UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Tesis de Máster

Diseño e Implementación de Operaciones Aritméticas en Punto Flotante Decimal según el Estándar IEEE 754-2008

Carlos Eduardo Minchola Guardia

Octubre 2010

Diseño e Implementación de Operaciones Aritméticas en Punto Flotante Decimal según el Estándar IEEE 754-2008

Autor: D. Carlos Minchola Guardia
Tutor: D. Gustavo Sutter

Escuela Politécnica Superior
Universidad Autónoma de Madrid

Octubre 2008

Agradecimientos

Quiero agradecer a mis padres, a pesar de estar separados siempre me alientan para seguir adelante, a mis hermanos que son el impulso que necesito, a Esteban mi primer sobrino y al amor de mi vida Esperanza.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1 INTRODUCCION

1.1 Motivación

Los ordenadores representan, almacenan y manipulan datos numéricos en formato binario. Muchas aplicaciones comerciales como análisis financiero, transacciones bancarias, cálculo de tasas, y operaciones contables son realizadas mediante operaciones aritméticas binarias lo cual introduce un determinado error de precisión al convertir un número decimal a binario y viceversa. Por este motivo la aritmética decimal surge como alternativa para contrarrestar la pérdida de precisión.

La mayor parte de procesadores de propósito general no proveen instrucciones o soporte hardware (*HW*) para aritmética de punto flotante decimal (*DFP*) [7]. Como consecuencia de esto se utilizan dos alternativas para mitigar la eventual pérdida de precisión: a) los números decimales, son leídos, convertidos a números binarios y procesados utilizando aritmética de punto flotante binario, aceptando la consecuente pérdida de precisión. b) se utilizan diferentes librerías software que realizan el cálculo sin pérdida. La primera estrategia obtiene los mejores resultados en velocidad, siendo la opción utilizada por la mayor parte de las aplicaciones, en tanto la segunda es utilizada en casos más críticos, aunque con penalizaciones en tiempo respecto a la estrategia anterior. La demanda y el crecimiento de aplicaciones en aritmética decimal durante los últimos años proponen diferentes técnicas para contrarrestar el problema de precisión, por eso existen soluciones que provienen de ambos ejes: hardware y software. En lo referente al software, existen lenguajes de programación, incluyendo *COBOL* [15], *XML* [29], *Visual Basic* [17], *Java* [23], *C* [22] que proveen soporte para aritmética decimal. Las soluciones hardware aparecieron en los primeros ordenadores digitales de las décadas de los 50`s y 60`s como *ENIAC* y *UNIVAC* [10]. Sin embargo, recientemente se encuentran arquitecturas como *CADAC* [4], *IBM's Z9000* [3] o *IBM Power6* [7] que implementan a nivel hardware ciertas operaciones en aritmética decimal.

Debido a la importancia de la aritmética decimal, surgió una revisión borrador (*IEEE 754r*) del estándar para aritmética en punto flotante. Diferentes especificaciones incorporadas al *IEEE 754r* han originado el nuevo estándar *IEEE 754-2008* [1]. Una operación fundamental en aritmética *DFP* es la multiplicación debido al amplio rango de aplicaciones que posee, por eso en los últimos años varios diseños de multiplicación decimal en punto fijo y flotante han sido propuestos con diferentes resultados manteniendo un compromiso entre parámetros como latencia y área. Por eso estudiar y proponer novedosas alternativas de multiplicaciones en formato *DFP* resulta atractivo para encontrar adecuados compromisos de diseño.

El trabajo consiste en diseñar e implementar las operaciones aritméticas de suma y multiplicación considerando los parámetros de velocidad, área utilizada dentro de la *FPGA*; además el circuito ha sido sintetizado y simulado sobre la *FPGA Virtex-5 (xcv25vtx240t)* de la familia *Xilinx* [34]. La propuesta para ambas operaciones se basó en implementaciones en

bloques segmentados (pipeline) de modo que podamos optimizar la performance del sistema. Este trabajo proporciona análisis, métrica, aplicabilidad y metodología de diseño para el desarrollo de algoritmos que implementen las operaciones en aritmética decimal de suma, resta y multiplicación sobre hardware reconfigurable (*FPGA´s*) en términos de área y velocidad.

1.2 Objetivos

El trabajo se puede dividir en dos objetivos fundamentales:

- Estudiar y comparar diferentes alternativas de diseño de multiplicadores y sumadores *DFP* implementados sobre tecnología *FPGA´s* y/o *ASIC*'s en base a la lectura de publicaciones recientes. Seleccionar algunas alternativas de diseño hardware en base al mejor performance de las publicaciones estudiadas, para esto debemos considerar y verificar parámetros como frecuencia, latencia, área, memoria, ciclos, eficiencia etc.
- Implementar un diseño en tecnologías *FPGA's* de un sumador y multiplicador *DFP*, conseguir un diseño hardware de considerable performance y un adecuado *trade-off* eficiencia/delay. Comparar nuestros resultados con los diseños estudiados para concluir que ventajas, desventajas y mejoras pueden ser consideradas para trabajos futuros.

La estructura del resto del trabajo es como sigue:

El *Capítulo 2* presenta un enfoque general sobre números decimales en punto flotante representados en el estándar *IEEE 754-2008*. Adicionalmente introduciremos los esquemas propuestos para la operación suma, resta y multiplicación *DFP*. Finalmente mencionaremos algunas aplicaciones de aritmética decimal sobre sistemas embebidos en un chip *(SoC)* y sobre hardware reconfigurable.

El *Capítulo 3* explica en detalle las etapas que forman la operación suma e intenta encontrar un óptimo compromiso de diseño. Se muestran otras alternativas de diseño que son comparadas con el diseño propuesto y discutidas posteriormente. Se muestran los resultados de la implementación hardware.

El *Capítulo 4* describe la operación de multiplicación y muestra comparaciones con otras alternativas. La propuesta de diseño es explicada detalladamente. Se muestran los resultados de la implementación hardware.

En el *Capítulo 5* muestra la implementación de una aplicación basada en una operación matemática de alta precisión como la tarifación telefónica. El objetivo es medir la performance de la aplicación mencionada mediante la instanciación ilimitada de los módulos diseñados. La aplicación es implementada en Software *(SW)* y en un *Soft Core Processor (SCP)* sobre hardware reconfigurable *(FPGA).* Discutimos sobre el compromiso Software/Hardware *(SW / HW).*

En el Capítulo 6 se presentan conclusiones, consideraciones de mejoras para nuestro diseño y una propuesta de trabajos futuros.

2 ESTADO DEL ARTE

En este capítulo informaremos sobre el estándar para números en punto flotante *IEEE 754-2008*, como son representados y operados en procesos de suma y multiplicación. Presentamos algunos términos que usaremos en el resto del trabajo, y enfocaremos de manera general las operaciones de suma y multiplicación *DFP*.

2.1 Números en punto fijo y punto flotante

Los números que un computador entiende son generalmente representados en notación binaria. Un número binario de 64 bits puede ser representado aproximadamente con un número entero de 19 dígitos decimales. El redondeo de un número real puede ser representado en punto fijo pero existen diferentes aplicaciones en que los usuarios necesitan precisión y valores máximos, por eso el punto fijo requiere mayor cantidad de bits para satisfacer lo explicado. La figura 2.1 muestra lo explicado anteriormente.

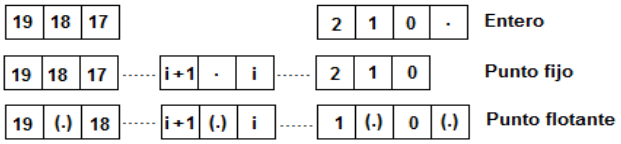

Figura 2.1 Diferencias entre número entero, punto fijo y flotante

Una mejor solución es reservar un conjunto de bits, *E*, para codificar una posición dinámica i^{th} observada en la figura 2.1. En 64 bits podemos usar 6 bits para codificar la posición de cada bit dejando espacio para casi 17 dígitos decimales, *D*, como se observa en la figura 2.2. En notación matemática es equivalente a $D.2^{E}$.

6 bit (encode)	17.16.15.14.13.12.11.10.9.8.7.6.5.4.3.2.1.0

Figura 2.2 Codificación de la posición dinámica del punto decimal

Desafortunadamente esta representación es corta para números en notación científica que van desde 2^{-58} hasta 2^{58}-1 (aproximado a 10^{17}) e inadecuado para cálculos científicos. Observando la notación matemática podemos solucionar el problema aumentando más bits a la posición *E* y considerando signo. De esta manera el rango dinámico puede ser incrementado generando mayor exactitud en representaciones numéricas. Lo explicado justifica el uso de números en punto flotante y los estándares actuales muestran un buen *trade-off* entre precisión y exponente.

