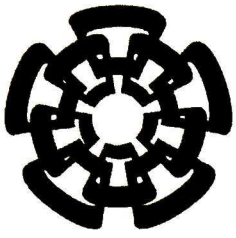
OT - 799-551
DON. 7014



Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional
Unidad Guadalajara

DISEÑO ASÍNCRONO EN DOBLE RIEL: Una metodología para la construcción de estructuras avanzadas

Tesis que presenta:
Edgardo de Jesús Pérez Casas

para obtener el grado de:
Maestro en Ciencias

en la especialidad de:
Ingeniería Eléctrica

Director de Tesis
Dra. Susana Ortega Cisneros

CINVESTAV del IPN Unidad Guadalajara, Guadalajara, Jalisco, Noviembre de 2013.

DISEÑO ASÍNCRONO EN DOBLE RIEL: Una metodología para la construcción de estructuras avanzadas

Tesis de Maestría en Ciencias
Ingeniería Eléctrica

Por:
Edgardo de Jesús Pérez Casas
Ingeniero en Electrónica
Universidad Tecnológica de la Mixteca 2004-2009

Becario de conacyt, expediente no. 365637

Director de Tesis
Dra. Susana Ortega Cisneros

CINVESTAV del IPN Unidad Guadalajara, Noviembre de 2013.

Agradecimientos

Esta tesis es el resultado del esfuerzo de varias personas e instituciones, que directa o indirectamente me ayudaron a terminarla. Al Cinvestav con sus académicos, en especial a los doctores Federico Sandoval Ibarra, José Raúl Loo Yao y Juan Luis del Valle, a los amigos de la generación que me acompañaron en la trayectoria y dieron fuerza durante el, al Conacyt por administrar los pocos recursos que se le confieren para la ciencia y educación, a mis padres y hermano que siempre están ahí para mí, a la Doctora Susana Ortega Cisneros, por haberme dado su confianza para realizar este trabajo y haberme empujado a terminarlo cuando más lo he necesitado.

Resumen

La tesis aquí presentada plantea una metodología para el diseño de circuitos asíncronos de Doble Riel. Se revisará la información existente, sentando las bases para establecer una metodología en el diseño de estos circuitos, así como variaciones a los protocolos existentes para realizar diseños más complejos y eficientes.

Los sistemas asíncronos de Doble Riel fueron introducidos a finales de los 50's y principalmente a principios de los 60's por David Muller [1]. Estos circuitos se definieron como aquellos capaces de funcionar sin necesidad del uso de una señal de reloj. Las ventajas principales que ofrecieron al momento de su aparición, eran la de un uso eficiente de energía, un espectro de ruido más distribuido, menor latencia y mayor velocidad de respuesta, entre otras.

Sin embargo, entre algunas de las desventajas, se encuentra la gran ocupación de espacio requerido para diseñar circuitos de este tipo, así como la dificultad para diseñarlos, que junto con la poca documentación que existe al respecto establecen una barrera inicial al diseñador, que tiene que generar en la práctica la mayoría de las estructuras.

Por esto, el objetivo principal de esta tesis, es de establecer las bases para el diseño en Doble Riel, describiendo primero cómo se construyen las estructuras más básicas, para luego construir unidades más complejas y de mayor utilidad.

Un aspecto importante de esta tesis es también abonar a la discusión acerca del tipo de circuitos que pueden ser construidos con esta señalización, delimitando su aplicación a campos donde verdaderamente sean eficientes y presenten una mejoría respecto a su contraparte síncrona. De esta manera, se pretende esclarecer por qué los circuitos asíncronos no son un mero sustituto a sus versiones síncronas, sino una forma de diseñar y pensar diferente, que en ciertos casos puede generar diseños más eficientes en velocidad, uso de energía, espacio y distribución de ruido, es decir, tener las ventajas de ambos tipos de diseño.

Así, este trabajo presenta las bases para diseñar circuitos asíncronos de Doble Riel con un protocolo de 4 fases. Después se presentan modificaciones a este protocolo y sus ventajas respecto al protocolo ya establecido. Se incluyen comparaciones al diseñar circuitos síncronos, asíncronos de riel sencillo, asíncronos de Doble Riel de 4 fases y asíncronos de Doble Riel de 4 fases modificado. Por último, la mayor contribución de este trabajo será la construcción de un circuito más complejo, un multiplicador de Booth de 8 bits con una sola línea de sumadores en el protocolo de 4 fases modificado propuesto.

Abstract

This thesis presents a methodology for Dual Rail Asynchronous design. It will review the current information for a better methodology construction and more complex and efficient designs through modifications to existing protocols.

Dual Rail Asynchronous designs were introduced in the late 50's and early 60's by David Muller. These circuits were defined as those capable of working without clock signaling, introducing *handshaking* for circuits communication. Among the major advantages of these circuits, we can highlight efficient use of energy (Low-power design), electromagnetic emissions better distributed along the spectrum, less latency, and faster response speed.

However, among some disadvantages for these circuits, we have the amount of space required for their physical design and complexity, which along with the poor documentation about these type of circuits makes for a very difficult start to any interested designer.

For te above stated reasons, the objective of this thesis is to establish the basis for Dual Rail Design, describing how basic structures are built, and then explaining the construction of more complex and common structures.

An important aspect of this thesis is to discuss the type of circuits that can be designed with asyncrhonous structures, establishing their use in fields where they are truly efficient and present an improvement with respect to synchronous design.

Índice General

Índice de Figuras

Índice de Tablas

Lista de Acrónimos

ASIC. *(Application specific integrated circuit)* Circuito integrados de aplicación especifica.

BCA. Bloque de control asíncrono.

CI. Circuito Integrado.

LUT. *(Lookup table)*

VHDL. *(Very high speed Intregated Circuit & HDL)* Lenguaje de descripción de hardware de alta velocidad para circuitos integrados.

FPGA. *(Field Programmable Gate Array)* Arreglo de compuertas programables en campo.

ALU. *(Arithmetic Logic Unit)* Unidad Aritmético Lógica.

FIFO. *(First In, First Out)* Primero en entrar, primero en salir.

HDL. *(Hardware Description Language)* Lenguaje descriptor de Hardware.

Capítulo 1

Introducción

En este captítulo se presenta el objetivo general de la tesis, así como los objetivos particulares a lograr; también se precisa la metodología a seguir en el trabajo y la justificación por la que este trabajo es importante. Este capítulo fundamentalmente describirá la intención de la tesis y lo que terminará aprendiendo el lector.

1.1 Planteamiento del problema.

La construcción de circuitos asíncronos de Doble Riel, en cualquiera de sus variantes, es un proceso que ha sido caracterizado por realizarse con poca o nula metodología establecida. Cada circuito se da al entendimiento de quien diseña (el mismo que tiene que realizar todos el proceso necesario hasta su construcción) y el establecimiento de una metodología para su diseño, no existe como tal.

A lo largo de su historia, desde que fueron presentados los circuitos asíncronos, han sido resaltadas varias de sus ventajas, como la eficiencia en el uso de energía, o una mejor distribución del ruido electromagnético que genera, aunque no se han esclarecido algunas de las desventajas que uno enfrenta al querer diseñarlos, entre ellas, una de las más importantes es la falta de una metodología para su diseño, verificación, fabricación, etc. Al menos comparable con la existente en el diseño de circuitos síncronos.

Otro problema común en quien busca realizar circuitos asíncronos, especialmente en Doble Riel, es la dificultad de apartar la mentalidad síncrona al diseñarlos. Esto significa que el diseñador actual está acostumbrado (incluso sin que se dé cuenta) en una forma de pensar en la cual todo tiene que estar perfectamente dividido por bloques, los cuáles solo ejercen una sola tarea y en la mayoría de los casos no son reusables para otros fines, estableciendo una forma de funcionamiento mecánica y poco flexible, con poca o nula capacidad para reaccionar de manera distinta ante diferentes estímulos, procurando siempre la regularidad. También la existencia de circuitos asíncronos de riel sencillo, es una muestra de esta necesidad de pensar síncronamente, más bien delimitada por la industria actual.

El hecho de que esto no suene a una desventaja para la mayoría de los lectores y diseñadores, es incluso un efecto de esta forma de pensar síncrona. Y es que prácticamente el que toda la industria electrónica funcione de acuerdo a este concepto, no significa que es la manera más eficiente que tienen los circuitos de operar, significa que es la manera más eficiente que tiene la industria para crearlos, y venderlos.

Una de las principales ventajas de los circuitos asíncronos, es precisamente esta capacidad de adaptación, de rehusar sus circuitos y responder de manera diferente ante cada estímulo que se le dé, en muchos casos siendo eficiente tanto en velocidad de respuesta como en uso de energía para responder, utilizando solo los recursos necesarios para desarrollar su tarea. En el caso de los circuitos síncronos, la mayoría hacen trabajar todos sus bloques independientemente de si la información que procesan es necesaria.

De aquí podemos tratar de dar solución a dos problemas perfectamente alcanzables, la necesidad de establecer una metodología para el diseño de circuitos asíncronos de Doble Riel, en la cual se incluya una nueva forma de pensar el diseño de circuitos asíncronos, y la segunda, complementando a la primera, establecer para qué circuitos es conveniente ya que no todas las estructuras deberían ser portables.

1.2 Justificación.

Si damos solución a los problemas mencionados, podremos establecer una base más sólida, que sirva precisamente a un diseño más avanzado, y no meramente sustitutivo de los circuitos síncronos. La idea de esta tesis, radica

principalmente en el potencial de los circuitos asíncronos de Doble Riel, uno no explotado aún, por desviar sus aplicaciones a meramente sustituir bloques síncronos, lo cual por definición, pone en desventaja a los primeros.

De esta manera, establecer el potencial de los circuitos asíncronos, especialmente de Doble Riel, no radicará en que pueden sustituir por completo a su contraparte síncrona, sino por el contrario, complementarlos, y en algunos casos, realizar algunos de sus equivalentes de manera más eficiente, debido a su naturaleza "asíncrona" original.

La importancia de dejar claro esto, incluso ayudará a aquellos que diseñan circuitos en un riel sencillo, pues estos no son más que una derivación de los circuitos asíncronos originales, aquellos que deberían ser definidos por su capacidad de respuesta individual a cada estímulo, y no a la de un sistema general que lo coordina al funcionamiento de otros bloques.

1.3 Objetivos

1.3.1 Objetivos Generales

- Establecer una metodología para la construcción de circuitos asíncronos de Doble Riel.
- Determinar para que circuitos es conveniente usar este tipo de diseño. En base a esto crear una estructura compleja de naturaleza asíncrona y demostrar las principales ventajas sobre su equivalente síncrono.

1.3.2 Objetivos específicos.

- Realizar bloques de interconexión para la comunicación entre estructuras asíncronas de Doble Riel, asíncronas de riel sencillo y síncronas.

  Realizar un circuito en riel sencillo, demostrando algunas ventajas y desventajas de este tipo de diseño, además de su similitud con los circuitos síncronos más que con los asíncronos.

  Demostrar en qué plataformas es conveniente diseñar circuitos asíncronos de Doble Riel, esclareciendo las limitaciones de usar aquellas que han sido diseñadas para la creación de circuitos síncronos..

1.4 Hipótesis

El desarrollo de estructuras asíncronas de Doble Riel que sean más eficientes que sus contrapartes síncronas, en cuanto a espacio, energía y velocidad de respuesta, es una posibilidad.

El desarrollo de una forma de pensar asíncrona como una necesidad primordial para poder diseñar circuitos puramente asíncronos se vuelve un aspecto fundamental para la correcta consturcción de estructuras asíncronas.

Una vez establecida la nueva forma de pensar, con las herramientas básicas y una metodología encaminada a la creación de ciertas estructuras donde sea necesario el diseño asíncrono para una eficiencia máxima, nos garantizarán la creación de estructuras avanzadas, con desempeños inalcanzables mediante metodologías de diseño síncronas o asíncronas equivocadas.

1.5 Metodología.

La metodología a seguir en para la realización de esta tesis consite en:

a) Una introducción a los circuitos asíncronos convencionales y estudio del estado de arte.

b) Establecer los tipos de circuitos asíncronos. En especial sobre la diferencia de Riel Sencillo y Doble Riel, de 2 y 4 fases y la definición de circuitos Self-Timed.

c) Discusión acerca de la necesidad de una forma de pensar asíncrona por sobre el simple uso de estructuras asíncronas en sustitución de las síncronas.

d) Se escogerá una implementación típica síncrona con posibilidad de ser mejorada en su versión asíncrona.

e) Repaso de algunas estructuras básicas e implementaciones sencillas en Riel Sencillo.

Una vez realizados estos pasos se continuará con la aportación principal de esta tesis:

i) Elaboración de un nuevo protocolo modificado, optimizando el protocolo de 4 fases para la elaboración de estructuras asíncronas más veloces y con facilidad de ser reducidas en ocupación.

ii) Establecer estructuras nuevas básicas que serán usadas para implementarse en estructuras complejas.

iii) Diseño de un Sumador Completo de N bits como base para su extensión a Multiplicador.

iv) Propuesta de Multiplicador Asíncrono en Doble Riel de 8 bits con codificación de Booth y el procolo modificado de 4 fases.

Capítulo 2

Introducción a la Lógica Asíncrona

En este capítulo se presentará una breve introducción al tema de circuitos asíncronos. De esta forma podremos entender los circuitos posteriores en los cuáles se presentarán las propuestas de este trabajo.

2.1 Diseño Asíncrono.

En el diseño asíncrono, no hay un reloj global que se encargue de sincronizar un circuito, pero sí hay señales de control que se encargan de la comunicación de los módulos que componen el circuito para una correcta transferencia de datos.

Ventajas de circuitos asíncronos

Bajo consumo. Los circuitos asíncronos requieren un mayor número de transiciones pero se puede habilitar solo las áreas necesarias para el procesamiento, ahorrando así consumo de potencia.

Latencia. En el diseño asíncrono si se conoce el tiempo que requiere cada proceso se optimiza el flujo de información por ser independientes las transiciones en el circuito y no depender de un reloj global.

Modularidad. Los circuitos asíncronos no se ven limitados a la latencia de cada módulo, al no depender de la señal global de reloj.

Baja radiación electromagnética. Los circuitos asíncronos generan una menor radiación electromagnética comparada con los síncronos ya que la radiación se incrementa cuando el sistema usa mayores frecuencias de reloj.

Limitaciones de circuitos asíncronos

Los circuitos asíncronos no tienen los problemas que se generan del uso de un reloj, sin embargo, tienen sus propios problemas. A continuación se describen los más habituales.

Hazards. Los circuitos asíncronos deben ser diseñados cuidadosamente para evitar hazards [16], los cuales pueden activar falsas transiciones en las etapas posteriores del bloque en donde se presentan.

Variedad en metodologías de diseño. Debido a que hay una variedad de metodologías de diseño asíncrono, resulta inconsistente especificar e implementar estilos de diseño parecidos.

Metodologías de síntesis. Aunque el diseño asíncrono es una alternativa al diseño síncrono, es difícil encontrar herramientas de diseño que manejen la síntesis de diseños asíncronos.

A pesar de sus limitaciones, el diseño asíncrono sigue siendo una opción viable. Uno de usos principales es en el de procesos con una reducción de latencia y el bajo consumo de energía. En aplicaciones anteriores, es donde se tiene la ventaja más significativa del diseño asíncrono contra el síncrono. Podemos encontrar como ejemplo de esta cualidad de los circuitos asíncronos en [2, 3].

2.2 Clasificación de circuitos asíncronos

Existen varios modelos utilizados para el diseño de circuitos asíncronos. Estos modelos se diferencian por la manera de interpretar el retardo de los componentes y sus interconexiones. A continuación se describen algunos de estos modelos de diseño, los cuales son utilizados en muchos diseños de microprocesadores asíncronos.

2.2.1 Circuito insensible al retardo (*DI*)

En los circuitos asíncronos DI (del inglés Delay Insensitive) [4], los retardo de los componentes lógicos y el de sus interconexiones se suponen finitos pero sin limitaciones. Esto quiere decir que independientemente del retardo de los componentes del circuito, este funcionará correctamente. La implementación de este tipo de circuitos frecuentemente resulta en circuitería muy compleja. Sólo se pueden construir un reducido número de circuitos simples utilizando compuertas sencillas.

2.2.2 Circuito casi insensible al retardo (*QDI*)

Los circuitos asíncronos QDI (del inglés Quasi Delay Insensitive) [5], son muy parecidos a los DI por asumir retardos arbitrarios en componentes lógicas y sus interconexiones. Este tipo de circuitos hace uso de dos ramificaciones de salida por componente (isochronic fork [6]). Estos componentes pueden enviar información a dos destinos diferentes, pero sólo pueden recibir la confirmación del envío de uno. A diferencia de los circuitos DI, el uso del método QDI puede simplificar el modelado de circuitos asíncronos, pero requiere que el retardo de las ramificaciones sean insignificantes en comparación con los componentes del circuito. Además, los circuitos QDI requieren que los componentes de origen y destino de las ramificaciones tengan el mismo valor de umbral.

2.2.3 Circuito escalable insensible al retardo (*SDI*)

Los circuitos asíncronos SDI (del inglés Scalable Delay Insensitive) [7], surgen como una solución a los modelos DI y QDI, debido a que estos últimos requieren de una gran cantidad de componentes y pueden ocasionar un bajo

desempeño en la velocidad de los circuitos con respecto a otros modelos. El SDI es un modelo en el que no se asume ningún límite superior para el retardo de los componentes y de sus interconexiones. A diferencia de los modelos DI y QDI, el SDI asume que la relación del retardo relativo entre dos componentes tiene un límite. Los circuitos con componentes SDI pueden funcionar con mayor velocidad que los que utilizan el modelo SQI.

2.2.4 Circuito independiente de la velocidad (*SI*)

Los circuitos asíncronos independientes de la velocidad (Speed-Independent) [8], consideran que el retardo de los componentes es arbitrario y el de las interconexiones es cero o nula, por lo que puede permitir menor cantidad de lógica compleja. Los aspectos más importantes de este modelo funcionan como el modelo QDI, pero minimizando una gran cantidad de problemas que implican el uso del modelo QDI, como mayor tiempo de cómputo y complejidad al modelar cricuitos grandes. Como desventaja, son menos robustos.

2.2.5 Circuito auto-temporizado (*Self-Timed*)

En los circuitos asíncronos se requiere identificar cuándo un bloque combinacional produce un dato válido a su salida. Una forma de lograrlo es utilizando circuitos auto-temporizados o Self-Timed (ST) [9]. Éste método utiliza lógica redundante, donde las salidas de control de los circuitos ST representan que en alguno de los bloques a sincronizar existen datos válidos o inválidos, dependiendo del estado lógico en que se encuentren las señales de control (1 ó 0 respectivamente). Este método ST utiliza un protocolo de comunicación que asegura la transferencia de datos. Los circuitos Self-Timed utilizan interconexiones

de elementos básicos [10], los cuales pueden ser implementados con el modelo *Speed-Independent;* estos elementos básicos pueden comunicarse mediante el modelo DI [11].

También se puede hacer uso de la lógica Delay Maching [12], la cual utiliza lógica convencional que incluye una señal de control adicional que indica cuándo hay un dato válido. Habitualmente la señal de control pasa a través de la lógica del módulo a sincronizar, ésta señal debe de tener un retardo mayor al de los datos, para asegurar que hay un dato válido antes de la transferencia.

2.3 Protocolos de Comunicación Asíncrona

2.3.1 Simple

La principal característica de este protocolo de comunicación es que los datos son transportados en una sola línea de transmisión. Utiliza dos señales de control para hacer la comunicación entre los bloques emisor/receptor; estas señales son: la de request (petición) y la de acknowledge (reconocimiento). La señal de request, es enviada por el emisor e indica que hay un dato listo para ser transferido. La señal de acknowledge, enviada por el receptor, indica que el dato ha sido recibido. La codificación de los datos es la misma que se utiliza en los diseños síncronos, pero dependen del tiempo en que se hace la petición, ya que proviene de un retardo, el cual debe ser mayor al tiempo de latencia del dato del emisor para garantizar que el dato a transferir sea el correcto.

2.3.2 Doble

A diferencia del riel simple, la comunicación en Doble Riel utiliza dos líneas de transmisión para codificar un dato. En este protocolo la señal de request se encuentra implícita en la codificación de los datos. Al enviar un dato válido se genera automáticamente un request.

En la Tabla 2.3.1 se muestra la codificación utilizada para representar cada bit mediante dos líneas de transmisión.

Tabla 2.3.1 Codificación de datos en Doble Riel

DATO CODIFICADO	dt	df
Reset	0	0
Dato válido (0)	0	1
Dato válido (1)	1	0
No usado	1	1

En la Tabla 2.3.1, se puede observar que los valores "0" y "1" son codificados idénticamente en una señal "dt" y complementados en una señal "df" Para que se realice el request de una transmisión en Doble Riel, es necesario que todos los bits codificados en dos líneas contengan un dato válido, así la señal de acknowledge del bloque receptor se activará. Si alguno de los bits codificados del emisor se encuentra en reset, el receptor seguirá esperando hasta que todos los datos codificados tengan un dato válido. En la Figura 2.3.1 se puede apreciar la interación entre el receptor y el emisor.

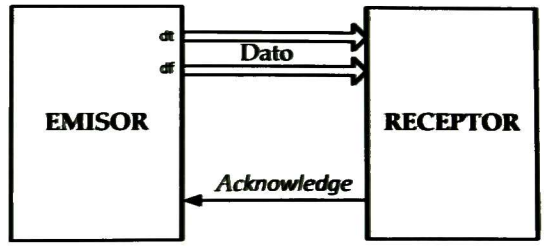


Figura 2.3.1 Comunicación básica en Doble Riel.

2.4 Protocolo *Self-Timed* en riel simple

2.4.1 Protocolo de 2 fases

En [13] se describe ampliamente el protocolo de comunicación de 2 fases, el cual se sintetiza en la Figura 2.4.1. Este protocolo utiliza una señalización de transición (sin retorno a cero), con dos señales de control, la de request y la de acknowledge.

Se le llama protocolo de 2 fases debido al número de transiciones que intervienen en la comunicación. La manera en que se realiza la comunicación es la siguiente: primero el emisor entrega un dato válido a la salida y produce un evento en la señal de request, después el receptor recibe los datos y produce un evento en la señal de acknowledge.

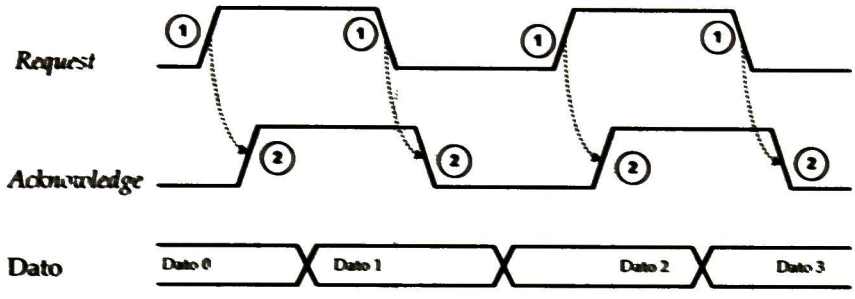


Figura 2.4.1 Protocolo de 2 fases en riel simple

2.4.2 Protocolo de 4 fases

El protocolo de 4 fases utiliza más transiciones en las líneas de comunicación (request y acknowledge), pero presenta una metodología de comunicación mejorada y es mejor adaptado a los requerimientos de circuitos estables. Este protocolo utiliza una señalización de nivel (con retorno a cero), y señales de control request y acknowledge.

Se le llama protocolo de 4 fases por el número de transiciones que intervienen en la comunicación. Estas transiciones son las siguientes: primero el emisor pone un dato a la salida y activa la señal de request en alto. Después el receptor toma el dato y activa la señal de acknowledge en alto. En seguida el emisor responde desactivando la señal de request en bajo. Por último el receptor cambia la señal de acknowledge a bajo.

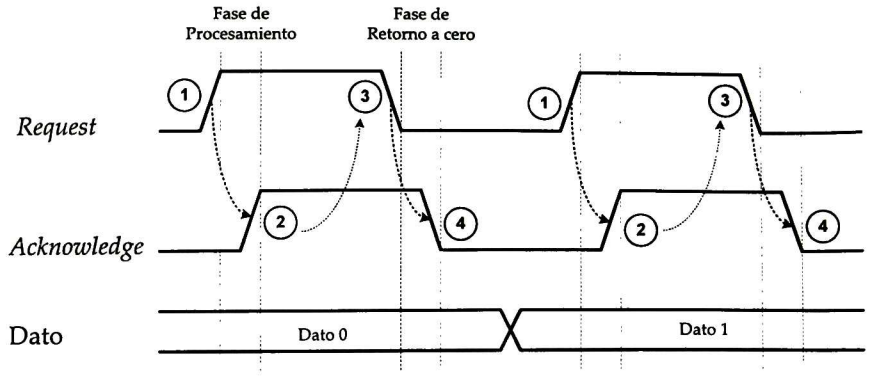


Figura 2.4.2 Protocolo de 4 fases en riel simple

El protocolo de 2 fases consume menor cantidad de energía que el de 4 fases, debido a que este último por cada transición de datos (fase de procesamiento) tiene que regresar a cero sus señales de control (fase de retorno a cero) para volver a hacer una transición, pero tiene la ventaja de que asegura una comunicación más estable.

2.5 Módulos elementales para el diseño *Self-Timed*

El diseño asíncrono se basa en transiciones producidas por eventos en las señales de control, debido a esto, para implementar los protocolos Self-Timed es necesario el uso de células elementales que funcionen con eventos y no con lógica convencional.

A continuación se describen algunas de las células elementales propuestas por Ivan E. Sutherland, en su artículo titulado "Micropipelines" [13]. Además se describen algunos tipos de implementación de retardos.

2.5.1 Muller C

El bloque Muller C es la implementación de una AND de eventos. Su señal de salida sólo cambiará a cero una vez que ambas señales de entrada valgan cero, pero para modificar el valor de salida a 1, las señales de entrada tendrán que haber cambiado a 1 (y mantenerse en ese valor). En otras palabras, la combinación de un 0 y un 1 en la entrada no producirá cambio alguno en la salida. En la Tabla 2.5.1 se muestra el comportamiento de ésta compuerta.

Tabla 2.5.1 Eventos del bloque Muller C

(I_1, I_2)	O
(0,0)	0
(0,1)	O
(1, 0)	O
(1,1)	1

A estos circuitos se puede agregar una entrada de reset, para poder inicializarlos con un cero lógico cuando sea necesario. En la Figura 2.5.1 se muestra como agregar la señal de reset a una Muller C de dos entradas. El uso de la señal de reset en estos bloques asíncronos ayuda a inicializar los circuitos correctamente, para evitar estados lógicos indeseables.

