V 8 Bits Signed Multiplier Two's Complement Version

Contreras R Orlando[1,2*] and Garcia B Iker[1,2*†]

[3]Departamento de Sistemas Electrónicos, UAA, Av. Universidad 940, Aguascalientes, 20131, Aguascalientes, México.

*Corresponding author(s). E-mail(s): {al348390,al307630}@edu.uaa.mx;
†Ambos autores contribuyeron de forma equitativa a este trabajo.

Abstract

El presente documento busca implementar la multiplicación con signo de la forma complemento a dos, para poder implementarlo de forma efectiva, se agregarán dos entradas correspondientes al signo.

Keywords: VHDL, FPGA, Signo, Complemento a 2, Multiplicador con Signo

1 Introducción

Este multiplicador se destaca por realizar la operación mediante la conversión a complemento a dos cuando detecta que uno o ambos operandos son negativos. Esta técnica permite manejar correctamente los signos durante la multiplicación y sirve como una base fundamental para comprender métodos más avanzados y eficientes de multiplicación en sistemas digitales.

El procesamiento eficiente de señales digitales y aplicaciones de alto rendimiento requieren operaciones aritméticas rápidas y eficientes. Entre ellas, la multiplicación de numeros con signo es una de las más importantes, ya que es fundamental en algoritmos de procesamiento de imágenes, inteligencia artificial y sistemas embebidos.

En este contexto, el uso de circuitos diseñados en **FPGA** ofrece ventajas significativas en términos de velocidad, paralelismo y consumo energético en comparación con soluciones basadas en procesadores convencionales.

2 Metodología

2.1 Complemento a 2

Para realizar el bloque de complemento a 2 deberemos de hacer un bloque que tenga como entrada el valor a complementar, su signo y su salida, en este caso nosotros manejamos el bloque como única entrada el valor a complementar y la salida, de forma que siempre estaremos complementando el valor, ya en la arquitectura principal tendremos un multiplexor que seleccionará entre el valor complementado y el valor sin complementar.

El funcionamiento del bloque se define como negar el valor (Aplicar una not) y agregar uno, en este caso se agrego el uno por medio de un propagador. Otras formas de hacerlo podría serlo aplicar directamente not$(val) + 1$, o podemos directamente castearlo a signed por medio de la librería unsigned. En la práctica se hizo el primer caso por cuestiones de aprendizaje pero ambos construyen el mismo circuito.

2.2 Extensión de Signo

En este caso fue necesario hacer una extensión de signo del doble de bits para evitar que quedara basura, de esta forma deberemos ampliar el multiplicador haciéndolo de 16*16 bits, para esto lo que hizo fue aprovecharse del vector previamente definido e instanciarlo 16 veces.

El ejemplo más claro del funcionamiento correcto de esto es si nosotros multiplicamos $-1 \times -1 = 1$, en binario deberemos de aplicar complemento a 2 al número 1 es decir lo negamos 0001 $\rightarrow^{Ca1}$ 1110 para posteriormente sumar 1 quedando el -1 como 1111 si se extiende a 16 bits sería 1111 1111 1111 1111 = $FFFF$. Nosotros al realizar esta multiplicación esperamos un 1 de forma positiva al multiplicarse dos negativos, si bien no lo obtenemos de forma pura, se puede apreciar en los 16 bits de la figura 1.

Fig. 1 Multiplicación de negativos

2.3 Diseño de los bloques de 16 bits

Al igual que en la práctica pasada, se aprovechó del diseño de los bloques de 16 bits, e incluso se realizó una versión optimizada en cuanto a código; es decir, genera el mismo hardware pero permite hacer bloques de forma más dinámica usando la estructura del generate. Esto permite crear versiones prototipadas de los códigos y entenderlo de una mejor forma; asimismo, las instancias también fueron generadas con Generate en esta versión e incluso se aprovechó del tipo de dato llamado Array para generar una

matriz de propagadores e instanciarlo de una mejor forma. De igual forma, se realizó la versión sin estos complementos, pero nos pareció interesante abarcarlo desde esta otra forma.

3 Resultados

Se puede observar en la simulación como los valores de la multplicacion son correctos y el signo de salida esta invertido, esto es ya que el led de salida de signo que usamos es un led interno de la tang nano y prende con 0 en ves de con unos.