2.2 Estándar IEEE 754 para números en punto flotante decimal

En 1985, el *IEEE* (*Institute of Electrical and Electronics Engineers*) publicó la norma *IEEE 754*. Una especificación relativa a la precisión y formato de los números de punto flotante. Incluye una lista de las operaciones que pueden realizarse con dichos números, entre las que se encuentran las cuatro básicas: suma, resta, multiplicación, división. Así como el resto, la raíz

cuadrada y diversas conversiones. También incluye el tratamiento de circunstancias excepcionales, como manejo de números infinitos y valores no numéricos. El estándar se desarrolló para facilitar la portabilidad de los programas de un procesador a otro y para alentar el desarrollo de programas numéricos sofisticados. Una versión actual es el estándar *IEEE 754-2008* que fue publicado en Agosto del 2008.

El estándar define lo siguiente:

- Formatos Aritméticos: conjunto de datos binarios o decimales en punto flotante, los cuales consisten de números finitos (ceros y números subnormales), infinitos y valores especiales de *NaNs (Not a Number).*
- Formatos intercambiables: los números punto flotante pueden ser intercambiados y compactados en formato binario y decimal de diferentes tamaños de bits.
- Algoritmos de redondeo: varios métodos pueden ser usados por los números durante las operaciones.
- Operaciones: permiten operaciones con formatos aritméticos.
- Excepciones: indica si hay condiciones de excepción como *overflow*, *underflow*, división por cero, etc.

El nuevo estándar *IEEE 754-2008* específica formatos para representar números binarios en punto flotante (*BFP*) y decimal en punto flotante (*DFP*). La diferencia entre ambas proviene no solo de la base utilizada sino también de la normalización de las mantisas, el *BFP* se normaliza con el punto decimal a la derecha del bit más significativo (*MSB*) mientras el *DFP* solo se representa por decimales. El *DFP* es representado en formatos intercambiables de 32, 64 y 128 bits. Un número *DFP* es representado por un signo, un exponente *E* que es normalizada a un exponente desplazado (*biased exponent*) *q*, y un significando o mantisa *M* con *p* dígitos de precisión. El valor de un número *DFP D* es representado como:

$$D=-1^{S} \times M \times 10^{q}, \qquad q=E - bias \qquad (1)$$

El número *D* es expresado en formato *DFP* como se muestra en el esquema

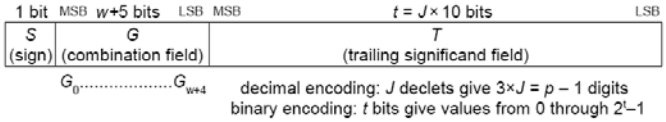

Figura 2.3 Representación de un número en el estándar IEEE 754-2008

La codificación contiene *K* bits (*K* múltiplo de 32) y presenta 3 campos:

- El campo Signo formado por 1 bit.
- El campo combinado *G* formado por w+5, en el que se encuentra implicado el cálculo del exponente q como el dígito más significante (*MSD*) de la mantisa *M*. Permite también la verificación de excepciones.

- El campo *T* formado por *J×10* bits y contienen los *p-1* dígitos codificados de la mantisa *M*. La codificación consiste en *Densely Packed Decimal* (*DPD*) [1] que comprime 3 dígitos decimales (12 bits) en 10 bits. Un *Declet* está representado por los 10 bits anteriores, lo que significa que *J* es el número de *Declets* que dispone el campo *T*. La relación entre la cantidad de dígitos decimales y *J* esta dado por *p=3×J+1*.

Si trabajamos con *decimal64* entonces las variables involucradas en el formato *k, p, t, w,* y *bias* poseen los siguientes valores 16, 50, 12, y 398 respectivamente. En la Tabla 2.1 podemos observar los valores para los diferentes formatos:

Parameter	decimal32	decimal64	decimal128	decimal{*k*} ($k \geq 32$)
k, almacenamieto de bits	32	64	128	multiple of 32
p, precisión en dígitos	7	16	34	$9 \times k/32 - 2$
emax	96	384	6144	$3 \times 2^{(k/16+3)}$
Encoding parameters				
bias, $E - q$	101	398	6176	$emax + p - 2$
signo de 1 bit	1	1	1	1
w+5, campo combinado en bits	11	13	17	$k/16 + 9$
t, representación mantisa en DPD en bits	20	50	110	$15 \times k/16 - 10$
k, almacenamieto de bits	32	64	128	$1 + 5 + w + t$

Tabla 2.1 Tabla sobre los formatos intercambiables soportados estándar IEEE 754-2008

Por ejemplo podríamos tener un *decimal256* con parámetros *p*=70 y *emax*=1572864.

Si bien el formato *decimal32* no pertenece al estándar, lo usaremos por simplicidad para explicar en detalle el almacenamiento del número.
Un número flotante *decimal32* se almacena en una palabra de 32 bits, el primer bit es el bit de signo (*S*), los siguientes 8 son los bits del exponente (*G*) y los restantes son la mantisa (*T*):

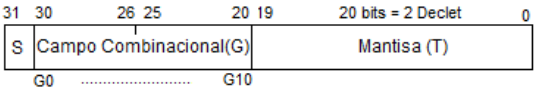

Figura 2.4 Representación de un número Decimal32

Al estar normalizado, es decir expresado en formato punto flotante decimal (*DFP*), se observan las posiciones de los bits que posee cada campo:

- El signo del número está representado en el bit 31.
- El campo combinacional desde el bit 26 al 30, posee 5 bits.
- El campo exponente dispone de 6 bits, empezando por el bit 25 y finaliza en bit 20.
- El campo mantisa dispone de 20 bits, empezando por el bit 19 terminando en el bit 0. Contienen 20 bits lo que representa 2 *Declets*, esto fue explicado anteriormente.

A continuación especificaremos los posibles casos especiales *(CS)* existentes en un número *Decimal32* (*D*). Los *CS* mencionado serán utilizados en el resto del diseño.

- Si G_0 .. G_4*=11111*, entonces *D* es *NaN* (*Not a Number*) sin considerar *S*. Además si G_5 *=1* entonces *D* es *sNaN* caso contrario es *qNaN*. El resto de los bits son ignorados.
- Si G_0 .. G_4*=11110*, entonces $D=-1^S x(+infinito)$ y los bits restantes no son considerados.

Para números finitos, *D* es considerado como (*S, E–bias, C*) y $D=-1^S xMx10^{E-bias}$ donde *M* contiene el número de *Declets* que al ser descomprimidos representan los 6 dígitos menos significativos de la mantisa, esto quiere decir que de los 7 dígitos que posee el *Decimal32* $(d_0d_1d_2d_3d_4d_5d_6)$ solo los 6 últimos los representa los *Declets* y el d_0 proviene del campo combinacional.

Los 6 bits menos significativos del campo *G* (G_5.. G_{10}) forman el exponente. El *MSD* de la mantisa (d_0) es encontrado como sigue:

- Si los 5 bits *MSB* de *G* son *110xx* o *1110x* entonces d_0 está determinado como $8+G_4$.
- Si los 5 bits *MSB* de *G* son *0xxxx* o *10xxx*, entonces el d_0 está determinado como $4G_2+2G_3+G_4$.

Se describe las siguientes definiciones utilizadas en los diferentes modelos de redondeo decimal:

- ***Dígito de Guarda (GD):*** *es* el primer dígito luego de los *p* dígitos de precisión establecido por el formato *DFP* a utilizar. Por ejemplo si trabajamos con *Decimal64*, sabemos que este formato posee una precisión de 16 dígitos. Si el resultado a codificar cuenta con más de *p* dígitos normalizados (sin ceros principales) entonces se seleccionan los *p* dígitos *MSD* del resultado y el primer dígito posterior a los *p* dígitos mencionados es el *GD*.
- ***Dígito de Redondeo (RD):*** es el segundo dígito de los 16 dígitos mencionados en la definición anterior.
- ***Sticky-bit (SB):*** formando por el procesamiento de los bits restantes de lo explicado en las dos definiciones anteriores, el procesamiento consta de una operación *OR* de los bits mencionados.

Cabe señalar que si la precisión es exacta, o sea si nuestro resultado final posee 16 dígitos normalizados, entonces *GD, RD y SB* son nulos.