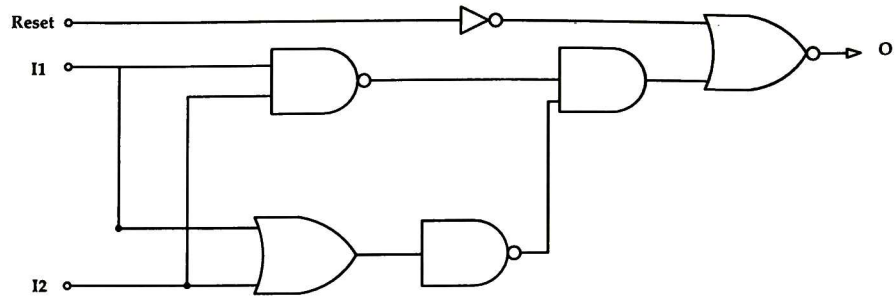


Figura 2.5.1 Circuito Muller C con *reset*

2.5.2 OR de eventos

Este bloque es comúnmente llamado elemento de unión, debido a que es utilizado para unir dos eventos en uno. Este bloque se puede construir simplemente con una compuerta XOR. En la Figura 2.5.2 se muestra su comportamiento.

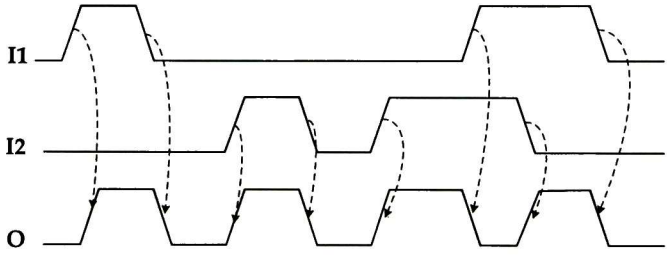


Figura 2.5.2 Comportamiento del bloque OR de eventos.

2.5.3 TOOGLE de eventos

El TOGGLE se compone de una entrada y dos salidas. Este bloque recibe un evento por la entrada "R" y después dirige eventos hacia las salidas de forma alternada, dependiendo de la transición de estado lógico (de '0' a '1', o de '1' a '0')

de la entrada "R" [14, 15]. El comportamiento de este bloque se muestra en la Figura 2.5.3a.

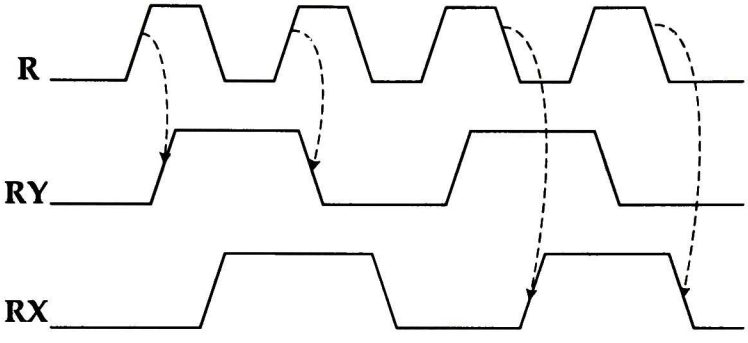


Figura 2.5.3a Comportamiento del bloque TOGGLE

En la Figura 2.5.3b se muestra un circuito para implementar el bloque TOGGLE. El punto negro que se encuentra en la salida "RY", indica que al hacer la primer transferencia en la entrada "R" ('0' a '1') el primer evento saldrá por esta salida.

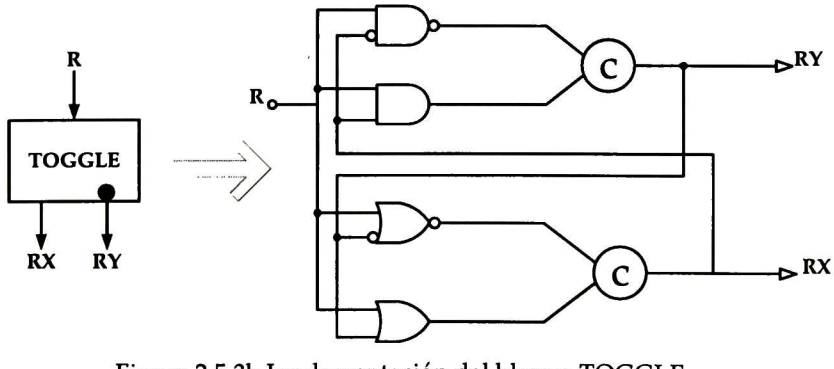


Figura 2.5.3b Implementación del bloque TOGGLE

Capítulo 3

Diseño de circuitos Síncronos y Asíncronos de Riel Sencillo y Doble Riel

En este capítulo se presentará un análisis acerca de la portabilidad de circuitos síncronos a sus equivalentes asíncronos, analizando en que casos podrían los circuitos asíncronos, especialmente de Doble Riel, superarlos en uso de área, energía usada y velocidad de operación. Al final del capítulo se describirá el desarrollo de una UART de Riel sencillo auto-conFigurable.

3.1 Diferencias entre diseños Síncronos y Asíncronos.

A lo largo de su historia, uno de los objetivos de los circuitos asíncronos ha sido el de reemplazar las versiones síncronas exitentes por éstos. Sin embargo, y como lo menciona [1], una vez comprendido el diseño en Doble Riel, que es como originalmente surgieron los circuitos asíncronos, es natural observar que la mera suplantación de cada bloque síncrono dentro de un circuito, por su equivalente asíncrono, no generará, en la mayoría de los casos, los mejores beneficios posibles de este diseño. A lo largo de este capítulo se presentará una serie de ejemplos para ayudar a establecer una mejor idea acerca de esta portabilidad y cuál debe ser el objetivo al crear circuitos asíncronos de Doble Riel, identificando que no es posible aprovechar su máxima utilidad pensando síncronamente y construyendo asíncronamente.

3.1.1 Pensamiento Síncrono vs Pensamiento Asíncrono.

Como se menciona en [1], el desarrollo del diseño de circuitos integrados ha sido siempre bajo el concepto de sincronía. La forma compartida del trabajo en la elaboración de tecnología ha impregnado incluso al diseño mismo de esta particularidad, poder fraccionar el funcionamiento de cualquier circuito en bloques que puedan ser probados por separado, obedeciendo al final solo lineamientos interpuestos, en este caso, por una señal de reloj que determinará el tiempo en el que debe estar computada la información requerida.

El diseño de circuitos asíncronos, sin embargo, vino a romper este esquema. Permitiendo una mayor interacción entre los bloques, y no obedeciendo a una

señal global, la velocidad en la que son procesadas las operaciones varía siempre dependiendo de los datos a computar. Cada bloque puede ahorrar trabajo y tiempo de ejcución al resto de los circuitos, es decir, contribuir al desempeño global en términos de eficiencia, no así con los circuitos síncronos, en el cual, en el mejor de los casos sólo puede contribuir con el ahorro de espacio o energía consumida.

Incluso, yendo más allá, se podría decir que la mayoría del diseño de los circuitos asíncronos, no tiene por que ser visto desde el punto de vista de bloque, al menos de uno aislado del resto. El diseño de un circuito asíncrono, desde el punto de vista conceptual, debería ser más bien un entramado de bloques que comparten ciertas partes físicas con el resto del circuito para reutilizar espacio y componentes, reducir el ruido, ocupación y energía necesaria para funcionar, siendo en la mayoría de los casos mucho más rápido que un circuito síncrono, es decir, en lugar de aislarlo para hacer una tarea específica, debería ser un componente dentro del circuito que no tenga límites definidos y pueda realizar varias funciones de manera inteligente.

Para fundamentar lo anterior, se describirá de manera rápida el funcionamiento de los circuitos síncronos y el de los dos tipos principales de circuitos asíncronos (de Riel Sencillo y Doble Riel) en base a la señal de reloj.

3.1.2 La señal de reloj como base para diferenciar el diseño síncrono y asíncrono.

En esta sección, se describirá el funcionamiento de los circuitos síncronos y asíncronos, para entender de mejor forma la diferencia entre estas dos formas de diseño. De manera general, se puede observar la modularización del diseño síncrono, el diseño en riel sencillo elimina la necesidad de una señal global de reloj, sin embargo aún sigue dependiendo de pequeñas señales de reloj locales, las cuáles están determinadas por el peor tiempo de cada bloque, lo que sin duda ya es un avance. Por último, el diseño en Doble Riel implicará que desaparezca por completo cualquier tipo de señal de reloj, los bloques podrán avisar cuando sus señales ya hayan sido computadas de forma no solo independiente entre bloques, sino independiente de cada operación. De esta forma, cada señal computada tendrá un tiempo de procesamiento diferente no solo con otras, sino consigo misma a lo largo del tiempo de funcionamiento del circuito, tratando siempre de aprovechar las condiciones en las que se ejecute para el mayor ahorro de energía y mayor rapidez para su ejecución.

3.1.2.1 Señal de Reloj y los Circuitos Síncronos

El funcionamiento de una señal de reloj dentro de un circuito síncrono, puede ser descrita usando la Figura 3.1.2.1. En ella, podemos observar que el funcionamiento del sistema está perfectamente comportamentado. El tiempo que determina la señal de reloj está determinado siempre, por el proceso más tardado del bloque más tardado del circuito.

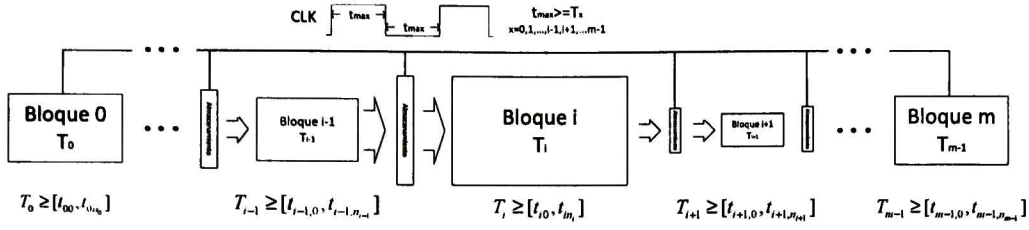


Figura 3.1.2.1. Uso de una señal de reloj global en un circuito síncrono.

Esto quiere decir, en otras palabras, que para asegurar que en cada ciclo de reloj todos los bloques hayan terminado de procesar la información que se les pide computar, se debe esperar siempre el tiempo de procesamiento más largo de todos los bloques, incluso si éste no sucede, y así la información se asegurará que es correcta en todos los casos. De esta forma, cada bloque debe saber el tiempo más largo posible que le podría tomar procesar sus datos de entrada. Una vez que cada bloque sabe esta información, se comparan estos tiempos, tomando del más largo, el tiempo que determinará el ciclo de reloj.

Usando la Figura 3.1.2.1, tenemos m bloques, en donde cada bloque tiene un número n de posibles tiempos para ejecutar la información que se le solicita. Usando el ejemplo sencillo de una compuerta NOT, podríamos decir que tiene dos posibles tiempos de respuesta, de pasar de 0-1 y de 1-0, que aunque son cercanos, no son iguales. En este caso n sería 2. De aquí el valor que nos interesa, es el más grande, a este se le llamará, para el bloque 0, T_0, el cual debe cumplir la Ec 3.1.2.1:

$$T_0 \geq [t_{00}, t_{0n_0}] \qquad \text{Ecuación 3.1.2.1}$$

Repitiendo esto para cada bloque, tendríamos los máximos tiempos de cada uno, donde para el bloque x se tendría que cumplir en general la Ecuación 3.1.2.2.

$$T_x \geq [t_{x0}, t_{xn_x}] \qquad \text{Ecuación 3.1.2.2}$$

Cabe señalar que este tiempo máximo para cada bloque no se presenta con tanta regularidad para la mayoría de los circuitos, esto por que el tiempo máximo siempre tiene que juntar ciertas condiciones que propicien el peor de los casos, que en lógica como sumadores, multiplexores siempre se traduce en casos de baja probabilidad pero que sin embargo tienen que ser tomados en cuenta siempre.

Una vez reunidos estos valores, podríamos determinar la señal de reloj usando el mayor de éstos, esto sería obedeciendo la Ecuación 3.1.2.3:

$$t_{max} \geq [T_0, T_{m-1}] \qquad \text{Ecuación 3.1.2.3}$$

De aquí se obtienen algunas particularidades de los circuitos síncronos, como el hecho de que con una variación de temperatura o voltaje, al moverse la señal de reloj ligeramente, esta ya no pueda garantizar que todos los bloques vayan a tener sus datos listos para cuando ésta cambie.

3.1.2.2 Circuitos Asíncronos de Riel Sencillo.

Curiosamente, los circuitos asíncronos de Riel Sencillo, son más parecidos a los circuitos síncronos que a los circuitos asíncronos, ya que operan bajo la lógica de suplantar una señal de reloj global por señales de reloj locales. Usando la Figura 3.1.2.2 podemos observar ahora que la señal de reloj global ha sido sustituida, por pequeñas señales de reloj. Aunque físicamente no existe como tal una señal de

reloj, ni osciladores, es más fácil comprender como funcionan estos circuitos de esta forma.

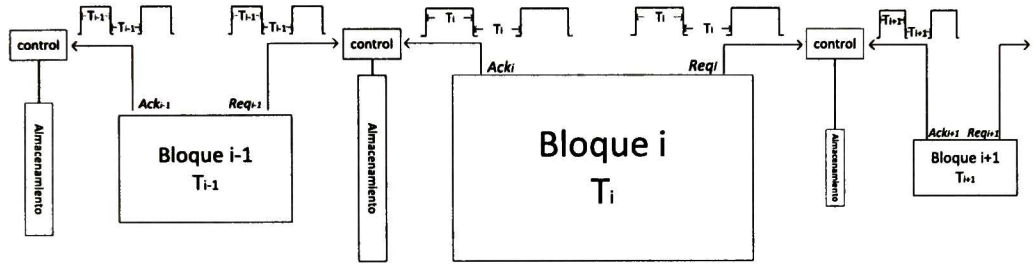


Figura 3.1.2.2. Señal de reloj globlal sustituida por señales locales.

Ahora, consideremos que no debemos limitar a todos los bloques a funcionar de acuerdo a los tiempos del más lento, pensemos que podemos pedirle a cada bloque que nos avise cuando su información haya sido procesada y esté listo para recibir nueva, manejando su propia señal de reloj. Entonces tendríamos un diseño asíncrono de Riel Sencillo. En este caso, las señales de petición y reconocimiento serán necesarias para manejar estos estados en los que se pude recibir información, se está procesando y se ha finalizado.

Esto quiere decir que tendremos que utilizar retardos en las señales de petición y reconocimiento para que aseguremos que en cada bloque los datos son computados correctamente, para esto cada retardo estaría determinado por:

$$t_{0i_0} \geq [t_{00}, t_{0n_0}]$$

Así pues, aunque cada bloque no tiene que tomar en cuenta el tiempo que le toma al resto de los bloques procesar su información, debe al menos tomar el tiempo más grande de todos los que el tendrá.

Sin embargo, aunque esto es una mejora, especialmente para el consumo de energía, esto no aprovecharía enteramente la paritucularidad de los circuitos asíncronos.

3.1.2.3 La verdadera asincronía en los Circuitos Asíncronos de Doble Riel.

El caso de los circuitos Asíncronos de Doble Riel, es muy interesante. El punto de alcanzar total asincronía o no necesitar una señal de reloj para nada se lograría si en los casos explicados con anterioridad (circuitos síncronos y asíncronos de Riel Sencillo), ninguna operación tuviera que estar determinada por otra, ni de su propio bloque, ni mucho menos de otros. Esto, traducido al funcionamiento que se ha venido describiendo, significaría que las señales de petición y reconocimiento no tuvieran que depender más que de la operación realizada y la desaparición total de una señal de reloj global. Dicho de otra forma, cada grupo de datos a procesar tendría que tener la capacidad de poder determinar cuándo han sido ya computados correctamente, pasando esta información al siguiente bloque, y requeriendo nueva información para procesar inmediatamente.

De esta forma, cada conjunto de datos a procesar tendrá un tiempo de ejecución totalmente diferente al resto de las demás, único.

Otra característica a resaltar es la del aprovechamiento de estos tiempos diferentes; mientras que en el Riel Sencillo había una pequeña ganancia, pero debido a los cuellos de botella que un bloque con gran retardo puede generar en el pipeline, no se aprovechaban correctamente, en este caso, como los bloques más tardados podrán ser más eficientes en la mayoría de los casos (recordando que los

peores casos se presentan con poca regularidad), las operaciones serán ejecutadas más rápidamente.

De aquí, se puede observar que para implementar esto, los datos, al no poder solos determinar cuando son ya válidos, tienen que tener un tercer estado diferente del de 1 ó 0, el estado de inválido. Este estado nos ayudará a saber cuando la información aún no esté lista. Para poder dotar a cada dato con esta propiedad se tendrá entonces que codificar con dos líneas de información, en lugar de una. Cada dato contará con un par de líneas de información, la línea de información *.f* nos ayudará a determinar cuando el dato tenga un valor 0 válido y la línea *.t* cuando sea un 1 válido.

El mayor reto entonces será el de ocupación, pues prácticamente codificar dos líneas de información por señal, en lugar de una, nos haría ocupar el doble de ocupación. Este problema y su análisis se detallarán en la sección 3.1.4.3.

3.1.3 El uso estratégico de la energía en los circuitos asíncronos.

Un aspecto a resaltar acerca de los circuitos asíncronos, es el uso eficiente de la energía que usan para funcionar. Dependiendo de si es de 2 ó 4 fases, será el consumo ahorrado (siendo el de 2 fases el más eficiente). En las siguientes secciones se describirá el comportamiento general de cada tipo de diseño en cuanto al uso de energía.

3.1.3.1 Circuitos Síncronos y la ejecución de operaciones en todo tiempo.

Para el consumo de energía, los circuitos síncronos tienen el inconveniente de generar transiciones falsas. Esto significa que antes de que su valor sea válido, la mayoría de las compuertas generarán estados intermedios. Analicemos el circuito de la Figura 3.1.3.1, en el cual dos compuertas generarán transiciones falsas en el valor de salida.

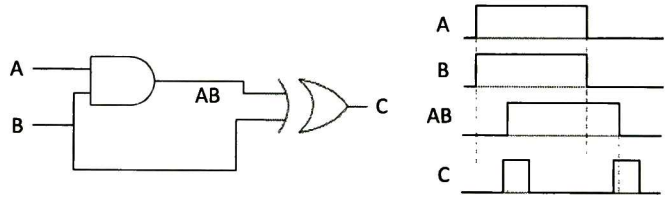


Figura 3.1.3.1. Transiciones Falsas en compuertas digitales.

Supongamos que en un tiempo dado, A y B pasan del valor 0 al valor 1 en el mismo instante. Al hacer esto, la señal de salida del XOR cambiará de 0 a 1, al mismo tiempo la compuerta AND a su salida estará cambiando de igual manera, de 0 a 1. Una vez que la salida de la compuerta AND es 1, el valor de la XOR cambiará de 1 a 0 nuevamente. Este valor momentáneo de 1 es a lo que se le llama una falsa transición.

¿Por qué es una desventaja para el uso de energía? Sencillamente por el consumo de energía que le hizo pasar de 0→1→0. Si tomamos en cuenta que este valor de igual manera está conectado a más compuertas, no sería extraño que éstas presenciaran una falsa transición, lo que produciría más falsas transiciones, como una reacción en cadena. Al final, si la señal B pasara al valor de 0, se generaría una falsa transición nuevamente, de 0→1→0. Es decir, en ambos casos la señal de salida siempre fue de 0, pero cambió 4 veces, produciendo seguramente muchas más transiciones falsas a lo largo de la lógica que le precede.

De esta forma, en los circuitos síncronos el consumo de energía es elevado, debido a la gran cantidad de switcheos de los transistores al cambiar de 1→0 ó de 0→1, siendo una significaitva cantidad de éstos, falsas transiciones.

3.1.3.2 Riel Sencillo, una mejora en el consumo de energía.

En este aspecto, los circuitos asíncronos de Riel sencillo aportan una mejora liegramente significativa, ya que sus bloques no están funcionando todo el tiempo, solo cuando estos se requieren, las falsas transiciones no son evitadas, ya que al usar una línea de información por dato no permite establecer cuándo una señal ya es válida y por lo tanto pasar información incorrecta antes de que todas las señales se estabilicen.

Sin embargo hay un ahorro significativo cuando el circuito contien muchos bloques que normalmente deberían de funcionar todo el tiempo, ya que se procura mantenerlos sin procesar información.

3.1.3.3 Doble Riel y el uso eficiente de la energía.

En el caso del Doble Riel, el uso de la energía es optimizada al máximo. Supongamos el caso mostrado en la Figura 3.1.3.3a. Como se explicó en los circuitos síncronos, las falsas transiciones consumirán gran parte de la energía, en Doble Riel, se eliminan por completo. Independientemente de si es de 2 o 4 fases o fuerte/débilmente indicante, como las salidas de cualquier bloque cambiarán una sola vez, es decir, cuando ya esté lista su información, no generará una falsa información que haya que corregir.

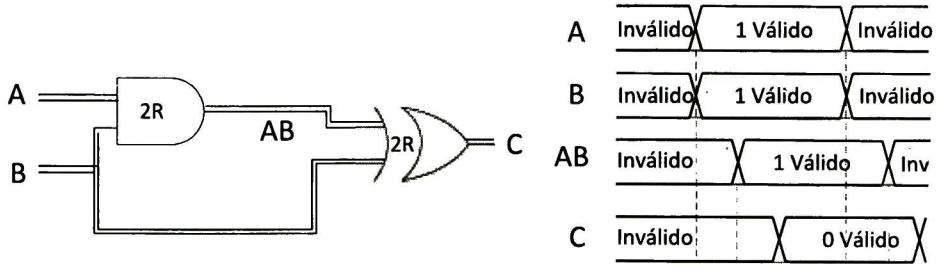


Figura 3.1.3.3a. Compuertas en Doble Riel evitando transiciones falsas.

Supongamos que en A se da un cambio, a un 0 ó 1 válido, si el bloque 1 puede generar una salida válida, lo hará, el cual tomará el bloque 2 y hará lo mismo, en caso contrario, su salida permanecerá en 00 (inválido), luego entonces el bloque no se modificará para nada, evitando una falsa transición. En pocas palabras, en los circuitos asíncronos de Doble Riel no están permitidas las falsas transiciones (un requisito fundamental para determinar datos válidos o inválidos) y para una operación en un bloque, solo utilizará la energía necesaria para procesar esa información.

Cabe señalar que en el caso del protocolo de 4 fases, habrá un gasto de energía necesaria para regresar todos los valores a 00 y que el bloque esté listo para una nueva operación, como se puede ejemplificar en la Figura 3.1.3.3b. En este caso, una vez que la información ha sido procesada y está lista para el siguiente bloque, todos los datos deben regresar a su estado inválido, antes de recibir nueva información.

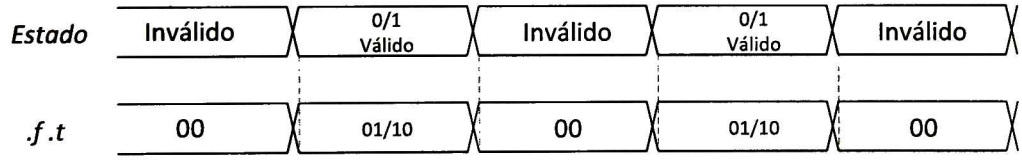


Figura 3.1.3.3b. Transiciones entre datos válidos e inválidos necesarias.

En el caso del protocolo de 2 fases, el ahorro es máximo, ya que las señales solo cambiarán una línea de información por cada paquete de datos a procesar. En la Figura 3.1.3.3c se puede apreciar esto. Cada vez que se requiera procesar información, solo cambiarán los circuitos necesarios, además de que no tendrá que regresar a estados inválidos o válidos que ocasionarían switchear a los transistores.

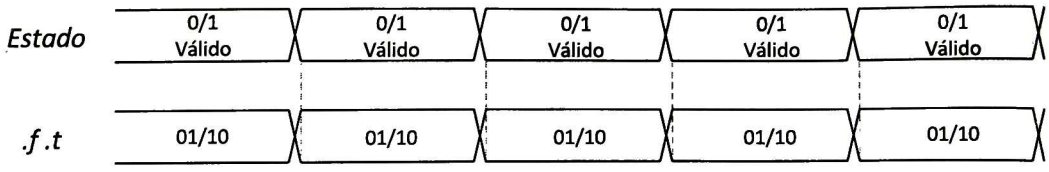


Figura 3.1.3.3c.Protocolo de 2 fases sin transiciones a datos inválidos.

Incluso, el uso de una señalización débilmente indicante, ayudará a ahorrar aún más energía. Si tomamos en cuenta que no es necesario esperar a todas las entradas para generar una respuesta de un bloque dado y que sea utilizada esta propiedad para evitar seguir requieriendo datos, estos se evitarán procesar.

3.1.4 El uso del espacio en los circuitos asíncronos.

Uno de los principales problemas que se tienen al diseñar en circuitos asíncronos de Doble Riel, es el de la ocupación. Hay que recordar que el uso de una doble línea de información por cada dato a procesar, consumirá, al menos a primera vista, no solo doble espacio requerido para su traslado a lo largo del circuito, sino doble número de compuertas, al menos compuertas a nivel de diseño digital.

Sin embargo, tampoco hay que olvidar que el espacio requerido siempre se podrá disminuir cuando los diseños sean portados finalmente a nivel transistor, donde lo que consideramos como un conjunto de compuertas lógicas, agrupadas, pasan a ser vistas como funciones lógicas, permitiendo disminuir el espacio requerido para implementarlas.

En la primera parte de esta sección se explicará con un ejemplo claro esta capacidad a la hora de diseñar CI's, para el caso de un circuito síncrono. Más adelante se tratará con detalle como portar ciertos circuitos o bloques básicos de nivel compuerta (donde la ocupación es máxima) a nivel transistor.

En términos generales, aunque la ocupación puede ser disminuida, al hablar en términos de comparar un bloque síncrono con su contraparte asíncrona, aún así existiría un aumento considerable de espacio usado. No es si no hasta que se agrupan más funcionalidades o la reutilización de bloques que se puede observar un verdadero aprovechamiento del espacio, aunque esto se verá a más detalle en el capítulo 4.

3.1.4.1 Circuitos Síncronos, el diseño y traslado de nivel compuerta a nivel transistor.

En el diseño de circuitos síncronos, una vez elaborados los bloques, es necesario agregar la circuitería y control que permitirá que la señal de reloj, en cada ciclo, permita que cada bloque obtenga un nuevo paquete de datos a procesar. Para esto, se utilizan medios de almacenamiento, que por una parte ayuden a retener los datos que están siendo procesados, mientras que en cada ciclo de reloj, se dejen pasar los nuevos. En la Figura 3.1.4.1a se muestra el funcionamiento de 2 formas de almacenamiento, el Flip Flop y el Latch.

Los Flip Flops, permiten el reconocimiento del cambio de flanco que hace la señal dereloj, esto es, pasar de 0→1 o de 1→0, siendo éste el único instante en el cual la información es pasada de un punto a otro.

