Laboratorio 3: Lógica Combinacional y Aritmética I

David Cordero, Jonathan Guzmán, Darío Rodríguez
dcorderoch@ieee.org jonathana1196@gmail.com darior1227@gmail.com
Área Académica de Ingeniería en Computadores
Instituto Tecnológico de Costa Rica

***Resumen*—Una unidad lógica aritmética es un circuito digital que calcula operaciones aritméticas tales como sumas y restas, y operaciones lógicas (con valores de verdad, es decir verdadero o falso). En este documento se detalla la implementación de una de estas que permite las operaciones fundamentales lógicas, NOT, OR, AND, XOR, además de corrimiento de bits lógico, y aritmético, además de sumas y restas.**

***Palabras clave*—ALU, Lógica combinacional, FPGA, frecuencia, SystemVerilog.**

I. Introducción

Una ALU (Arithmetic Logic Unit) es un circuito digital usado para realizar operaciones lógicas y aritméticas, es el bloque fundamental de la unidad central de procesamiento (CPU) de una computadora. Los CPUs modernos contienen varias ALUs poderosas y complejas. Además de las ALUs, los modernos CPUs contienen una unidad de control (CU).

La mayoría de las operaciones de un CPU son realizadas por una o mas ALUs, las cuales cargan los datos de entrada desde un banco de registros. Un registro es una unidad de almacenamiento pequeña, pero extremadamente rápida disponible como parte de un CPU.

La unidad de control le dice a la ALU que operaciones realizar con qué datos y la ALU almacena el resultado en un registro de salida. La unidad de control mueve los datos entre estos registros de la ALU y la memoria.

La ALU realiza operaciones básicas, tanto aritméticas como lógicas. Algunas operaciones aritméticas que realiza son: suma y resta; mientras que algunas operaciones lógicas son: NOT, AND, OR.

Toda la información es procesada en forma de números binarios (0 y 1). Los interruptores de los tansistores se usan para manipular números binarios ya que solo hay dos estados posibles de un interruptor: abierto y cerrrado. Un transistor abierto a través del cual no hay corriente representa un 0, un transistor cerrado a través del cual hay corriente, representa un 1 (a menos que se use un sistema de lógica negativa, que funciona de manera opuesta).

Las operaciones pueden realizarse conectando múltiples transistores. Un transistor puede ser usado como control de un segundo transistor. La operación de compuerta más simple, NOT utiliza un solo transistor, el cual siempre brinda una salida contraria la entrada que recibe.

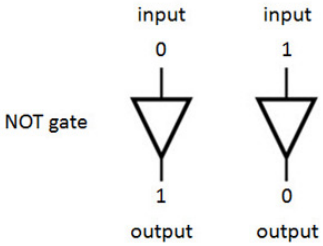

Figura 1: NOT gate

A	C_{out}
0	1
1	0

Cuadro I: Tabla de verdad de NOT Gate

Otras compuertas más complejas consisten de múltiples transistores y usan más entradas.

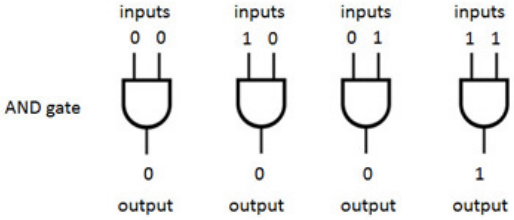

Figura 2: AND gate

A	B	C_{out}
0	0	0
1	0	0
0	1	0
1	1	1

Cuadro II: Tabla de verdad de AND Gate

A	B	C_{out}
0	0	0
1	0	1
0	1	1
1	1	1

Cuadro III: Tabla de verdad de OR Gate

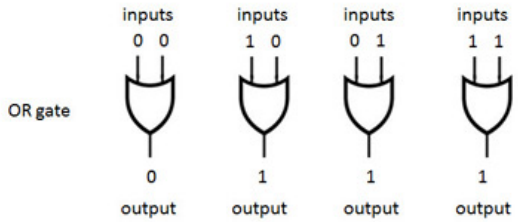

Figura 3: OR gate

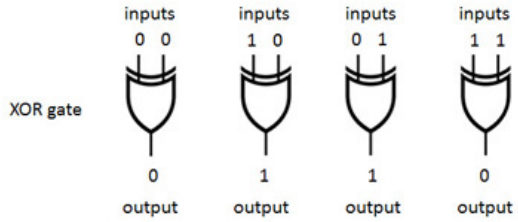

Figura 4: XOR gate

De estas se componen otros circuitos más complejos.