Mencionaremos también algunas excepciones que son consideradas en los diseños propuestos:

- Operación inválida: $\infty \pm \infty$, $0 \times \infty$, $0 \div 0$, $\infty \div \infty$, x mod 0, $\sqrt{x}$ cuando $x < 0$, $x=\infty$
- Inexacto: el resultado redondeado no coincide con el real
- *Overflow y underflow*.
- División por cero.

En el trabajo y como nomenclatura nombraremos las mantisas como *MAX*, *MBX* con sus respetivos exponentes *EAX*, *EBX*, también existen nombres como *EX*, *GDX*, *RDX* y nombres adicionales con terminación en un dígito *X*. La letra *X* es un número que denota la entrada y la salida de diferentes módulos. Encontramos también símbolos como "$(N)_Z{}^T$" que se refiere T^{th} bit de el dígito Z^{th} en un número *N*, donde el menos significativo bit/dígito tiene el índice 0. Por ejemplo, $(A1)_2{}^5$ es el quinto bit del segundo dígito *BCD* de un numero *A1*. *A1*puede representar entrada a un módulo y *A2* es la salida del mismo.

2.3 Operación Suma/Resta en formato punto flotante decimal

En este apartado mostramos el esquema general del diseño propuesto para la suma *DFP*. Se utiliza el formato *Decimal64* considerando *p*=16 dígitos *BCD*. La extensión al formato *Decimal128* es directo. Para realizar la suma/resta de dos operandos en representación punto flotante debemos realizar previamente la separación de los exponentes y de las mantisas para su tratamiento posterior, se realizan los siguientes pasos:

- Normalización de las mantisas.
- Seleccionar el número con menor exponente y desplazar su mantisa a la derecha tantas veces como indique la diferencia en módulo de los exponentes.
- Hacer que el exponente resultado sea igual al mayor de los exponentes.
- Realización de las operaciones de suma o resta en punto fijo con las mantisas.

Considerando como base lo mencionado anteriormente, se detalla la estrategia que se utilizo para el sumador/restador considerando la performance del sumador propuesto en [27]. La estrategia consiste de los siguientes módulos: decodificador (*decoder IEEE 754-2008*), detección de ceros principales (*leading zero deteccion*, *LZD*), intercambio de operandos (*swapping*), desplazamiento a la derecha (s*hifting right amount*, *SRA*), paralelamente se procesan las etapas generación de señales previas (*pre-signal generation*) y sumador punto fijo *BCD* de 16 dígitos (*16-BCD adder*), corrección posterior (*post-correction*), redondeo (*rounding*) y finalmente la etapa codificador (*coder IEEE 754-2008*). El sumador empieza capturando dos operandos *(OPA, OPB)* en formato *Decimal64 IEEE 754-2008* y la operación seleccionada *(OP)*, la etapa decodificación transforma los operandos en sus respectivos signos *(SA, SB)*, exponentes *(EA0, EB0)* y mantisas *(MA0, MB0).* La etapa de decodificación también detecta los casos especiales *(CS).* Las etapas son detalladas en los capítulos siguientes. En la etapa de redondeo intervienen algunos parámetros como dígito de guarda, dígito de redondeo y sticky-bit que fueron detallados anteriormente.

Los signos y la operación generan la operación efectiva a realizar *(EOP)*, esta señal es calculada como:

$$EOP = SA \text{ xor } SB \text{ xor } OP \qquad (2)$$

Debemos indicar que la operación suma/resta es controlada por *EOP*, cuando *EOP = 0* la suma es procesada caso contrario ejecutamos la resta. Luego la etapa *LZD* normaliza las mantisas

(MA0, MB0) y los exponentes *(EA0, EB0)* eliminando los ceros iniciales y actualizando las señales *MA0, MB0, EA0 y EB0*. Este proceso genera nuevas mantisas *(MA1, MB1)* y exponentes *(EA1, EB1).* La etapa *swapping* intercambia las mantisas y sus respectivos exponentes solo si *EA1 < EB1*, esta etapa produce como salidas: mantisas actualizadas *MA2* (posee el mayor de los dos exponentes), *MB2* (posee el menor de los exponentes), un exponente *E2 = max (EA1, EB1),* una señal *SWAP* para indicar la presencia de un proceso *swapping*.

Para realizar la suma decimal debemos alinear ambas mantisas de acuerdo al punto decimal, para esto analizamos los exponentes: recordemos primeramente que trabajamos con mantisas normalizadas *MA2, MB2* y con el exponente *E2*, ahora debemos desplazar *MB2* a la derecha de *MA2* una cantidad de dígitos que es definido por la diferencia absoluta *Ed=|EA1-EB1|* y designado como *RSA (shifting right amount)*. Lo explicado garantiza la alineación de las mantisas para la suma correcta. La siguiente etapa (*shifting right*) realiza el desplazamiento a la derecha de *MB2* una cantidad de *RSA* dígitos generando *MB3*. La cantidad *RSA* debe estar limitado a un rango de valores de *0* a *p+2* (recordemos que *p* representa los dígitos de precisión), el límite superior de *RSA* posee dos dígitos adicionales para poder predecir y computar el *GD* y *RD* que son utilizados en la etapa de redondeo. Aparte de generar el *GD* y *RD*, registramos un dígito adicional para el caso de suma con *overflow* por tal razón *MB3* contiene 19 dígitos. La etapa *shifting right* también genera una señal que predice el correcto *sticky-bit* a utilizar, esta señal se denomina *predicted sticky-bit (PSB).*

Luego el módulo *pre-signal generation* captura como entrada *MA2, MB3, EOP* y *PSB* y a partir de ellas genera las siguientes señales: el acarreo de inicio *(CIN)* del sumador *BCD* decimal, la señal *AGTB* que verifica si *MA2>MB3* que es muy usada cuando la operación es resta (*EOP=1*), el dígito de guarda inicial *(GD1),* el dígito de redondeo inicial *(RD1)* y un dígito extra *(ED)* usado en casos de sumas con *overflow*. Paralelamente el *16-BCD Adder* procesa los sumados *MA2* y los 16 *MSD* de *MB3*, la operación *EOP*, el acarreo *CIN* y la señal *AGTB.* El *BCD Adder* genera una suma parcial de 16 dígitos con su respectivo acarreo de salida *(COUT).*

La etapa *post-correction* produce los valores finales para los dígitos de guarda y redondeo *(GD2, RD2)* y el *sticky-bit* final *(FSB),* la suma parcial y el exponente *E2* son también corregidos (designados como *S2 y E3* respectivamente) y listos para ser procesados por la etapa de redondeo. La etapa *rounding* captura las salidas *(FSB, GD2, RD2, S2, E3)* de la etapa previa y ejecuta el redondeo y produce la suma y el exponente final. Mencionemos que el signo final es determinado por la siguiente operación:

$$FS = (SA \text{ and not}(EOP))\text{or}(EOP \text{ and } (AGTB \text{ xnor}(SA \text{ xor } SWAP))) \quad (3)$$

Finalmente el módulo codificación recibe las señales finales del signo, exponente y suma transformándolos a la suma final expresada en formato *IEEE 754-2008*. Esta etapa también maneja los casos especiales. El algoritmo siguiente muestra un ejemplo general de lo explicado.

Algoritmo 1 Ejemplo para una suma Decimal Punto Flotante IEEE 754-2008

MA0[27 .. 0] = 0005678, EA0=-5, SA=0

MB0[27 .. 0] = 0000478, EB0=-7, SB=1

OP=1

EOP=SA xor SB xor OP=0

Etapa Leading zero detection

LZDA=LZD(MA0)=3 y EA1=EA0-3=-8

LZDB=LZD(MB0)=4 y EB1=EB0-4=-11

MA1=5678000

MB1=4780000

Verificar swapping

Como EA1=-8>EB1=-11 entonces SWAP=0

ED=|EA1-EB1|=3 y E2=max(EA1-EB1)=-8

MA2=MA1

MB2=MB1

Etapa Shifting right

SRA=ED=3, MB3=0004780 & 0 & 0 & 0

Etapa suma (EOP=0)

MB2=5 6 7 8 0 0 0 +

MB3=0 0 0 4 7 8 0 0 0 0

S1=5 6 8 2 7 8 0 0 0 0

Generación señales

*GD=S1[11 .. 8]=*0

*RD=S1[7 .. 4]=*0

Sticky-bit = (S1[3] or S1[2] or S1[1] or S1[0])=0

Redondeo

Analizamos los parámetros GR, RD y Sticky-bit y aplicamos el modelo de redondeo seleccionado.