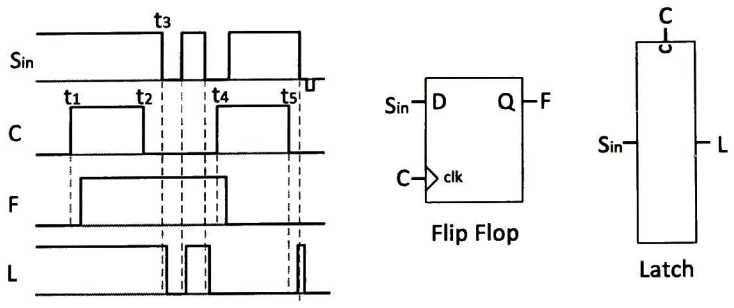


Figura 3.1.4.1a. Funcionamiento del Flip Flop y Latch.

Los latches, por el contrario, solo almacenan su información ante un 1 ó 0 lógico en su señal de control, es decir, responde ante valores lógicos y no cambios de flanco de estos valores.

Ahora bien, en realidad, cuando los Circuitos Integrados son fabricados, se hacen a nivel transistor. En la Figura 3.1.4.1b podemos observar cómo las compuertas AND y OR pueden ser implementadas a este nivel.

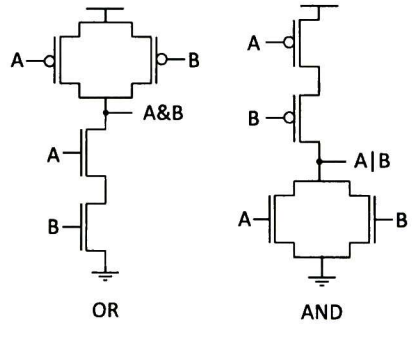


Figura 3.1.4.1b Compuertas AND y OR a nivel transistor.

Actualmente el diseño a nivel transistor se realiza con herramientas tales como Tanner [16]. Esto por que realizar la construcción a nivel transistor de millones de transistores nos tomaría demasiado tiempo, lo cual es evitado pasando el diseño de nivel RTL (Register Transfer Level), el cual ya contiene la descripción del circuito en lógica digital y de registros.

3.1.4.2 Riel Sencillo y el ligero aumento de espacio.

En el caso del diseño de circuitos asíncronos en Riel Sencillo, la ocupación no incrementará en cuanto a la lógica usada en el procesamiento de la información, pues ésta es la misma que en los circuitos síncronos, sin embargo cada bloque contará con los Bloques de Control Asíncronos (BCA's [48]) y los retardos, principalmente. El construir retardos en las FPGA's consume varios LUT's, una cantidad considerable conforme los retardos requeridos son de más tiempo.

En los trabajos [17, 18] se puede probar que este es el mayor problema a la hora de implementar circuitos asíncronos en Riel Sencillo.

Más adelante, en la sección 3.2 se describirá una propuesta para disminuir el número de compuertas usadas para estos retardos, basándonos en la construcción de un retardo genérico y mediante un contador, iterarlo hasta alcanzar el retardo requerido.

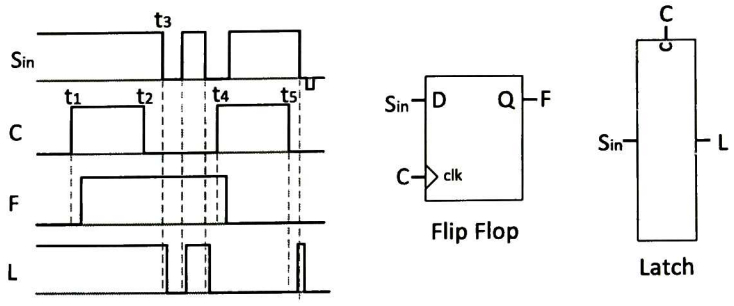


Figura 3.1.4.1a. Funcionamiento del Flip Flop y Latch.

Los latches, por el contrario, solo almacenan su información ante un 1 ó 0 lógico en su señal de control, es decir, responde ante valores lógicos y no cambios de flanco de estos valores.

Ahora bien, en realidad, cuando los Circuitos Integrados son fabricados, se hacen a nivel transistor. En la Figura 3.1.4.1b podemos observar cómo las compuertas AND y OR pueden ser implementadas a este nivel.

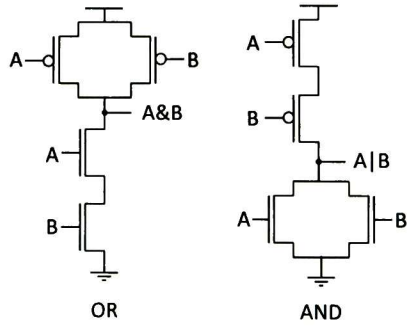


Figura 3.1.4.1b Compuertas AND y OR a nivel transistor.

Actualmente el diseño a nivel transistor se realiza con herramientas tales como Tanner [16]. Esto por que realizar la construcción a nivel transistor de millones de transistores nos tomaría demasiado tiempo, lo cual es evitado pasando el diseño de nivel RTL (Register Transfer Level), el cual ya contiene la descripción del circuito en lógica digital y de registros.

3.1.4.2 Riel Sencillo y el ligero aumento de espacio.

En el caso del diseño de circuitos asíncronos en Riel Sencillo, la ocupación no incrementará en cuanto a la lógica usada en el procesamiento de la información, pues ésta es la misma que en los circuitos síncronos, sin embargo cada bloque contará con los Bloques de Control Asíncronos (BCA's [48]) y los retardos, principalmente. El construir retardos en las FPGA's consume varios LUT's, una cantidad considerable conforme los retardos requeridos son de más tiempo.

En los trabajos [17, 18] se puede probar que este es el mayor problema a la hora de implementar circuitos asíncronos en Riel Sencillo.

Más adelante, en la sección 3.2 se describirá una propuesta para disminuir el número de compuertas usadas para estos retardos, basándonos en la construcción de un retardo genérico y mediante un contador, iterarlo hasta alcanzar el retardo requerido.

3.1.4.3 El Doble Riel y el espacio, su mayor desventaja.

Tengamos en consideración el circuito mostrado en la Figura 3.1.4.3a, es un sumador completo, a nivel compuerta. Usando este diseño en Doble Riel, trasladando todas las compuertas a su equivalente muller C ocasionará un uso excesivo de espacio, como se puede observar en la Figura 3.1.4.3b. estamos hablando de cerca de 110 compuertas.

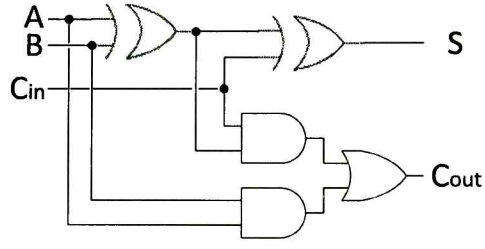


Figura 3.1.4.3a Sumador completo con compuertas digitales.

Esta es, como se puede ver, una de las principales desventajas al diseñar circuitos en Doble Riel con compuertas digitales, sin llegar al diseño a nivel transistor. Por eso es importante realizar las pruebas necesarias para comprobar la señalización, tiempos y protocolos propuestos, dedicando una parte significativa de tiempo al diseño a nivel transistor.

Ahora bien, como la verdadera comparación y menor diferencia en el uso de espacio, viene del diseño a nivel transistor, a lo largo de esta sección, se explicará el uso de todas las estructuras con sus versiones a este nivel.

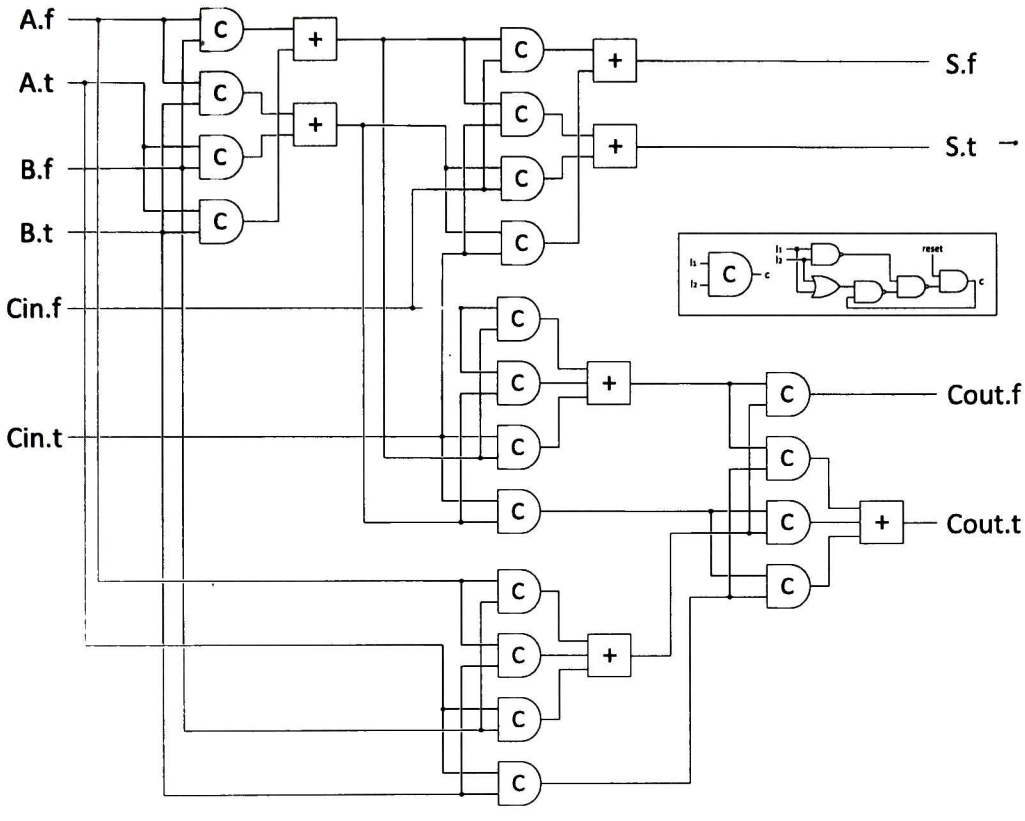


Figura 3.1.4.3b Sumador completo con compuertas Muller-C.

3.1.4.3.1 Comparación entre una señalización débil y fuertemente indicante.

Una de las propuestas más completas para implementar circuitos asíncronos de Doble Riel a nivel transistor se puede encontrar en [19]. A diferencia de una señalización fuertemente indicante, donde todos los valores de entrada son requeridos para generar una señal válida, la señalización débilmente indicante establece que para hacer válida una de las dos líneas, *.f* o *.t*, no siempre es necesario tener todos los valores de entrada válidos.

Un ejemplo concreto es el de las compuertas convencionales, las cuáles reducen la lógica implementada al ser diseñadas con este tipo de señalización. A continuación se detallará el diseño para cada compuerta, la cual no pudo ser encuentradado descrito en algún libro o artículo de forma resumida.

3.1.4.3.1.1 Compuerta AND

De acuerdo con la Tabla 3.1.4.3.1a, el diseño a nivel compuerta tiene un comportamiento diferente dependiendo del tipo de señalización, en este caso, una vez que se ha recibido uno de los datos con un valor 0 válido, para la señal débilmente indicante (ANDd) será suficiente para dar un valor 0 válido. En el caso de la señal fuertemente indicante, tendrá que esperar a obtener un dato válido de el otro dato de entrada.

Tabla 3.1.4.3.1a. Salida para una compuerta AND débil y fuertemente indicante.

a.f	a.t	b.f	b.t	andD.f	adnD.t	andF.f	andF.t
0	0	0	0	0	0	0	0
1	**0**	**0**	**0**	**1**	**0**	**0**	**0**
0	1	0	0	0	0	0	0
0	**0**	**1**	**0**	**1**	**0**	**0**	**0**
1	0	1	0	1	0	1	0
0	1	1	0	1	0	1	0
0	0	0	1	0	0	0	0
1	0	0	1	1	0	1	0
0	1	0	1	0	1	0	1

En la Figura 3.1.4.3.1a se muestran ambas implementaciones.

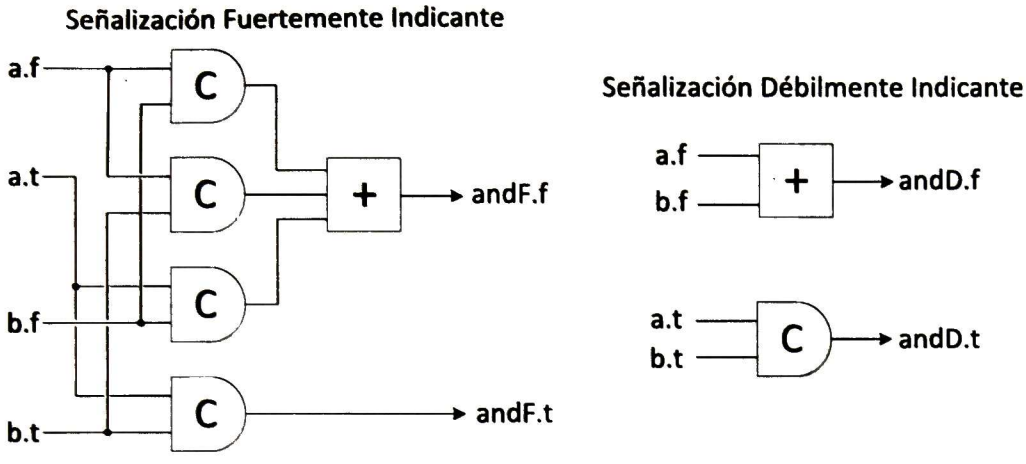


Figura 3.1.4.3.1a. Diseño de AND fuerte y débilmente indicante.

3.1.4.3.1.2 Compuerta OR

De acuerdo con la Tabla 3.1.4.3.1b, en este caso de la OR, una vez que se ha recibido uno de los datos con un valor 1 válido, para la señal débilmente indicante (ORd) será suficiente para dar un valor 1 válido. En el caso de la señal fuertemente indicante, tendrá que esperar a obtener un dato válido de el otro dato de entrada.

Tabla 3.1.4.3.1b. Valores de salida para una compuerta OR débil y fuertemente indicante.

a.f	a.t	b.f	b.t	orD.f	orD.t	orF.f	orF.t
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
0	**1**	**0**	**0**	**0**	**1**	**0**	**0**
0	0	1	0	0	0	0	0
1	0	1	0	1	0	1	0
0	1	1	0	0	1	0	1
0	**0**	**0**	**1**	**0**	**1**	**0**	**0**
1	0	0	1	0	1	0	1
0	1	0	1	0	1	0	1

En la Figura 3.1.4.3.1b se muestran ambas implementaciones.

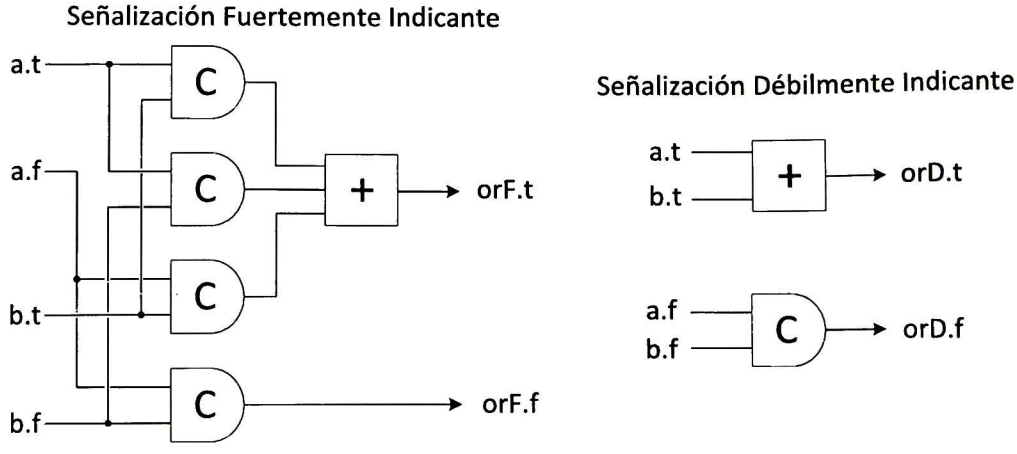


Figura 3.1.4.3.1b Construcción de OR fuerte y débilmente indicante.

3.1.4.3.1.3 Compuerta XOR

Para el caso de la compuerta XOR, no hay diferencia entre el tipo de señalización, dado que esta compuerta siempre requiere de sus dos datos de entradad para determinar su valor de salida. En la Tabla 3.1.4.3.1c podemos ver el comportamiento de sus señales de salida.

Tabla 3.1.4.3.1c Valores de entrada para una compuerta XOR en Doble Riel.

a.f	a.t	b.f	b.t	xor.f	xor.t
0	0	0	0	0	0
1	0	0	0	0	0
0	1	0	0	0	0
0	0	1	0	0	0
1	0	1	0	1	0
0	1	1	0	0	1
0	0	0	1	0	0
1	0	0	1	0	1
0	1	0	1	1	0

En la Figura 3.1.4.3.1c se muestrea su implementación a nivel compuerta.

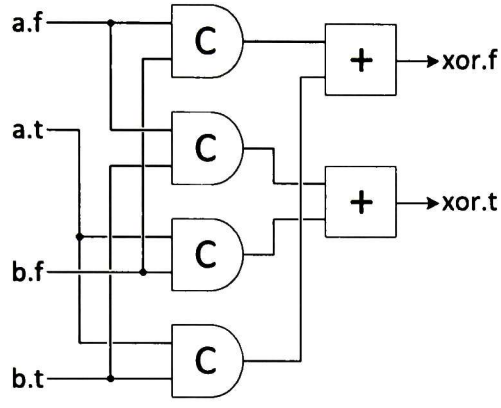


Figura 3.1.4.3.1c Construcción de OR con Muller-C.

3.2 Circuitos asíncronos de Riel Sencillo. Un ejemplo de diseño utilizando sus ventajas.

A lo largo de esta sección se describirá el uso de estructuras asíncronas de Riel Sencillo para la implementación de un circuito específico, una UART Self Timed con ajuste dinámico de velocidad de transmisión. Esta propuesta de UART, realiza dos contribuciones, la capacidad para reconFigurarse en tiempo de funcionamiento a nuevas velocidades de transmisión (sin necesidad de una señal de reloj) y el uso de retardos lineales y de menor ocupación, para los BCA's usados en su funcionamiento.

A manera de introducción, una unidad UART en un circuito síncrono, es usada para transmitir y recibir datos de manera serial, a una velocidad constante, a otro puerto, independiente de su señal de reloj de funcionamiento.

3.2.1 Características básicas de una UART

El protocolo de transmisión es sencillo, sólo se tienen que seguir los tiempos mostrados en la Figura 3.2.1a. En ella, se aprecia que la señal de datos siempre está en 1, cuando un nuevo dato va a ser enviado, se baja a 0 y después siguen los 8, 9 ó 10 datos a transmitir, se tiene como opcional un bit de paridad (para ayudar a saber si un dato fue enviado erróneamente) y se requiere un bit de paro forzosamente, esto es, que antes de volver a enviar un dato exista un tiempo en el que esté en alto nuevamente la señal de datos, para sí reconocer el bit de inicio de transmisión.

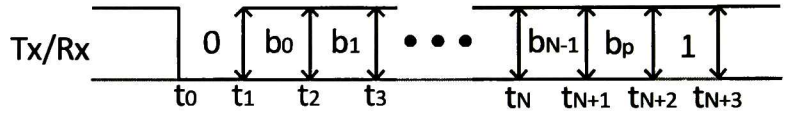


Figura 3.2.1a. Diagrama de tiempos en la transmisión de datos por UART.

La aspecto más importante de una UART, es el uso de los tiempos para enviar y recibir datos. Estos determinarán el valor del dato recibido, y de ser mal calculados pueden dar transmisiones con datos erróneos. En el caso de circuitos síncronos, generalmente se usa la señal de reloj como base para generar la señal de reloj (de mucho menor frecuencia) que usará la UART.

La transmisión de datos, cuando estos se generan más rápido de lo que pueden ser enviados, utilizan el recurso de FIFO's [20, 21, 22], que no son mas que medios de almacenamiento temporal.

3.2.2 UART en circuitos asíncronos de Riel Sencillo

La implementación de una unidad UART en Riel Sencillo, es relativamente similar a la versión síncrona. En la Figura 3.2.2 se puede apreciar el la composición de la UART en términos del transmisor y receptor, las FIFOs usadas como buffers de entrada o salida y las interfaces entre transmisor/receptor y éstos FIFOs, así como del bloque que captura y proporcionará el BAUD rate.

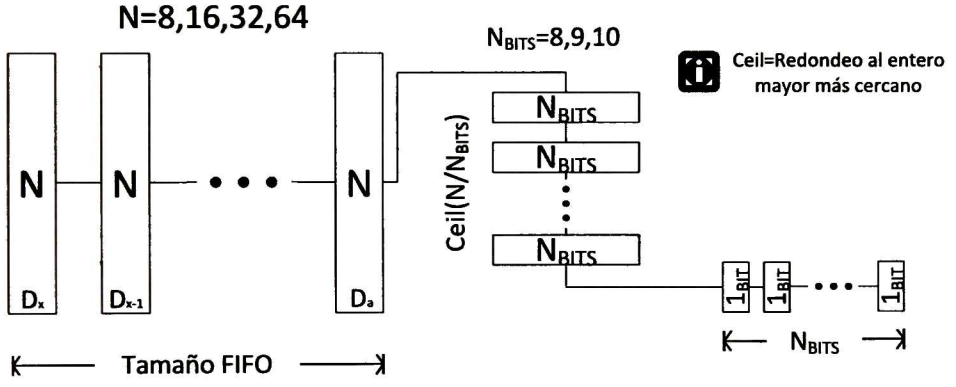


Figura 3.2.2 Construcción de Transmisor y Receptor de la UART en bloques generales.

Una implementación adicional que se realizó, al realizar el código en Verilog, es la posibilidad de enviar paquetes de N cantidad de bits, donde N puede ser no necesariamente una potencia de 2 (aunque para mejor aprovechamiento se recomienda esto). El circuito dividirá automáticamente esos paquete en paquetes más pequeños de Nbits, donde estos son la conFiguración de envío, 8, 9 o 10 datos. Para realizar esta opción se usa la función generate, que generá todos los bloques necesarios. En el Apéndice A se puede encontra el código para realizar esto en cada uno de los bloques necesarios para construir esta unidad. En la Figura 3.2.2 se

muestra en que consiste esta transformación de N a Nbits y luego 1 bit, esto sirve tanto para una transmisión, como para una recepción de datos.

3.2.2.1 Construcción del Transmisor

De manera general, se busca obedecer al diagrama de tiempos de la Figura 3.2.2.1a. Cada vez que hay datos en la FIFO por enviar, se enviarán cada uno de ellos, en paquetes de Nbits, hasta acompletar los N datos. La FIFO se irá vaciando y llenando conforme se lleguen a usar todos los valores y lleguen nuevos, mientras tanto, el transmisor no parará de enviar bit por bit de cada paquete tomado.

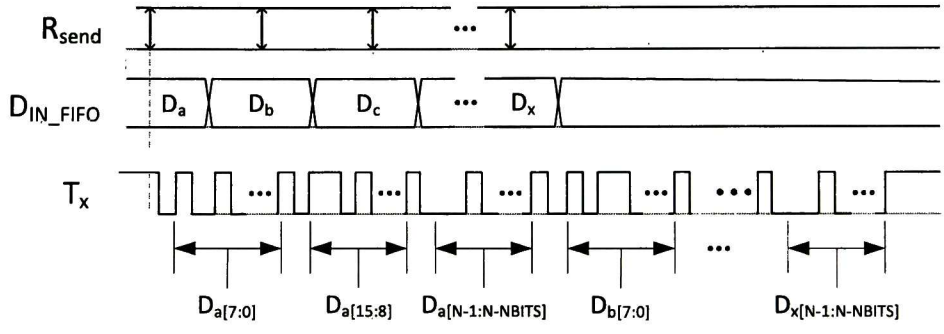


Figura 3.2.2.1a. Diagrama de tiempos para enviar NBITS en paquetes de 8 bits.

Para la construcción del transmisor se usó el diagrama mostrado en la Figura 3.2.2.1b, donde básicamente se utilizan dos bloques, un generador de la señal de reloj local (BAUD GEN), que utiliza el valor del BAUDrate, un contador (Cont_Bits), para determinar cuándo ya se han enviado los datos y las señales de control, tales como el bit de inicio y de paro.

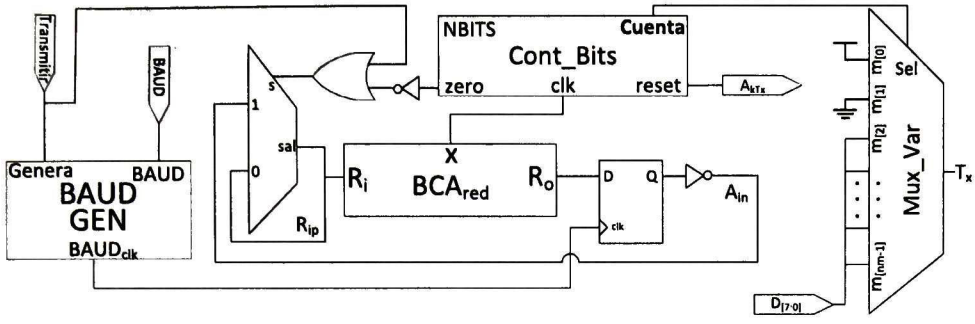


Figura 3.2.2.1b. Bloque para construir el transmisor.

Existe lógica extra para regresar la señal de akwnoledge, avisando que el envío de ese paquete de datos ya está realizada. El BCAred básicamente es un Bloque de Congrol Asíncrono Reducido, ya que no se necesitan todas sus propiedades. En la Figura 3.2.2.1c se muestra su diagrama de tiempos y su construcción con compuertas.

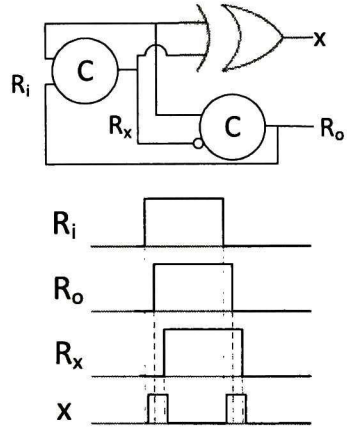


Figura 3.2.2.1c. BCA reducido (BCAred), una variante para usar menos circuitería basada en el Bloque de Control Asíncrono original.

Con este circuito, se respeta los diagramas de tiempos mostrados en la Figura 3.2.2.1d, los cuales son necesarios para interactuar a este bloque con los bloques adyacentes. Se puede observar que se genera una señal de reloj interna, determinada por el bloque BAUD_GEN, de ahí, este bloque solo generará una señal de reconocimiento (AkTx) cada vez que termine de enviar un paquete de datos de Nbits.

