	0,999357193	3,07625

### Estimación de potencia multiplicador Booth 4 bits, 8bits, 16 bits

Tabla 5.27: Estimación de potencia para el multiplicador Booth de 4 bits

Booth-4bits						
Frecuencias (MHz)		25	50	100	150	200
Estimación(W)	Potencia estática	0,12	0,12	0,12	0,12	0,12
	Potencia dinámica	0,002	0,005	0,009	0,013	0,018
	Potencia total	0,122	0,125	0,129	0,133	0,138

Tabla 5.28: Estimación de potencia para el multiplicador Booth de 8 bits

Booth-8bits						
Frecuencias (MHz)		25	50	100	150	200
Estimación(W)	Potencia estática	0,122	0,122	0,122	0,122	0,122
	Potencia dinámica	0,003	0,009	0,02	0,03	0,043
	Potencia total	0,125	0,131	0,142	0,152	0,165

Tabla 5.29: Estimación de potencia para el multiplicador Booth de 16 bits

Booth-16bits						
Frecuencias (MHz)		25	50	100	150	200
Estimación(W)	Potencia estática	0,122	0,122	0,123	0,124	0,125
	Potencia dinámica	0,017	0,034	0,065	0,092	0,128
	Potencia total	0,139	0,156	0,188	0,216	0,253

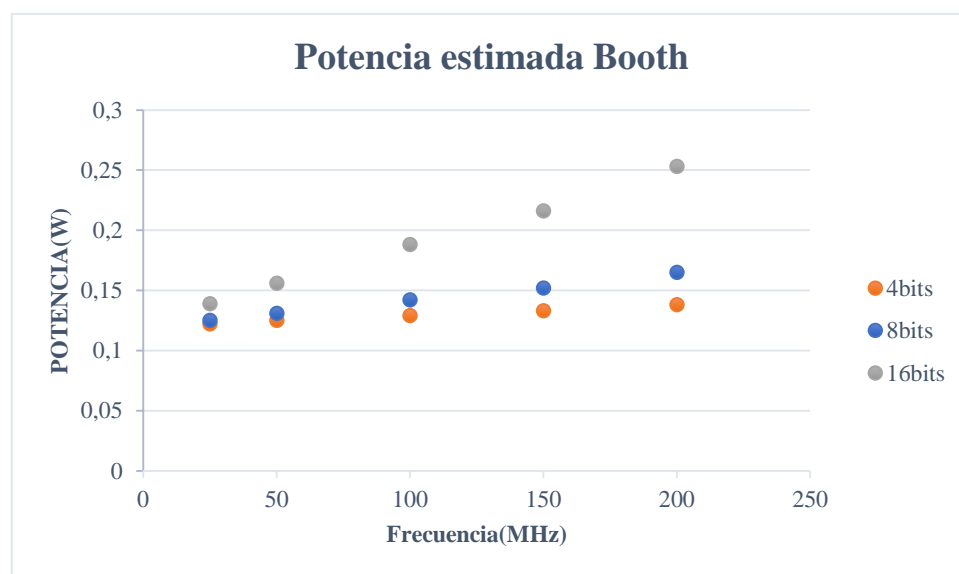


Figura 5.14: Potencia estimada del multiplicador de Booth por Vivado

Estimación	Correlación	Covarianza
4 bits	0,99939859	1,27625
8bits	0,999501607	2,59375
16bits	0,998577116	6,21125

## Estimación de potencia multiplicador Baugh-Wooley 4 bits, 8bits, 16 bits

Tabla 5.30: Estimación de potencia para el multiplicador Baugh-Wooley de 4bits

Baugh-Wooley-4bits						
Frecuencias (MHz)		25	50	100	150	200
Estimación(W)	Potencia estática	0,12	0,12	0,12	0,12	0,12
	Potencia dinámica	0,003	0,005	0,01	0,015	0,02
	Potencia total	0,123	0,125	0,13	0,135	0,14

Tabla 5.31: Estimación de potencia para el multiplicador Baugh-Wooley de 8 bits

Baugh-Wooley-8bits						
Frecuencias (MHz)		25	50	100	150	200
Estimación(W)	Potencia estática	0,122	0,122	0,122	0,122	0,122
	Potencia dinámica	0,003	0,009	0,019	0,027	0,039
	Potencia total	0,125	0,131	0,141	0,149	0,161

Tabla 5.32: Estimación de potencia para el multiplicador Baugh-Wooley de 16bits

Baugh-Wooley-16bits						
Frecuencias (MHz)		25	50	100	150	200
Estimación(W)	Potencia estática	0,122	0,122	0,123	0,124	0,125
	Potencia dinámica	0,016	0,031	0,061	0,087	0,12
	Potencia total	0,138	0,153	0,184	0,211	0,245

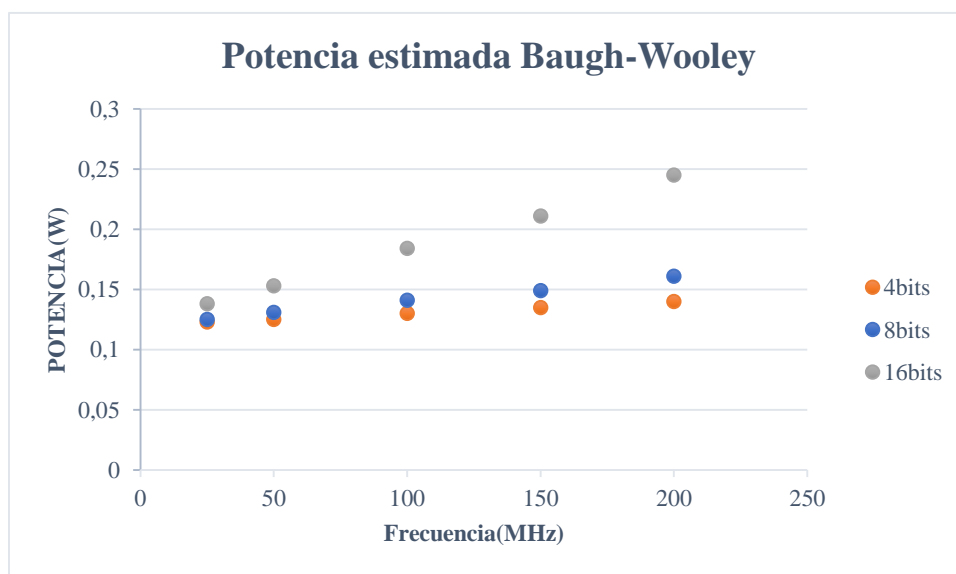


Figura 5.15: Potencia estimada del multiplicador Baugh-Wooley por Vivado

Estimación	Correlación	Covarianza
4 bits	0,99939859	1,27625
8bits	0,999459235	2,665
16bits	0,999459235	5,755

### Estimación de potencia multiplicador Vivado HLS 4 bits, 8bits, 16 bits

Tabla 5.33: Estimación de potencia para el multiplicador de Vivado HLS de 4bits

Vivado HLS-4bits						
Frecuencias (MHz)		25	50	100	150	200
Estimación(W)	Potencia estática	0,12	0,12	0,12	0,12	0,12
	Potencia dinámica	0,003	0,005	0,01	0,015	0,021
	Potencia total	0,123	0,125	0,13	0,135	0,141

Tabla 5.34: Estimación de potencia para el multiplicador de Vivado HLS de 8bits

Vivado HLS-8bits						
Frecuencias (MHz)		25	50	100	150	200
Estimación(W)	Potencia estática	0,122	0,122	0,122	0,122	0,122
	Potencia dinámica	0,003	0,008	0,018	0,026	0,038
	Potencia total	0,125	0,13	0,14	0,148	0,16

Tabla 5.35: Estimación de potencia para el multiplicador de Vivado HLS de 16 bits

Vivado HLS-16bits						
Frecuencias (MHz)		25	50	100	150	200
Estimación(W)	Potencia estática	0,122	0,122	0,123	0,124	0,125
	Potencia dinámica	0,016	0,03	0,058	0,082	0,114
	Potencia total	0,138	0,152	0,181	0,206	0,239

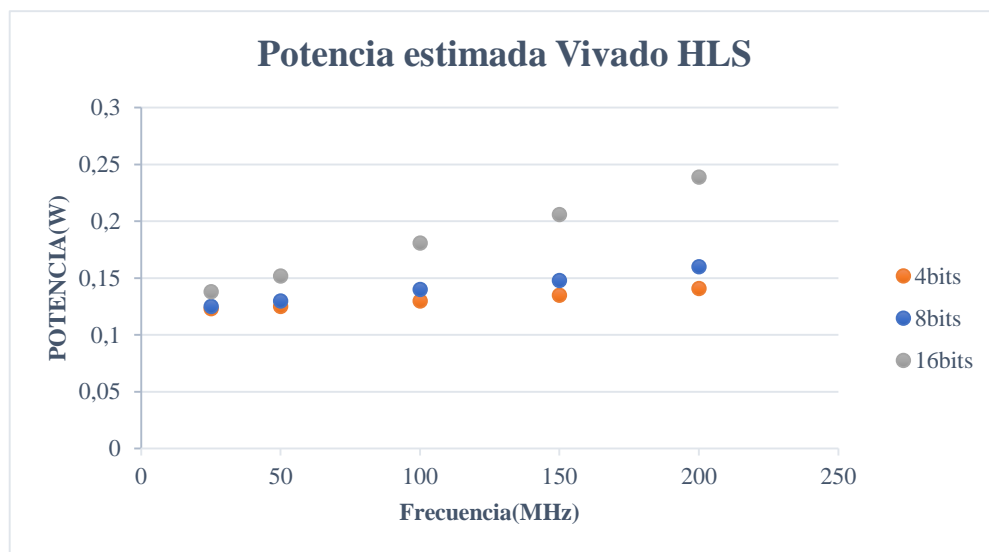


Figura 5.16: Potencia estimada del multiplicador Vivado HLS por Vivado:

Estimación	Correlación	Covarianza
4 bits	0,998494164	0,52625
8bits	0,998216857	1,0025
16bits	0,999140174	2,9175

## Análisis Estimación de consumo

En las gráficas 5.17 y 5.18 podemos ver la estimación de potencia total para cada multiplicador respecto a la frecuencia. Se puede observar que la evolución del consumo respecto a la frecuencia es similar para todos los diseños y ocurre de forma lineal. Además, se puede ver que las diferencias son pequeñas entre los distintos diseños siendo los más eficientes el multiplicador de Wallace y el generado por Vivado HLS.