Resultado

Suma final[27 .. 0]=5682780, Exponente fina=E2=-8, Signo final=0

En la figura 2.5 representamos en forma general el esquema del sumador propuesto.

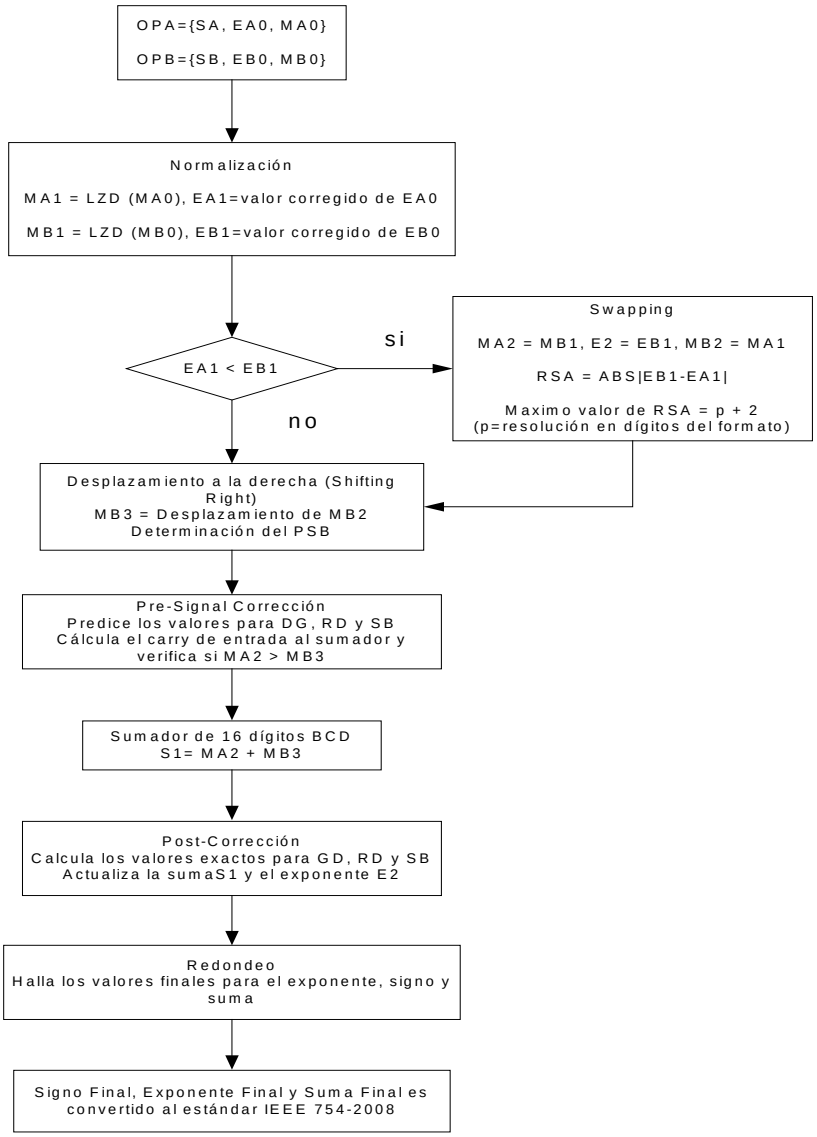

Figura 2.5 Esquema general de una suma/resta DFP

2.4 Operación multiplicación decimal en formato punto flotante decima

Apartado que revisa el esquema general propuesto para un multiplicador *DFP* bajo el estándar *IEEE 754-2008*, algunos módulos también son usados en el sumador. El algoritmo de la multiplicación está fundamentado en los siguientes pasos:

- Normalizar las mantisas, si es necesario.
- Realizar la suma de los exponentes.
- Multiplicar las mantisas y determinar el signo del resultado.
- Redondeo.

En base a estos puntos explicaremos el esquema adoptado. La estrategia usada considera parte del modelo propuesto en [12]. Los módulos que forman la operación de multiplicación son: decodificador (*decoder IEEE 754-2008*), detección de ceros principales (*leading zero deteccion,*

LZD), multiplicación punto fijo *BCD* de 16 dígitos (*16-BCD multiplier*), redondeo (r*ounding*) y finalmente la etapa codificador (*coder IEEE 754-2008*).

La lectura de publicaciones [8] [19] [12] nos permitió seleccionar diseños de óptimo performance y es la base del esquema propuesto. El multiplicador empieza capturando dos operandos *(OPA, OPB)* expresado en formato *decimal64 IEEE 754-2008* y luego descompone ambos números en sus respectivos signos *(SA, SB), exponentes (EA, EB)* y mantisas *(MA, MB).* Recordemos que el tamaño en bits de los dos últimos depende si trabajamos con *decimal64 o decimal128*. La normalización de los operando remueve los ceros iniciales, este proceso es llevado a cabo por la etapa *LZD* generando las nuevas mantisas *MA1, MB1* y actualizando los exponentes *EA1 y EA2*. EL módulo *LZD* garantiza también normalización en el resultado de la multiplicación final. Luego se desarrolla una multiplicación *BCD* de 16 dígitos y resulta el producto *IP (IP=MA1xMA2,* conocido como producto intermedio*)* con una dimensión de 32 dígitos *BCD*, desde *IP* debemos producir una expresión de 16 dígitos para poder operar en el estándar *decimal64*. En la etapa de redondeo se capturan los 16 dígitos *MSD* de *IP*, como dígito de guarda considera el dígito siguiente al *LSD* del producto intermedio *IP*, como digito de redondeo toma el dígito siguiente a *GD* y como *sticky-bit* se realiza una operación *OR* con todos los bits restantes. Luego se procesa redondeo seleccionado, esto produce la multiplicación correcta *P* que junto al signo final (*SF=SA xor SB*) y al exponente final *(E=EA1+EA2)* representan las señales necesarias para ser codificadas al formato *IEEE 754-2008*. Las señales *SF, E*, y *P* codificadas generan el resultado final de la multiplicación *DFP IEEE 754-2008.* Los casos especiales son detectados tanto en la etapa de decodificación como codificación, y son realizados paralelamente a la multiplicación.

En la figura 2.6 representamos en forma general el esquema propuesto.

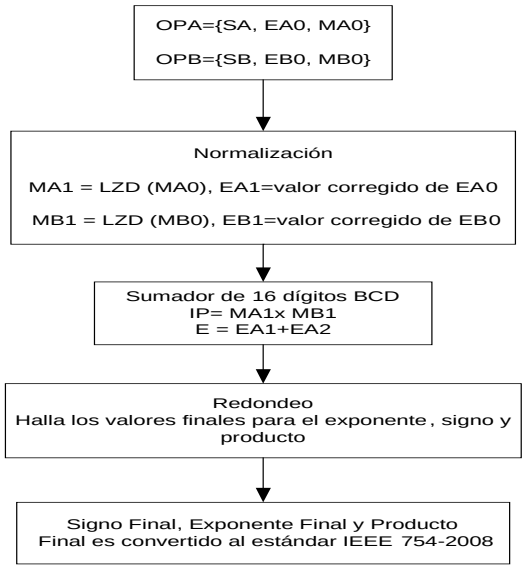

Figura 2.6 Esquema general de una multiplicación DFP

2.5 Aplicaciones de Sistemas Embebidos on-Chip sobre FPGA

Los sistemas embebidos *on-chip (SoC)* consisten en diseños de complejidad media o alta que se integran totalmente en un solo circuito integrado o chip, integran parte digital y analógica.

El uso de *FPGA* para la implementación de *SoC* cada vez es más demandante, por su velocidad de procesamiento y flexibilidad. En este sentido, en el mercado se ofertan diferentes tarjetas de evaluación y desarrollo, las cuales incluyen una diversidad de periféricos para aumentar su versatilidad. La flexibilidad de las *FPGA* ha favorecido la implementación de sistemas complejos utilizando componentes en solución hardware y software (*HW/SW*).