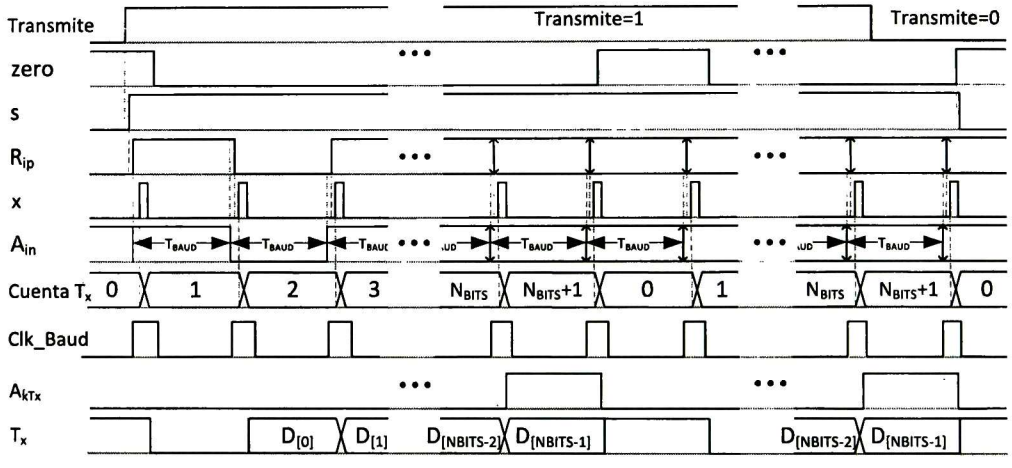


Figura 3.2.2.1d. Cambios de señal en el bloque transmisor de la UART.

Más adelante se detallará como se construye el bloque BAUD_GEN, un bloque importante en donde se podrá determinar de forma dinámica el tiempo que tardará en generarse la señal Clk_Baud, la señal de reloj generada localmente.

3.2.2.2 Construcción de FIFO's asíncronos en Riel Sencillo

Para construir FIFO's con circuitos asíncronos de Riel Sencillo, nos podemos remontar a [22]. En la Figura 3.2.2.2a se muestra la implementación con BCA's y latches. Al construir FIFO's de tamaño N, cada uno de los latches será controlado

por un mismo BCA. La profundidad del FIFO determina cuántos paquetes de datos puede almacenar antes de llenarse.

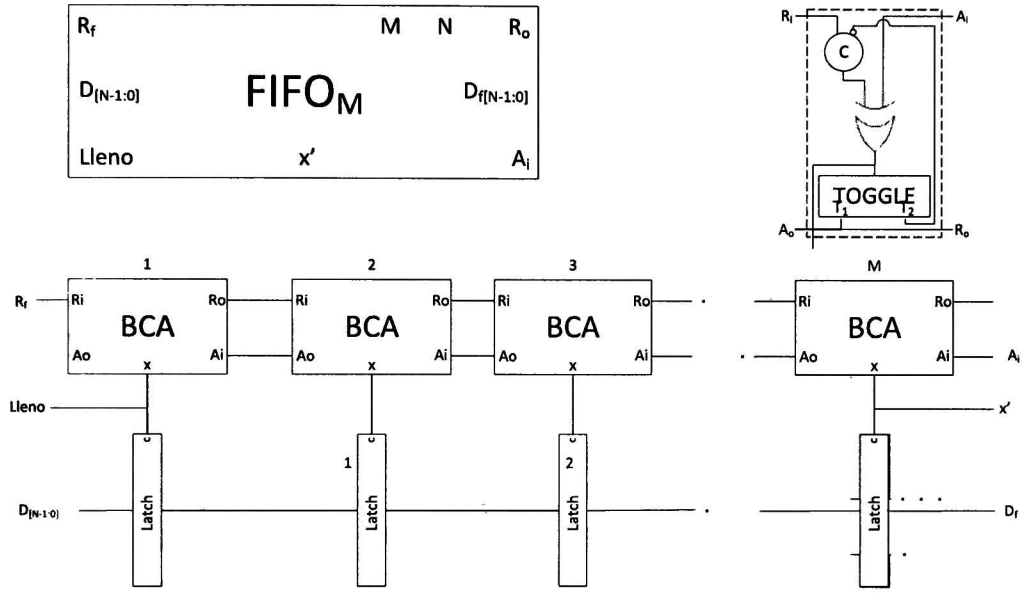


Figura 3.2.2.2a. Implementación de una FIFO con BCA's.

La construcción de estos bloques de forma dinámica también está contemplada, de esta forma se pueden crear FIFO's de profundidad M, dependiendo de las necesidades de diseño.

Luego, esta FIFO, para ser conectada a el transmisor, deberá tener una interface, que básicamente convertirá todos los datos (N) en paquetes manejables por el transmisor. En la Figura 3.2.2.2b podemos ver este bloque que interconectará al transmisor con su FIFO.

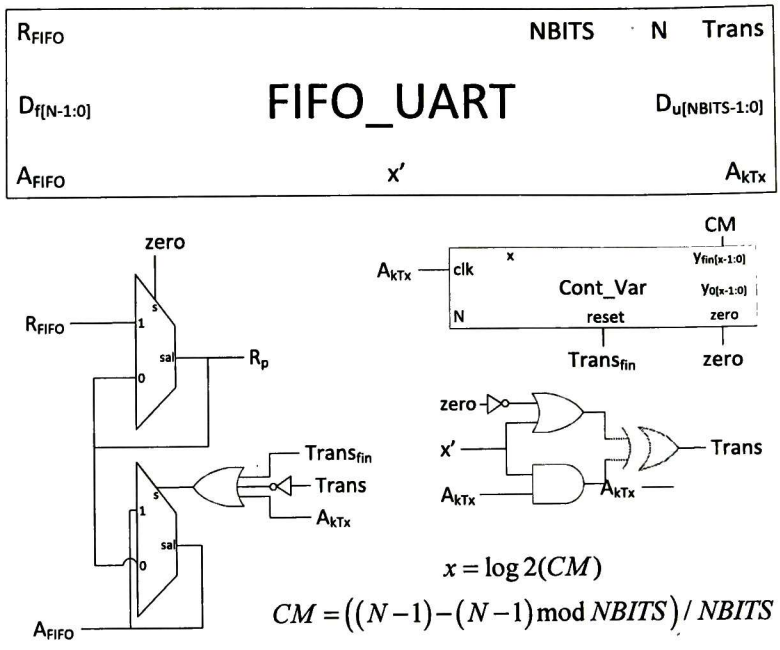


Figura 3.2.2.2b. Bloque intermedio entre FIFOs y el Transmisor.

En la Figura 3.2.2.2c podemos además, la serie de multiplixores que se usarán para hacer esta conversión gradual en paquetes de 8, 9 o 10 bits.

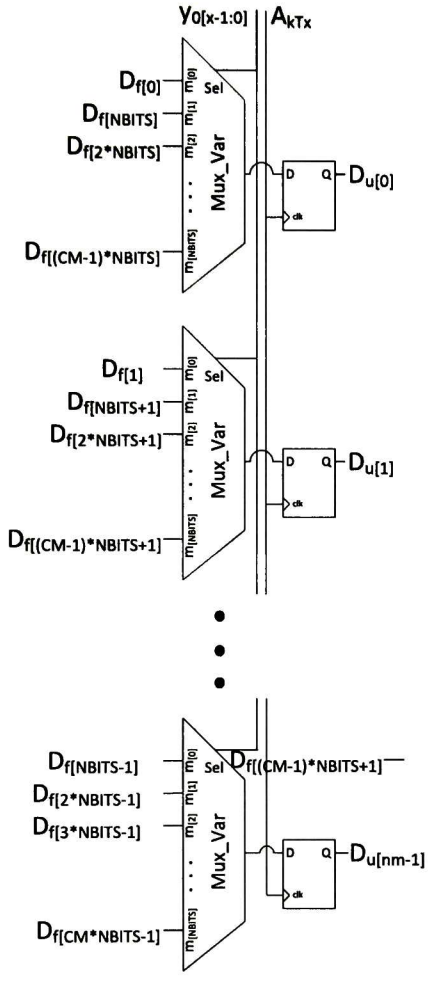


Figura 3.2.2.2c. Multiplexor variable para transmitir los N bits en grupos de nm bits.

Este bloque tendrá que comportarse como se muestra en la Figura 3.2.2.2d. De esta forma, cada vez que haya una señal de reconocmiento (A_{kTx}), debe significar que un paquete de datos ha sido enviado exitosamente, por lo que habrá que escoger otro.

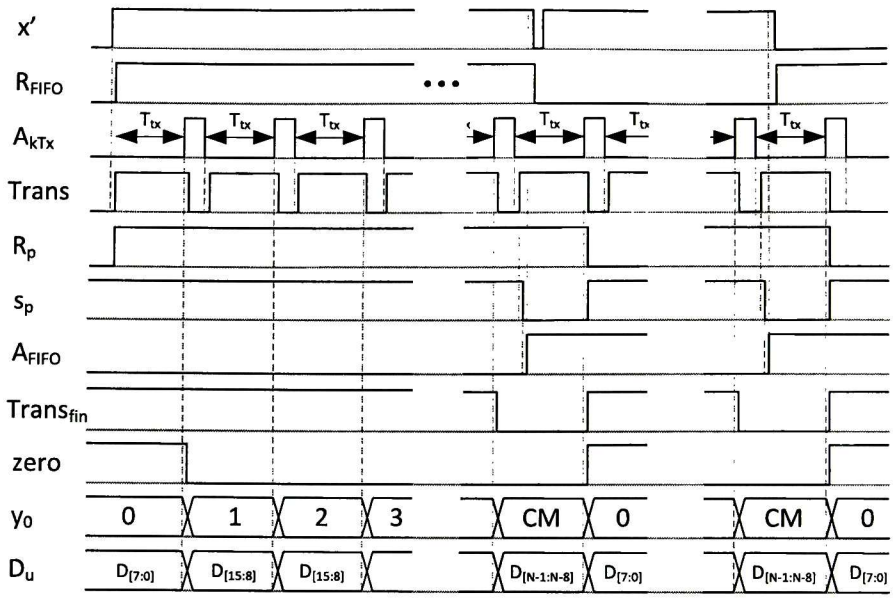


Figura 3.2.2.2d Bloque más general donde se muestra el comportamiento de las señales principales del transmisor variable.

El número de veces que se ejecutará esta transmisión por paquete de datos está determinada por la ecuación 3.2.2.2. Donde el operando $a\%b$ significa a módulo b, es decir, el residuo al dividir a entre b por el entero más grande posible.

$$CM = \frac{(N-1) - (N-1)\%N_{BITS}}{N_{BITS}} \qquad (3.2.2.2)$$

3.2.2.3 Implementación de Generador de Baud Rate

En una UART, el BAUD rate no es más que un valor que nos ayudará a saber durante la transmisión y recepción de datos, cada cuanto debemos esperar para obtener datos válidos, uno tras otro. Desde el punto de vista de implementación, es la frecuencia de reloj interna de la UART (clkUART), dividiendo

(DIV veces) la frecuencia general del sistema (clk_{sys}) y está determinada por la siguiente ecuación:

$$DIV = \frac{Clk_{sys}}{Clk_{UART}} \quad donde \quad clk_{UART}[Hz] = BAUD_{rate}[bits/seg]$$

De esta forma, determinando este valor, podremos determinar el valor de un contador de pulsos para la señal de reloj del sistema. De aquí que podamos determinar con el sistema en funcionamiento, el valor de este contador, independientemente de la señal de reloj del sistema (el cual no hay en los circuitos asíncronos de Riel Sencillo) ni del que envía. Esto se verá a detalle en la sección 3.2.2.5.

El bloque encargado de esta función es el llamado BAUD_GEN. En la Figura 3.2.2.3a se muestra la implementación en bloques de este generador de baud rate.

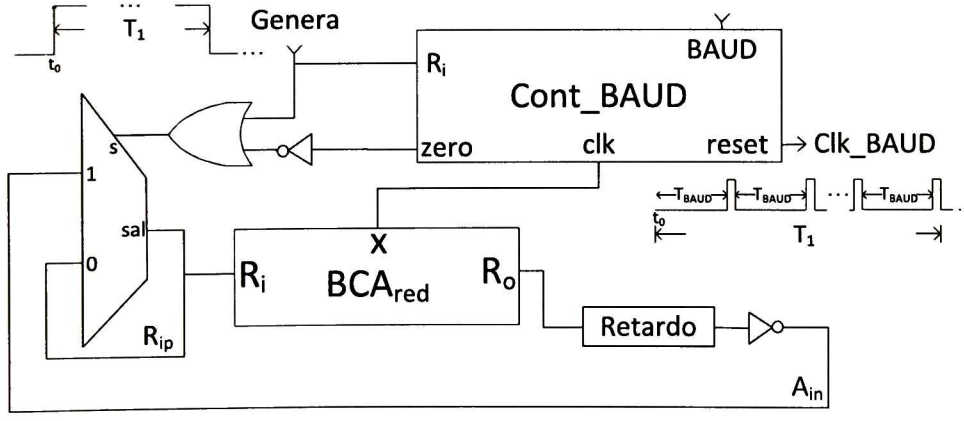


Figura 3.2.2.3a. El bloque generador del Baud Rate, el cual tiene como base el valor binario de BAUD, determinado en el modo de captura de éste.

En la Figura 3.2.2.3b se muestra el diagrama a tiempos de las señales más importantes de este bloque, en él, se puede observar que siempre y cuando la señal

Genera sea 1, habrá un conteo y cambio de señales, por lo cual la señal de reloj interna solo funcionará cuando se necesite transmitir, apagando prácticamente todo funcionamiento dentro del circuito transmisor, ya que esta señal es la que genera cambios en el resto de la implementación.

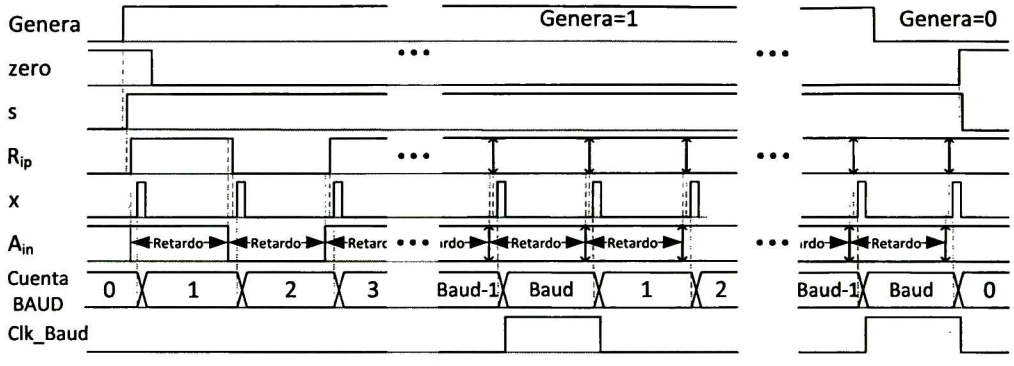


Figura 3.2.2.3b. Este bloque no es más que un contador que al alcanzar el valor de BAUD generará una señal de reloj para el resto dle circuito.

También podemos observar que el tiempo que tardará en activarse la señal de Clk_Baud dependerá del valor del retardo, por lo que un incremento lineal en el retardo se traducirá en un incremento lineal en la señal de reloj para cambiar.

3.2.2.4 Aumento del retardo con menor ocupación

Uno de los m ayores problemas al usar retardos en FPGA's, es el uso de retardos usando las LUT's [23, 24, 25]. En este caso, usando un retardo genérico y un contador será suficiente para generar retardos lineales, con menor ocupación aunque a costa de un ligero incremento en el consumo de energía. De la Figura 3.2.2.4a se obtiene que cada vez que se requiera aumentar al doble un retardo, solo será necesario añadir la lógica necesaria para aumentar en un bit al contador, la

cuál es mínima, ya que consistirá en añadir un flip flop, una compuerta XOR, dos AND.

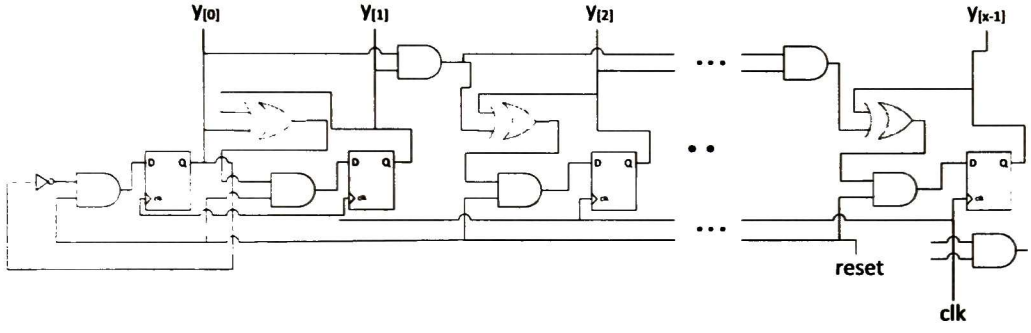


Figura 3.2.2.4a. Contador binario caracterizado por Flip Flops y compuertas AND, OR y XOR.

Adicionalmente se creará circuitería para la señal zero, la cual puede variar dependiendo de si pasa el umbral de alguna potencia de 2, lo cual creará más compuertas que aquella que continúe en el rango. Esto se aprecia en la Figura 3.2.2.4b.

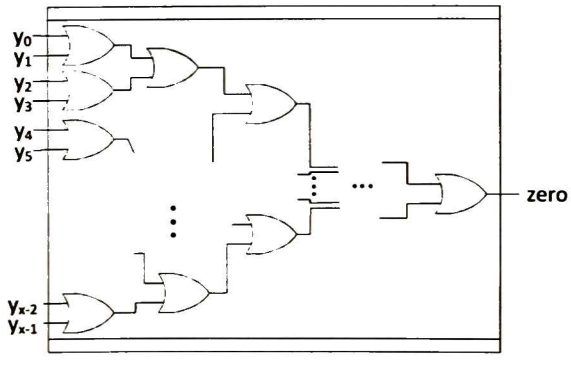


Figura 3.2.2.4b. Circuito para generar la señal que indica que el contador tiene le valor de 0.

Por último, la señal de reset para el caso de que sea una UART reconFigurable, necesitaremos añadir de igual forma que con la señal de zero, circuitería basado en la lógica de una potencia de dos. Esta circuitería, que en lugar de usar compuertas OR, usa compuertas AND se muestra en la Figura 3.2.2.4c.

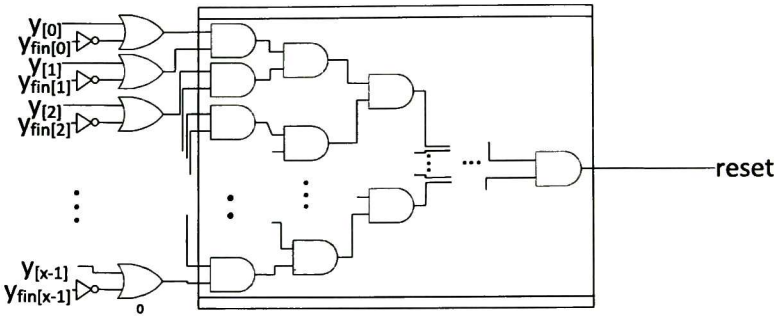


Figura 3.2.2.4c. Circuito para generar la señal que reseteará el contador de acuerdo a un valor de terminación especificado por otra parte del circuito.

3.2.2.5 Auto conFiguración en tiempo de operación para ajustar velocidad de transferencia

Una propiedad añadida al diseño de esta UART, es la capacidad de reconFigurarse a un baud rate en tiempo de operación, esto significa que el baud rate incluso no tiene por que ser determinado de manera fija a algunos valores, sino incluso aplicar esta propiedad poder entrenar las líneas de comunicación a la máxima tasa de transferencia posible, aún si esta es mayor que la máxima usada convencionalmente.

Para realizar esta propiedad a la UART en Riel Sencillo, en realidad no se añade tanta circuitería ni grandes bloques, ya que de hecho, se tiene un contador conFigurable (el retardo lineal) al cual solo hay que añadirle la opción de que la

cuenta final pueda ser fijada en tiempo de operación, esto es, con un valor que pueda ser modificado en cualquier momento, en nuestro caso, una vez que está en tiempo de captura del baud rate.

3.2.2.6 Obtención del baud rate en tiempo de operación

El método para obtener la velocidad de transferencia, es en realidad sencillo. Al entrar en un modo de captura, el transmisor solo envía un dato que contenga el bit *menos* significativo en alto. Tomando en cuenta que al iniciar cualquier transmisión, habrá un bit en bajo, al detectar el flanco de bajada, se iniciará un contador, que parará en cuanto se encuentre un valor 1 lógico, de aquí la importancia que le bit menos significativo siempre sea 1, sino, se tomaría incorrectamente el valor del bit de inicio de transmisión.

Una vez obtenido este valor, es el que será usado en futuras transmisiones o recepciones, es decir, el valor usado por el transmisor y receptor para determinar el cambio de bits al enviarlos o recibirlos. De esta manera, incluso se pueden tener valores no convencionales de transferencia de datos, de acuerdo al estado de las líneas, aumentando así la robustez del diseño. Cabe señalar que es importante que cada vez que el circuito se inicie, tener un valor precargado, de acuero al que corresponda al baud rate de mayor uso.

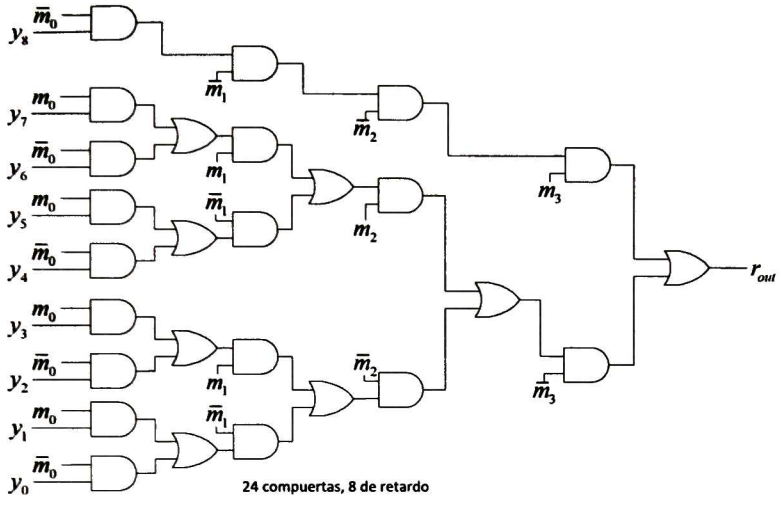
Una de las mayores ventajas al realizar esta calibración, es la independencia con el sistema. No importará la frecuencia de reloj a la que funcionen ambos sistemas, tanto la velocidad de transmisión de un circuito síncrono podrá ser calculada con los recursos actuales del sistema asíncrono, como si se usaran dos sistemas asíncronos, en el que se podría incluso, en trabajos posteriores, realizar un

modo de entrenamiento para alcanzar las mayores tasas de transmisión permitidas por el cableado y condiciones físicas externas.

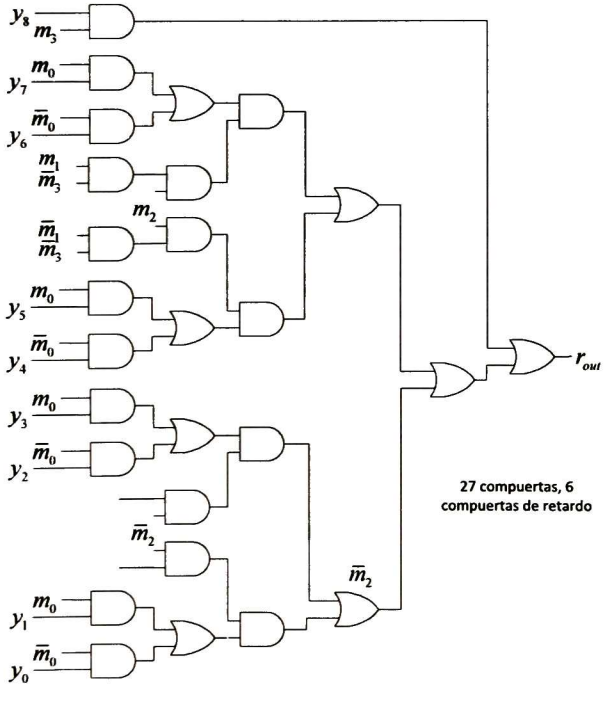
3.2.2.7 Importancia del diseño a nivel compuerta

Tomando en cuenta que para los circuitos asíncronos no hay una señal de reloj, y que la intención del diseño en FPGA's es sincronía, podemos determinar la utilidad de esta herramienta bajo el hecho de que también existe diseño comportamental que no usa ciclos de reloj [26, 27, 28]. De esta forma, teniendo el control completo del número de compuertas y de como son éstas conectadas, esta se convertiría en la forma más conveniente para iniciar el diseño de circuitos asíncronos.

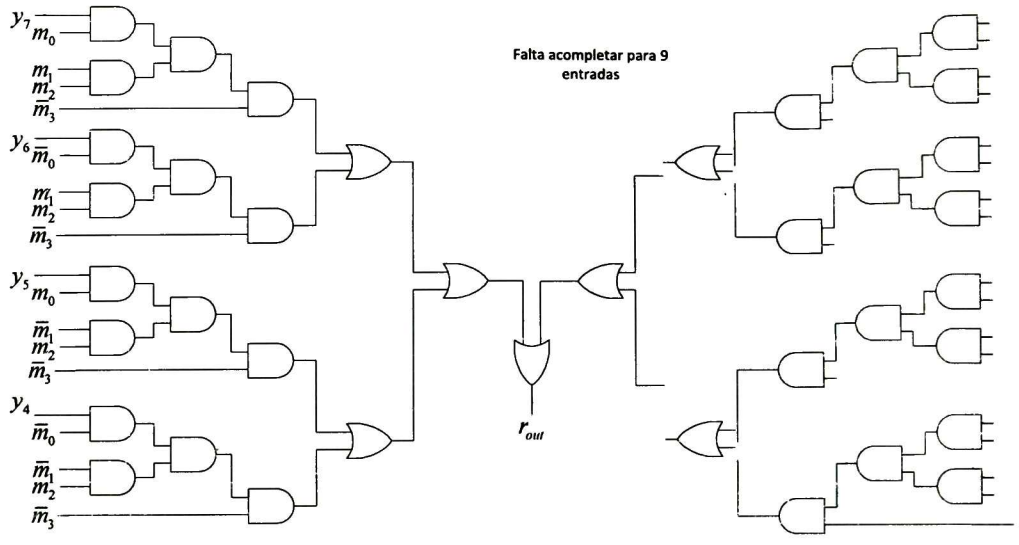
Veamos el caso de un multiplexor, el uso de verilog y la opción generate. Para crear un multiplexor variable, de un número de compuertas X, podemos referirnos a las Figuras 3.2.2.7a, 3.2.2.7b y 3.2.2.7c. En ellas se encuentran formas diferentes de implementación, variando la latencia y el uso de compuertas principalmente.



3.2.2.7a. Mulplexor variable tipo A. Menos compuertas a más latencia.



3.2.2.7b. Multiplexor variable tipo B. Ligeramente más compuertas pero menos latencia.



3.2.2.7c. Multiplexor convencional. Mucho mayor número de compuertas y con ligeramente mejor latencia.