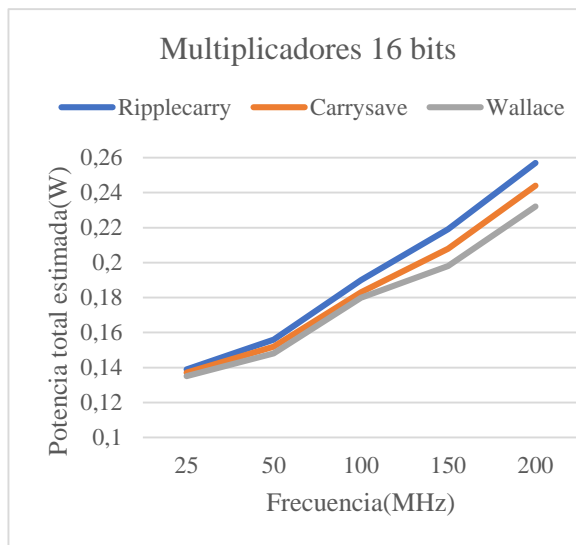


Figura 5.17: Potencia total estimada multiplicadores sin signo

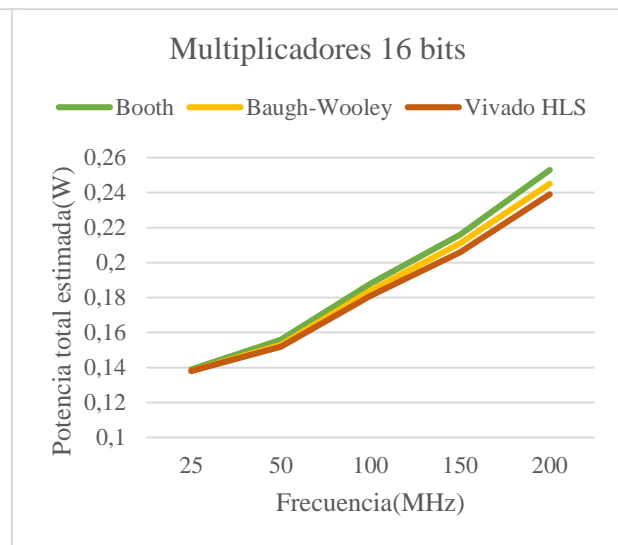


Figura 5.18: Potencia total estimada multiplicadores con signo

En las gráficas 5.19 y 5.20 podemos ver la estimación de potencia dinámica para cada multiplicador respecto a la frecuencia. En este caso, podemos ver como la evolución de la potencia dinámica es idéntica a la evolución de la potencia total. Este resultado es consecuencia de que la potencia estática se mantiene prácticamente constante con el aumento de la frecuencia, y es idéntica para todos los multiplicadores.

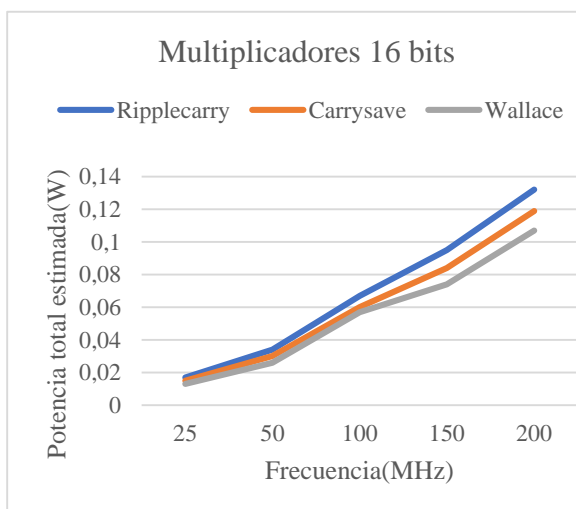


Figura 5.19: Potencia dinámica estimada multiplicadores sin signo

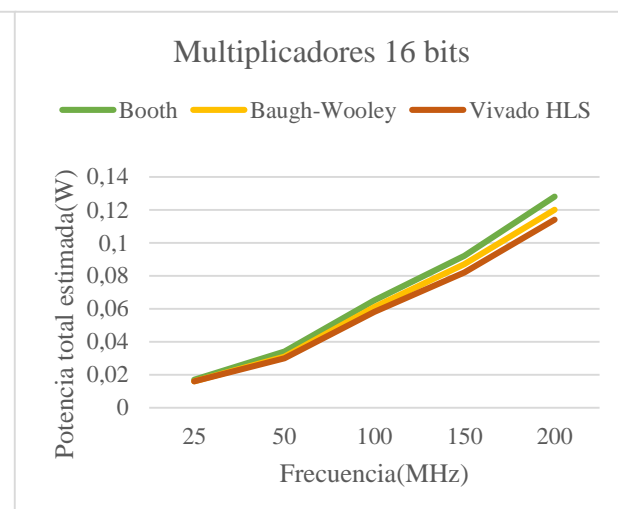


Figura 5.20.: Potencia dinámica estimada multiplicadores con signo

A continuación, se analiza el multiplicador Carrysave de 16 bits, aunque ocurre un fenómeno similar para todos los diseños. En la gráfica inferior, se observa como la potencia estática se mantiene prácticamente constante con el aumento de la frecuencia. Es la potencia dinámica la que provoca el aumento de la potencia total consumida, a partir de los 200MHz la potencia dinámica es mayor que la potencia estática y esta diferencia se hará mayor a medida que aumentemos la frecuencia.

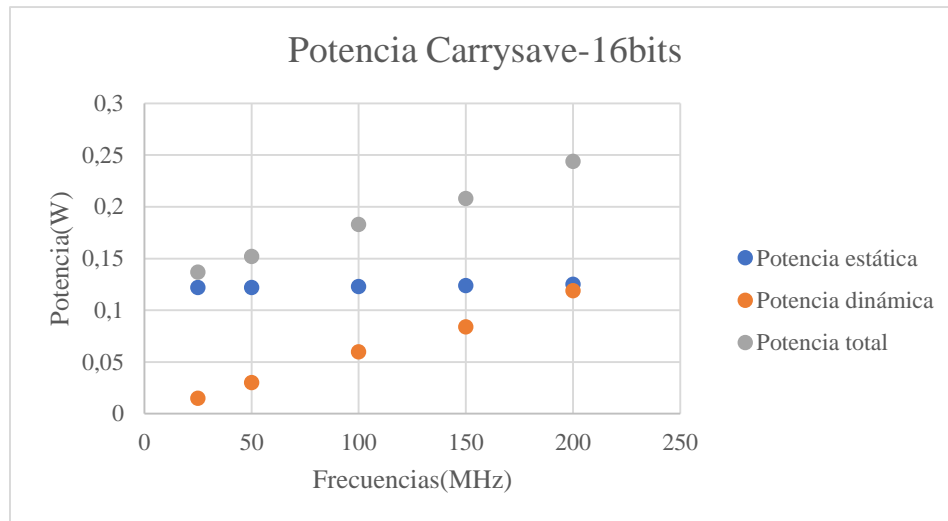


Figura 5.21: Potencia estática, dinámica y total

En la imagen inferior podemos ver como el consumo de potencia dinámica se encuentra en el 49% para 200MHz. Si desgranamos la potencia dinámica, se puede dividir en la consumida por el reloj (3%), la consumida por las señales (20%), la consumida por la lógica interna (11%) y la que abarca mayor porcentaje, los puertos de entrada/salida (66%). Estos porcentajes apenas varían, aunque variemos la frecuencia, incluso entre los distintos diseños. El gran consumo de los puertos se debe a que estos llevan una serie de drivers para dar la potencia suficiente para la actuación sobre los elementos exteriores del circuito.

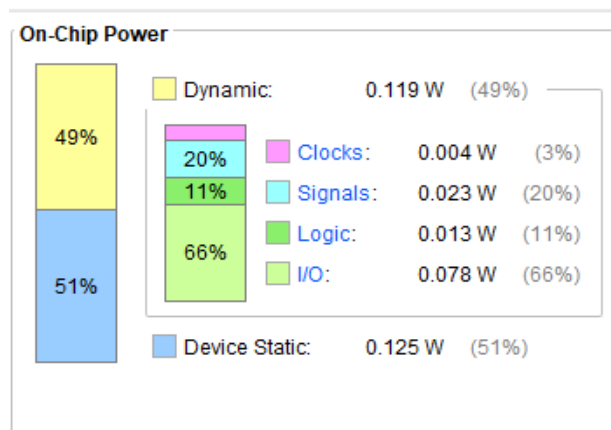


Figura 5.22: Consumo Carrysave de 16bits a 200MHz

Los puertos de entrada y salida, y el reloj son los mismos para todos los diseños y representan el 69% de la potencia dinámica. El 31% restante corresponde a las señales internas y la lógica, que sí presenta diferencias según el multiplicador. Por lo que se puede ver, que, modificando la arquitectura interna de los diseños, solo podemos influir en el 15,5% del consumo total. Esto puede explicar las pequeñas diferencias que observamos en el consumo entre los diseños y se puede deducir, que hay un consumo inherente a las estructuras de los multiplicadores que hemos diseñado.

### 5.4.2. Consumo Real

En este punto se muestra el consumo real de los multiplicadores, el procedimiento para medirlos se explica el punto 4.10. El sample time utilizado en la simulación es de 10 nanosegundos y la potencia estimada se corresponde con una frecuencia de 100 MHz. Los multiplicadores a los que se les ha medido el consumo son los de 16 bits.