Como aplicaciones de sistemas *SoC* sobre *FPGA* se menciona: diseño de una unidad aritmética decimal sobre *FPGA*; *networks on-chip* (*NOC*) implementadas sobre *FPGA* como en [11]; aceleración para procesamiento de gráficos 3D basado en *SoC* sobre *FPGA* [13]; aceleración para cálculos matemáticos como la *DWT* (*direct wavelet transform*) mediante *SoC* en *FPGA* [2]; evaluación de operaciones en punto flotante en un *soft-processor Rockwell´s 8bit core 6502* sobre *FPGA*, los resultados obtenidos son comparados con los procesadores *8051* y *PIC's* confirmando la mayor flexibilidad del *FPGA/soft-processor* [16]; otra aplicación desarrolla la *transformada directa coseno* en el *soft-core processor Microblaze*, mostrando la performance de una solución software contra una solución *HW/SW*, se confirma que la Interface *FSL* del *Microblaze* mejora la performance reemplazando tiempos críticos por hardware [21].

2.6 Aplicaciones de multiplicaciones y sumadores sobre FPGA

En las publicaciones estudiadas notamos que la mayoría de diseños en aritmética punto flotante decimal se basan en tecnología *ASIC's* que en tecnología *FPGA's*. La publicación [9] es una de las primeras soluciones de suma en punto flotante decimal sobre *FPGA's*. Podemos argumentar que la ventaja de los *ASIC's* radica en el diseño de circuitos específicos para aplicaciones fijas y por lo tanto presentan un rendimiento superior para tareas computacionales. Una desventaja de los *ASIC´s* es la carencia de reprogramabilidad. Las modernas *FPGA's* poseen una ventaja sobre los *ASICs*: la flexibilidad para implementar diferentes funciones en el mismo dispositivo.

Se puede mencionar otras aplicaciones sobre *FPGA´s* como en la publicación [28] que propone diseños de sumadores/restadores decimales basados en *LUT´s* sobre *Virtex-5*; en [18] se presenta un multiplicador en *decimal64 IEEE 754-2008* desarrollado sobre *Virtex-4*; en [24] se propone alternativas de multiplicadores decimales (combinacionales y secuenciales) con aceptable *trade-off* área/latencia; en [20] se presenta un multiplicador *decimal64 IEEE 754-2008* en arquitectura paralela como pipeline así como sus respectivas comparaciones de *trade-off* área/latencia, la verificación es realizada sobre una *NIOS II* en Altera *Cyclone II FPGA.*

3 DISEÑO DE LA SUMA/RESTA PUNTO FLOTANTE DECIMAL

En el capítulo anterior se explicó de forma general la suma. Ahora detallaremos el circuito que realiza esta operación con números decimales en punto flotante. El circuito considera los casos especiales, diferentes técnicas de redondeo y manejo de excepciones. Para mayor aceleración consideramos ocho etapas de pipeline como mostramos en la figura 3.1. Los números, en formato *decimal64*, se representan como cadenas de bits. En la tabla 3.1 se aprecia la cantidad de bits correspondiente a cada campo perteneciente al formato estudiado. La columna decimal *N* es genérica y aplicado para cualquier formato. Nótese que *e_lim+m_lim=N-6.*

	Decimal32	***Decimal64***	***Decimall28***	***DecimalN***
Signo	1	1	1	1
campo combinacional	5	5	5	*5*
continuación exponente	6	8	12	*e_lim*
continuación mantisa	20	50	110	*m_lim*

Tabla 3.1 Cantidad de bits de los campos que forman un número *DFP IEEE 754-2008*

La siguiente tabla 3.2 muestra la posición de los bits involucrados en los campos que componen la representación decimal.

	Decimal 32	***Decimal 64***	***Decimal 128***	***Decimal N***
Signo	*31*	*63*	*127*	*N*-1
campo combinacional	*30 .. 26*	*62 .. 58*	*126 .. 122*	*N*-2..*N*-6
continuación exponente	*25..20*	*57..50*	*121..110*	*N*-7..*m_lim* ó *N*-7..*N*-6-*e_lim*
continuación mantisa	*19..0*	*49..0*	*109..0*	*m_lim*-1..0 ó *N*-7-*e_lim*..0

Tabla 3.2 Posición de los bits en los campos que forman un número *DFP IEEE 754-2008*

La figura 3.1 muestra el circuito propuesto para la suma/resta *DFP*. Los bloques sombreados representan los módulos principales del sistema, las líneas punteadas representan las 8 etapas de pipeline del circuito propuesto.

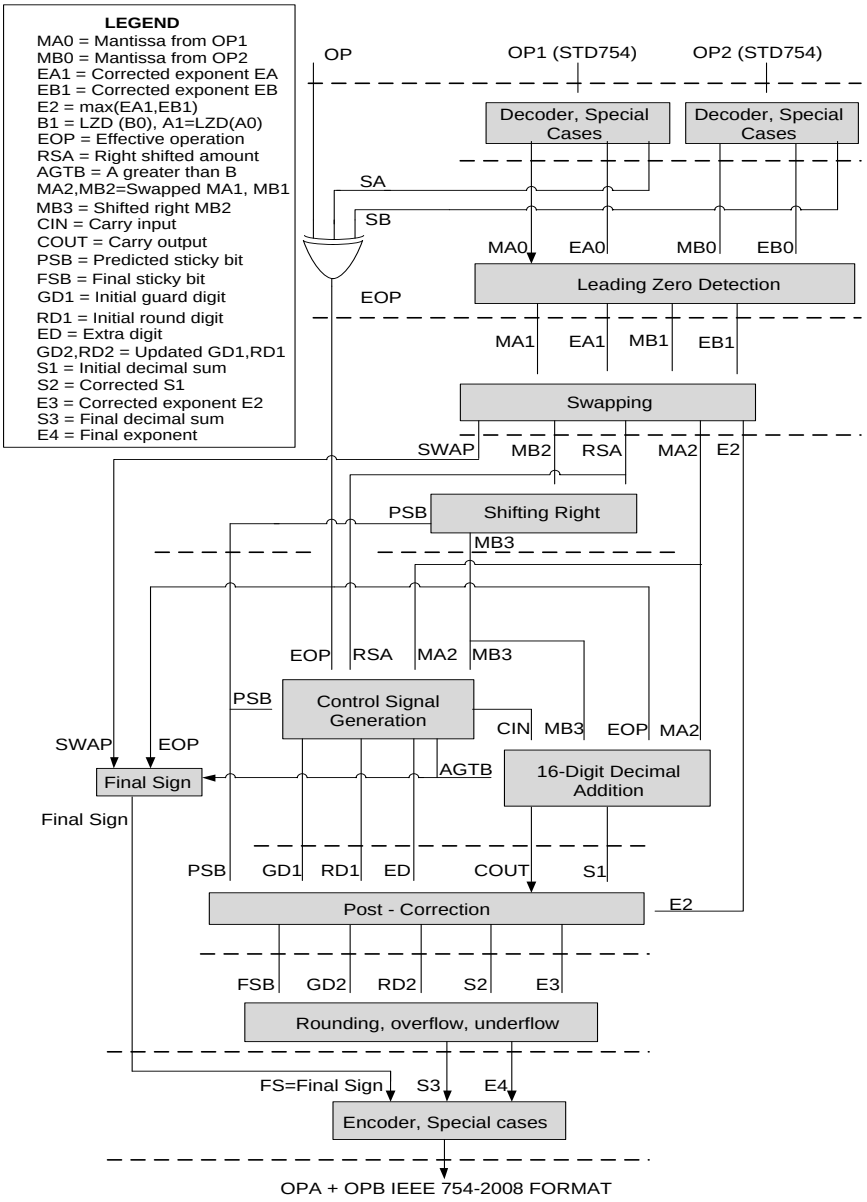

Figura 3.1 Circuito propuesto para la suma *DFP*

A continuación se detalla las etapas involucradas en el diseño, para una mejor explicación consideraremos las señales como vectores de bits, el modelo es basado en *decimal64* y la cantidad de dígitos serán designados como *p* y el número de bits como *N=4.p*. Los vectores serán representadas casi siempre como *V* [*N-1* .. 0], la multiplicación es representada por (.) y [*m* .. *n*] indica *m to*/*down to n* bits. Los comentarios presentes en algunos algoritmos serán precedidos por el símbolo -- como en los editores de lenguaje *HDL*.

3.1 Módulo Decoder IEEE 754-2008

El circuito procesa la decodificación de un número punto flotante decimal representado según el estándar *IEEE 754-2008*. Decodifica el número en signo, mantisa y exponente. Un número en *decimal64* es descompuesto en 1 bit de signo, 10 bits para el exponente, y 64 bits para representar los 16 dígitos *BCD*. Detallamos las entradas y salidas a este módulo.

Entradas

OPx [*N*-1 .. 0], representa al operando *OPA* y *OPB* (*x=A o B*) en formato *decimal64.*

Salidas

Sx, signo de un bit. Es 1 cuando *OPx* es negativo y 0 cuando es positivo.