Ahora bien, el diseño síncrono, no toma importancia en cómo son generadas estas compuertas, en nuestro caso, es de importancia saber como son conectadas y creadas todas las instancias de un circuito, dado que la ocupación y el diseño asíncrono dependen de ello. *Mientras más control se tenga sobre los bloques que son usados al construirse el circuito en una FPGA, más fácil será de convertir a nivel transistor de manera manual por el diseñador.*

¿Por qué se asegura esto? Primero por que el paso de niveles de RTL a transistor han sido optimizados para diseño síncrono, segundo por que en la mayoría de los casos no es de interés cómo se crearon los bloques que lo conforman, siempre y cuando realicen la función que se les requiere. Una

desventaja con el diseño asíncrono, es precisamente el hecho de tener que supervisar tanto el diseño a nivel compuerta como transistor.

Capítulo 4

Multiplicador de Booth en Doble Riel

En este capítulo se explicará el protocolo modifcado para Doble Riel propuesto. Con él, podremos crear estructuras más complejas. Aunado con el hecho de los capítulos anteriores, donde se explica el pensamiento asíncrono, se podrá crear un multiplicador asíncrono de Doble Riel con codificación de Booth, con una sola línea de sumadores, reduciendo drásticamente la ocupación y aumentando la eficiencia de estos.

4.1 Protocolo de Doble Riel de 4 fases Modificado.

En esta sección se describirá la modificación al protocolo convencional de 4 fases [29, 30,31, 32], introduciendo la posibilidad de realizarlo con compuertas AND y OR, sin las compuertas Muller C [33], además de introducir algunas compuertas nuevas, específicas para este protocolo.

La idea, básicamente, es la de aprovechar la capacidad de los circuitos de Doble Riel de utilizar al máximo el tiempo de procesamiento de cada bloque en un diseño. Esto significa, que cada vez que un bloque ha procesado la información requerida, y ésta es procesada por el bloque que la usará, pueda estar lista para nueva información, acelerando en la mayoría de los casos el procesamiento general de los datos.

Tomemos como base la Figura 4.1a. En ella, nuestra idea es mostrar que en el diseño de circuitos y la asignación de tareas específicas a bloques conlleva a una completa interrelación entre éstos. Esto, en otras palabras, significa que cada bloque puede tener una relación con un número indeterminado de bloques y dependencia en dos sentidos, ya sea en esperar datos o en entregarlos. En este caso, las líneas azules serán las señales salientes de información, mientras que las rojas serán los datos que entren para ser procesados. De manera interna las líneas grises representarán la lógica interna que sólo es necesaria para este bloque. Para ejemplificar como están interrelacionados los circuitos en Doble Riel, podríamos incluso decir que este bloque se puede convertir nuevamente en una pequeña parte de un circuito más grande, con la misma particularidad de interrelación.

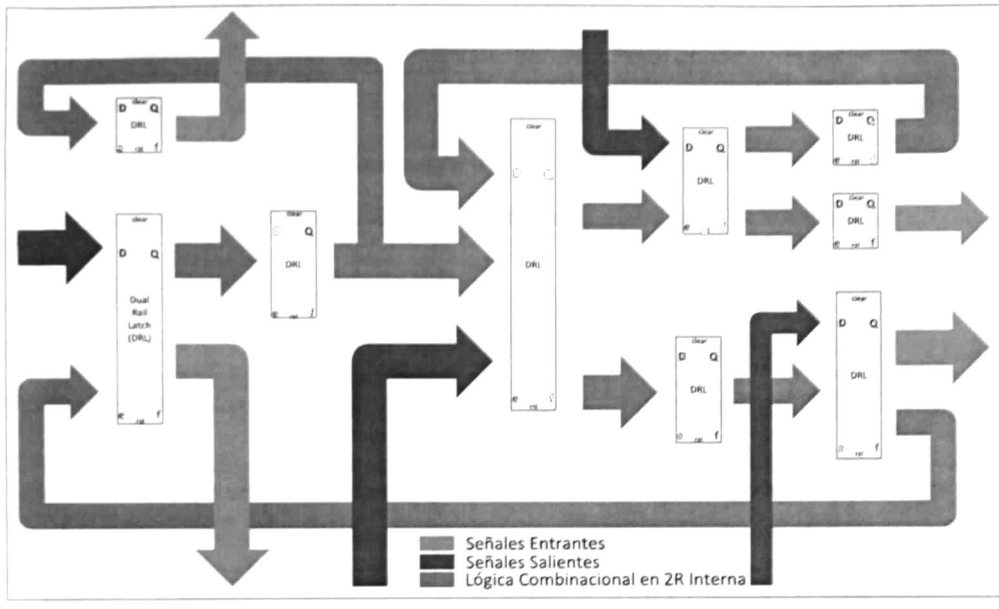


Figura 4.1a. Diagrama de bloques de estructuras asíncronas interactuando.

Ahora bien, supongamos que de estos bloques, uno de ellos es el más tardado, en un circuito síncrono convencional, el tiempo de operación funcionaría de acuerdo al tiempo más largo de éste bloque. En un circuito asíncrono de Riel Sencillo, cada bloque funcionaría de acuerdo al tiempo más largo que le pudiese tomar realizar una operación a cada bloque en particular. Sin embargo, un detalle acerca de esto, son los cuellos de botella. Mientras que un bloque puede haber terminado ya su tarea, tendrá que esperarse, en algunos casos, para la resolución de bloques posteriores, quizá más tardados.

En los circuitos asíncronos de Doble Riel, sin embargo, debe ser explotada la particularidad de que cada bloque funciona a sus tiempos, de esta manera, cada operación tendrá un tiempo específico de funcionamiento.

4.1.1 Descripción del protocolo modificado

Tomemos como ejemplo el circuito mostrado en la Figura 4.1.1a. En ella se pueden observar tres bloques de lógica de Doble Riel. Ahora supongamos la introducción de datos válidos en la entrada, estos se propagarán hasta llegar al otro extremo. Sabemos que para que se vuelvan a procesar datos, éstos tienen que pasar por el estado de inválidos, para volver a calcular datos válidos.

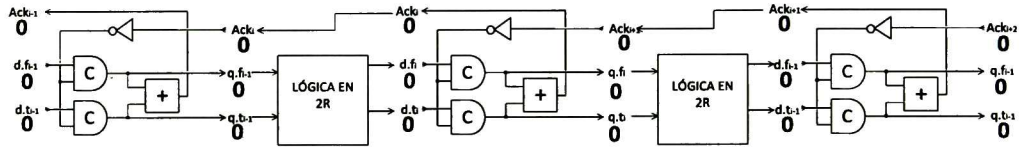


Figura 4.1.1a. Proceso mediante el cual se propagan datos válidos en una línea de bloques de Doble Riel para recibirlos.

En este caso, puede llegar el momento en el que si un bloque previo no se limpia, el resto tampoco lo hará, como lo muestra la Figura 4.1.1b. En este caso, una vez que éste se limpie, se iniciará con el proceso de dejar nuevamente todas las líneas de datos en 00 (inválido). Esto sin duda generará cierta latencia extra.

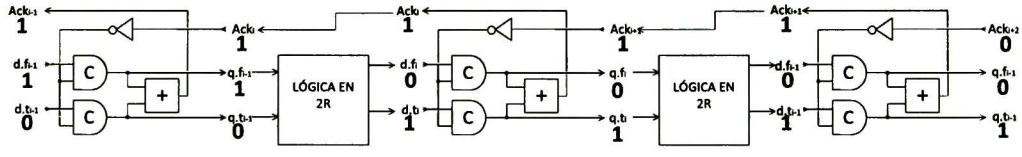


Figura 4.1.1b. Un inconveniente natural es el tener que esperar a bloques anteriores para que pueda volver a estar listo para recibir datos nuevos.

En la Figura 4.1.1c podemos apreciar, cómo una vez que empieza la limpieza de datos desde un punto dado, ésta tendrá que empezar a propagarse, antes de que pueda pasar datos váliso.

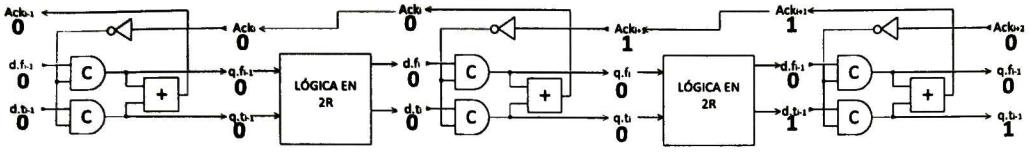


Figura 4.1.1c. Una vez iniciada la limpieza de un bloque, ésta se propagará.

¿Qué pasaría si pudiésemos ahorrar tiempo una vez que se han usado los datos? Se mejorarían los cuellos de botella y tiempos de procesamiento de datos. En otras palabras, si lográsemos que los bloques de procesamiento no tuviesen que esperar a bloques anteriores para pasar al estado inválido (limpiar toda la circuitería y dejarla en 00), aseguraríamos un tiempo ganado, ya que estarían listos para información válida inmediatamente.

Al realizar esto, tendríamos que implementar algo similar a lo mostrado en la Figura 4.1.1d, donde cada bloque tenga la capacidad de limpiarse, una vez utilizados sus datos. Pero esto, significaría poder tener la capacidad de almacenar, durante algún tiempo, los datos que usan los bloques, ya que algunos de ellos podrían tener datos vacíos (lo que en el diseño convencional significaría que en adelante se podrían limpiar). Dicho de otra forma, para que los bloques puedan limpiarse y estar listos para nuevos datos, también deben de tener la capacidad de mantener datos válidos a pesar de que los bloques con los que interactúan tengan inválidos. Esto es una gran modificación al protocolo, ya que se tendrían que usar medios de almacenamiento, ya sea una variante de muller-C o latches convencionales.

Analizemos el flujo que debería seguir una serie de operaciones bajo este esquema. En la Figura 4.1.1e podemos observar el paso de una operación, Op1, a través de la circuitería.

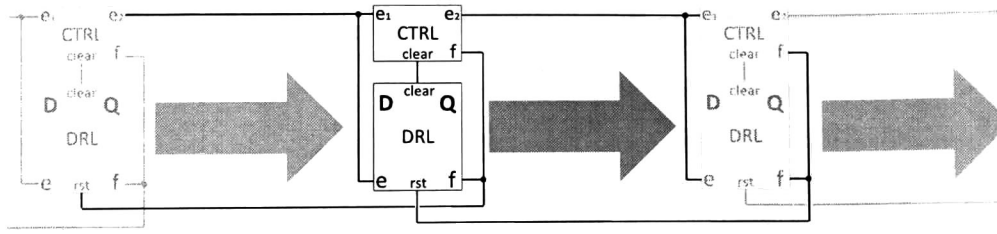


Figura 4.1.1d. Propuesta para el funcionamiento del procolo modificado.

Las flechas vacías significan datos inválidos (00), una vez que ya hay válidos están determinados por la operación. El proceso de limpiado tomará un tiempo, mientras tanto una nueva serie de datos no podrá ser tomada por un bloque mientras no sea limpiado. En la Figura 4.1.1f se muestra la intrusión de otra operación, Op2, con su respectivo conjunto de datos.

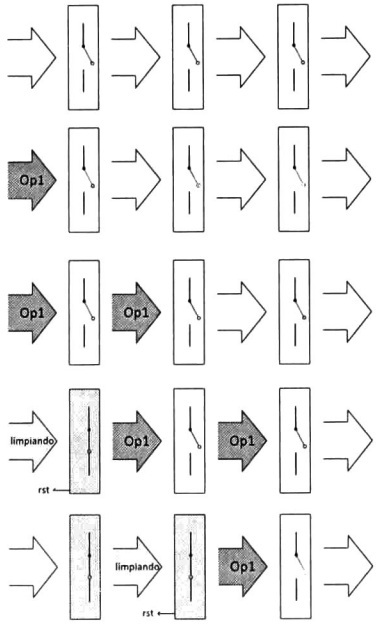


Figura 4.1.1e. Paso de la operación 1.

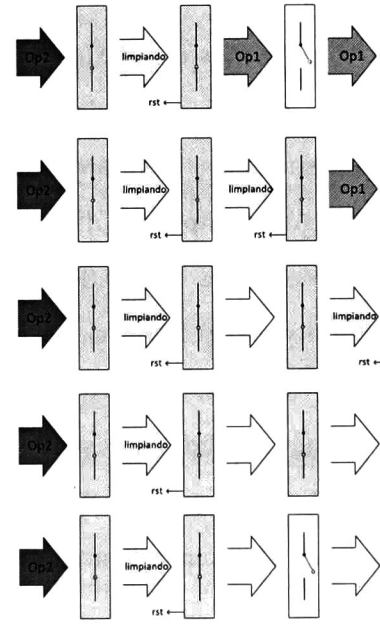


Figura 4.1.1f. Paso de la operación 2.

De esta forma, cuando lleguen más operaciones por ser tratadas, habrá momentos en el circuito en el que ciertas partes ya se encuentren en estados inválidos, listos para nueva información, aunque su antecesor aún no lo esté. Esto se puede apreciar en las Figura 4.1.1g y 4.1.1h.

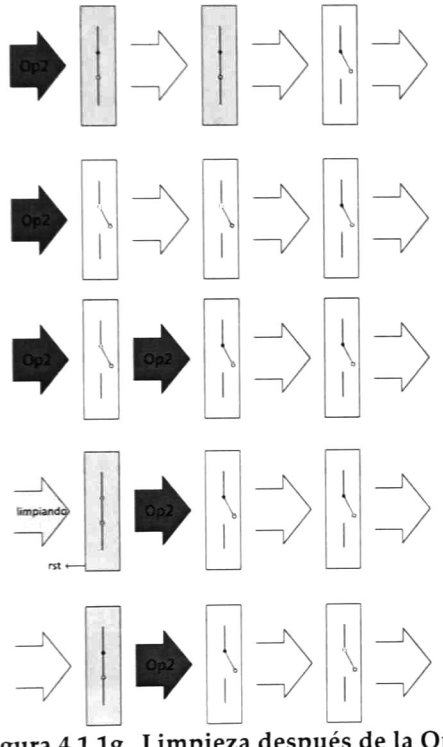


Figura 4.1.1g Limpieza después de la Op2.

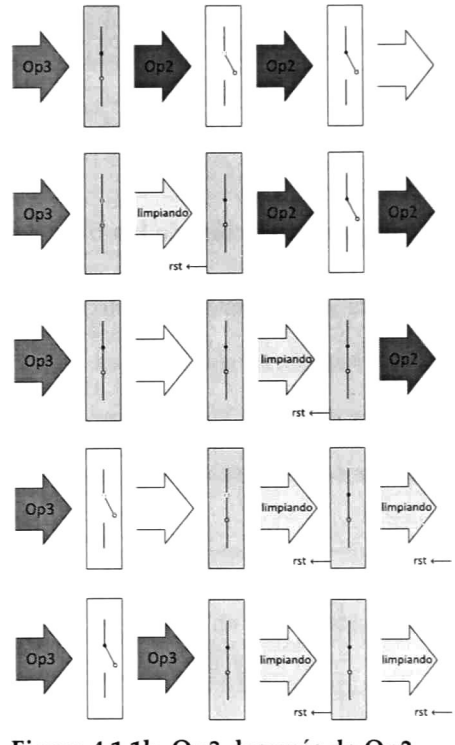


Figura 4.1.1h. Op3 después de Op2.

Aquí se puede apreciar que para el momento que un grupo de datos ya sea válido es probable que el resto de los bloques ya esté listo para nueva información, contrario a lo que podría suceder con el protocolo de 4 fases anterior, como se muestra en las 4.1.1i.

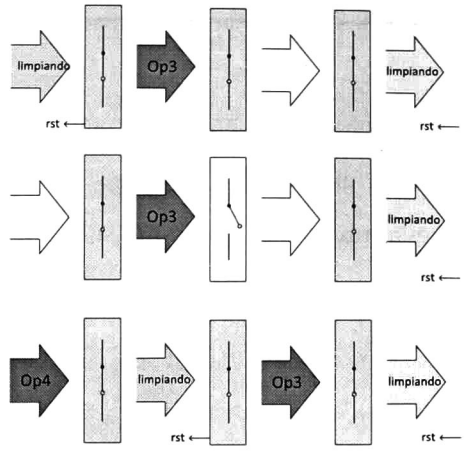


Figura 4.1.1i. Nuevas operaciones y su paso más fluido.

4.1.2 Implementación de los bloques necesarios

Para implementar los circuitos que permitan este protocolo modificado de 4 fases, es necesario usar lógica extra de control, que use señales que indiquen cuándo los datos ya pueden ser tomados y también limpiados. Para esto se siguen usando las señales de vacío y lleno (*e* y *f*) como se muestra en la Figura 4.1.2a.

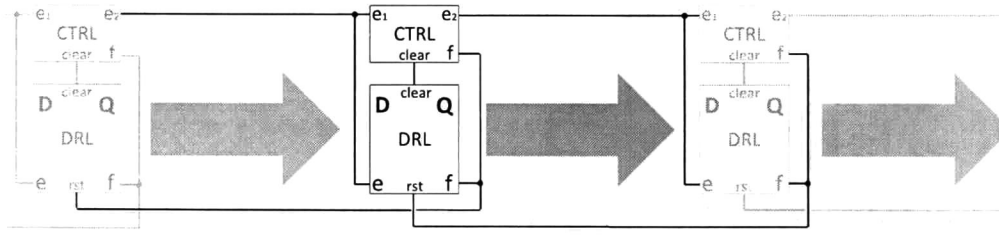


Figura 4.1.2a. Las señales clear y srt son de control. Las señales e y f son de estados vacío y lleno del bloque para sus datos de entrada y salida.

Aunque originalmente la forma de implementar esto se hacía como se muestra en la Figura 4.1.2b [1], la propuesta de este trabajo es el de crear una unidad que pueda almacenar y al mismo tiempo tener un control para los datos que maneje.

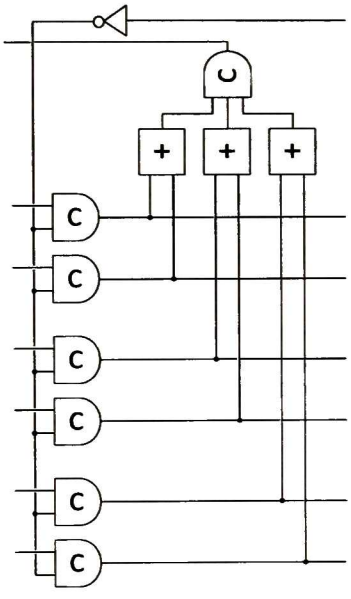


Figura 4.1.2b. Forma original para la señal de reconocimiento de que todas las señales de entrada son válidas.

En el caso inicial, todos los bloques tendrán sus señales de de vacío (*e*) en 1, mientras que la de lleno (*f*) en 0. En este estado hay una completa transparencia entre todos los bloques. Esto quiere decir que todos los datos válidos se propagarán a lo largo de todo el circuito.

Esta es una ventaja extra en el diseño de Doble Riel. Suponiendo que el circuito conste de N etapas, el tiempo que le tomaría al primer grupo de datos a procesar desde el inicio al final del bloque, en un circuito síncrono, será de N ciclos de reloj (N veces el peor tiempo estimado para todos los bloques). Sin en cambio, en el diseño en Doble Riel, la señal se propagará en el tiempo justo que le tome cruzar por cada bloque, que aún en el peor de los casos, estos tiempos juntos difícilmente sumarían N ciclos de reloj.

Una vez que un grupo de datos empieza a mostrar señales válidas a la entrada del bloque a analizar (bloque i), las señales e y f tendrán el valor de 0, ya que ambas no serán ciertas, los datos de entrada deben seguir pasando. *La puerta de entrada para datos válidos debe cerrarse solo cuando todos los datos de entrada sean válidos.*

Una vez cerrado este acceso, a diferencia del protocolo normal, el paso de válido a inválido no está determinado por bloques anteriores, sino por los bloques que le suceden. Para que este paso de válido a inválido suceda, se inicializan todos los datos a la entrada del bloque, después de los medios de almacenamiento, esto se realiza con una señal de reset. ¿Cómo determinar cuando se acciona esta señal de reset?

Esta señal de reset es activada una vez que los datos que produjo (todos válidos) ya han sido tomados por el bloque correspondiente. Es decir, la señal de reset la genera el bloque sucesor, con la señal *f*, pero solo cuando hay una transición de 0 a 1. ¿Por qué es importante detectar la transición y no solo el valor? Por qué se puede dar el caso que un bloque posterior se encuentre lleno y aún no se vacíe, si el bloque actual se vacía y se vuelve a llenar, detectará al final de este nuevo llenado, que el bloque posterior está lleno, pero lo será por el resultado anterior, por lo que tendrá que esperar a que se vacíe y vuelva a llenar, para poder asegurar que han sido tomados los datos nuevos, esto es, asegurar que paso de 0 a 1.

Ahora bien, ¿Cuándo debe permitirse nuevamente la entrada de datos? Esta es la parte más complicada. Una vez que se ha cerrado el acceso a los datos de entrada a un bloque, solamente puede volver a abrirse cuando se juntan dos condiciones.

La primera es que esa serie de datos ya haya sido procesada y el resultado haya sido tomado por el bloque correspondiente, esto es, que se haya producido una señal de reset y que su señal de vacío (*e*) sea de 1. La transición de la señal de reset de 0 a 1 debe ser detectada, para corroborar que se ha mandado la indicación de que ya no son necesarios los datos, que ya han sido tomados. Adicionalmente se requiere que la señal de vacío sea de 1 para asegurar que en realidad sí están limpios los datos, así no se producirá un traslape entre los datos viejos y los nuevos, produciendo datos indeseados.

La segunda condición, sucederá cuando los datos del bloque predecesor se hayan vaciado al menos una vez. Esto es de vital importancia, también para evitar que los datos viejos sean tomados nuevamente. Debe tenerse cuidado que después de vaciarse los datos actuales, este intentará obtener nuevos datos, pero si el bloque anterior no ha terminado su limpiado (no ha habido una transición de 0 a 1 en la señal *e* anterior) los datos no deben ser tomados, ya que estos datos ya han sido procesados y habría que esperar a que empiecen a fluir datos válidos nuevos.

Una vez que determinamos estas condiciones, para la señal de reset y de control de los latches, utilizaremos los bloques mostrados en la Figura 4.1.2c. Aquí se puede observar que existen dos formas en las que se puede bloquear la entrada del latch. La más inmediata es la del mismo par de líneas de información para el dato (*.f* y *.t*), aprovechando que sólo una de estas será 1, una vez que esto suceda, no se debería esperar un cambio adicional, pudiendo cerrar el latch.

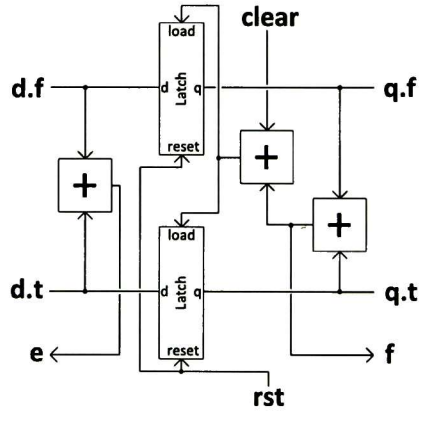


Figura 4.1.2c. Bloque de control y almacenamiento de datos válidos.

Para el caso de entradas de más de un dato, tendríamos interfaces como las que se muestran en la Figura 4.1.2d. De aquí, aparte de los datos, es importante recolectar las señales de vacío y llenado, que serán usadas para las señales tanto de control para el paso de valores como el reset de los latches.

La parte más interesante consta al elaborar el control para generar la señal que "liberará" a los latches. Para esto, se propone el diseño mostrado en la Figura 4.1.2d. Es importante recalcar la necesidad de detectar flancos, y que aunque estos mecanismo de detección son utilizados por circuitos síncronos no tienen que ser exclusivamente usados por estos. A nivel transistor será posible, incluso, realizar una propuesta más acorde con la lógica asíncrona.

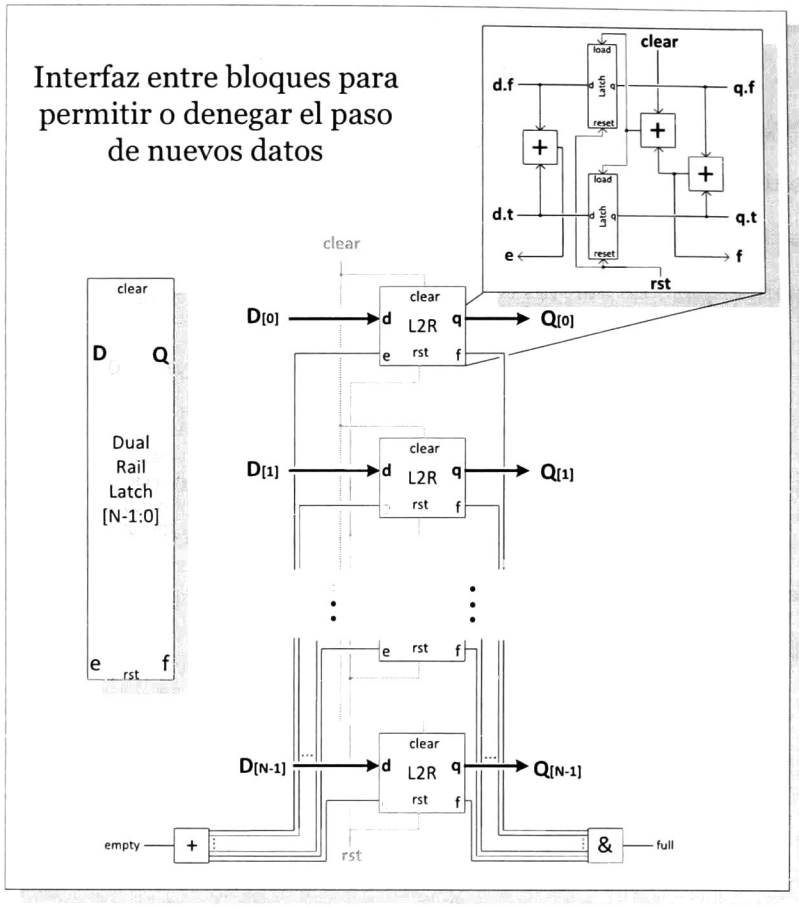


Figura 4.1.2d. Bloque de control y almacenamiento para N datos válidos.

4.1.3 Cambio de variable.

Existe una alternativa, para usar un menor número de compuertas, aunque a costa de aumentar la latencia del circuito. Observemos la Figura 4.1.3a. Sabiendo que una vez que una de las dos líneas es 1, la otra ya no cambiará, podríamos usar esta información para sólo usar un latch y menos compuertas.

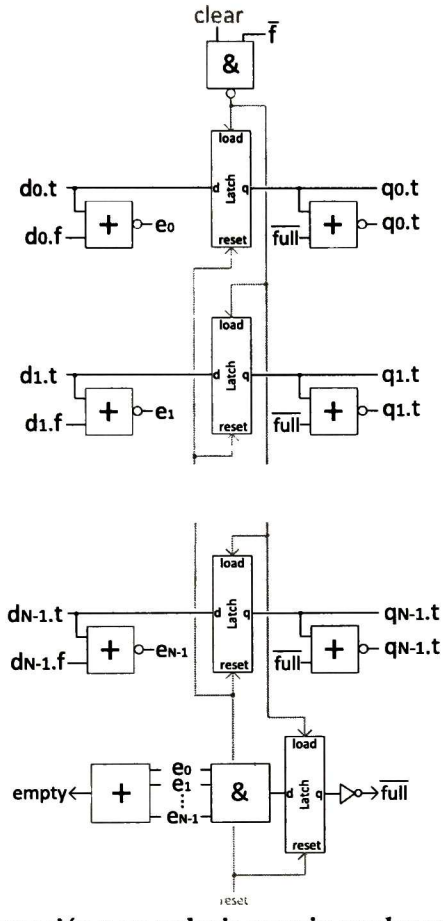


Figura 4.1.3a. Transformación para reducir espacio usado por los bloques de control.