#### Multiplicadores 16 bits

Consumo(W)	Lecturas	Potencia(W)	Estimación(W)
Ripplecarry	2685	1,625	0,19
Carrysave	2640	1,598	0,183
Wallace	2754	1,667	0,18
Booth	3384	2,049	0,188
Baugh-Wooley	2690	1,628	0,184
Vivado HLS IP	2597	1,572	0,181

Como podemos ver, los datos de medición de consumo en la realidad no son consistentes con los datos estimados por la herramienta Vivado. Esto podría suponer que la estimación no se ha realizado de manera correcta. Sin embargo, la diferencia entre los resultados es demasiado significativa, por lo que nos llevar a pensar que FPGA-in-the-Loop introduce un consumo extra en nuestra FPGA. Esto, unido a la dificultad para sincronizar las mediciones, nos lleva a concluir, que es una herramienta muy útil para testear la funcionalidad de nuestros diseños en hardware, no obstante, no permite estimar el consumo de forma precisa.



## Capítulo 6: Conclusiones

A partir del análisis de los distintos parámetros de diseño, podemos ver que el multiplicador con mejores características es el multiplicador de **Wallace**. Este diseño ocupa menos área, alcanza el mayor throughput y consume menos que el resto. Por otro lado, el multiplicador diseñado con la herramienta **Vivado HLS** mejora en algunos casos al multiplicador de Wallace. Por lo que se puede concluir que es una alternativa para el diseño de operadores aritméticos eficientes, y, por lo tanto, una herramienta muy potente para el desarrollo de bloques de mayor complejidad que tengan numerosas multiplicaciones.

Como se observa en los resultados obtenidos anteriormente, los parámetros **velocidad, área y consumo** están fuertemente relacionados entre sí. Consecuencia de esto, la modificación de alguno de estos en el sistema nos lleva a tener que analizar los demás parámetros, porque el diseño podría salirse del rango validez establecido en las especificaciones. En general, se observa que a medida que aumentamos el número de bits de los multiplicadores, las diferencias en los distintos diseños van aumentando. La velocidad, el consumo y el área estimado para cada multiplicador de 4 bits son prácticamente idénticos, para 8 bits se aprecian diferencias notables entre los distintos multiplicadores, y para 16 bits se plasman aún más las diferencias en los diseños.

La técnica de pipeline es un ejemplo claro de lo mencionado anteriormente. En la sección dedicada a esta técnica, se puede observar que se consigue aumentar el throughput del diseño de forma considerable, consiguiendo hasta triplicar la velocidad en algunos casos. Pero, esto tiene dos consecuencias, por un lado, aumenta la latencia de nuestro diseño y por otro, aumenta el área de forma lineal según vamos añadiendo etapas. Además, vemos como la velocidad de los multiplicadores está limitada por la unidad de procesamiento más lenta (camino crítico). Esta unidad es el sumador Ripplecarry, que se encuentra al final de la estructura de cada multiplicador.

**El consumo** está fuertemente ligado tanto a la frecuencia de funcionamiento, como al número de bits. Por otro lado, según las estimaciones realizadas con Vivado los consumos de cada multiplicador son muy parecidos entre sí. Analizando los porcentajes del consumo dinámico, vimos que aproximadamente un 30% correspondía a las señales y la lógica interna, siendo el resto del consumo provocado por elementos que son idénticos para todos diseños. Por lo que se puede concluir que la diferencia de consumo entre distintas arquitecturas es pequeña, y se presentan mayores diferencias en la velocidad o en el área.

Vemos como concentrar esfuerzos en plantear distintas arquitecturas para nuestro diseño y testearlas es imprescindible en el diseño electrónico. En este trabajo podemos ver, como algunos multiplicadores tienen mejores características que otros, aunque todos tienen la misma funcionalidad. El multiplicador de Wallace es más rápido, consume menos y ocupa menos

recursos que las otras alternativas planteadas. Por lo que, aunque nuestro diseño realice la funcionalidad requerida, es importante plantear posibles alternativas de diseño que sean más eficientes.

**FPGA-in-the-Loop (FIL)** es un software que permite testear diseños VHDL, sin embargo, la implementación que realiza esta herramienta modifica las características del proyecto y no siempre es adecuado utilizarla. Por otro lado, Simulink no funciona a la frecuencia que trabajaría la FPGA si utilizásemos un software como Vivado. Por tanto, aunque es una herramienta muy útil, no es adecuada para todo tipo de pruebas.

## Capítulo 7: Líneas futuras

Existen varias operaciones aritméticas fundamentales en el campo electrónico, una de ellas es la multiplicación. Los sistemas digitales son creados a partir de bloques funcionales, como puede ser un multiplicador digital. Existen multitud de trabajos referentes al diseño e implementación de distintas arquitecturas de multiplicadores. El objetivo de este proyecto consiste en estudiar distintas arquitecturas digitales diseñadas con VHDL. Existen múltiples formas de abordar el diseño de un multiplicador digital, y por cuestiones del alcance del proyecto solo se han diseñado cinco arquitecturas. Por lo podría quedar como futuro trabajo, el estudio de arquitecturas adicionales de multiplicadores u otros operadores aritméticos.

El diseño de sistemas electrónicos tiende a la utilización de herramientas de alto nivel, como Vivado HLS, para la mejora de la productividad. En este trabajo hemos hecho una primera aproximación al diseño HDL con herramientas de síntesis de alto nivel. Un posible campo de estudio en posteriores trabajos, podría ser evaluar con mayor profundidad, el impacto que tiene en el consumo utilizar Vivado HLS para diseñar nuestros operadores aritméticos.

FPGA-in-the-Loop ha sido la herramienta utilizada en este trabajo, para testear los diseños realizados en la FPGA, y para medir el consumo de estos. Para hacer una medición precisa del consumo utilizando esta herramienta, habría que hacer una modificación en la metodología propuesta en este trabajo. Para ello, se debería sincronizar el sistema FPGA-in-the-Loop con el sistema de medida utilizado. Así, podríamos conocer el momento exacto en el que se producen las multiplicaciones, y evaluar el consumo de potencia provocado por estas.

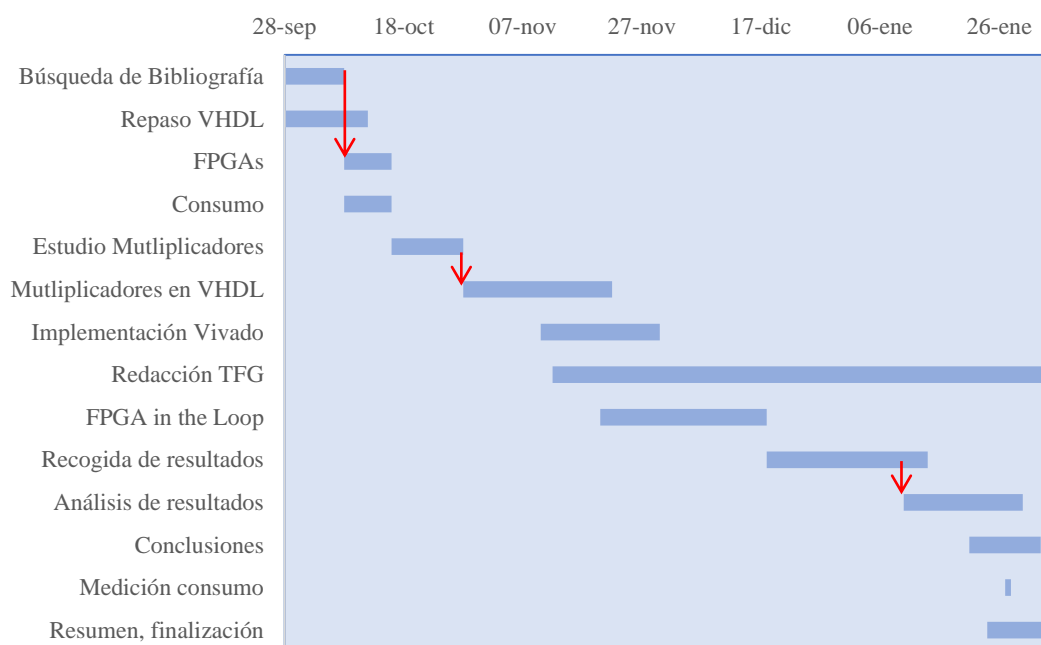
## Capítulo 8: Planificación temporal y presupuesto

### 8.1. Presupuesto

Concepto	Precio Ud.	Unidades	Total
Amortización Herramientas software	100,00 €	1	100,00 €
Amortización Placa PYNQ Z-1	12,5 €	1	12,50 €
Horas dedicadas	10,00 €	350	3.500,00 €
<b>Presupuesto Total</b>			<b>3612,5 €</b>

### 8.2. Planificación temporal

#### Diagrama de Gantt



## A. Anexo I: Código VHDL

### A.1. Declaración de librerías

Se omite la declaración de las librerías ya que se utilizan las mismas para todos los códigos VHDL.

```
1. library IEEE;
2. use IEEE.STD_LOGIC_1164.ALL;
3. use IEEE.NUMERIC_STD.ALL;
```

### A.2. Proceso entrada/salida de datos

Además, todos los multiplicadores tienen un proceso idéntico de entrada/salida de datos que también se omitirá de los posteriores códigos para hacer más sencilla su lectura.

```
1. entradasalida: process (clk, reset)
2. begin
3.
4. if reset='1' then
5.
6. reg_M<=(others=>'0');
7. reg_n<=(others=>'0');
8. P<=(others=>'0');
9.
10. elsif clk'event and clk='1' then
11.
12. reg_M<=M;
13. reg_n<=n;
14. P<=reg_P;
15. end if;
16. end process;
17. end estructural;
18. end Behavioral;
```

### A.3. Código sumador full-adder

```
1. entity full_adder is
2.   Port (sum1: in std_logic;
3.         sum2: in std_logic;
4.         Cin: in std_logic;
5.         S: out std_logic;
6.         Cout: out std_logic);
7.
8. );
9. end full_adder;
10.
11. architecture Behavioral of full_adder is
12. begin
13. S <= sum1 XOR sum2 XOR Cin;
14. Cout <= (sum1 AND sum2) OR (Cin AND sum1) OR (Cin AND sum2);
15.
16. end Behavioral;
17.
```