A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Cuadro IV: Tabla de verdad del Sumador Completo

Las banderas más comunes en una ALU son: zero, carry, overflow, negative.

- Zero flag: usada para determinar si dos valores son iguales, generalmente se usa una resta, y se revisa si el resultado es cero, si es así, esta bandera se enciende.
- Overflow flag: usada para determinar si el resultado de una operación (usualmente suma o multiplicación), tiene un tamaño superior al máximo representable, o inferior al mínimo representable en la ALU, es posible que existan como banderas separadas.
- Negative flag: usada para determinar si el resultado de una resta es menor a cero
- Carry/Borrow flag: Esta bandera indica cuando una operación resulta en un valor más largo del cual el acumulador puede representar o menor del cual el acumulador puede representar.

Definimos como tiempo de propagación al tiempo transcurrido desde que una señal de entrada pasa por un determinado circuito combinacional hasta que la salida reacciona a esa entrada.

Se le conoce como ruta crítica a la ruta por la que se puede dar un flujo de datos que pasan por la mayor cantidad de elementos combinacionales de un circuito digital, lo que implica que es la ruta con mayor retardo combinacional. Un aspecto importante es que siempre existen retardos entre la estimulación de la entrada de un elemento y su salida, además de la oscilación de los estados lógicos mientras se distribuye la información por el elemento a la que se le llama trasiente; lo que se busca con la ruta crítica es encontrar los componentes que generen un retardo tan grande en la ruta de datos, que la repuesta del circuitos final en sus salidas se prolongue por mucho tiempo.

La ruta crítica es en realidad la frecuencia máxima de trabajo por el conjunto de elementos que componen un procesador. El tiempo de retraso causado por la ruta crítica es inversamente proporcional a la frecuencia de trabajo.

Para minimizar el efecto de la ruta crítica, los procesadores modernos usan pipelines que distribuyen el cálculo en más unidades, a costo de más componentes, y por ende mayor área.

La ventaja primordial de este sistema es que, una vez que el canal está lleno, es decir, después de una latencia de N (con N la cantidad de pasos en el pipeline), los resultados de cada instrucción vienen uno tras otro cada flanco de reloj y sin latencia extra por estar encadenados dentro del mismo canal, todo esto habiendo maximizado la frecuencia máxima de trabajo.

Cabe destacar que el alto rendimiento y la velocidad de los procesadores modernos, se debe principalmente a los pipelines con segmentación.

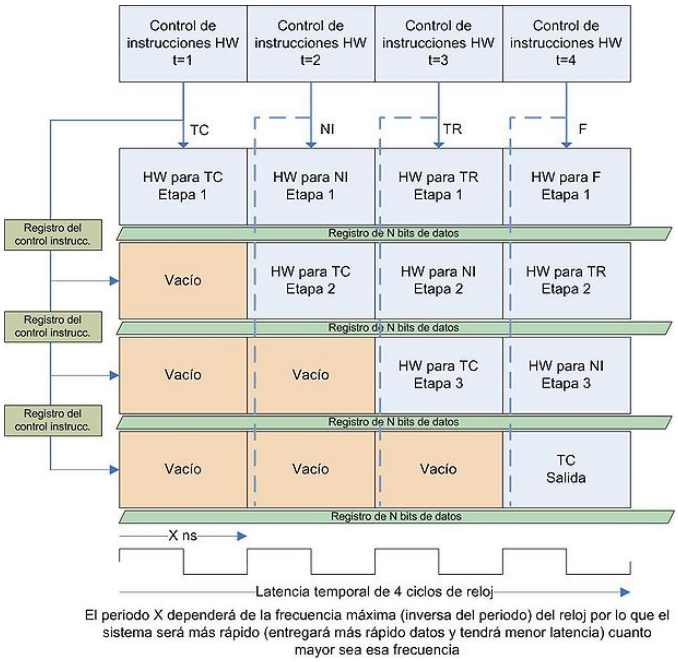

Figura 5: Detalle de la segmentación de instrucciones

La segmentación consiste en descomponer la ejecución de cada instrucción en varias etapas para poder empezar a procesar una instrucción diferente en cada una de ellas y trabajar con varias a la vez. Las etapas de segmentación pueden ser: IF (búsqueda), ID (decodificación), EX (ejecución), MEM (memoria), WB (estritura), en una arquitectura relativamente sencilla.