Aquí podemos observar algunos detalles. Las compuertas OR para generar las señales ei se mantienen, pero ahora estas determinarán cuando ya se tengan todos los valores válidos, con la compuerta AND para todas ellas. Una vez que pasa este valor a través del Latch, el circuito con la compuerta NAND donde también se encuentra la señal de clear negada, cerrará todos los latches. Mientras que la señal de full resultante de la AND de todas las señales *e*, no sea 1, todas las señales .f serán 0, esto por que la señal full negada será siempre 1, lo que dará a la salida de cada NOR un cero.

Una vez que se asegure que todos los datos ya son válidos, la señal full levantará a todas las señales *.f* en las cuáles su señal *.t* no haya sido levantada. ¿Por qué sabemos que esto? Supongamos que el dato 0 recibe un 1 en la línea *.f*. Aunque este dato no pasará, la señal *e0* se levantará, lo que no hará nada a la AND que genera el full general a menos que todas las demás ya sean 1 también. En este caso, el valor de full será de 1, lo que hará que su negación con el dato *q0.t* aplicado a la NOR, levanten a *q0.f*.

De aquí se puede observar que se perderá capacidad del circuito para responder ante cualquier cambio inmediatamente. Supongamos que llega un dato 0 válido, este valro no se verá reflejado en el siguiente bloque, hasta que se hayan acompletado todos, hasta que la señal *full* levante a su respectiva línea *.f*. Sin encambio, con solo intercambiar las líneas, podríamos darle prioridad al valor 0 válido (*.f*). Dependiendo de la aplicación podría ser el valor que le demos a los 0's o 1's válidos.

En realidad, este método propuesto, es un cambio de variable que se hace, codificar todos los pares de datos 2*N en un grupo de N+1 datos.

4.2 Implementación de Multiplicador de Booth con una sola línea de sumadores.

Para comprender la potencialidad del diseño de circuitos asíncronos en Doble Riel, se explicará la implementación de un multiplicador de Booth [34, 35, 36], primero con una construcción convencional, usando la misma lógica que en los circuitos síncronos. Aquí comprobaremos que un circuito asíncrono, por naturaleza, es más rápido y eficiente para ejecutar operaciones que las versiones síncronas. De hecho, será más fácil ver por qué *los circuitos electrónicos son asíncronos por naturaleza.*

Después, se explicará el diseño final, en el que se eliminan todas las líneas de sumadores excepto una, se agregará circuitería y control adicional para poder añadir otras funciones, como el de utilizar el bloque como la misma Unidad Aritmética Lógica. Visto desde otra perspectiva, potenciaremos la utilidad de una ALU asíncrona en Doble Riel, para hacer una unidad mulitiplicadora, añadiendo un mínimo de control, ejemplificando, por una parte, la reutilización de bloques dentro del diseño y por otra, la posible reducción de área ocupada sin perder funcionalidad. Algunos ejemplos de multiplicadores asíncronos se pueden ver en [37, 38] y con codificación de Booth en [39].

4.2.1 Sumador binario, principio básico de diseño.

Dado que todas las operaciones en los circuitos digitales se encuentran codificadas binariamente, cualquier operación matemática debe ser realizada bajo esta base.

Sin embargo, la suma en cualquier base es similar a la que conocemos en base 10. Cada posición (bit 0, 1,... o dígito 0, 1...), al representar una potencia de la base usada, tendrá la propiedad de que al serle sumada una catidad que le rebase y no pueda ser representada por ella misma, añadirá una unidad más a la potencia imediata superior.

Algunos ejemplos de esto lo tenemos a continuación. El número 9_{10} (que significa 9 en base 10) al sumarle una unidad, provocará un desborde que tendrá que ser representado como añadir una unidad a la siguiente posición y que reniciará su valor. Esto se traduce en un 10_{10}.

Ahora bien, en los números binarios, como se dijo, no es la excepción, veamos el siguiente ejemplo. Se tiene le número 101011_2. Si la cantidad 001001_2 se le es sumada, tendremos lo siguiente:

De aquí podemos observar la necesidad de construir al menos 2 bloques genéricos. Uno que solo observe los dos bits al que se le pide sumar (conocido como medio sumador). Y otro que tome en cuenta, aparte de los dos sumandos, un posible bit de acarreo anterior (Sumador Completo o FA por sus siglas en inglés).

A continuación veremos como se construyen los bloques anteriores.

4.2.1.1 Sumador Completo (Full Adder) y Medio Sumador.

Para construir estos dos bloques [41, 42], usemos la Tabla 4.2.1.1a. De ella determinaremos la lógica necesaria para la construcción con compuertas del Medio Sumador.

Tabla 4.2.1.1a. Tabla para construir un medio sumador en lógica digital.

Medio Sumador			
A	B	Cout	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Para la construcción de el sumador completo, tenemos la Tabla 4.2.1.1b. Esta lógica servirá tanto para circuitos síncronos como asíncronos.

Tabla 4.2.1.1b. Lógica para un Sumador Completo.

Sumador Completo				
Cin	A	B	Cout	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

En la Figura 4.2.1.1a se muestra la construcción básica del Medio Sumador, el cual es usado cuando no se requiere tomar en cuenta un bit de acarreo de entrada.

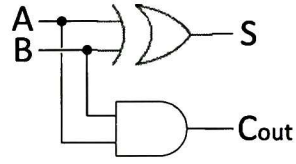


Figura 4.2.1.1a. Medio sumador en compuertas digitales.

En la Figura 4.2.1.1b se muestra la construcción del Sumador Completo a nivel compuerta.

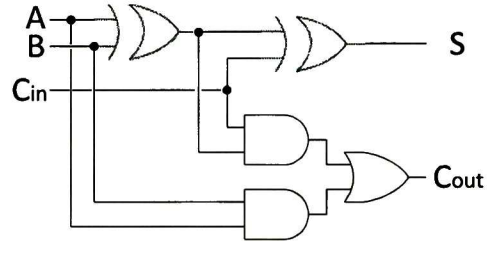


Figura 4.2.1.1b. Sumador Completo con compuertas digitales.

La construcción a nivel transistor es un proceso separado. Aunque se puede sustituir cada compuerta por su equivalente a nivel transistor, es recomendable utilizar mecanismos de reducción de área cuando los circuitos a convertir son grandes. En la Figura 4.2.1.1c podemos ver la construcción a nivel transistor del medio sumador usando la directa sustitución de las compuertas por sus equivalentes en transistor.

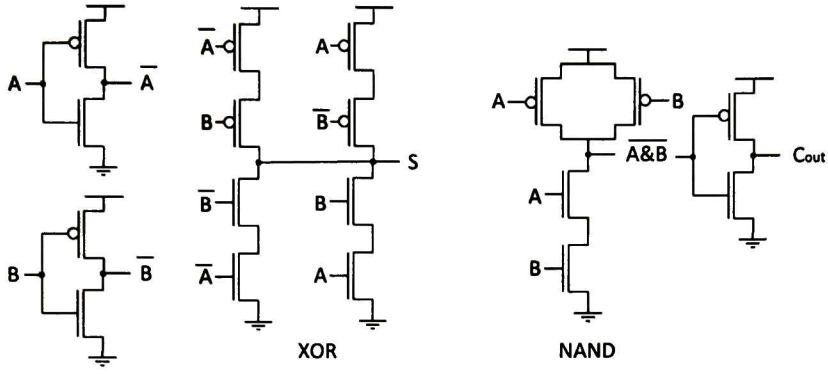


Figura 4.2.1.1c. Construcción del medio sumador a nivel transistor.

Y por último, en la Figura 4.2.1.1d se muestra la construcción del Sumador Completo a nivel transistor. Esta primera aproximación resulta de sustituir cada compuerta por su equivalente a nivel transistor, lo que nos da 38 transistores.

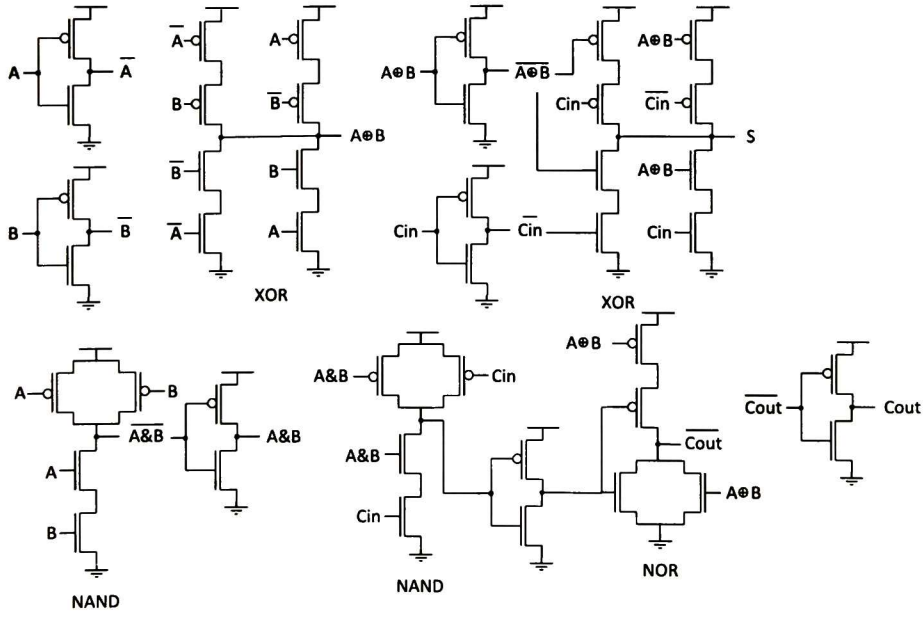


Figura 4.2.1.1d. Construcción del sumador completo a nivel transistor.

4.2.1.2 Sumador de N bits

Para construir el sumador de N bits [42, 43], usaremos ambas estructuras, el Sumador Completo (FA) y el Medio Sumador (HA). En la Figura 4.2.1.2a podemos observar la interconexión entre estos bloques para el sumador. Notemos la regularidad en el alambrado de los sumadores completos.

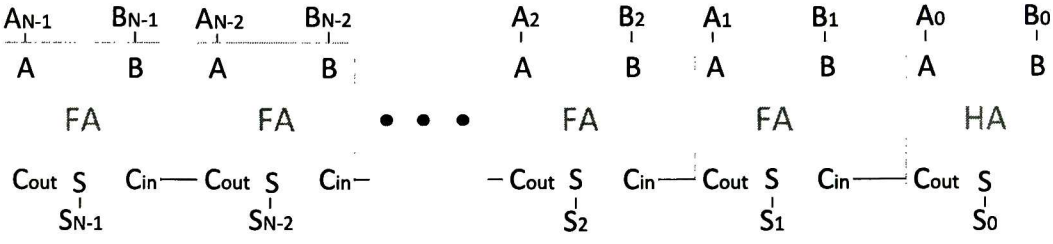


Figura 4.2.1.2a. Línea de sumadores para un sumador de N bits.

Como se ha recalcado, en los circuitos síncronos, es de vital importancia detectar el tiempo crítico de operación de un bloque. En este caso, el ¡*(path?) crítico tendrá que tomar en cuenta el hecho de que en algunos casos se necesite esperar a todos los retardos antes de dar el resultado como válido. Para ejemplificar esto, veamos dos ejemplos, la suma de 2 números de 4 bits.

Supongamos primero la suma del número 0001 y 1001. En este caso, basta ver que el único acarreo que existirá, será el del primer bit, entonces, se podría decir que en dos tiempos se sabría el resultado. En un tiempo se suman los respectivos bits, y en el otro sólo se ajusta el bit 1.

Ahora bien, la suma del número 0001 y 1111, sólo será válido después de que todos los acarreos hayan sido calculados. Siguiendo la tabla de tiempos mostrada a continuación, podemos ver que se necesitará más tiempo para saber el resultado correcto una vez que se dan todas las falsas transiciones.

Luego, es entendible por qué en los circuitos síncronos se tiene que esperar el peor de los casos, ya que no se sabe con certeza cuando se presentará. Para esto existen formas de anticipar el acarreo, con lo cual se puede reducir a la mitad o más el tiempo de espera, aunque añade bastante circuitería al sumador.

4.2.2 Sumador binario en Doble Riel

El funcionamiento del sumador en Doble Riel [44, 45], no es muy diferente a su versión síncrona. Tendrá, sin embargo, ciertas ventajas. Aquí se podrá observar el consumo eficiente de recursos, tanto de energía, al evitar falsas transiciones, como de rapidez, al usar sólo el tiempo necesario para cada operación, en vez de esperar el peor de los casos.

Al final de esta sección, podremos observar algunas estimaciones de tiempo ahorrado en términos probabilísticos. Desafortunadamente, la desventaja que pueden presentar estos circuitos, además de la ocupación, es el ligero incremento de la latencia, para el peor de los casos. Esto significa que aunque en promedio hará operaciones más rápidamente (de manera aleatoria), pudiesen existir casos, en los que al presentarse el peor de los casos de manera seguida, el tiempo de ejecución podría ser mayor comparado con el sumador síncrono.

4.2.2.1 Diseño de Sumador Completo y Medio Sumador en Doble Riel

El comportamiento de estos bloques no es muy diferente al descrito en la sección síncrona. Existe sólo una diferencia fundamental y que le dará potencialidad a cualquier bloque que utilice al Sumador Completo, la señalización débilmente indicante. Supongamos que de las 3 entradas (los dos bits a sumar y el bit de acarreo) dos son iguales, inmediatamente podremos decir que el bit de acarreo de salida será de este valor. Si dos de las entradas son 0, no podrá haber bit de acarreo de salida con un solo 1. De igual manera, si dos entradas son 1, el bit de acarreo inmediatamente será de 1, independientemente del valor de el otro bit. Esto será de mucha utilidad, como se verá en la sección siguiente, para un sumador de N bits.

4.2.2.2 Sumador de N bits en Doble Riel

Para el diseño del sumador de N bits, seguiremos el diseño básico síncrono. En la Figura 4.2.1.2 se muestra cómo estaría formado este bloque.

Al utilizar una señalización débilmente indicante, y la particularidad de que con dos datos iguales es suficiente para calcular el bit de acarreo de salida de cada Sumador Completo usado, en muchos casos eliminaremos la necesidad de esperar bits de acarreo.

Supongamos la suma de los los números 101000 y 011101. Como se puede observar, para las posiciones 0, 1 y 3, es posible determinar el acarreo de salida de esos bits. En el primer tiempo de ejecución, esos bits darán el bit de acarreo 0, 0 y 1. Es decir, en el mismo tiempo ya tendremos listos al menos dos bits de acarreo y no tener que esperar la propagación de este a través de la circuitería convencional. Si manejamos esto por tiempos, podemos ver en la Tabla 4.2.2.2a en cuantos se tendría el resultado final.

Tabla 4.2.2.2a. Resultado en tiempos para un sumador con bits de acarreo adelantados al sumar 101000 y 011101.

Tiempo	S0	C0	S1	C1	S2	C2	S3	C3	S4	C4	S5	C5
t1	1	0	-	0			-	1	-			
t2	1	0	0	0	1	0		1	0	1		-
t3	1	0	0	0	1	0	0	1	0	1	0	1

De aquí se puede observar que al saber los acarreos en algunos puntos dentro de la suma, es posible adelantar resultados (como en el caso de los dos últimos bits), en lugar de esperar el largo proceso de esperar el acarreo.

De hecho, con que se encuentre un acarreo justo en medio del número de bits que tienen las palabras, ahorrará la mitad de tiempo en encontrar la suma. Esto significa que para una suma de 8 bits, con el bit 4 de ambos sumandos iguales, la suma sólo consumirá el tiempo máximo de un sumador de 4 bits.

Incluso, en el mejor de los casos, cuando todos los bits sean iguales en ambos sumandos, el tiempo que le tomaría sería el del equivalente de un Sumador de 2 bits, ya que en dos tiempos podrían tener todos los valores de S (en el primer tiempo se tendrían todos los acarreos).

En el peor de los casos, donde todos los bits sean diferentes unos de otros, el sumador se comportará como un sumador de N bits. En otras palabras, *un sumador en Doble Riel no podrá empeorar su tiempo estimado de respuesta, si no al contrario, en la mayoría de los casos, la mejorará.*

4.2.2.3 Desempeño del Sumador de N bits

Una vez determinado que el encontrar bits similares ayudará a acelerar la ejecución de las operaciones del sumador, es importante aclarar, cada cuánto sucederá esto. De esta manera, si se determina que la mayoría de las veces se comportará como un sumador de menos bits, será una razón importante para ser usado, debido a que en promedio, ejecutará muchas más operaciones que un sumador síncrono.

Supongamos que hablamos de un sumador de 3 bits. Debido a que el bit cero no cuenta pues no necesita de un acarreo anterior y obtiene siempre su bit de acarreo de salida en el primer tiempo, no se toma en cuenta. Tampoco tomamos en cuenta el último bit, ya que este no tendrá un bit sucesor que espere acarreo. Esto quiere decir que sólo debemos analizar al bit 1. La mitad de las veces ambos sumandos serán iguales (00 y 11) y la otra mitad serán diferentes (01 y 10), por lo que la mitad de las veces se comportará como un sumador de 2 bits (en dos tiempos dará el resultado) y la otra mitad como el peor de los casos (como un sumador de 3 bits).

Ahora bien, para analizar correctamente esto, supongamos estos dos números a sumar, 00000 y 10101. La solución se obtendrá en sólo dos tiempos, como se observa en la Tabla 4.2.2.3a.

Tabla 4.2.2.3a. En sólo dos tiempos se puede saber el resultado de sumar 00000 y 10101.

Tiempo	S0	C0	S1	C1	S2	C2	S3	C3	S4	C4
t1	1	**0**		**0**				**0**		
t2	1	0	**0**	**0**	**1**	**0**		0	**1**	**0**
t3	1	0	0	0	1	0	0	0	1	1

En el caso anterior, teníamos sólo 2 bits iguales, el bit 1 y 3. En el siguiente ejemplo, pondremos 3 bits iguales, sumando 00000 y 01100. En la Tabla 4.2.2.3b se muestra los tiempos que le tomará obtener el resultado.

Tabla 4.2.2.3b. Más tiempos para sumar 00000 y 01100 a pesar de tener más bits iguales.

Tiempo	S0	C0	S1	C1	S2	C2	S3	C3	S4	C4
t1	0	**0**		**0**	-					**0**
t2	0	0	**0**	**0**	**1**	**0**				**0**
t3	0	0	0	0	1	0	1	0		
t4	0	0	0	0	1	0	1	0	0	0

Un resultado muy importante, con mayor número de bits que coincidan, existen casos en los que van a tardar más. Pero en realidad, lo que pareciera perjudicar a la naturaleza asíncrona, es en realidad un beneficio, por que en realidad son los bits consecutivos que son diferentes los que harán que el sumador se vaya comportando en sus peores casos. Esto por que al tener cadenas consecutivas de bits diferentes, para llegar a determinar el valor del último reuslltado, se tendrá que esperar a cada uno de los acarreos. En otras palabras, para medir el comportamiento de un sumador asíncrono en Doble Riel ante una serie de palabras, es suficiente con saber cuál es la cadena mayor de bits diferentes.

Por ejemplo, supongamos los números a sumar son 010100 y 100101, traduciendo esto a bits iguales o diferentes, tendríamos las siguienteDDIIID, donde los bits que nos interesan son los que están entre el primero y el último. ¿Por qué? Por que el primer bit independientemente de si son iguales o diferentes los bits siempre podrá determinar su bit de acarreo de salida en el primer tiempo. El último bit no provee un bit de acarreo final, por lo tanto tampoco influye en el tiempo en el que la suma ya está lista. En este ejemplo, el sumador tendrá listo el resultado en 3 tiempos, en el primero calculará S0, en el segundo, S1, S2, S3 y S4 (por que en el primer tiempo C0, C1, C2 y C3 ya estarán calculados);

Ahora analizemos un sumador de 5 bits. Para los bits 1, 2 y 3, existen varias posibilidades, dependiendo de estas combinaciones, es el comportamiento esperado. Como siempre, el tener todos los bits iguales generará el mejor de los casos (y se comportará como un sumador de 2 bits), así como el tener todos diferentes nos dará el peor, o más bien, generará que el sumador se comporte como el número de bits que tiene.

En la Tabla 4.2.2.3c podemos observar las posibles combinaciones y el comportamiento que tendría de acuerdo a ella, para los tres bits que nos interesan. Para hacer este análisis, tomemos en cuenta que los casos críticos provienen del número de bits diferentes seguidos que existen, ya que estos determinan cuándo hay que esperar a resultados previos dentro del circuito.

Tabla 4.2.2.3c. Las posibles combinaciones de bits iguales o diferentes en un sumador de 5 bits (tomadno en cuenta sólo los 3 intermedios).

Sumador de X bits	A3=B3	A2=B2	A1=B1
X=2	I	I	I
X=3	I	I	D
X=3	I	D	I
X=4	I	D	D
X=3	D	I	I
X=3	D	I	D
X=4	D	D	I
X=5	D	D	D

Podemos ver que aunque haya diferencia de bits, pero si están salteadas sus posiciones, no afectarán tanto como cuando son seguidas. Esta es una propiedad benéfica para el sumador, ya que determinará una tendencia a comportarse como un sumador de menos bits, como vemos en la tabla, veremos más adelante cuando analicemos un mayor número de bits. En este caso en particular, sólo una vez se comportará como un sumador de 5 bits, una vez como de 2 bits, 4 veces como de 3 bits y 2 veces como de 4 bits. Desde aquí podemos ver que a partir de la media, no hay una distribución equitativa, habrá una tendencia a comportarse por debajo de ésta la mayoría de las veces.

Es importante recalcar que no existen 8 posibles operaciones, sino 1024 (32x32), que sería el conjunto de sumas:

{00000 + {00000, 00001, ..., 11111}, 00001+{00000, 00001, ..., 11111},...,+{11111, 00001, ..., 11111}

Sin embargo, eliminando el primer y el último bit, tendríamos en realidad 64 posibilidades para cada comparación entre bits, pero en términos de bits iguales o diferentes, en realidad sólo tenemos 8 posibilidades (que se repiten ocho veces). Tendríamos en realidad, comparando bit por bit, sólo ocho posibilidades, desde que todas estas comparaciones sean iguales (III), hasta que sean todas diferentes (DDD), lo que nos dará ocho combinaciones regulares.

Ahora veamos el comportamiento de un sumador de 8 bits. En la Tabla 4.2.2d podemos observar cómo la tendencia a desempeñarse por debajo de la media se conserva y se acentúa.

Tabla 4.2.2d. Comportamiento para un sumador de 8 bits dependiendo de la igualdad o diferencia de sus bits intermedios.

Sumador de X bits	Bits	Sumador de X bits	Bits	Sumador de X bits	Bits	Sumador de X bits	Bits
X=2	IIIII	X=3	IDIII	X=3	DIIII	X=4	DDIII
X=3	IIIID	X=3	IDIID	X=3	DIIID	X=4	DDIID
X=3	IIIDI	X=3	IDIDI	X=3	DIIDI	X=4	DDIDI
X=4	IIIDD	X=4	IDIDD	X=4	DIIDD	X=4	DDIDD
X=3	IIDII	X=3	IDIDII	X=3	DIIDII	X=4	DDIDII
X=3	IIDID	X=3	IDIDID	X=3	DIIDID	X=4	DDIDID
X=4	IIDDI	X=4	IDIDDI	X=4	DIIDDI	X=4	DDIDDI
X=5	IIDDD	X=5	IDIDDD	X=5	DIIDDD	X=5	DDIDDD

X=3	IIDIII	X=4	IDDIII	X=3	DIDIII	X=5	DDDIII
X=3	IIDIID	X=4	IDDIID	X=3	DIDIID	X=5	DDDIID
X=3	IIDIDI	X=4	IDDIDI	X=3	DIDIDI	X=5	DDDIDI
X=4	IIDIDD	X=4	IDDIDD	X=4	DIDIDD	X=5	DDDIDD
X=4	IIDDII	X=5	IDDDII	X=4	DIDDII	X=6	DDDDII
X=4	IIDDID	X=5	IDDDID	X=4	DIDDID	X=6	DDDDID
X=5	IIDDDI	X=6	IDDDDI	X=5	DIDDDI	X=7	DDDDDI
X=6	IIDDDD	X=7	IDDDDD	X=6	DIDDDD	X=8	DDDDDD

En la Figura 4.2.2.a se muestra la distribución del comportamiento para este sumador. Podemos observar que 31.25% de las veces se comportará como un sumador de 3 bits, esto quiere decir que cerca de una tercera de las veces el sumador de 8 bits resolverá las sumas en un tiempo aproximado de lo que lo haría un sumador convencional de 3 bits.

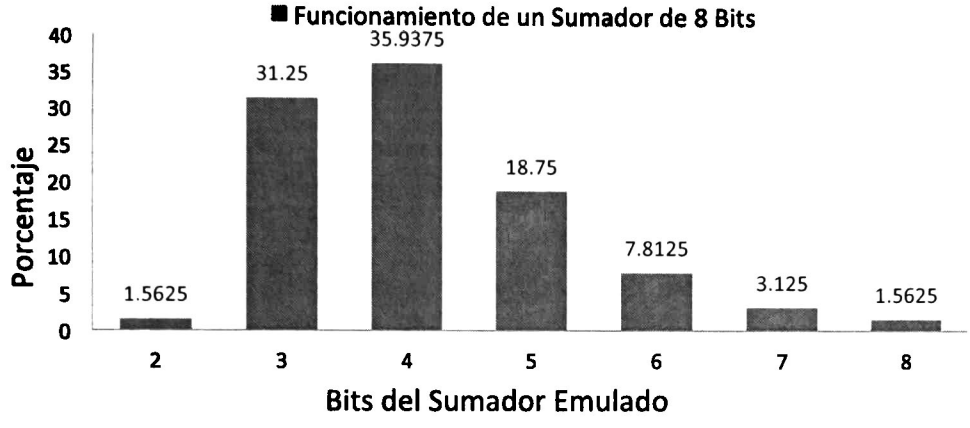


Figura 4.2.2d. Probabilidad de que un sumador de 8 bits se comporte en sumadores de menor número de bits.

Teniendo en cuenta estos datos y observando el comportamiento de la sucesión de números formada para cada sumador, es posible obtener una serie para determinar cuantas veces se comporta un sumador de N bits como un sumador de 3 bits, y se describe en la ecuación 4.2.2a.

$$f_N^3 = f_{N-1}^3 + f_{N-2}^3 + 1, \quad N > 3 \qquad (4.2.2a)$$

Teniendo en cuenta los casos base de f(3) = 1 y f(2) = 0, podemos determinar cualquier valor posterior. Para f(4) = 2, f(5) = 4, f(6) = 7 y así sucesivamente. En la Figura 4.2.2e podemos ver el comportamiento de los sumadores conforme se alcanzan niveles altos en número de bits sumados.