#### A.4.Código sumador Ripplecarry

```
1. entity ripplecarry_Adder is
2.
3. Generic (
4.   n_bits: positive := 8);
5.
6.
7.   Port (
8.       pr: in std_logic;
9.       sum1: in std_logic_Vector (7 downto 0);
10.      sum2: in std_logic_vector (7 downto 0);
11.      Cin: in std_logic;
12.      S: out std_logic_Vector (7 downto 0);
13.      Cout: out std_logic
14.
15.
16.   );
17. end ripplecarry_Adder;
18.
19. architecture Behavioral of ripplecarry_Adder is
20.
21.   component full_adder
22.   Port (
23.       sum1: in std_logic;
24.       sum2: in std_logic;
25.       Cin: in std_logic;
26.       S: out std_logic;
27.       Cout: out std_logic);
28.
29.   end component;
30.
31.   type carry is array (0 to 7) of std_logic;
32.   signal c: carry; -- acarreos intermedios
33.   signal sum_and: std_logic_Vector (7 downto 0);
34.
35.   begin
36.
37.     sum_and<=sum2 when pr='1' else (others=>'0');
38.
39.     FA1: full_adder port map(sum1=>sum1(0), sum2=>sum_and (0), Cin=>Cin,
40.       S=>S(0), Cout=>C(1));
41.
42.     multi_FA: for i in 1 to 6 generate
43.       FA2: full_adder port map(sum1=>sum1(i), sum2=>sum_and(i), Cin=>c(i),
44.         S=>S(i), Cout=>C(i+1));
45.     end generate;
46.
47.     FA3: full_adder port map(sum1=>sum1(7), sum2=> sum_and (7),
48.       Cin=>C(7), S=>S(7),Cout=> Cout);
49.   end Behavioral;
```

### A.3 Código Multiplicador Ripplecarry

```
1. entity mRipplecarry_8 is      -- M=multiplicando n=multiplicador
2.
3.   Port (
4.       reset: in std_logic;
5.       clk: in std_logic;
6.       M: in std_logic_vector (7 downto 0);
7.       n: in std_logic_vector (7 downto 0);
8.       P: out std_logic_vector (15 downto 0)
9.   );
10.  end mRipplecarry_8;
11.
12.  architecture estructural of mRipplecarry_8 is
13.
14.
15.
16.  component ripplecarry_Adder is
17.
18.      Port (
19.          pr: in std_logic;
20.          sum1: in std_logic_Vector (7 downto 0);
21.          sum2: in std_logic_Vector (7 downto 0);
22.          Cin: in std_logic;
23.          S: out std_logic_Vector (7 downto 0);
24.          Cout: out std_logic
25.
26.
27.      );
28.  end component;
29.
30.
31.  type acumulacion is array (1 to 8) of std_logic_Vector (8 o 0);
32.  signal acum: acumulacion;
33.  signal reg_P: std_logic_Vector (15 downto 0) := (others=>'0');
34.  signal reg_M, reg_n: std_logic_Vector (7 downto 0) := (others='');
35.
36.  begin
37.
38.  acum(1)<= ( '0' &reg_M) when reg_n(0)='1' else (others=>'0');
39.
40.  multi_rc: for i in 1 to 7 generate
41.
42.      C_suma: ripplecarry_Adder
43.
44.      port map (
45.
46.          pr=>reg_n(i),
47.          sum1 => acum(i) (8 downto 1),
48.          sum2 => reg_M,
49.          Cin=>'0',
50.          Cout=>acum(i+1) (8),
51.          S=>acum(i+1) (7 downto 0)
52.
53.      );
54.
55.      reg_P(i) <= acum(i+1) (0);
56.
57.  end generate;
```

```

58.         reg_P (15 downto 8) <= acum (8) (8 downto 1);
59.         reg_P(0)<=acum(1) (0);  end structural;

```

#### A.4 Código Sumador carrysave de 8 bits

```

1.  entity carrysave_sum is
2.
3.  Generic (n_bits: positive:=8);
4.
5.  Port (
6.      sum1: in std_logic_vector (6 downto 0);
7.      sum2: in std_logic_vector (6 downto 0);
8.      sumC: in std_logic_vector (6 downto 0);
9.      S: out std_logic_vector (6 downto 0);
10.     Cout: out std_logic_vector (6 downto 0)
11.
12. );
13. end carrysave_sum;
14.
15. architecture structural of carrysave_sum is
16.
17.     component full_adder
18.
19.     port (sum1: in std_logic;
20.           sum2: in std_logic;
21.           Cin: in std_logic;
22.           S: out std_logic;
23.           Cout: out std_logic
24.
25.     );
26.     end component;
27.
28.     begin
29.
30.     CSA: for i in 0 to 6 generate
31.
32.         FA: full_adder port map (sum1=>sum1(i), sum2=>sum2(i), Cin=>sumC(
33.             i), S=>S(i), Cout=>Cout(i));
34.
35.     end generate;
36. end structural;

```

#### A.5.Código Multiplicador carrysave de 8 bits

```

1.  entity mcarrysave_8 is
2.
3.  Generic (n_bits: positive:=8);
4.  Port (
5.      reset: in std_logic;
6.      clk: in std_logic;
7.      M: in std_logic_vector (7 downto 0);
8.      n: in std_logic_vector (7 downto 0);
9.      P: out std_logic_vector (15 downto 0));
10.
11. end mcarrysave_8;

```



```
12.
13.     architecture structural of mcarrysave_8 is
14.
15.     component ripplecarry_Adder
16.
17.     Generic (n_bits: positive := 8);
18.     Port (
19.
20.         sum1: in std_logic_vector (6 downto 0);
21.         sum2: in std_logic_vector (6 downto 0);
22.         Cin: in std_logic;
23.         S: out std_logic_vector (6 downto 0);
24.         Cout: out std_logic
25.
26.     );
27.
28.     end component;
29.
30.     component carrysave_sum
31.     generic (n_bits: positive:=8);
32.     Port (
33.
34.         sum1: in std_logic_vector (6 downto 0);
35.         sum2: in std_logic_vector (6 downto 0);
36.         sumC: in std_logic_vector (6 downto 0);
37.         S: out std_logic_vector (6 downto 0);
38.         Cout: out std_logic_vector (6 downto 0)
39.
40.     );
41.
42.     end component;
43.
44.     type acumulacion is array (0 to 6) of std_logic_vector (5 downto
45.     0);
46.     signal acum: acumulacion;
47.     type acarreo is array (1 to 7) of std_logic_vector(6 downto 0);
48.     signal carry: acarreo;
49.     type suma_and is array(0 to 7) of std_logic_Vector(7 downto 0);
50.     signal sum_and: suma_and;
51.
52.     signal reg_M,reg_n: std_logic_vector(7 downto 0):=(others=>'0');
53.     signal reg_P: std_logic_vector (15 downto 0):=(others=>'0');
54.
55.     begin
56.
57.         sum_and(0)<= reg_M(7 downto 0) when reg_n(0)='1' else (others=>'0');
58.
59.         sum_and(1)<= reg_M(7 downto 0) when reg_n(1)='1' else (o>'0');
60.
61.         reg_P(0)<= sum_and(0) (0);
62.
63.         CSA1: carrysave Sum
64.
65.         generic map(n_bits=>n_bits)
66.
67.         port map(
68.
69.             sum1=>sum_and (1) (6 downto 0),
70.             sum2=>sum_and (0) (7 downto 1),
71.             SumC=>(others=>'0'),
72.             S (n_bits-2 downto 1) => acum (0),
```

```

68.             S (0) =>reg_P (1),
69.             Cout=>carry (1)
70.         );
71.
72.     multi_cs: for i in 2 to 7 generate
73.
74.         sum_and(i)<= reg_M when reg_n(i)='1' else (others=>'0');
75.
76.         CSA2: carrysave_Sum
77.             generic map(n_bits=>n_bits)
78.
79.             port map (
80.                 sum1(n_bits-3 downto 0) =>acum(i-2)(5 downto 0),
81.                 sum1(n_bits-2) => sum_and(i-1)(7),
82.                 sum2=>sum_and(i)(6 downto 0),
83.                 SumC=>carry(i-1),
84.                 S(0)>= reg_P(i),
85.                 S(n_bits-2 downto 1)>= acum(i-1),
86.                 Cout=>carry(i)
87.             );
88.
89.     end generate;
90.
91.     RCA: ripplecarry_Adder
92.
93.     generic map(n_bits=>n_bits)
94.
95.     port map (
96.         sum1(n_bits-2) => sum_and(7)(n_bits-1),
97.         sum1(n_bits-3 downto 0) => acum(6)(n_bits-
98.         3 downto 0),
99.         sum2 => carry (7),
100.        Cin => '0',
101.        S => reg_P (14 downto 8),
102.        Cout => reg_P (15)
103.    );
104.    end structural;

```

## A.6.Código Multiplicador Wallace de 8 bits

```

1. entity mwallace_8 is
2.     generic( n_bits: positive := 8);
3.     Port (
4.         reset: in std_logic;
5.         clk: in std_logic;
6.         M: in std_logic_vector (7 downto 0);
7.         n: in std_logic_vector (7 downto 0);
8.         P: out std_logic_vector(15 downto 0)
9.     );
10.    end mwallace_8;
11.
12.
13.    architecture structural of mwallace_8 is
14.
15.        type suma_and is array (0 to n_bits-1) of std_logic_vector ( n_bits-
16.        1 downto 0);
17.        signal sum_and: suma_and;
18.        type carrys is array (0 to 3) of std_logic_vector ( 7 downto 0);