Ex0 [*e_lim*+1 .. 0], exponente expresado como vector de (*e_lim*+2) bits.

Mx0 [*N*-1 .. 0], mantisa expresada como vector de *p* dígitos decimales, cada dígito codificado en *BCD* (4 bits).

qNaN, un bit. Es 1 cuando *OPx* es *qNaN* y 0 cuando no lo es.

sNaN, un bit. Es 1 cuando *OPx* es *sNaN* y 0 cuando no lo es.

Inf, un bit. Es 1 cuando *OPx* es ±∞ y 0 cuando no lo es.

En la figura 3.2 podemos observar el diagrama de flujo del decodificador y la generación de los casos especiales. Observamos un procedimiento de decodificación *DPD*, como se explico anteriormente, este proceso transforma los *Declets* que posee el operando en los *p*-1 dígitos *BCD*. El dígito *MSD* proviene del campo combinacional del operando *DFP*, lo cual es mostrado en el diagrama (figura 3.2).

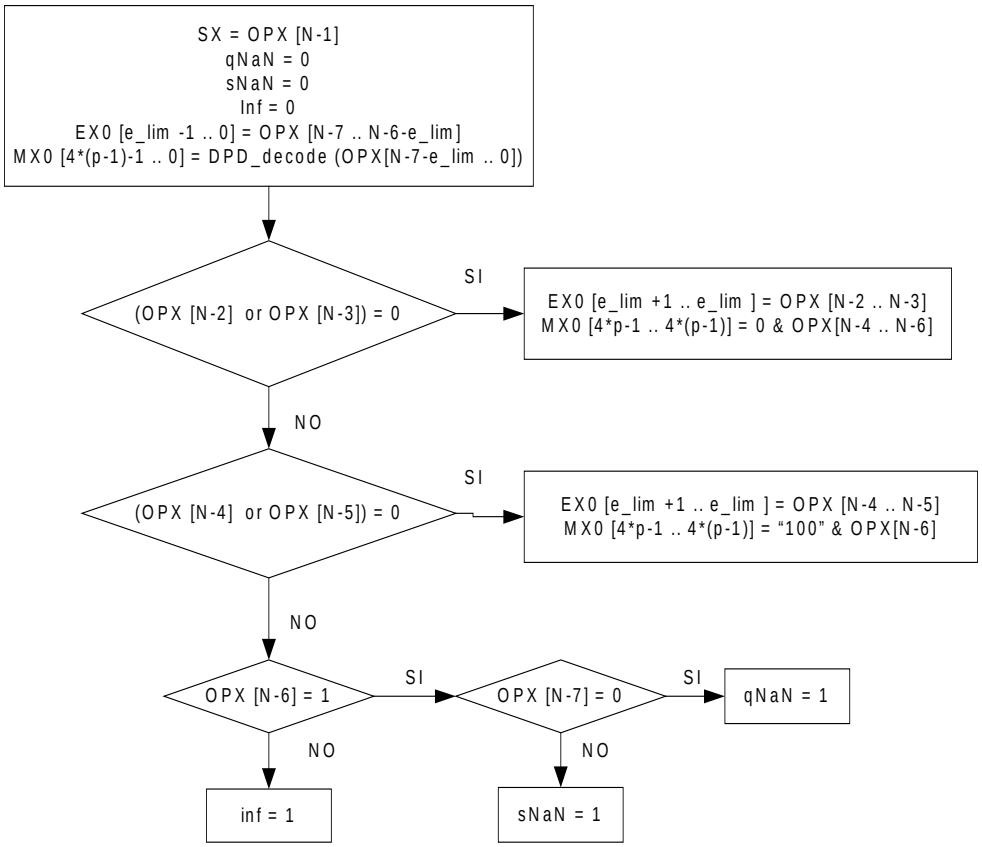

Figura 3.2 Diagrama de flujo que representa la decodificación y los casos especiales

3.2 Módulo Codificación

El circuito realiza la codificación de una mantisa, un exponente y su respectivo signo generando un operando en formato *IEEE 754-2008*. En la figura 3.3 vemos el diagrama de flujo utilizado, el procedimiento de codificación *DPD* arma los *Declets* del campo continuación mantisa a partir de los *p*-1 dígitos *MSD* de la mantisa. Se detallan las entradas y salidas a este módulo.

Entradas

SF, un bit. Es 1 cuando *R* es negativo y 0 cuando es positivo.

EF[e_*lim*+1 .. 0], exponente expresado como vector de (*q_lim*+2) bits. El exponente se encuentra en cero desplazado.

MF[4.*p*-1 .. 0], mantisa expresada como vector de *p* (precisión) dígitos decimales, cada dígito codificado en 4 bits.

qNaN_F, un bit. Es 1 cuando *R* es *qNaN* y 0 cuando no lo es.

sNaN_F, un bit. Es 1 cuando *R* es *sNaN* y 0 cuando no lo es.

inf_F, un bit. Es 1 cuando *R* es ±∞ y 0 cuando no lo es.

Salida

RF[*N*-1..0], operando decimal en formato *IEEE 754-2008.*

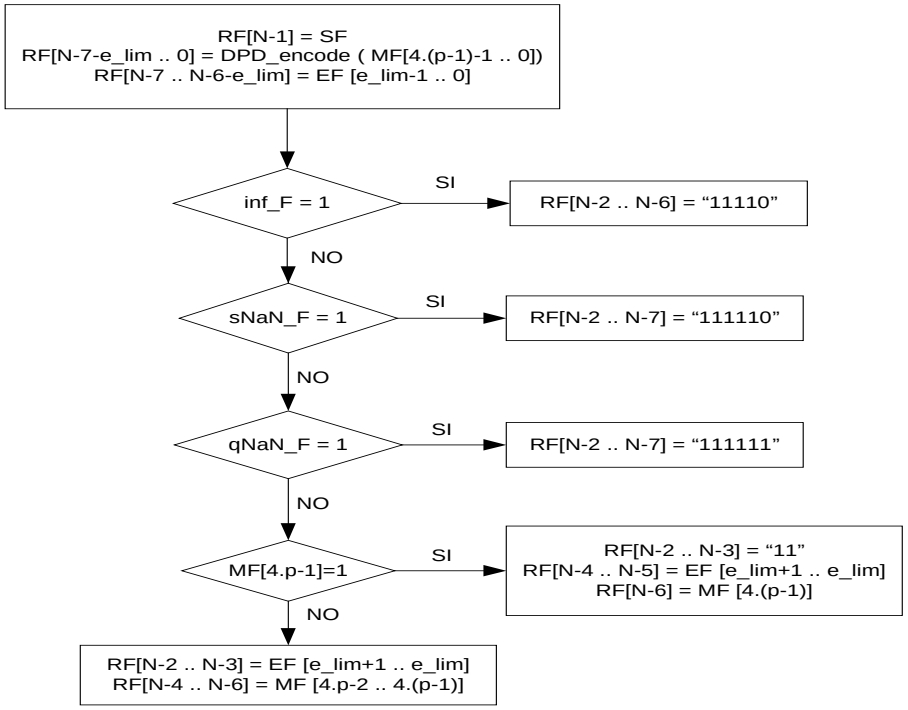

Figura 3.3 Diagrama que representa la codificación y los casos especiales

3.3 Módulo de Alineamiento: Detección de ceros principales (LZD) y Swapping

Etapa que captura los datos de la etapa de decodificación *(MA0, EA0, MB0 y EB0).* Lo siguiente es la normalización de las mantisas y exponentes, lo cual es realizado por el proceso *LZD* cuya función es eliminar los ceros principales. La cantidad de ceros principales son representadas por *LZDA* y *LZDB,* ambos son generados para cada mantisa *MA0* y *MB0*. Las salidas normalizadas *MA1* y *MB1* son obtenidas asignando las entradas desplazadas, esto es mostrado en la siguiente relación:

$$MA1(4.p\text{-}1 \,..\, 4.LZDA) = MA0\,(4.p\text{-}1\text{-}4.LZDA \,..\, 0) \qquad (4)$$

$$MB1(4.p\text{-}1 \,..\, 4.LZDB) = MB0(4.p\text{-}1\text{-}4.LZDB \,..\, 0)$$

Antes del proceso *LZD* y para mejorar la performance del diseño se detecta los dígitos que son diferentes de ceros de ambas mantisas *MA0* y *MB0*; solo es necesario 1 bit para verificar lo anterior. Las mantisas *MA0* y *MB0* son transformadas en una palabra de 16 bits donde cada bit indica si el dígito es diferente de cero, la posición tanto de los dígitos como de los bits se mantiene. La transformación se consigue realizando una operación *OR* con los bits de cada dígito. Esto se puede visualizar en la figura 3.4.