En la ecuación 2 se describe la serie necesaria para determinar cuantas veces un sumador de N bits se comportaría como un sumador de 4 bits. En esta serie, se usan los tres números anteriores, además de usar un elemento de la serie anterior (ecuación 4.2.2b). Es claro que para cualquier N<4, f = 0.

$$f_N^4 = f_{N-1}^4 + f_{N-2}^4 + f_{N-3}^4 + f_{N-3}^3 + 1 \qquad (4.2.2b)$$

En general, podemos determinar para un sumador de x bits, cuántas veces opera como un sumador de 3, 4, 5, ..., N-1 bits, donde si N<x, f = 0, y si N = x, f = 1.

$$f_N^x = f_{N-1}^x + f_{N-2}^x + \cdots + f_{N-x-1}^x + f_{N-x-1}^{x-1} + \cdots + f_{N-x-1}^3 + 1 \qquad (4.2.2b)$$

Con esta ecuación se puede construir la tabla para el comportamiento de cualquier sumador. En la Figura 4.2.2d se muestra la distribución de la forma de operar de un sumador de 8 bits, en ella se puede notar que la mayor parte de las operaciones (casi el 70%) las realiza como un sumador de menor o igual a 4 bits.

En la Figura 4.2.2e se muestra cómo se comporta un sumador de 32 bits. En esta Figura se puede apreciar que la mayor parte de los datos generarán un funcionamiento de un sumador menor a los 8 bits, al sumar los porcentajes por debajo de este comportamiento, nos da un 90% de los casos. Incluso, más del 60% de las operaciones se realizan en tiempos de sumadores de 4, 5 y 6 bits.

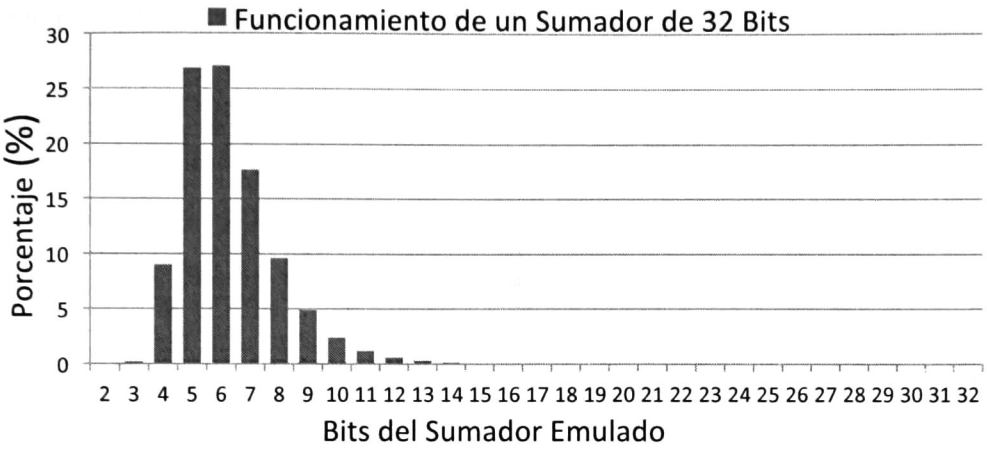


Figura 4.2.2e. Probabilidad de que un sumador de 32 bits se comporte como un sumador de menor número de bits.

De aquí podemos concluir que las operaciones en Doble Riel de este tipo, son mucho más eficientes en proporción con sus equivalentes síncronos. Pensar que para un sumador de 32 bits, 90% de los casos ocuparán menos de la cuarta parte del tiempo requerido, nos dice que para operaciones de mayor cantidad de bits, la relación será todavía más contrastante.

4.2.3 Construcción a nivel transistor de Sumador en Doble Riel

Construir a nivel transistor, como se ha visto en el diseño síncrono, conlleva a ahorrar cierto espacio, para algunas bloques. El caso de los circuitos asíncronos no es la excepción.

4.2.3.1 Sumador Completo

En la Figura 4.2.3.1 podemos observar cómo se construiría un sumador completo a nivel transistor [1], con señalización débilmente indicante. Podemos observar que el número de transistores es de 34, lo cual no es muy alejado del número de transistores usados en la versión síncrona (el cual en su versión sin reducir es de transistores).

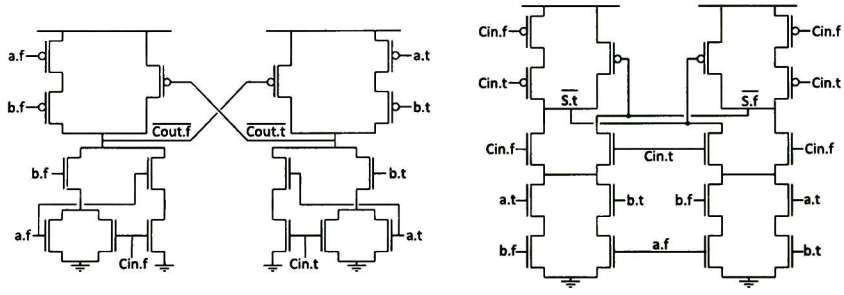


Figura 4.2.3.1. Sumador completo a nivel transistor para señalización débilmente indicante en Doble Riel.

4.2.4 Multiplicador Binario en Doble Riel, la primera aproximación.

Para diseñar el multiplicador, primero revisaremos como se construye uno en su versión síncrona. Para multiplicadores de N bits, además, se describirá su construcción en base a sumadores de N+1 bits.

4.2.4.1 Multiplicación de dos números binarios.

Analicemos la multiplicación de A*B, que por definición, es la suma de A veces B ó B veces A. Es decir, 100*101 significaría sumar 4 veces 5 ó 5 veces 4. Realizar esto en un circuito, sería improductivo, ya que no podríamos tener un número de sumadores de acuerdo a el valor de uno de los multiplicandos. Significaría tener

1023 sumadores de 10 bits (al multiplicar 1111111111xB9B8...B0), lo que se traduciría en algo irreazible en términos de ocupación (y también de tiempo de operación). Una mejor forma de realizar la multiplicación, es realizar N sumas de uno de los multiplicandos, es decir, menos sumas. A continuación se describe el desarrollo necesario para llegar a esta forma más eficiente y que además pueda ser traducida a un circuito.

$$M = A * B$$

$$M = A_2 * [(B_0 * 2^0) + (B_1 * 2^1) + (B_2 * 2^2) + \cdots + (B_{N-1} * 2^{N-1})]$$

$$M = A * (B_0 * 2^0) + A * (B_1 * 2^1) + A * (B_2 * 2^2) + \cdots + A * (B_{N-1} * 2^{N-1})$$

$$M = B_0 * (A * 2^0) + B_1 * (A * 2^1) + B_2 * (A * 2^2) + \cdots + B_{N-1} * (A * 2^{N-1})$$

$$M = B_0 * (A) + B_1 * (A \ll 1) + B_2 * (A \ll 2) + \cdots + B_{N-1} * (A \ll N - 1)$$

Ahora bien, esto lo podemos traducir de la siguiente manera, el multiplicar 0101x1010 se puede hacer de dos formas, de acuerdo a esta última ecuación:

$$M = 1 * (1010) + 0 * (10100) + 1 * (101000) + 0 * (1010000)$$

$$M = 0 * (0101) + 1 * (01010) + 0 * (010100) + 1 * (0101000)$$

***En ambos casos, el resultado es 50. A) 10 + 0 + 40 + 0. B) 0+ 10 + 0 + 40**

De esta forma es más fácil observar cómo se desarrollaría un circuito para realizar esta suma.

Aunque en los circuitos síncronos existen diversas formas para implementar la multiplicación, en general para ambas topologías, la circuitería usada usará de N sumadores de al menos N bits (entre Sumadores completos y Medios Sumadores).

El acarreo, como se ha visto, es un problema común para los sumadores síncronos y asíncronos, pero para el caso de los circuitos asíncronos de Doble Riel,

lo es en menor medida. Como se vió en la sección anterior, el acarreo ayudará, de forma natural, a acelerar el proceso mediante el cual se obtienen los resultados. El añadir circuiteria para el adelanto del acarreo aumentaría la probabilidad de que el sumador se comportara como uno de menor catidad de bits dependiendo de donde fueran añadidos los bloques correspondientes.

El desempeño de este multiplicador en Doble Riel, potenciará los beneficios vistos en el sumador. En realidad, para calcular el resultado final, se tuvieron que haber obtenido todos los valores y acarreos previos, lo que nos dice que si un sumador intermedio acelera la obtención de su resultado o acarreos, acelerará la obtención del resultado final en general.

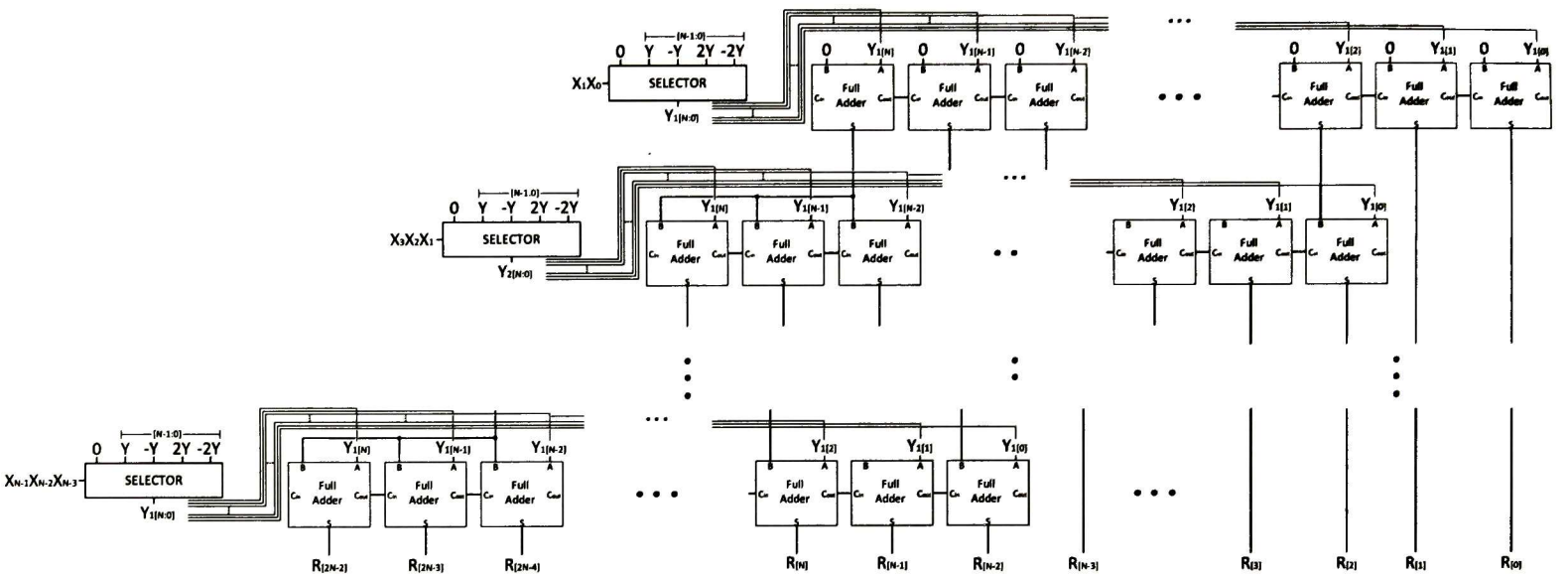


Figura 4.5.2. Multiplicador de Booth para reducir espacio y acelerar la respuesta.

4.2.5 Multiplicador de Booth en Doble Riel, una mejora sustancial.

Se puede observar que realizar un multiplicador de esta forma, consumiría muchos recursos. Existe una alternativa para el ahorro de tiempo y espacio, el cual utiliza la Codificación de Booth.

Booth se dio cuenta que para obtener un resultado, existen varias formas de obtenerlo. Por ejemplo, multiplicar a*b es lo mismo que multiplicar (a+1)*b-b.

4.2.5.1 Codificación de Booth, una mejora al algoritmo de multiplicación

La Codificación de Booth [34, 35, 36], agrupa bits para uno de los operandos (supongamos y), de tal forma que se vayan creando grupos de x de bits, tomando los N elementos de y. A continuación se muestra cómo se van agrupando estos bits que después usaremos para determinar los valores del otro multiplicador (supongamos x) que se irán sumando para obtener el valor final de la multiplicación.

$$[y_0, y_1, \cdots, y_{x-2}]$$

$$[y_{x-2}, y_{x-1}, \cdots, y_{2x-3}]$$

$$\cdots$$

$$[y_{i*x-i-1}, y_{i*x-i}, \cdots, y_{(i+1)*x-i-2}]$$

$$\cdots$$

$$[y_{\left[\frac{N-1}{x-1}\right]*(x-1)-1}, y_{\left[\frac{N-1}{x-1}\right]*(x-1)}, \cdots, y_{N-1}]$$

Donde $i < [N\text{-}1/x\text{-}1]$ y [a/b] significa el entero de la división de a entre b, por ejemplo: [8/3]=2, [14/4]=3, etc. Supongamos dos ejemplos, codificar un número de 8

bits, en grupos de 3 ($N = 8$, $x = 3$ y $[(N-1)/(x-1)] = 3$) y codificar 18 bits en grupos de 5 ($N = 18$, $x = 5$ y $[(N-1)/(x-1)] = 4$).

Para el primer caso, tendríamos los grupos $[y_0, y_1]$, $[y_1, y_2, y_3]$, $[y_3, y_4, y_5]$ y $[y_5, y_6, y_7]$. Para el segundo, $[y_0, y_1, y_2, y_3]$, $[y_3, y_4, y_5, y_6, y_7]$, $[y_7, y_8, y_9, y_{10}, y_{11}]$ y $[y_{11}, y_{12}, y_{13}, y_{14}, y_{15}]$ y $[y_{15}, y_{16}, y_{17}]$.

Esta codificación permitirá que el número de sumas sea reducido exactamente al número de códigos obtenidos menos 1. Es decir, que para una multiplicación de 2 número de 8 bits, se requerirán sólo 3 líneas de sumadores cuando el multiplicador se codifique en grupos de 3 bits. Para el caso de multipliaciones de 32 bits, por ejemplo, en lugar de tener más de 30 líneas de sumadores, tendremos sólo 9 líneas de sumadores, codificando en grupos de 4 bits y 14 líneas de sumadores en grupos de 3 bits, una reducción bastante drástica en cuanto a ocupación, y como veremos después, en tiempo de operación.

En la Tabla 4.2.5.1 se aprecia la codificación de Booth para grupos de 3 bits. Con esto, podrá construirse el multiplicador a nivel de bloques y compuertas, descrito en la sección siguiente. Se asume que el multiplicando es Y, que i es la iteración para el control de cada línea de sumador de N+1 bits.

Tabla 4.2.5.1. Codificación de Booth para grupos de 3 bits.

2i+1	2i	2i-1	Pi
0	0	0	0
0	0	1	Y
0	1	0	Y
0	1	1	2Y
1	0	0	-2Y
1	0	1	-Y
1	1	0	-Y
1	1	1	0

4.2.5.2 Implementación de la codificación de Booth en Doble Riel

Para implementar el multiplicador de Doble Riel con codificación de Booth se seguirá la estructura mostrada en la Figura 4.2.5.2. En ella se puede observar que el número de sumadores será reducido como el algoritmo de Booth lo hace para su versión síncrona.

4.2.6 Multiplicador de Booth en Doble Riel con el mínimo de ocupación.

Aunque el uso de la codificación de Booth es una mejora significativa respecto al uso de espacio y velocidad de respuesta, la propuesta de este trabajo consiste en incluso dejar de usar un número de N-1 líneas de sumadores para solo usar una.

Obervando el hecho de que la multiplicación es recursiva, es posible deducir que existe alguna forma para usar el mismo sumador, reusar el valor obtenido en cada suma (iteración) y recorrer el dato de acuerdo a la codificación para calcular el siguiente, hasta obtener el resultado final. En la Figura 4.2.6a podemos observar la lógica mediante la cual funciona esta idea, usando una codificación de Booth de 3 bits.

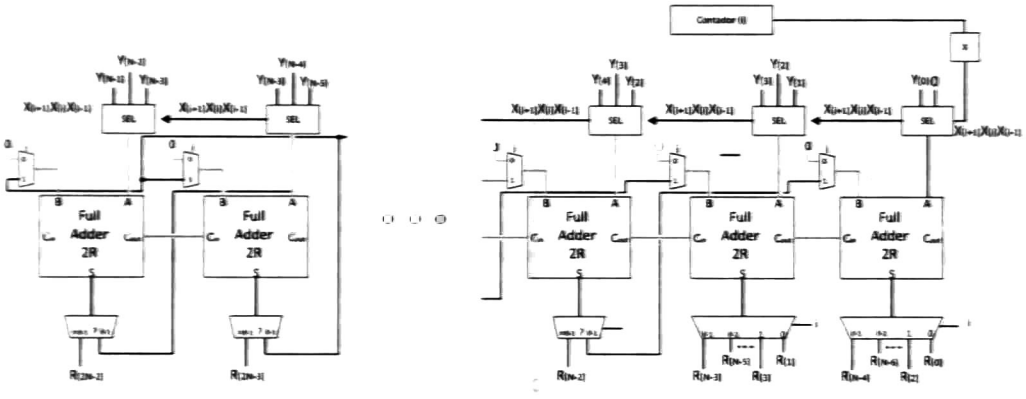


Figura 4.2.6a. Codificación de Booth para 3 bits en un sumador de Doble Riel.

Para los circuitos síncronos también es posible usar esta propuesta, sin embargo, no traerá mejoras en el rendimiento, ya que siempre estará determinado por el peor de los tiempos, el cual se incrementará por la lógica de control y el hecho de tener que esperar siempre el resultado de cada suma.

Sin embargo, las propiedades de adelanto de acarreo para los circuitos asíncronos en el sumador de N bits se conservarán para la versión en Doble Riel, ejecutando de manera prácticamente igual las operaciones que en la versión con N-1 líneas de sumadores, pero ahorrando una gran cantidad de espacio.

4.2.6.1 La misma unidad multiplicadora como ALU

Incluso, con los bloques de control propuestos, es posible usar este multiplicador como una unidad ALU [46, 47], lo que implicaría poder ser además sumador, restador, comparador lógico (AND, OR, XOR, igual a cero, etc). Aquí es donde podemos comprender más la naturaleza asíncrona, la reutilización de bloques.

4.2.6.2 Multisegmentación

Otra ventaja significativa, es la posible potencialización para el uso de una multisegmentación. Esto significa implementar en dos dimensiones el pipeline usado para la construcción de circuitos. De forma convencional, el pipeline de un diseño síncrono o asíncrono es secuencial. En el caso de el uso de circuitos de Doble Riel como el propuesto, es posible establecer incluso que la segmentación sea en dos dimensiones, es decir, que los paquetes de datos en los bloques no sea siempre el mismo.

Dicho de otra forma, con este diseño, es posible tener en una ALU varios paquetes de información procesándose al mismo tiempo. En un momento dado podríamos tener una multiplicación, una comparación, una suma y una resta de diferentes conjuntos de datos. Para esto, faltará implementar no sólo FIFOs para los datos (o propagar este tipo de diseño a través del circuito), sino lógica que permita ir guardando estados previos en la lógica de control, y no sólo para datos.

Capítulo 5

Conclusiones

A lo largo de este trabajo se realizaron varias propuestas nuevas, tanto para Riel Sencillo como para Doble Riel, incluso, algunas de estas utilizables en diseño síncrono.

Para Riel Sencillo, se diseñó una UART ajustable y reconFigurable. Ajustable por que puede enviar una cantidad N de datos en paquetes de 8, 9 o 10 bits, creando todas las instancias necesarias en HDL, usando siempre compuertas (con la opción generate de Verilog), de tal forma que pueda ser extraído a nivel transistor por herramientas externas o el mismo diseñador. ReconFigurable por que en tiempo de ejecución es posible ajustar su BAUD rate, sin necesitar señal de reloj, tan sólo entrando en un modo especial de reconFiguración y recibiendo un dato, se podrá calcular el valor del BAUD rate que el otro puerto usa para comunicarse.

Para el diseño en Doble Riel, se describieron las compuertas más usadas en esta topología, como la AND, OR, XOR, NOT, MUX, entre otras, tanto para el diseño a nivel compuerta como a nivel transistor. Esta documentación no se encontró en otro lado.

Adicionalmente, se propuso un nuevo protocolo de comunicación de 4 fases, basado en el convencional que usa compuertas Muller C. Con este protocolo modificado, la propagación de los datos es más rápida, aunque al costo de añadir cierta dificultad y circuitería extra.

También se propuso una forma diferente para transportar la información que se lleva de un bloque a otro, de tal forma que se utilice menos espacio, usando una codificación para el cambio de variable, ahorrando latches y control a la mitad, aunque aumentando la latencia y disminuyendo la capacidad de propagar información rápidamente. Para los circuitos síncronos y asíncronos, se propuso una metodología de diseño a nivel transistor, para ahorrar espacio sabiendo que éstos funcionan con lógica negada, evitando para la mayoría de los casos usar inversores.

Adicionalmente, se expusieron las principales ventajas del diseño asíncrono para la resolución de problemas aritméticos, específicamente hablando del sumador y multiplicador. Analizando el comportamiento de un Sumador de N bits, donde se comprobó que estadísticamente se comportará mucho mejor que su versión síncrona.

La mayor contribución de este trabajo fue el multiplicador en Doble Riel, con una sola línea de sumadores. Con esto, el ahorro de espacio es ya considerable. Posteriormente, con circuitería mínima añadida, incluso se puede agregar la función de la ALU, con el multiplicador, lo que se traduce en aún más ahorro de esapacio, pues ambas unidades se localizarían en el mismo espacio.

Apéndice B

Archivos Fuente y Salida

En este apéndice se adjuntan los archivos fuente de la unidad UART asíncrona reconFigurable, así como el código en HDL para construir las unidades asíncronas simples para el protocolo modificado propuesto y la implementación del sumador y multiplicador con codificación de Booth.

Arbol

```
module arbol #(parameter N=8, Op=1)
        (input [N-1:0] y, output sal);
wire [(N+N%2)/2-1:0] r;
genvar j;
generate if (N!=2)
        begin: G1
                arbol #((N+N%2)/2,Op) arbol2 (r,sal);
        end
        begin: G2
                for (j=0;j<(N-N%2)/2;j++)          begin
                        if(Op==1)
                                assign r[j]=y[2*j] |
y[2*j+1];
                        if(Op==2)
                                assign r[j]=y[2*j] &
y[2*j+1];
                        if(2*(j+1)!=N && j==(N-N%2)/2-1)
                                assign r[j+1]=y[N-1];
                end
        end
endgenerate
generate if (N==2)
        begin: G3
                if(Op==1)
                        assign sal=y[0] | y[1];
                if(Op==2)
                        assign sal=y[0] & y[1];
        end
endgenerate
endmodule
```

BCA

```
module BCA
        (input Ri,Ai,RST,
         output Xo,Ao);

wire C,T2;

assign Xo=C ^ Ai;
MullerC CM (Ri,~T2,RST,C);
TOGGLE CT (Xo,RST,Ao,T2);
endmodule
```

Cont_Var_Fijo

```
module Cont_Var_Fijo #(parameter C=4, val=7)
        (input Cont,rst,
         output reset,              output [C-1:0] Sal);

wire [C-1:0] Sal2,Sal3;
wire [C-2:0] ands,ors;
wire [num_unos(C)-1:0] ands_reset;
genvar k,i;


//Señal para resetear el contador
generate for(k=0;k<C;k=k+1) begin:RESETEO
                if (busca(k)==1) begin:SITUAR_AND
                        assign
ands_reset[busca(k)]=Sal[k
                        end
        end
endgenerate

generate
        if(num_unos(C)>1)
                arbol #(num_unos(C),2) res
(ands_reset,reset);
        if(num_unos(C)==1)
                assign reset=ands_reset[0];
endgenerate

//Sal(0) solo es la negaciÛn de la salida del Flip-Flop
assign Sal2[0]=~Sal[0], Sal3[0]=Sal2[0] & reset,
ands[0]=Sal[0];
Flip F0 (Cont,rst,Sal3[0],Sal[0]);
//Las demÁs salidas siguen un patrÛn
generate  for(k=2;k<=C;k=k+1) begin:PATRON
                if (k==2)
                        assign Sal2[1]=(Sal[1] ^ Sal[0]) &
reset;
                if (k>2)
                        assign Sal2[k-1]=Sal[k-1] ^
ands[k-2], ands[k-2]=Sal[k-2] & ands[k-3];
                assign Sal3[k-1]=Sal2[k-1] & reset;
                Flip FC (Cont,rst,Sal3[k-1],Sal[k-1]);
        end
endgenerate

function integer busca; // constant function
        input integer pos;
        integer pot,val2;
        begin
        for (pot=C-1,val2=val;pot>=pos;pot=pot-1)
                if(val2>=2**pot) begin
                        val2=val2-2**pot;
                        busca=1;
                end
                else
                        busca=0;
        end
endfunction
function integer num_unos; //Funcion de doble uso!!
        input integer val;
        integer pos;
        begin
        for (pos=0,num_unos=0;pos<val;pos=pos+1)
                if(busca(pos)==1)
                        num_unos=num_unos+1;
        end
endfunction
endmodule
```

Bloque Serial

```
/*N:número de bits a enviar, NBITS:lo que puede enviar
la UART,
TAM_FIFO: capacidad de almacenamiento de la FIFO.*/
module BloqueSerial #(parameter
N=30,NBITS=10,TAM_FIFO=3,TAM_BAUD=5)
```

```
        (input Ri,rst,      input [N-1:0] D, input
[TAM_BAUD-1:0] BAUD,
        output Tx,Ru,Au,Ro,Ai,xp,Ao,    output [NBITS-
1:0] Dtx,          output [N-1:0] Dfifo);

//wire Ru,Au,Ro,Ai,xp,Ao;
//wire [N-1:0] Dfifo;
//wire [NBITS-1:0] Dtx;
/*Memoria FIFO, en ella se guardan los datos para
enviar, de N bits, se cambia
la seÒal Ri y un reset al principio*/
FIFO #(N,TAM_FIFO) F (D,Ri,Ai,rst,Ao,Ro,xp,Dfifo);
Transmisor #(NBITS,TAM_BAUD) TT
(rst,Ru,BAUD,Dtx,Tx,Au);
UART_FIFO #(N,(N-1-(N-1)%NBITS)/NBITS,clog((N-1-
(N-1)%NBITS)/NBITS),NBITS) UF
        (rst,Ro,Au,xp,Dfifo,Ru,Ai,Dtx);
/*El transmisor recibe los datos provenientes de
UART_FIFO y se comunica con
Ru y Au*/
//Transmisor #(NBITS,TAM_BAUD) T
(rst,Ru,BAUD,Dtx,Tx,Au);

function integer clog
(input integer num);
begin
        for (clog=1; num>=2**clog; clog=clog+1);
        end
endfunction
endmodule
```