```

```
19.  signal carry: carrys;
20.  signal acum: carrys;
21.  signal carry3: std_logic_vector (9 downto 0):=(others=>'0');
22.  signal carry4: std_logic_vector (10 downto 0):=(others=>'0');
23.  signal acum3: std_logic_vector (9 downto 0):=(others=>'0');
24.  signal acum4: std_logic_vector (10 downto 0):=(others=>'0');
25.
26.  signal reg_P: std_logic_vector (15 downto 0);
27.  signal reg_M, reg_n: std_logic_vector (7 downto 0);
28.  signal c: std_logic;
29.
30.  component ripplecarry_Adder
31.    generic (n_bits: positive:= 11);
32.    Port (
33.
34.        sum1: in std_logic_vector(10 downto 0);
35.        sum2: in std_logic_vector(10 downto 0);
36.        Cin: in std_logic;
37.        S: out std_logic_vector (10 downto 0);
38.        Cout: out std_logic
39.
40.    );
41.  end component;
42.
43.
44.  component carrysave_sum
45.    generic (n_bits: positive := 8);
46.    Port (
47.
48.        sum1: in std_logic_vector(n_bits-1 downto 0);
49.        sum2: in std_logic_vector(n_bits-1 downto 0);
50.        sumC: in std_logic_vector(n_bits-1 downto 0);
51.        S: out std_logic_vector(n_bits-1 downto 0);
52.        Cout: out std_logic_vector(n_bits-1 downto 0)
53.
54.    );
55.  end component;
56.
57.
58.  begin
59.
60.    sum_and(0)<= reg_M when reg_n(0)='1' else (others=>'0');
61.    sum_and(1)<= reg_M when reg_n(1)='1' else (others=>'0');
62.    sum_and(2)<= reg_M when reg_n(2)='1' else (others=>'0');
63.    sum_and(3)<= reg_M when reg_n(3)='1' else (others=>'0');
64.    sum_and(4)<= reg_M when reg_n(4)='1' else (others=>'0');
65.    sum_and(5)<= reg_M when reg_n(5)='1' else (others=>'0');
66.    sum_and(6)<= reg_M when reg_n(6)='1' else (others=>'0');
67.    sum_and(7)<= reg_M when reg_n(7)='1' else (others=>'0');
68.
69.
70.    CSA11: carrysave_sum  ---Stage1
71.    generic map(n_bits=>8)
72.    port map(
73.
74.        sum1(6 downto 0) => sum_and(0)(7 downto 1),
75.        sum1(7) => '0',
76.
77.        sum2(7 downto 0) => sum_and(1)(7 downto 0),
78.
```

```

79.         sumC(0) => '0',
80.         sumC(7 downto 1) => sum_and(2)(6 downto 0),
81.
82.         S      => acum(0),
83.         Cout   => carry(0)
84.
85.     );
86.
87. CSA12: carrysave_sum    ---Stage1
88. generic map(n_bits=>8)
89. port map(
90.
91.         sum1(6 downto 0) => sum_and(3)(7 downto 1),
92.         sum1(7) => '0',
93.
94.         sum2=> sum_and(4),
95.
96.         sumC(0)='0',
97.         sumC(7 downto 1)=>sum_and(5)(6 downto 0),
98.
99.         S      => acum(1),
100.        Cout   => Carry(1)
101.
102.    );
103.
104. CSA21: carrysave_sum    --stage2
105. generic map(n_bits=>8)
106. port map(
107.
108.        sum1(6 downto 0)  => acum(0)(7 downto 1),
109.        sum1(7) => sum_and(2)(7),
110.
111.        sum2 =>carry(0),
112.
113.        sumC(0)='0',
114.        sumC(1)=> sum_and(3)(0),
115.        sumC(7 downto 2)=>acum(1)(5 downto 0),
116.
117.        S      => acum(2),
118.        Cout   => Carry(2)
119.    );
120. CSA22: carrysave_sum    --stage2
121. generic map(n_bits=>8)
122. port map(
123.
124.        sum1(3 downto 0) => carry(1)(4 downto 1),
125.        sum1(5 downto 4) => acum(1)(7 downto 6),
126.        sum1(6)=> sum_and(5)(7),
127.        sum1(7)=> '0',
128.
129.        sum2=> sum_and(6),
130.
131.        sumC(7 downto 1)=>sum_and(7)(6 downto 0),
132.        sumC(0)='0',
133.
134.        S      => acum(3),
135.        Cout   => Carry(3)
136.
137.    );
138.
139. CSA3: carrysave_sum    --stage3

```

```
140.         generic map(n_bits=>10)
141.         port map (
142.
143.             sum1(6 downto 0) => acum(2)(7 downto 1),
144.             sum1(7) => carry(1)(5),
145.             sum1(8) => '0',
146.             sum1(9) => '0',
147.
148.             sum2(7 downto 0) => carry(2),
149.             sum2(8) => carry(1)(6),
150.             sum2(9) => carry(1)(7),
151.
152.             sumC(0) => '0', sumC(1) => '0',
153.             sumC(2) => carry(1)(0),
154.             sumC(9 downto 3) => acum(3)(6 downto 0),
155.
156.
157.             S      => acum3,
158.             Cout   => carry3
159.
160.     );
161.
162.     CSA4: carrysave_sum    --stage4
163.         generic map(n_bits=>11)
164.         port map (
165.
166.             sum1(2 downto 0) => (others=>'0'),
167.             sum1(9 downto 3) => carry(3)(6 downto 0),
168.             sum1(10) => '0',
169.
170.             sum2(8 downto 0) => acum3(9 downto 1) ,
171.             sum2(9) => acum(3)(7),
172.             sum2(10) => carry(3)(7),
173.
174.             sumC(9 downto 0) => carry3,
175.             sumC(10) => sum_and(7)(7),
176.
177.             S      => acum4,
178.             Cout   => carry4
179.
180.     );
181.
182.     RC1: ripplecarry_Adder
183.         generic map(n_bits=>11)
184.
185.         port map (
186.
187.             sum1(9 downto 0) => acum4(10 downto 1),
188.             sum1(10) => '0',
189.
190.             sum2(10 downto 0) => carry4(10 downto 0) ,
191.             Cin => '0',
192.
193.             S => reg_P(15 downto 5) ,
194.             Cout => c
195.
196.     );
197.     reg_P(0) <= sum_and(0)(0);
198.     reg_P(1) <= acum(0)(0);
199.     reg_P(2) <= acum(2)(0);
```

```

200. reg_P(3)<= acum3(0);
201. reg_P(4)<= acum4(0);
202.
203. end structural

```

### A.7.Código multiplicador de Baugh-Wooley de 8 bits

```

1. entity mbooley_8 is
2.
3.   Generic(n_bits: positive:=8);
4.   Port (
5.       reset: in std_logic;
6.       clk: in std_logic;
7.       M: in std_logic_vector (7 downto 0);
8.       n: in std_logic_vector (7 downto 0);
9.       P: out std_logic_vector (15 downto 0));
10.
11. end mbooley_8;
12.
13. architecture structural of mbooley_8 is
14.   component ripplecarry_Adder
15.     Generic (n_bits : positive := 9);
16.     Port (
17.
18.         sum1: in std_logic_vector(8 downto 0);
19.         sum2: in std_logic_vector(8 downto 0);
20.         Cin: in std_logic;
21.         S: out std_logic_vector (8 downto 0);
22.         Cout : out std_logic
23.
24.     );
25.   end component;
26.
27.   component carrysave_sum
28.     generic (n_bits: positive: =8);
29.     Port (
30.
31.         sum1: in std_logic_vector (6 downto 0);
32.         sum2: in std_logic_vector(6 downto 0);
33.         sumC: in std_logic_vector(6 downto 0);
34.         S: out std_logic_vector(6 downto 0);
35.         Cout: out std_logic_vector(6 downto 0)
36.
37.     );
38.   end component;
39.
40.   component carrysave_sum2
41.     generic(n_bits: positive :=9);
42.     Port (
43.
44.         sum1: in std_logic_vector(7 downto 0);
45.         sum2: in std_logic_vector(7 downto 0);
46.         sumC: in std_logic_vector(7 downto 0);
47.         S: out std_logic_vector(7 downto 0);
48.         Cout: out std_logic_vector(7 downto 0)
49.
50.     );
51.   end component;
52.
53.

```

```

54.  type acumulacion is array (0 to 5) of std_logic_vector (5 downto 0);
55.  signal acum: acumulacion;
56.  type acarreo is array (0 to 6) of std_logic_vector(6 downto 0);
57.  signal carry: acarreo ;
58.  type suma_and is array (0 to 7) of std_logic_Vector(7 downto 0);
59.  signal sum_and: suma_and;
60.  signal acum2: std_logic_vector(6 downto 0);
61.  signal carry2: std_logic_vector(7 downto 0);
62.  signal reg_M, reg_n: std_logic_vector(7 downto 0):=(others=>'0');
63.  signal reg_P: std_logic_vector(15 downto 0):=(others=>'0');
64.  signal c: std_logic;
65.  signal c1: std_logic;
66.  signal m7,n7: std_logic;
67.
68.  begin
69.
70.      carry(0)<=(others=>'0');
71.      sum_and(0) (6 downto 0)<= reg_M(6 downto 0) when reg_n(0)='1' else (o
thers=>'0');
72.      sum_and(0) (7)<= reg_M(7) and( not reg_n(0));
73.
74.      sum_and(1) (6 downto 0)<= reg_M(6 downto 0) when reg_n(1)='1' else (o
thers=>'0'); --- Etapa1
75.      sum_and(1) (7)<= reg_M(7) and( not reg_n(1));
76.
77.      sum_and(7) (6 downto 0)<= not reg_M(6 downto 0) when reg_n(7)='1' els
e (others=>'0');
78.      sum_and(7) (7)<= (reg_M(7)) and (reg_n(7));
79.
80.
81.      m7<= not reg_M(7);
82.      n7<= not reg_n(7);
83.
84.      reg_P(0)<= sum_and(0) (0);
85.
86.      CSA1: carrysave_Sum
87.
88.          generic map(n_bits=>n_bits)
89.
90.          port map(
91.
92.              sum1=>sum_and(1) (6 downto 0),
93.
94.              sum2=>sum_and(0) (7 downto 1),
95.
96.              Sumc=>carry(0),
97.
98.              S(6 downto 1) => acum(0),
99.              S(0)=>reg_P(1),
100.             Cout=>carry(1)
101.          );
102.
103.  multi_cs: for i in 2 to 6 generate
104.
105.      sum_and(i) (6 downto 0)<= reg_M(6 downto 0) when reg_n(i)='1' else (o
thers=>'0');
106.      sum_and(i) (7)<= reg_M(7) and (not reg_n(i));
107.
108.
109.      CSA2: carrysave_Sum