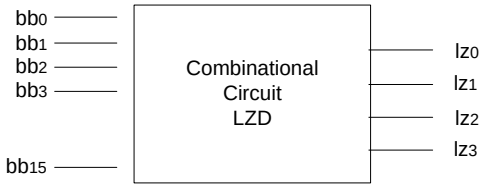

Figura 3.4 Bloque combinacional *Leading Zero Detection*

LZDA y *LZDB* son palabras de 4 bits para representar los 16 desplazamientos que pueden existir en un formato de 16 dígitos de precisión como el *dicemal64*. La palabra de 16 bits *"bb"* es generada como:

$$bb(i) = (MX0)_i^{3} \text{ or } (MX0)_i^{2} \text{ or } (MX0)_i^{1} \text{ or } (MX0)_i^{0} \qquad (5)$$

La transformación anterior es usada continuamente en el trabajo por eso a la función la denominaremos *Dig_to_bits*, y podría usarse como *bb = Dig_to_bits*(*MX0*). La señal *bb* consta de 16 bits y es la entrada al proceso *LZD* obteniendo como resultado una señal en representación binaria de 4 bits que indica la cantidad de ceros principales. La entrada al bloque *LZD* (figura 3.4) es la señal *bb*[15 .. 0] y la salida es un señal *offset*[3 .. 0] que indica la cantidad de ceros principales en las mantisas *MA0* y *MB0*. La señal *offset* representada a las señales *LZDA* y *LZDB* mencionadas anteriormente.

Procedemos con la actualización de los exponentes: *EA1=EA–LZDA* y *EB1=EB–LZDB*. Luego que las mantisas y exponentes están normalizados, podemos empezar con el cómputo de la siguiente etapa. El proceso suma o resta depende de la operación seleccionada y de los signos de los operandos *(SA, SB)*, a esto se conoce como operación efectiva *EOP* (la suma es seleccionada

si *EOP*=0, caso contrario es resta). Para simplicidad del análisis se considera que las mantisas en las operaciones aritméticas respetan el orden *MA1+/-MB1* y además que *EA1>=EB1*, siempre se debe sumar o restar la mantisa de mayor exponente con la de menor exponente. Si *EA1<EB1* entonces la etapa *swapping* es llevada a cabo generando una señal *SWAP* para identificar el proceso (si se produce *swapping* entonces *SWAP=1*, caso contrario *SWAP=0*). Los dos procesos anteriores (*LZD* y *swapping*) producen nuevas mantisas como *MA2* (con su respectivo exponente igual al *max*(*EA1, EB1*)) y *MB2* (con exponente igual al *min*(*EA1, EB1*)). También se considera como salida el nuevo exponente *E2* que pertenece a la mantisa *MA2*. La suma/resta requiere la alineación de las mantisas, este proceso depende de la diferencia absoluta de los exponentes *Ed=|EA1-EB1|*. Por lo tanto se debe desplazar *MB2* una cantidad de dígitos a la derecha de *MA2*, la cantidad a desplazar se conoce como *RSA* (*right shifting amount*) y su cálculo es como sigue:

$$\begin{array}{llll} If & (Ed<=p\_max) & RSA=Ed & (6) \\ else & & RSA=p\_max & \end{array}$$

Donde *p_max*=18 dígitos, como trabajamos con *MB2* de 16 dígitos entonces *RSA* tendría un valor máximo de desplazamiento de 16, pero para simplificar el cálculo de la etapa de *rounding* entonces consideramos 2 dígitos más que nos ayudará a calcular los dígitos de guarda y redondeo [30]. Por eso el *RSA* es limitado a *p_max*=18. La figura 3.5 muestra el bloque para el proceso de alineamiento, la línea sombreada indica el camino crítico de la etapa.

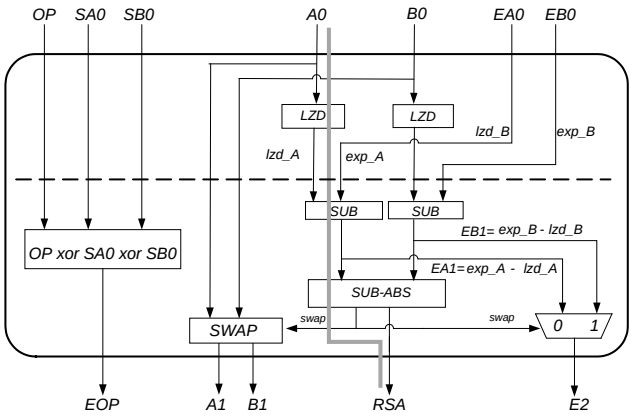

Figura 3.5 Etapa de Alineamiento: *Leading zero detection* y *swapping*

3.4 Módulo desplazamiento a la derecha (Shifting Right)

Etapa que realiza el desplazamiento a la derecha de *MB2* y produce una señal de 2 bits, que genera los *sticky-bits* iniciales, llamado *PSB* (*predicted sticky-bit*).

MB2 es desplazado *SRA* dígitos a la derecha; con esto las mantisas están alineadas y listas para iniciar la operación suma/resta, el exponente a usar sería el *E2* que fue explicado anteriormente.

La señal desplazada es llamada *MB3* y almacenada en una palabra de 19 dígitos. Inicialmente usamos un palabra de 18 dígitos ya que el máximo valor de *RSA* es 18; pero debido a que puede ocurrir un cierto estado de *overflow* en el módulo suma/resta cuando ésta presente un acarreo de salida (*COUT*), lo que obliga a considerar *COUT* como el nuevo dígito más significativo (*MSD*) de la suma final y descartar el dígito menos significativo (*LSD*) originando modificación de las entradas en el módulo de *rounding*, se dispone de 1 dígito más llamado dígito extra inicial (*IED*) razón por la cual la señal desplazada se almacena en un registro de 19 dígitos.

Para el desplazamiento usamos una versión nueva para *RS*A llamada *RSA_new* que es calculada de acuerdo al diagrama de la figura 3.6. En el mismo diagrama observamos la formación de la señal *PSB*. Para acelerar el diseño usamos la función *Dig_to_bit* explicado en la etapa anterior, para verificar que dígitos de *MB2* son distintos de cero. La función genera *MB2_bits* = *Díg_to_bits*(*MB2*) y es la entrada al proceso que calcula el *RSA_new* y *PSB*.

Las entradas y salidas a este módulo son:

Entradas

MB2_bits[15 .. 0], señal de 16 bits que indica que dígitos de *MB2* son diferentes a cero.

RSA[4 .. 0], cantidad de dígitos a desplazar a la derecha.

Salidas

RSA_new[4 .. 0], desplazamiento usado bajo condiciones explicadas posteriormente.

PSB[1 .. 0], señal cuyos bits son usados para generar futuros *sticky-bits.*

La figura 3.6 muestra el algoritmo para generar las señales de salida. En la figura anterior se observa la función *conv_int* que se encarga de convertir un vector a un número entero.

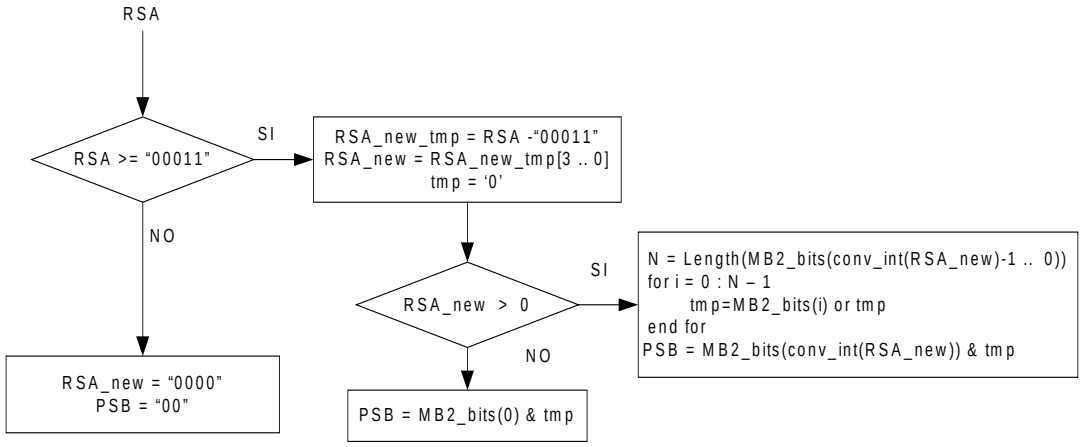

Figura 3.6 Generación de las señales *RSA_new* y *Predicted Sticky-bit*

Al obtener las señales de salida, el desplazamiento se lleva cabo mediante la siguiente asignación:

Algoritmo 2

if RSA>="00011" then

MB3(4.p-1–4.conv_int(RSA_new) .. 0)=MB2(4.p-1 .. 4.conv_int(RSA_new))

else

MB3(4.(p+3)-1-4.conv_int(RSA) .. 12–4.conv_int(RSA) =MB2

end if

En la figura 3.7 se aprecia las diferentes formas de desplazamiento de *MB2* y las posiciones donde son almacenadas en el vector *MB3*, recordemos que *MB3* está alineado con *MA2*. Se observa también los dígitos *GR*, *RD* y *IED* que pronostican los dígitos de guarda, de redondeo y el dígito adicional. La señal *RSA_new* es calculada como *RSA*-3.