II. Sistema Desarrollado

Se pretendió implementar el diseño del diagrama en la figura 6.

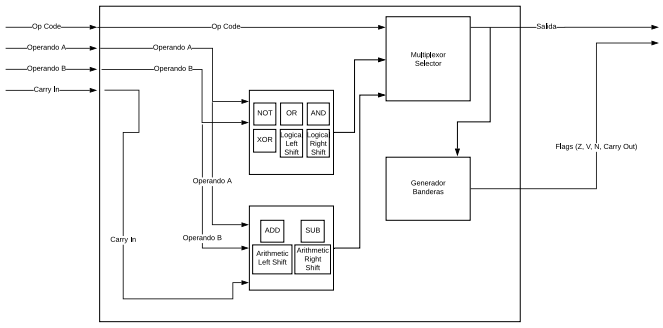

Figura 6: Diagrama de sistema a implementar

Se empezó elaborando un módulo que implementa la funcionalidad de un Desplazador de bits lógico, al cual se le agregó un testbench de auto-checkeo, verificando la salida esperada con diferentes entradas, una vez que se determinó que el módulo presentaba el comportamiento correcto, tanto en los casos comúnes, como en los casos extremos, se prosiguió a realizar lo mismo con módulos implementando Desplazadores de bits aritméticos, y luego las compuertas lógicas NOT, OR, AND, y XOR, además de un sumador y un restador, para instanciarlos en un módulo implementanto la Unidad Lógico-Aritmética, una vez que se verificó el comportamientod de cada uno de los módulos que implementan cada operación que la ALU debe soportar, se procedió a instanciar cada módulo en un módulo superior que contiene la implementación de la ALU, a la cual también se le agregó un testbench de autocheckeo. Luego de verificar que la selección de cada operación funcionaba (es decir que al usar el op-code de shift lógico izquierdo, la salida correspondía con esa operación, y así mismo para todas las demás operaciones), se procedió a implementar el cálculo de las banderas en la ALU, dentro del módulo de la ALU.

III. Resultados

Cada módulo implementado presenta el comportamiento esperado dadas las señales de entrada proporcionadas en los testbench de autocheckeo en *SystemVerilog*, usando señales *logic* de 4 bits, esto se comprobó en ModelSim desde Quartus, en diferentes máquinas para verificar la reproducibilidad de los resultados obtenidos.

IV. Análisis de resultados

Fue posible implementar una Unidad Lógico-Aritmética relativamente simple en *SystemVerilog* al "conectar" los submódulos que la componen, e implementando el generador de banderas, usando una cláusula *case* como un decodificador/-multiplexor para seleccionar la función de salida dependiendo del código de operación de entrada, aunque algunas prticularidades de *SystemVerilog* causaron confusión y atrasos.

V. Conclusiones

El diseño de componentes con lenguages de descripción de hardware puede parecer sencillo a primera instancia, cuando se ven ejemplos pequeños y simples, pero algunas sutilezas del lenguaje pueden resultar desafiantes, ya que el lenguaje usado (*SystemVerilog*) tiene aspectos confusos debido a su herencia de *Verilog*, y además difiere en aspectos importantes de lenguajes de programación, que probablemente muchos hayan usado antes, como C, en que para usar entradas/salidas con signo, sólamente se usa *int*, y para usar entradas/salidas sin signo, se agrega o se reemplaza con *unsigned*, mientras que en *SystemVerilog*, debe agregarse *signed*, y esto en algunos casos cambia el comportamiento de algunas funciones lógicas que se puede considerar intuitivo de manera considerable, por lo que los testbench, y en especial los de autocheckeo, resultan una herramienta indispensable para verificar los resultados esperados.

Referencias

[1] Harris, S. Harris, D. *Digital Design and Computer Architecture: ARM Edition*. Morgan Kaufmann, 2015.