```

```

110.     generic map(n_bits=>8)
111.
112.     port map(
113.         sum1(5 downto 0)=>acum(i-2)(5 downto 0),
114.         sum1(6) => sum_and(i-1)(7),
115.
116.         sum2=>sum_and(i)(6 downto 0),
117.
118.         SumC=>carry(i-1),
119.
120.         S(0)=> reg_P(i),
121.         S(6 downto 1)=> acum(i-1),
122.         Cout=>carry(i)
123.
124.     );
125.
126. end generate;
127.
128. CSA3: carrysave_Sum2
129.
130.     generic map(n_bits=>9)
131.
132.     port map(
133.
134.         sum1(5 downto 0)=>acum(5)(5 downto 0),
135.         sum1(6)=>sum_and(6)(7),
136.         sum1(7)=>sum_and(7)(7),--?
137.
138.         sum2(6 downto 0)=>sum_and(7)(6 downto 0),
139.         sum2(7)=>m7,
140.
141.         SumC(6 downto 0)=>carry(6),
142.         sumC(7)=>n7,
143.
144.         S(7 downto 1) => acum2,
145.         S(0)=>c,
146.         Cout=>carry2
147.     );
148.
149. RCA:ripplecarry_Adder
150.
151. generic map(n_bits=>9)
152. port map(
153.
154.     sum1(0)=>reg_M(7),
155.     sum1(7 downto 1)=>acum2,
156.     sum1(8)=>'1',
157.
158.     sum2(0) => reg_N(7),
159.     sum2(8 downto 1) => carry2,
160.     Cin => c,
161.     S => reg_P(15 downto 7),
162.     Cout => c1
163.
164. );
165. end structural;

```



## A.8. Código multiplicador Booth de 8 bits

```

1. entity Boothpar_8 is
2.
3.     generic(n_bits : positive := 8);
4.     Port (
5.         reset: in std_logic;
6.         clk: in std_logic;
7.         M: in std_logic_Vector(7 downto 0);
8.         n: in std_logic_Vector(7 downto 0);
9.         P: out std_logic_Vector(14 downto 0)
10.
11.     );
12. end Boothpar_8;
13.
14. architecture structural of Boothpar_8 is
15.
16.     signal Pin1: std_logic_Vector(7 downto 0) := (others=>'0');
17.     signal Pin2: std_logic_Vector(8 downto 0) := (others=>'0');
18.     signal Pin3: std_logic_Vector(9 downto 0) := (others=>'0');
19.     signal Pin4: std_logic_Vector(10 downto 0) := (others=>'0');
20.     signal Pin5: std_logic_Vector(11 downto 0) := (others=>'0');
21.     signal Pin6: std_logic_Vector(12 downto 0) := (others=>'0');
22.     signal Pin7: std_logic_Vector(13 downto 0) := (others=>'0');
23.
24.     signal reg_P: std_logic_vector(2*n_bits-2 downto 0);
25.     signal reg_M, reg_n: std_logic_vector(n_bits-1 downto 0);
26.     signal H,D: std_logic_vector(n_bits-1 downto 0) := (others=>'0');
27.
28.     component Control
29.     generic(n_bits: positive:=8);
30.     Port (
31.         n0: in std_logic_vector(n_bits-1 downto 0 );
32.         n1: in std_logic_vector(n_bits-1 downto 0 );
33.         H: out std_logic_vector(n_bits-1 downto 0 );
34.         D: out std_logic_vector(n_bits-1 downto 0)
35.     );
36. end component;
37.
38.     component CASS_struct
39.     generic(n_bits: positive :=8);
40.     Port (
41.         H: in std_logic;
42.         D: in std_logic;
43.         a: in std_logic_vector(n_bits-1 downto 0);
44.         Pin: in std_logic_vector(n_bits-1 downto 0);
45.         Cin: in std_logic;
46.         Pout: out std_logic_vector(n_bits-1 downto 0)
47.     );
48. end component;
49.
50. begin
51.
52.     BOOTHCONTROL: Control
53.         generic map (n_bits=>8)
54.         port map( n0(0)=>'0',
55.         n0(n_bits-1 downto 1)=>reg_n(n_bits-2 downto 0),
56.         n1(n_bits-1 downto 0)=>reg_n, H=>H, D=>D);
57.

```

```

58.  CASS1: CASS_struct
59.
60.      generic map (n_bits=>8)
61.      port map(H=>H(7), D=>D(7), a=>reg_M , Pin=>(others=>'0'), Cin=>'0'
,Pout=>Pin1(n_bits-1 downto 0) );
62.
63.  CASS2: CASS_struct
64.
65.      generic map (n_bits=>9)
66.      port map(H=>H(6), D=>D(6), a(7 downto 0)=>reg_M ,
67. a(8) =>reg_M(n_bits-1),
68. Pin(8 downto 1) =>Pin1(7 downto 0), Pin(0)=>'0',
69. Cin=>'0', Pout=>Pin2(8 downto 0));
70.
71.  CASS3: CASS_struct
72.      generic map (n_bits=>10)
73.      port map(H=>H(5), D=>D(5), a(7 downto 0)=>reg_M ,
74. a(8)=>reg_M(n_bits-1),a(9)=>reg_M(n_bits-1),
75. Pin(9 downto 1)=>Pin2(8 downto 0), Pin(0)=>'0',Cin=>'0' ,
76. Pout=>Pin3(9 downto 0) );
77.
78.
79.  CASS4: CASS_struct
80.
81.      generic map (n_bits=>11)
82.      port map(H=>H(4), D=>D(4), a(7 downto 0)=>reg_M ,
83. a(8)=>reg_M(n_bits-1),a(9)=>reg_M(n_bits-1),
84. a(10)=>reg_M(n_bits-1) ,
85. Pin(10 downto 1)=>Pin3(9 downto 0),
86. Pin(0)=>'0',Cin=>'0' ,Pout=>Pin4( 10 downto 0));
87.
88.  CASS5: CASS_struct
89.
90.      generic map (n_bits=>12)
91.      port map(H=>H(3), D=>D(3), a(7 downto 0)=>reg_M
,a(8)=>reg_M(n_bits-1),a(9)=>reg_M(n_bits-1),a(10)=>reg_M(n_bits-
1) ,a(11)=>reg_M(n_bits-1) ,
92. Pin(11 downto 1)=>Pin4(10 downto 0), Pin(0)=>'0',Cin=>'0'
,Pout=>Pin5(11 downto 0) );
93.
94.  CASS6: CASS_struct
95.
96.      generic map (n_bits=>13)
97.      port map(H=>H(2), D=>D(2), a(7 downto 0)=>reg_M ,
98. a(8)=>reg_M(n_bits-1),
99. a(9)=>reg_M(n_bits-1),a(10)=>reg_M(n_bits-1) ,
100. a(11)=>reg_M(n_bits-1) ,a(12)=>reg_M(n_bits-1) ,
101. Pin(12 downto 1)=>Pin5(11 downto 0),
102. Pin(0)=>'0',Cin=>'0' ,Pout=>Pin6(12 downto 0));
103.
104.  CASS7: CASS_struct
105.
106.      generic map (n_bits=>14)
107.      port map(H=>H(1), D=>D(1), a(7 downto 0)=>reg_M ,
108. a(8)=>reg_M(n_bits-1),a(9)=>reg_M(n_bits-1),
109. a(10)=>reg_M(n_bits-1) ,a(11)=>reg_M(n_bits-1) ,
110. a(12)=>reg_M(n_bits-1) ,a(13)=>reg_M(n_bits-1) ,
111. Pin(13 downto 1)=>Pin6( 12 downto 0), Pin(0)=>'0',Cin=>'0' ,
112. Pout=>Pin7(13 downto 0) );
113.
114.  CASS8: CASS_struct

```

```

115.
116.         generic map (n_bits=>15)
117.         port map (H=>H(0), D=>D(0), a(7 downto 0)=>reg_M ,
118. a(8)=>reg_M(n_bits-1),a(9)=>reg_M(n_bits-1),
119. a(10)=>reg_M(n_bits-1) , a(11)=>reg_M(n_bits-1) ,
120. a(12)=>reg_M(n_bits-1) ,
121. a(13)=>reg_M(n_bits-1) ,a(14)=>reg_M(n_bits-1),
122. Pin(14 downto 1)=>Pin7(13 downto 0), Pin(0)=>'0', Cin=>'0' ,
123. Pout=>reg_P(14 downto 0) );
124. end structural;

```

### A.9. Código Control estructura de Booth

```

1. entity Control is
2.     generic(n_bits: positive:=16);
3.     Port (
4.         n0: in std_logic_Vector(n_bits-1 downto 0 );
5.         n1: in std_logic_vector(n_bits-1 downto 0 );
6.         H: out std_logic_vector(n_bits-1 downto 0 );
7.         D: out std_logic_vector(n_bits-1 downto 0)
8.     );
9. end Control;
10.
11.     architecture structural of Control is
12.
13.         component CTRL
14.             PORT(
15.                 n: in std_logic_vector(1 downto 0);
16.                 H: out std_logic;
17.                 D: out std_logic
18.             );
19.         end component;
20.
21.         begin
22.
23.             CONTROL: for i in 0 to n_bits-1 generate
24.                 C: CTRL
25.
26.                 port map( n(0)=>n0(i),n(1)=>n1(i), H=>H(i), D=>D(i) );
27.
28.             end generate;
29.         end structural;