Cont_Var

```
module Cont_Var #(parameter C=4)
        (input Cont,rst,  input [C-1:0] y,
         output reset,              output [C-1:0] Sal);

wire [C-1:0] Sal2,Sal3,xors;
wire [C-2:0] ands,ors;

genvar k;
//SeÒal para resetear el contador, se cambian las XOR's
por not's y OR's
generate for(k=0;k<C;k=k+1) begin
        //assign xors[k]=Sal[k] ^ y[k];
                assign xors[k]=Sal[k] | ~y[k];
        end
endgenerate
arbol #(C,1) res (xors,reset);
//Sal(0) solo es la negaciÛn de la salida del Flip-Flop
assign Sal2[0]=~Sal[0], Sal3[0]=Sal2[0] & reset,
ands[0]=Sal[0];
Flip F0 (Cont,rst,Sal3[0],Sal[0]);
//Las demÁs salidas siguen un patrÛn
generate  for(k=2;k<=C;k=k+1) begin
                if (k==2)
                        assign Sal2[1]=(Sal[1] ^ Sal[0]) &
reset;
                if (k>2)
                        assign Sal2[k-1]=Sal[k-1] ^
ands[k-2], ands[k-2]=Sal[k-2] & ands[k-3];
                assign Sal3[k-1]=Sal2[k-1] & reset;
                Flip FC (Cont,rst,Sal3[k-1],Sal[k-1]);
        end
endgenerate
endmodule

module Cont_Var_Ret #(parameter C=4)
        (input Cont,rst,Ri,       input [C-1:0] y,
         output reset,            output [C-1:0] Sal);

wire [C-1:0] xors,Sal2,Sal3;
wire [C-2:0] ands,ors;

genvar k;
//SeÒal para resetear el contador
generate for(k=0;k<C;k=k+1) begin
                assign xors[k]=Sal[k] ^ y[k];
        end
endgenerate
arbol #(C,1) res (xors,reset);
//Sal(0) solo es la negaciÛn de la salida del Flip-Flop
assign Sal2[0]=~Sal[0], ands[0]=Sal[0];
mux m1 (Sal2[0],Ri,reset,Sal3[0]);
Flip F0 (Cont,rst,Sal3[0],Sal[0]);
//Las demÁs salidas siguen un patrÛn
generate  for(k=2;k<=C;k=k+1) begin
                if (k==2)
                        assign Sal2[1]=(Sal[1] ^ Sal[0]) &
reset;
                if (k>2)
                        assign Sal2[k-1]=Sal[k-1] ^
ands[k-2], ands[k-2]=Sal[k-2] & ands[k-3];
                assign Sal3[k-1]=Sal2[k-1] & reset;
                Flip FC (Cont,rst,Sal3[k-1],Sal[k-1]);
        end
endgenerate
endmodule
```

CONTROL

```
module CONTROL
        (input wire I,C,S,RST,
         output wire O);

wire L,IC;

assign O = ((O & RST) & ~S) | ((I ^ L) & S), IC = I ^ C;
assign L = ((L & S) | (IC & ~S) | (L & IC)) & RST;

endmodule
```

FIFO

```
module FIFO #(parameter N=32,M=4)(
        input [N-1:0] Di, input Ri,Ao,RST,
        output Ai,Ro,Xp,          output [N-1:0] Do);

wire [M+1:0] Ra;
wire [M-1:0] X;
wire [N-1:0] D [M:0];
wire X0n,RSTn;
genvar j;

generate for (j=0;j<=M-1;j++)
```

```
        begin: G1
                BCA BCAj
(Ra[j],Ra[j+2],RSTn,X[j],Ra[j+1]);
                LTCH #(N) LTCHj (D[j],X[j],RSTn,D[j+1]);
        end
endgenerate
CONTROL Cout(Ao,Ro,X[M-1],RSTn,Ra[M+1]),
Cin(Ri,Ai,X0n,RSTn,Ra[0]);
assign D[0]=Di, Ai=Ra[1], Ro=Ra[M], Do=D[M],
X0n=~X[0], Xp=X[M-1],RSTn=~RST;
endmodule
```

Flip

```
module Flip
        (input logic clk,rst,D,
        output logic Q);

always_ff @(posedge clk,posedge rst)
        if(rst)         Q<=0;
        else            Q<=D;
endmodule

module Flip2
        (input logic clk,rst,prst,D,
        output logic Q);

always_ff @(posedge clk,posedge rst,posedge prst)
        if(rst)                 Q<='0;
        else if(prst)   Q<='1;
        else                    Q<=D;
endmodule
```

LTCH

```
module LTCH #(parameter N=8)
        (input [N-1:0] D, input C,RST,
         output [N-1:0] Q);

wire Cn;
genvar i;

assign Cn=~C;
generate for (i=0; i<N; i++)
        begin: G1
                assign Q[i]=((Q[i] & C)|(D[i] & Cn)|(D[i] &
Q[i]))&RST;
        end
endgenerate
//Se aÒade lÛgica redundate para evitar oscilaciones (D
& Q).
//Digital system design with SystemVerilog, Mark
Zwolinski, pp.279-280
endmodule
```

Muller C

```
module MullerC
        (input wire I1,I2,RST,
         output wire  C);

         assign C = ~(~((I1 && I2) && RST) && (~((C &&
RST) && (I1 || I2))));
endmodule
```

Multiplexor_Var

```
/*nm es el n'mero de entradas, x el n'mero de bits para
representarlo*/
module Multiplexor_Var #(parameter x=3, nm=7)
        (input [nm-1:0] y,          input [x-1:0] m,
        output r);

/*wire [nm-1:0] ca [x-1:0];
wire [nm-1:0] yca [x:0];*/
wire [x-1:0] ca [nm-1:0];
wire [x:0] yca [nm-1:0];
wire [nm-1:0] co;
genvar i;

generate for(i=0;i<nm;i=i+1) begin
                negx #(x,i) EN (m,ca[i]);
                assign yca[i]={y[i],ca[i]};
                arbol #(x+1,2) AA (yca[i],co[i]);
        end
endgenerate
arbol #(nm,1) AO (co,r);
endmodule
```

negx

```
module negx #(parameter N=3, val=7)
        (input [N-1:0] ent,
        output [N-1:0] sal);

wire neg;
wire [N-1:0] resto;
genvar i;

function integer busca // constant function
        (input integer pos);
        begin
        integer pot,val2;
        for (pot=N-1,val2=val;pot>=pos;pot=pot-1)
                if(val2>=2**pot) begin
                        val2=val2-2**pot;
                        busca=1;
                end
                else
                        busca=0;
        end
endfunction

generate for(i=0;i<N;i=i+1) begin
                if(busca(i)==1)
                        assign sal[i]=ent[i];
                else
                        assign sal[i]=~ent[i];
        end
endgenerate
```

Recon

```
//NBITS2 debe ser enviado desde Tx como NBITS+1, y
desde Rx como NBITS+2
module Recon #(parameter NBITS2=8, TAM_BAUD=5)
        (input rst,TxRxn,          input [TAM_BAUD-1:0]
BAUD, input [clog(NBITS2)-1:0] fin,
        output Ak,ZBaud,          output [clog(NBITS2)-
1:0] y0,
        output
Akn,clk,Ai,Aim,Ain,zero,Rip,Ai_Ro,r_ret,zeron,Ctrlm1,ini_
ret,
        output [TAM_BAUD-1:0] yB);

//wire
Akn,clk,Ai,Aim,Ain,zero,Rip,Ai_Ro,r_ret,zeron,Ctrlm1,ini_
ret;

/*Se utiliza la parte donde despuÈs de terminar la
cuenta, se asegura
que no habr· un pulso de m·s, checar rp1 y rp2 para el
reset y preset*/
assign Aim= ~rst & Rip, Ctrlm1= TxRxn & zero;
mux Mux1 (Aim,Ain,Ctrlm1,Rip);

/*Viene la parte donde una vez detectada la transmisiÛn
se envìa y se
retarda con el tiempo definido por Ret(el bloque de
retardo similar a este)*/
BCA B1 (Rip,Ai_Ro,~rst,clk,Ai_Ro);
Flip F2(r_ret,rst,Ai_Ro,Ai);
not n1(Ain,Ai);

/*Esta parte revisar que haya habido un cambio que es
una peticiÛn de retardo*/
assign ini_ret= ~Ctrlm1;
Ret #(TAM_BAUD) Rtrd (rst,ini_ret,BAUD,r_ret,yB);

/*Este es el bloque contador, se usa Unos y el negx para
terminar*/
Cont_Var #(clog(NBITS2)) Cont (clk,rst,fin,Akn,y0);
//Se crea el arbol para sacar ZBaud
arbol #(clog(NBITS2-1)-1,1) Zr (y0[clog(NBITS2)-
1:1],ZBaudn);
or O1 (zeron,ZBaudn,y0[0]);
not n2(zero,zeron), n3(Ak,Akn), n4(ZBaud,ZBaudn);

function integer clog
(input integer num);
begin
        for (clog=1; num>=2**clog; clog=clog+1);
        end
endfunction
endmodule
```

Ret

```
module Ret #(parameter ry=3)
        (input rst,Ri,            input [ry-1:0] fin,
        output reset,             output [ry-1:0] y0);

wire Rip,Ai_Ro,Ain,zeron,zero,Aim;
//wire [ry-1:0] y0;

Cont_Var_Ret #(ry) Cont (x,rst,Ri,fin,reset,y0);
BCA B1 (Rip,Ai_Ro,~rst,x,Ai_Ro);
not #1ps n1(Ain,Ai_Ro);

mux m1 (Ain,Rip&~rst,Ri,Aim), m2 (Aim,Ain,zero,Rip);

arbol #(ry,1) Zr (y0,zeron);
not n2(zero,zeron);

endmodule
```

TOGGLE

```
module TOGGLE
        (input I,RST,
         output T1,T2);

wire X1,X2,X3,X4;

assign X1=I & (~T2), X3=~(I | (~T1)), X2=~(I & T2), X4=I
| T1;
MullerC MC1(X1,X2,RST,T1), MC2(X3,X4,RST,T2);

endmodule
```

Transmisor

```
module Transmisor #(parameter
NBITS=8,TAM_BAUD=5)
        (input rst,Trans,          input [TAM_BAUD-1:0]
BAUD,
        input [NBITS-1:0] Dtx,   output Tx,TXAk);

//wire TXAkn,zeron;
wire TXAkn,clk,Ai,Aim,Ain,zero,Rip,Ai_Ro,r_ret,zeron;
wire [clog(NBITS+1)-1:0] Unos,fin,y0;
wire [NBITS+1:0] Dtx1;
wire [TAM_BAUD-1:0] yB;

/*Se cuenta hasta NBITS+1, y se envìa con clog el
n·mero de bits
necesarios para representar NBITS+1*/
assign Unos=2**clog(NBITS+1)-1;
negx #(clog(NBITS+1),NBITS+1) fin_cont (Unos,fin);

/*Se utiliza la parte donde despuÈs de terminar la
cuenta, se asegura
que no habr· un pulso de m·s, checar rp1 y rp2 para el
reset y preset*/
mux Mux1 (Ain,Rip&~rst,Trans,Aim),  Mux2
(Aim,Ain,zero,Rip);

/*Viene la parte donde una vez detectada la transmisiÛn
se envìa y se
retarda con el tiempo definido por Ret(el bloque de
retardo similar a este)*/
BCA B1 (Rip,Ai_Ro,~rst,clk,Ai_Ro);
Flip F2(r_ret,rst,Ai_Ro,Ai);
not n3(Ain,Ai);
```

```
//Esta parte revisar que haya habido un cambio que es
una peticiÛn de retardo
Ret #(TAM_BAUD) Rtrd (rst,Trans,BAUD,r_ret,yB);

//Este es el bloque contador, se usa Unos y el negx para
terminar
Cont_Var #(clog(NBITS+1)) Cont (clk,rst,fin,TXAkn,y0);
arbol #(clog(NBITS+1),1) Zr (y0,zeron);
not n2(zero,zeron), n4(TXAk,TXAkn);

//Recon #(NBITS+1,TAM_BAUD) RTx
(rst,~Trans,BAUD,fin,TXAk,y0);

/*El multiplexor para sacar los datos que se transmitir·n
uno por uno*/
assign Dtx1={Dtx,'0,'1};
Multiplexor_Var #(clog(NBITS+2),NBITS+2) TRANS
(Dtx1,y0,Tx);

function integer clog
(input integer num);
begin
        for (clog=1; num>=2**clog; clog=clog+1);
        end
endfunction
endmodule
```

UART_FIFO

```
/*      --N=N'mero de bits a enviar, CM: N'mero hasta
el que se debe contar
        --x=n'mero bits necesarios para representar CM,
NBITS= tamaÒo trama de UART
        --*CM=((N-1)-(N-1)mod(NBITS))/NBITS en su
generic map
        --**x se obtiene de una b'squeda por generates
en quien lo instancia*/
module UART_FIFO #(parameter
N=32,CM=2,x=2,NBITS=8)
        (input rst,Ro,Au,xp,      input [N-1:0] D,
        output Ru,Ai,                   output[NBITS-
1:0] Dtx);

wire [CM:0] D2 [NBITS-1:0];
wire mux1,mux1n,mux2,Rop,Ai2,reset,inicio,RSTn;
wire [x-1:0] Cont,Unos,Fin_Cont;
wire [NBITS-1:0] Dp;
genvar j,k;
//-------------------------------------------------------------------------
-------
//Unos es la seÒal que se usar· despuÈs para detener el
contador
assign Unos=2**x-1, mux1n=~mux1;
//-------------------------------------------------------------------------
-------
//La seÒal para dejar pasar Ro es cuando Cont=0
arbol #(x,1) mx1 (Cont,mux1);
mux m1 (Ro,Rop,mux1n,Rop);
//-------------------------------------------------------------------------
-------
//La seÒal para dejar pasar Rop es cuando se activa
reset del contador
assign Ai2=Ai & ~rst, mux2=reset | Au | ~Ru;
mux m2 (Ai2,Rop,mux2,Ai);
//-------------------------------------------------------------------------
-------
//SeÒal que pide una nueva transmisiÛn a la UART
assign inicio= xp | mux1, Ru= (Au & xp) ^ inicio;
//-------------------------------------------------------------------------
-------
//El contador variable, definido por el n'mero de envÌos
necesarios dados N y NBITS
negx #(x,CM) FC (Unos,Fin_Cont);
Cont_Var #(x) c4 (Au,rst,Fin_Cont,reset,Cont);
//-------------------------------------------------------------------------
-------
//Se crean los multiplexores para la salida de bits final
generate for(j=0;j<NBITS;j=j+1) begin
                for(k=0;k<=CM;k=k+1) begin
                        if(j+k*NBITS>N-1)
                                assign D2[j][k]='b0;
                        else
                                assign
D2[j][k]=D[j+k*NBITS];
                end
                Multiplexor_Var #(x,CM+1) m3
(D2[j],Cont,Dp[j]);
        end
endgenerate
//-------------------------------------------------------------------------
-------
//Se utiliza un arreglo Flip Flop, ya utilizando el
Transmisor asÌncrono
generate for(j=0;j<NBITS;j=j+1) begin
        Flip F1 (Ru,rst,Dp[j],Dtx[j]);
        end
endgenerate

function integer clog
(input integer num);
begin
        for (clog=1; num>=2**clog; clog=clog+1);
        end
endfunction
endmodule
```

Referencias

[1] Jens Sparso, Steve Furber, "Principles of Asyncrhonous Circuit Design, A Systems Perspective", European Low-Power Initiative for Electronic System Design, Kluwer Academic Publishers, Boston/Dordretch/London.

[2] Liljeberg Pasi Tuominen Johanna, Isoaho Jouni, "*Self-Timed Approach for Reducing On-Chip Switching Noise*", Proceedings of the IFIP WG 10.5 International Conference on Very Large Scale Integration of System-on-Chip (IFIP VLSI-SoC 2003), Darmstadt, Germany, no. 0, pp. 19-24, ISBN 3-901882-17-0, Diciembre, 2003.

[3] Nanmoto T. Karaki N., Ebihara H., Utsunomiya S., Inoue S., Shimoda T., "*A flexible 8b asynchronous microprocessor based on low-temperature poly-silicon TFT technology*", IEEE International Solid-State Circuits Conference 2005, Digest of Technical Papers (ISSCC 2005), vol. 1, pp. 272-598, ISSN: 0193-6530, ISBN: 0-7803-8904-2, Febrero, 2005.

[4] Danielli P. Antognetti P., De Gloria A., Faraboschi P., Oliveri, M.,, "*A standard cell set for delay insensitive VLSI design*", Proceedings of Fifth Annual IEEE International ASIC Conference and Exhibit, Rochester, NY, pp. 123, ISBN: 0-7803-0768-2, Sep, 1992.

[5] Martin A. J. Prakash Piyush, "*Slack Matching Quasi Delay-Insensitive Circuits*", Proceedings of 12th IEEE International Symposium on Asynchronous Systems and Circuits (ASYNC 2006), pp. 204, ISBN: 0-7695-2498-2, Marzo, 2006.

[6] Nanya T. tasereekul N., "*Eliminating isochronic-fork constraints in quasi-delay-insensitive circuits*", Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC 2001), pp. 437, ISBN: 0-7803-6633-6, Enero, 2001.

[7] Kuwako M. Takamura A., Imai M., Fujii T. Ozawa, M., Fukasaku I., Ueno Y., Nanya T.,, "*TITAC-2: an asynchronous 32-bit microprocessor based on scalable-*

delay-insensitive model", Proceedings IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD 1997) ISBN: 0-8186-8206-X, Octubre, 1997.

[8] Josephs M.B. Bush M.E., "*Some limitations to speed-independence in asynchronous circuits*", Proceedings of Second International Symposium on Advanced Research in Asynchronous Circuits and Systems, Fukushima pp. 104-111, ISBN: 0-8186-7298-6, Marzo, 1996.

[9] Akella V. Werner T., "*Asynchronous Processor Survey*", IEEE Computer Society, vol. 30, no. 11, pp. 67-76, ISSN: 0018-9162, Noviembre, 1997.

[10] Raygoza J.J. y Boemo E. Ortega S., "Sincronización *Self-Timed*: protocolo de 4 fases", Computación ReconFigurable & FPGAs, Madrid, España, Septiembre, 2003.

[11] Jacobson Hans, "*Asynchronous Circuit Design a Case Study of a Framework Called Ack*", Lulea University of Technology, Master thesis, 1996.

[12] S.B. Furber, "*Asynchronous Logic*", In IberChip, Sao Paulo, Brazil, Febrero, 1996, URL: http://www.cs.man.ac.uk/amulet/publications/papers.html.

[13] Sutherland I. E., "*Micropipelines*", Communications of the ACM, vol. 32, no. 6, pp. 720-738, ISSN: 0001-0782, Junio, 1989.

[14] Barriga A. Acosta A.J., Bellido M.J., Juan J., Valencia M., "Temporización en circuitos integrados digitales CMOS", Ed. Marcombo, Cap. 3, ISBN-13: 9788426712462, ISBN: 8426712460, Febrero, 2000.

[15] Jianwei Liu, "*Arithmetic and Control Components for an Asynchronous System*", Universidad de Manchester , Ph.D Thesis, 1997.

[16] Tanner EDA Tools. Copyright 2012, Tanner EDA. All Rights Reserved. www.tannereda.com

[17] Richard G. Burford, Xingcha Fan and Neil W. Bergmann, "An 180 Mhz 16 bit Multiplier Using Asyncrhonous Logic Design Techniques", CSIRO/Flinders Joint Research Centre in Information Technology, Flinders University, Adelaide, Australia.

[18] M. V. Joshi, S. Gosavi, V. Jegadeesan, A. Basu, S. Jaiswal, W. K. Al-Assadi, and S. C. Smith, "NCL Implementation of Dual-Rail 2s Complement 8x8 Booth2 Multiplier using Static and Semi-Static Primitives", University of Missouri – Rolla, Department of Electrical and Computer Engineering, 1870 Miner Circle, Rolla, MO 65409.

[19] A. J. Martin. "Asynchronous datapaths and the design of an asynchronous adder" Formal Methods in System Design, 1(1): 119-137, July 1992.

[20] E. Brunvand, "Low Latency Self-Timed Flow Through FIFOs," in 16th Conference on Advanced Research in VLSI, UC Santa Cruz, March 1995.

[21] Altera Corporation, Single & Dual-Clock FIFO Mega-functions User Guide, June 2003. [Online]. Available: http://www.altera.com/literature/ug/ug-fifo.pdf

[22] T. Chelcea and S. M. Nowick, "Low-latency asynchronous FIFO's using token rings," in 6th International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC 2000). IEEE, apr 2000.

[23] H. Gao , Y. Yang , X. Ma and G. Dong "Analysis of the effect of LUT size on FPGA area and delay using theoretical derivations", Proc. Int. Symp. QED, pp.370 -374 2005

[24] H. Li, W. Mak, S. Katkoori, "Efficient LUT-Based FPGA Technology Mapping for Power Minimization,' In proc. Asia Pacific Design Automation Conference (ASP-DAC), pp. 353-358, January 2003.

[25] Z.-H. Wang, E.-C. Liu, J. Lai, and T.-C. Wang, "Power minimization in LUT-based FPGA technology mapping," In proc. Asia Pacific Design Automation Conference (ASP-DAC), pp. 635-640, January 2001.

[26] Kapian Maheswaran, "Implementing Self-Timed Circuits in Field Programmable Gate Arrays", Master Thesis, UC.Davis, 1995

[27] Cuong Pham-Quoc, Anh-Vu Dinh-Duc, "Hazard-free Muller Gates for Implementing Asynchronous Circuits on Xilinx FPGA,' in Proc of 2010 fifth IEEE International Symposium on Electronic Design, Test & Applications, 2010.

[28] S. Hauck, S. Burns, G. Borriello, and C. Ebeling, "An FPGA for Asynchronous Circuits", IEEE Design & Test of Computer, Vol. 11, No. 3, pp. 60-69 (1994).

[29] J. Lechner, M. Lampacher, and T. Polzer, "A robust asynchronous interfacing scheme with four-phase dual-rail coding," in Application of Concurrency to System Design, 2012. ACSD 2012. Seventh International Conference on, June 2012.

[30] J. Teifel and R. Manohar, "An asynchronous dataflow fpga architecture," Computers, IEEE Transactions on, vol. 53, no. 11, pp. 1376-1392, nov. 2004.

[31] HAUCK, S. Asynchronous Design Methodologies: An Overview. In Proceedings of the IEEE, Vol. 83, No. 1, pp. 69-93, January, 1995.

[32] T. Verhoeff, "Delay-insensitive codes an overview," Distributed Computing, vol. 3, no. 1, pp. 1-8, 1988.

[33] T.-Y. Wuu and S. B.K. Vrudhula "A Design of a Fast and Area Efficient Multi-Input Muller C-Element", IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 1, no. 2, 1993

[34] K. J. Cho , K. C. Lee , J. G. Chung and K. K. Parhi "Design of low-error fixed-width modified Booth multiplier", IEEE Trans. Very Large Scale (VLSI) Syst., vol. 12, no. 5, pp.522 -531 2004

[35] M. A. Song , L. D. Van and S. Y. Kuo "Adaptive low-error fixed- width Booth multipliers", IEICE Trans. Fundamentals, vol. E90-A, no. 6, pp.1180 -1187 2007

[36] D. Gajski. Princples of Digital, Design,Prentice Hall, 1997

[37] R. Sankar and V. Kadiyala, "Implementation of Static and Semi-Static Versions of a Bit-Wise Pipelined Dual-Rail NCL 2s Complement Multiplier," 2007 IEEE Region 5 Technical Conference, p.59 (2007).

[38] R. G. Burford, X. Fan, and N. W. Bergman, "An 180 MHz 16 bit multiplier using asynchronous logic design techniques", IEEE Custom Integrated Circuit Conf., pp.215 -218 1994

[39] Jiaoyan Chen, Popovici, E. ; Vasudevan, D. ; Schellekens, M., "Ultra Low Power Booth Multiplier Using Asynchronous Logic", Dept. of Electr. & Electron. Eng., Univ. Coll. Cork, Cork, Ireland. ISSN:1522-8681.

[40] J. D. Garside, S. B. Furber, and S.-H. Chung, "AMULET3 revealed", Proc. Async',99, pp.51 -59 1997 :IEEE Computer Society Press

[41] S. J. Jou, C. Y. Chen, E. C. Yang, and C. C. Su, "A pipelined multiplier-accumulator using high-speed, low-power static and dynamic full adder design", IEEE J. Solid-State Circuits, vol. 32, pp.114 -118 1997

[42] G. Yang , S.-O. Jung , K.-H. Baek , S. H. Kim , S. Kim and S.-M. Kang "A 32-bit carry lookahead adder using dual-path all-N logic", IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol. 13, no. 8, pp.992 -996 2005

[43] Yuke Wang et al, "Design and analysis of low-power 10-transistor full adders using novel XOR-XNOR gates", IEEE Trans. on Circuits and Systems II: Analog and Digital Signal Processing, Vol. 49, No 1, pp. 25-30, 2002.

[44] P. Balasubramanian and D.A. Edwards, "Dual-sum single-carry selftimed adder designs," accepted for IEEE ISVLSI, 2009.

[45] K.M. Fant and S.A. Brandt, "Null convention logic: a complete and consistent logic for asynchronous digital circuit synthesis," Proc. Intl. Conf. on ASAP, pp. 261-273, 1996.

[46] T. Y. Tang, C. S. Choy, J. Butas, and C. F. Chan, "An ALU design using a novel asynchronous pipeline architecture," in Proc. IEEE ISCAS, May 2000, pp. vol. 5, 361-364.

[47] Beom Seon Ryu, Jung Sok Yi ; Kie Young Lee ; Tae Won Cho, "A design of low power 16-b ALU" Sch. of Electron. & Electr. Eng., Chung-Buk Nat. Univ., Cheongju, South Korea.

[48] Implementación de circuitos self timed de 2 y 4 fases en FPGAs, Ortega Cisneros S., Raygoza Panduro J.J., Sutters G., Boemo E., III Jornadas de Computación Reconfigurable y FPGAS Jcra03, madrid 10 al 12 sept. 2003, ISBN 84-600-9928-8, 2003, Vol.1, Pag.407-414, Memorias de congresos

CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS DEL I.P.N.
UNIDAD GUADALAJARA

El Jurado designado por la Unidad Guadalajara del Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional aprobó la tesis

DISEÑO ASÍNCRONO EN DOBLE RIEL: Una metodología para la construcción de estructuras avanzadas

del (la) C.

Edgardo de Jesús PÉREZ CASAS

el día 11 de Diciembre de 2013.

Dr. Federico Sandoval Ibarra
Investigador CINVESTAV 3C
CINVESTAV Unidad Guadalajara

Dra. Susana Ortega Cisneros
Investigador CINVESTAV 2C
CINVESTAV Unidad Guadalajara

Dr. Juan José Raygoza Panduro
Profesor Investigador
Universidad de Guadalajara