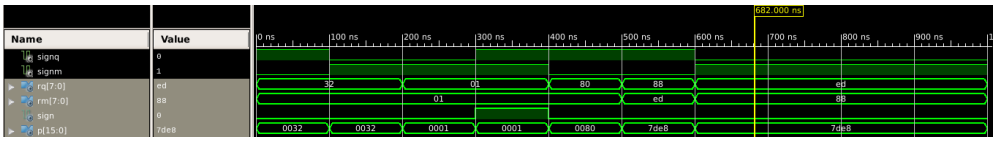

Fig. 2 Testbench realizado por Xilinx

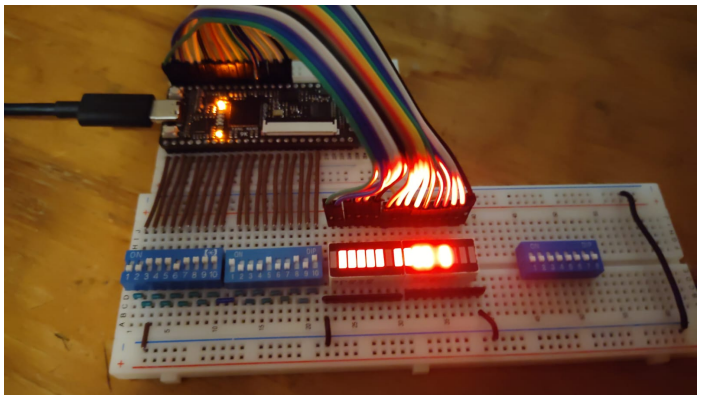

Fig. 3 Circuito Armado

Device Utilization Summary				
Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	0	11,440	0%	
Number of Slice LUTs	187	5,720	3%	
Number used as logic	187	5,720	3%	
Number using O6 output only	165			
Number using O5 output only	0			
Number using O5 and O6	22			
Number used as ROM	0			
Number used as Memory	0	1,440	0%	
Number of occupied Slices	85	1,430	5%	
Number of MUXCFs used	0	2,860	0%	
Number of LUT Flip Flop pairs used	187			
Number with an unused Flip Flop	187	187	100%	
Number with an unused LUT	0	187	0%	
Number of fully used LUT-FF pairs	0	187	0%	
Number of slice register sites lost to control set restrictions	0	11,440	0%	
Number of bonded IOBs	35	102	34%	
Number of RAMB16BWERs	0	32	0%	
Number of RAMB8BWERs	0	64	0%	
Number of BUFIO2BUFIO2_2CLKs	0	32	0%	
Number of BUFIO2FB/BUFIO2FB_2CLKs	0	32	0%	
Number of BUFG/BUFGMUXs	0	16	0%	
Number of DCM/DCM_CLKGENs	0	4	0%	
Number of ILOGIC2/ISERDES2s	0	200	0%	
Number of IODELAY2/IODRP2/IODRP2_MCBs	0	200	0%	
Number of OLOGIC2/OSERDES2s	0	200	0%	
Number of BSCANs	0	4	0%	
Number of BUFHs	0	128	0%	
Number of BUFPLLs	0	8	0%	
Number of BUFPLL_MCBs	0	4	0%	
Number of DSP48A1s	0	16	0%	

Fig. 4 Summary report generated by Xilinx

4 Conclusiones

Esta práctica se nos hizo más fácil a la hora de implementar, pero gracias a diversos factores se malinterpretó el funcionamiento del tamaño de la multiplicación y, pues, se tuvo que modificar el código que teníamos en la primer entrega. Nos facilitó mucho la práctica el uso de la tarjeta FPGA Tang Nano gracias a cómo hace las constraints files y la portabilidad.

5 Anexos

Esto es la parte más importante de los RTL, si se gusta observar con mayor detalle, los PDFs estarán en el documento adjunto. Asimismo se podrá observar el código actualizado en: GitHub Repository

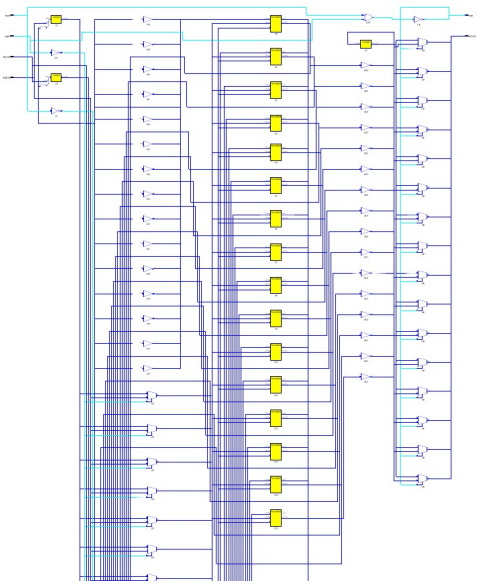

Fig. 5 RTL del Componente principal

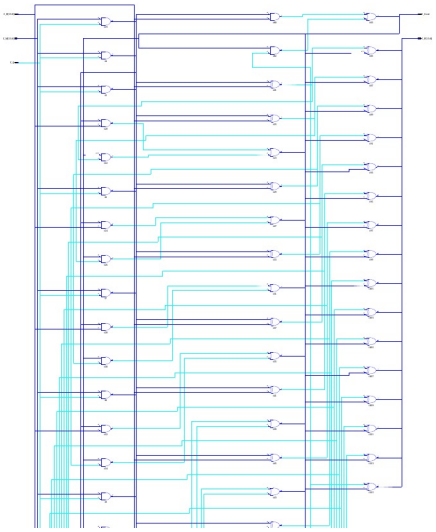

Fig. 6 RTL del Componente Multiplicador

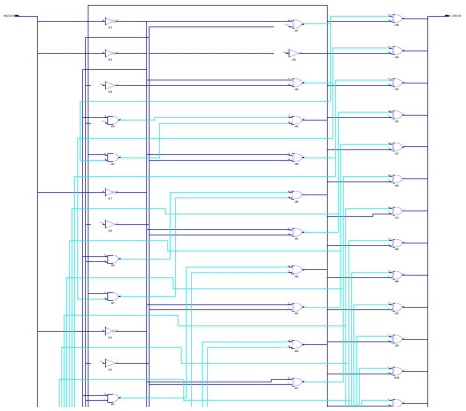

Fig. 7 RTL del Componente Complemento a 2