```

### A.10. Código Control

```

1. entity CTRL is
2.     Port (
3.
4.         n: in std_logic_Vector(1 downto 0);
5.         H: out std_logic;
6.         D: out std_logic
7.
8.     );
9. end CTRL;
10.
11.     architecture Behavioral of CTRL is
12.
13.         begin

```

```

14.
15.   with n select H<=
16.
17.   '0' when "11",
18.   '0' when "00",
19.   '1' when "01",
20.   '1' when others;
21.
22.   with n select D <=
23.
24.   '1' when "10",
25.   '0' when others;
26.
27.   end Behavioral;

```

### A.11. Código celda CASS

```

1. entity CASS is
2.   Port (
3.       Pin: in std_logic;
4.       a: in std_logic;
5.       H: in std_logic;
6.       D: in std_logic;
7.       Cin: in std_logic;
8.       Pout: out std_logic;
9.       Cout: out std_logic
10.    );
11. end CASS;
12.
13. architecture Behavioral of CASS is
14.
15.   begin
16.
17.   Pout <= Pin xor (a and H) xor (Cin and H);
18.   Cout <= ((Pin xor D) and (a or Cin)) or (a and Cin);
19.
20.   end Behavioral;

```

### A.12. Código estructura CASS

```

1. entity CASS_struct is
2.   generic(n_bits: positive :=16 );
3.   Port (
4.
5.       H: in std_logic;
6.       D: in std_logic;
7.       a: in std_logic_Vector(n_bits-1 downto 0);
8.       Pin: in std_logic_Vector(n_bits-1 downto 0);
9.       Cin: in std_logic;
10.      -- Cout: out std_logic;
11.      Pout: out std_logic_vector(n_bits-1 downto 0)
12.
13.    );
14. end CASS_struct;
15.
16. architecture structural of CASS_struct is
17.
18.   component CASS
19.     Port (

```

```
20.         Pin: in std_logic;
21.         a: in std_logic;
22.         H: in std_logic;
23.         D: in std_logic;
24.         Cin: in std_logic;
25.         Pout: out std_logic;
26.         Cout: out std_logic
27.     );
28.     end component;
29.
30.     type acarreo is array (1 to n_bits-1) of std_logic;
31.     signal C: acarreo ;
32.     signal Cout: std_logic;
33.
34.     begin
35.         CASS1: CASS
36.         port map( Pin=>Pin(0), a=>a(0), H=>H, D=>D, Cin=>Cin,
Pout=>Pout(0), Cout=>C(1) );
37.
38.         estructura_CASS: for i in 1 to n_bits-2 generate
39.
40.             CASS2: CASS
41.
42.             port map( Pin=>Pin(i), a=>a(i), H=>H, D=>D, Cin=>C(i),
Pout=>Pout(i), Cout=>C(i+1) );
43.
44.         end generate;
45.
46.         CASS3: CASS
47.
48.         port map( Pin=>Pin(n_bits-1), a=>a(n_bits-1), H=>H, D=>D,
Cin=>C(n_bits-1), Pout=>Pout(n_bits-1), cout=>cout);
49.
50.     end structural;
```

### A.13. Código Asociación de pines Multiplicadores de 8 bits

```
1.  set_property PACKAGE_PIN U5 [get_ports {P[9]}]
2.  set_property PACKAGE_PIN V5 [get_ports {P[7]}]
3.  set_property PACKAGE_PIN V6 [get_ports {n[0]}]
4.  set_property PACKAGE_PIN U7 [get_ports {n[7]}]
5.  set_property PACKAGE_PIN V7 [get_ports {n[3]}]
6.  set_property PACKAGE_PIN U8 [get_ports {n[1]}]
7.  set_property PACKAGE_PIN V8 [get_ports {n[4]}]
8.  set_property PACKAGE_PIN V10 [get_ports {P[5]}]
9.  set_property PACKAGE_PIN W10 [get_ports {P[4]}]
10. set_property PACKAGE_PIN W6 [get_ports {n[5]}]
11. set_property PACKAGE_PIN Y6 [get_ports {P[0]}]
12. set_property PACKAGE_PIN Y7 [get_ports {n[2]}]
13. set_property PACKAGE_PIN W8 [get_ports {P[1]}]
14. set_property PACKAGE_PIN Y8 [get_ports {n[6]}]
15. set_property PACKAGE_PIN W9 [get_ports {P[2]}]
16. set_property PACKAGE_PIN Y9 [get_ports {P[6]}]
17. set_property PACKAGE_PIN Y13 [get_ports {P[10]}]
18. set_property PACKAGE_PIN T14 [get_ports {P[13]}]
19. set_property PACKAGE_PIN U12 [get_ports {P[8]}]
20. set_property PACKAGE_PIN U13 [get_ports {P[14]}]
21. set_property PACKAGE_PIN V13 [get_ports {P[12]}]
22. set_property PACKAGE_PIN V15 [get_ports {P[11]}]
23. set_property PACKAGE_PIN T15 [get_ports {P[15]}]
24. set_property PACKAGE_PIN R16 [get_ports {M[7]}]
25. set_property PACKAGE_PIN U17 [get_ports {M[6]}]
26. set_property PACKAGE_PIN Y12 [get_ports {M[5]}]
27. set_property PACKAGE_PIN V18 [get_ports {M[4]}]
28. set_property PACKAGE_PIN T16 [get_ports {M[3]}]
29. set_property PACKAGE_PIN R17 [get_ports {M[2]}]
30. set_property PACKAGE_PIN P18 [get_ports {M[1]}]
31. set_property PACKAGE_PIN N17 [get_ports {M[0]}]
32. set_property PACKAGE_PIN Y11 [get_ports {P[3]}]
33. set_property IOSTANDARD LVCMOS18 [get_ports {n[*]}]
34. set_property IOSTANDARD LVCMOS18 [get_ports {M[*]}]
35. set_property IOSTANDARD LVCMOS18 [get_ports {P[*]}]
```

**A.14. Testbench números pseudoaleatorios para archivo SAIF**

```
1. entity Tb_LFSR is
2.   generic(
3.     n_bits: positive := 8);
4.   end Tb_LFSR;
5.
6. architecture tb of Tb_LFSR is
7.
8.   component mRipplecarry_8
9.     port(
10.       reset: in std_logic;
11.       M: in std_logic_vector(n_bits-1 downto 0);
12.       n: in std_logic_vector(n_bits-1 downto 0);
13.       clk: in std_logic;
14.       P: out std_logic_vector(2*n_bits-1 downto 0)
15.     );
16.   end component;
17.   signal M,n: std_logic_vector(n_bits-1 downto 0):=(others=>'0');
18.   signal P: std_logic_vector(2*n_bits-1 downto 0):=(others=>'0');
19.   signal clk: std_logic;
20.   signal reset: std_logic:='0';
21.   constant clk_period: time:= 40 ns;
22.
23.   signal s2: std_logic:='0';
24.   signal lfsr2: std_logic_vector (7 downto 0):="10111001";
25.   signal s: std_logic:='0';
26.   signal lfsr: std_logic_vector (7 downto 0):="11001000";
27.   begin
28.     clk_proc: process
29.     begin
30.       clk<='1';
31.       wait for clk_period/2;
32.       clk<='0';
33.       wait for clk_period/2;
34.     end process;
35.     uut:mRipplecarry 8
36.
37.     port map( M=>M, n=>n,clk=>clk, reset=>reset, P=>P);
38.     s <= (lfsr(3) xnor (lfsr(4) xnor (lfsr(5) xnor lfsr(7))));
39.     s2 <= (lfsr2(3) xnor (lfsr2(4) xnor (lfsr2(5) xnor lfsr2(7))));
40.
41.     stim_proc: process
42.     begin
43.
44.       wait for 10ns;
45.       for i in 0 to 10000 loop
46.
47.         reset<='0';
48.         lfsr2 <= (lfsr2(6 downto 0) & s2);
49.         lfsr <= (lfsr(6 downto 0) & s);
50.         n<=lfsr2;
51.         M<=lfsr;
52.         wait for clk_period;
53.       end loop; wait;
54.     end process; end tb;
```

## B. Anexo II: Pines placa PYNQ

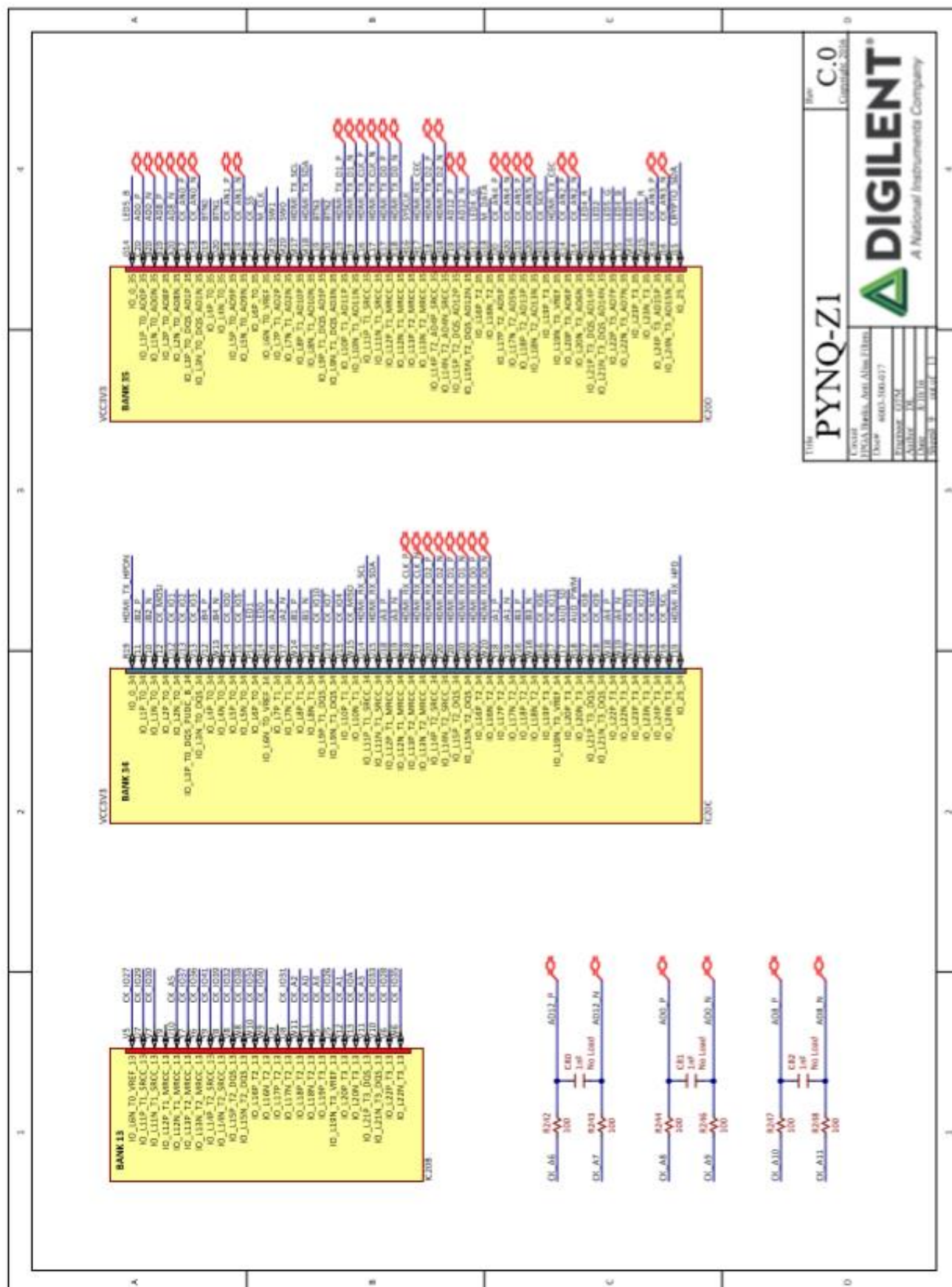


Figura B.1: Pines placa PYNQ [4]



## C. Anexo III: Referencias

### C.1. Bibliografía y referencias

- [1] Chen, D., Cong, J., He, L., & Li, F. (2005). *Power modeling and characteristics of field programmable gate arrays*. *IEEE Trans Comput Aided Des Integr Circ Syst*. IEEE.
- [2] D.Buell, T. El-Ghazawi, K.Gaj, & V. Kindratenko. (2007). FPGAs. Octubre.
- [3] DESCHAMPS, J.-P., BIOUL, G. J., & SUTTER, G. D. (2006). *SYNTHESIS OF ARITHMETIC CIRCUITS*. WILEY-INTERSCIENCE.
- [4] DIGILENT. (8 de OCTUBRE de 2016). *PYNQ-Z1.Schematic*. Obtenido de [https://reference.digilentinc.com/\\_media/reference/programmable-logic/pynq-z1/pynq-z1\\_sch.pdf](https://reference.digilentinc.com/_media/reference/programmable-logic/pynq-z1/pynq-z1_sch.pdf)
- [5] Digilent. (2019). <https://reference.digilentinc.com/>. Obtenido de <https://reference.digilentinc.com/reference/programmable-logic/pynq-z1/reference-manual>
- [6] García García, D. A. (s.f.). *Multiplificador digital*. Escuela de ingeniería y ciencias.
- [7] Machado Sánchez, F., Borromeo López, S., & Rodríguez Sánchez, C. (2011). *Diseño de sistemas digitales con VHDL*. Universidad Rey Juan Carlos.
- [8] Mandado Pérez, E., & Martín González, J. L. (2015). *SISTEMAS ELECTRÓNICOS DIGITALES*. Marcombo.
- [9] MathWorks. (2019). *FPGA-in-the-Loop Simulation*. Obtenido de [https://es.mathworks.com: https://es.mathworks.com/help/hdlverifier/ug/fpga-in-the-loop-fil-simulation.html](https://es.mathworks.com/help/hdlverifier/ug/fpga-in-the-loop-fil-simulation.html)
- [10] Mittal, S., Gupta, S., & Dasgupta, S. (s.f.). *System Generator: The State-of-art FPGA Design*. Indian Institute of Technology Roorkee.
- [11] Moustafa, K. (2014). *Aging Analysis of Datapath Sub-blocksBased onCET Map Model for Negative Bias Temperature Instability (NBTI)*. Universidad de Nile.
- [12] Mozos Muñoz, D. (s.f.). *Ampliación de estructura de computadores*. Facultad de informática.
- [13] Oliver, J. P. (2014). *Técnicas de Bajo consumo en FPGAs*. Montevideo: Universidad de la República.

- [14] Ordóñez-Fernández, G., López-López, L., & Velasco-Medina, J. (s.f.). *DISEÑO DE MULTIPLICADORES PARALELOS DE 16 BITS EN FPGAS*. Cali: Universidad del Valle A.A.
- [15] Rubio, A., Altet, J., Aragonés, X., González, J., Mateo, D., & Moll, F. (2000). *Diseño de circuitos y sistemas integrados*. Universidad Politécnica de Cataluña.
- [16] Suganya T.M, & Dr.M. Senthikumar. (2015). *A REVIEW OF UART ENABLING BIST ARCHITECTURE USING VHDL*. Coimbatore: INTERNATIONAL JOURNAL OF RESEARCH IN SCIENCE AND ENGINEERING.
- [17] Sung, R., Sung, A., Chan Patrick, & Mah, J. (s.f.). *Linear Feedback Shift Register*.  
Obtenido de [http://www.ece.ualberta.ca/~elliott/ee552/studentAppNotes/1999f/Drivers\\_Ed/lfsr.html](http://www.ece.ualberta.ca/~elliott/ee552/studentAppNotes/1999f/Drivers_Ed/lfsr.html)
- [18] Sutter, G. (2005). *Aportes a la Reducción de Consumo en FPGAs*. Madrid: Universidad Autónoma de Madrid.
- [19] Terés, L., Torroja, Y., Olcoz, S., & Villar, E. (1997). *VHDL LENGUAJE ESTÁNDAR DE DISEÑO ELECTRÓNICO*. McGRAW-HILL.
- [20] Vargas, E., & OCHOA, A. (s.f.). *Algoritmo de Auto-generación de Bloques Aritméticos*. Colima: Universidad de Colima.
- [21] Xilinx. (5 de Enero de 2006). *Using Digital Clock Managers (DCMs) in*.  
Obtenido de [https://www.xilinx.com/support/documentation/application\\_notes/xapp462.pdf](https://www.xilinx.com/support/documentation/application_notes/xapp462.pdf)
- [22] Xilinx. (30 de Mayo de 2014). *Vivado Design Suite User Guide High-level Synthesis*.
- [23] Xilinx. (27 de Septiembre de 2016). *7 Series FPGAs Memory Resources*.  
Obtenido de [https://www.xilinx.com/support/documentation/user\\_guides/ug473\\_7Series\\_Memory\\_Resources.pdf](https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf)
- [24] Xilinx. (20 de Diciembre de 2017). *Vivado Design suite User Guide: Getting Started*. Obtenido de <https://www.xilinx.com/support.html#documentation>
- [25] Xilinx. (20 de Diciembre de 2017). *Vivado Design Suite User Guide: Power Analysis and Optimization*. Obtenido de <https://www.xilinx.com/support.html#documentation>

- [26] Xilinx. (20 de Diciembre de 2017). *Vivado Design Suite User Guide: Using the Vivado IDE*. Obtenido de <https://www.xilinx.com/support.html#documentation>
- [27] Xilinx. (27 de Marzo de 2018). *7 Series DSP48E1 Slice*. Obtenido de [https://www.xilinx.com/support/documentation/user\\_guides/ug479\\_7Series\\_DSP48E1.pdf](https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf)
- [28] Xilinx. (27 de Septiembre de 2018). *7 Series FPGAs Configurable Logic Block User.* Obtenido de [https://www.xilinx.com/support/documentation/user\\_guides/ug474\\_7Series\\_CLB.pdf](https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf)
- [29] Xilinx. (6 de Junio de 2018). *Vivado Design Suite User Guide: I/O and Clock Planning*. Obtenido de <https://www.xilinx.com/support.html#documentation>
- [30] Xilinx. (6 de Junio de 2018). *Vivado Design Suite User Guide: Using Constraints*. Obtenido de <https://www.xilinx.com/support.html#documentation>
- [31] Xilinx. (2019). <https://www.xilinx.com>. Obtenido de <https://www.xilinx.com/products/design-tools/vivado.html>

## C.2. Abreviaturas, unidades y acrónimos

FPGA: Field Programmable gate array

SoC: System on a chip

SAIF: Switching Activity Interchange format

HDL: Hardware Description Language

VDD: Tensión de alimentación

GND: Ground

Is: Corriente subumbral

FIL: FPGA in the loop

CI: Circuito integrado

ASIC: Application-Specific Integrated Circuit

CMOS: Semiconductor complementario de óxido metálico

NMOS: Negative-channel Metal-Oxide Semiconductor

$I_s$ : corriente de saturación en inversa

$V_T$ : Tensión térmica

DSP: Digital signal processing

PLD: Programmable logic device

CLB: Configurable Logic Block

LUT: Look-up tables

BUGF: Global Clock Buffer

FF: Flip-Flop

HLS: High level synthesis

### **C.3. Glosario**

*Throughput*: número de multiplicaciones por segundo capaces de realizar los multiplicadores.

*Testbench*: Archivo de simulación para los diseños VHDL.

*Glitching activity*: Consiste en actividad innecesaria de nuestro diseño que provoca un mayor consumo de potencia dinámica.

*Signal rate*: número de veces que un elemento cambia su estado por segundo.

*Toogle rate*: porcentaje con el cual una salida de un elemento síncrono cambia comparado a el reloj.

*Static probability*: define el tiempo relativo respecto a la duración de un análisis durante el cual, el considerado elemento está a nivel lógico alto.