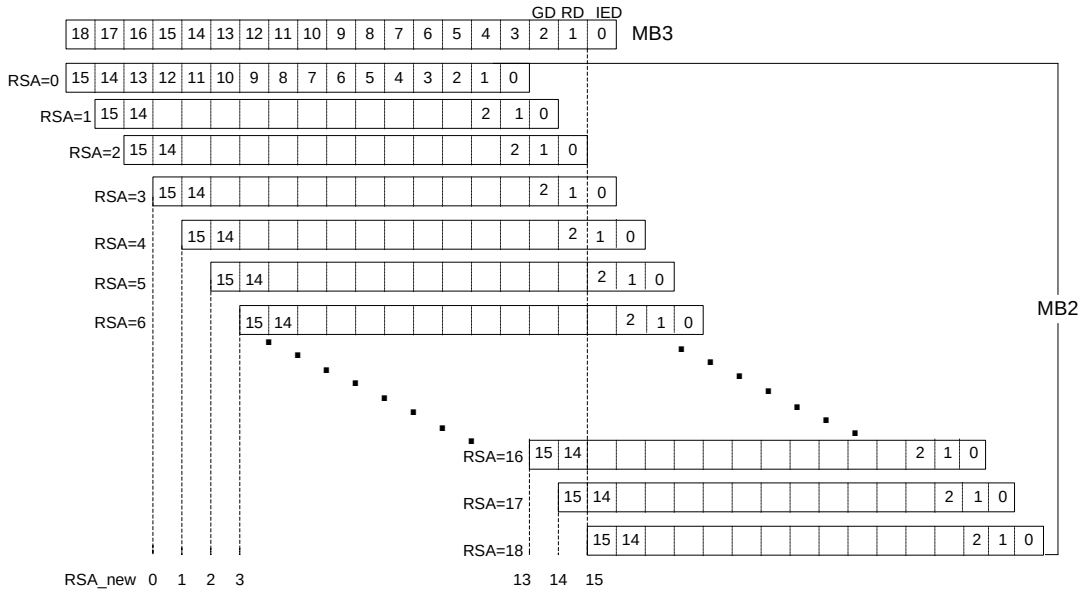

Figura 3.7 Posibles desplazamientos y almacenamientos de *MB2* en *MB3*

RSA	PREDICTED STICKY BIT=PSB (2 BITS)
0	'0' & '0'
1	'0' & '0'
2	'0' & '0'
3	$((MB2)_0 >0)$ & '0'
4	$((MB2)_1 >0)$ & $((MB2)_0 >0)$
5	$((MB2)_2 >0)$ & $(((MB2)_1>0) \text{ or } ((MB2)_0 >0))$
.	. .
.	. .
.	. .
15	$((MB2)_{12}>0)$ & $(((MB2)_{11}>0) \text{ or } ((MB2)_{10}>0) \text{ or } ((MB2)_9 >0) \ldots \text{ or } ((MB2)_0>0))$
16	$((MB2)_{13}>0)$ & $(((MB2)_{12}>0) \text{ or } ((MB2)_{11}>0) \text{ or } ((MB2)_{10}>0) \ldots \text{ or } ((MB2)_0>0))$
17	$((MB2)_{14}>0)$ & $(((MB2)_{13}>0) \text{ or } ((MB2)_{12}>0) \text{ or } ((MB2)_{11}>0) \ldots \text{ or } ((MB2)_0>0))$
18	$((MB2)_{15}>0)$ & $(((MB2)_{14}>0) \text{ or } ((MB2)_{13}>0) \text{ or } ((MB2)_{12}>0) \ldots \text{ or } ((MB2)_0>0))$

PSB (1) & PSB (0)

Figura 3.8 Generación de la señal *predicted sticky-bit* (*PSB*)

La figura 3.7 muestra la forma de calcular el *RSA_new* y la figura 3.8 la generación del *PSB*, observamos que la señal *RSA* es esencial para el procesamiento. Cuando *RSA<3* entonces la mantisa es completamente ubicada dentro de *MB3* respetando la posición como se ve en la figura 3.7, bajo está condición la señal de 2 bits *PSB* es igual a *'0'&'0'*. Para simplificar el cálculo de *PSB* generamos una nueva señal de desplazamiento llamada *RSA_new* conforme a lo observado en el diagrama de flujo de la figura 3.6, ahora los desplazamientos lo determina *RSA_new*. EL máximo desplazamiento de *RSA_new* ubica el *MSD* de *MB2* en la *LSD* de *MB3*, se adopta esté límite pues el *LSD* de *MB3* se usa como un dígito extra (*IED*) para algunas

condiciones de *overflow* pero realmente no forma parte de la operación suma/resta. Se observa también que el primer y segundo dígito representan los futuros dígito de guarda (*GD)* y de redondeo (*RD*) lo cual será usado para predecir los valores finales. Cuando *RSA*>=3 (es equivalente a *RSA_new*>=0, observado en el diagrama de la figura 3.6) se produce una señal de 2 bits *PSB* que es igual a la concatenación de 1 bit que verifica si $(MB3)_0$>0 y otro bit que proviene de una operación *OR* de todos los bits que no son almacenados en *MB3* y que pertenecen a *MB2*. La señal *PSB=PSB(1)&PSB(0),* donde *PSB(1)* es ($(MB3)_0$>0) y *PSB(0)* es como se explicó anteriormente. Predecimos dos señales *sticky-bit* mediante una combinación de *PSB(1)* y *PSB(0),* y son generadas como:

Primera señal Sticky-bit (SB1)=a la operación PSB(1) or PSB(0) *(7)*

Segunda señal Sticky-bit (SB2)=PSB(0)

Se muestran los algoritmos para producir la señal *PSB* como las señales *SB1* y *SB2*.

Algoritmo 3 Generación de la señal PSB

```
        --SB1=PSB(1) or PSB(0), SB2=PSB(0)
If      RSA > 4 then
        PSB=((MB2)RSA_new >0)&SB2
        --when SB1=((MB2)RSA_new > 0) or SB2
else if RSA = 4 then
        PSB=(MB2)1&SB2
        --when SB1=(MB2)1 or SB2
else if RSA=3 then
        PSB=(MB2)0&'0'
        --when SB1=(MB2)0
else
        PSB='0'&'0
        --when SB1='0'
end if
```

Algoritmo 4 Generación de las señales SB1 (PSB(1) or PSB(0)) y SB2 (PSB(0))

if RSA> 4 then

SB1=0 si todos los dígitos $\overleftarrow{(MB2)_{RSA-3}(MB2)_{RSA-4}.....(MB2)_0}$ *(RSA-3 Dígitos) son iguales a 0, e igual a 1 si cualquier dígito es diferente de 0*

SB2=0 si todos los dígitos $\overleftarrow{(MB2)_{RSA-4}.....(MB2)_0}$ *(RSA-4 Dígitos) son iguales a 0, igual a 1 si cualquier dígito es diferente de 0*

else if RSA = 4 then

SB1=0 si los dígitos $(MB2)_1(MB2)_0$ son iguales a 0, igual a 1 si cualquier dígito es diferente de 0

SB2=0 if $(MB2)_0$ es igual a 0, e igual a 1 en otro caso

else if RSA = 3 then

SB1=0 if $(B2)_0$ es 0, e igual a 1 en otro caso

SB2=0

else

SB1=0, SB2=0

end if

En la figura 3.8 presenta el circuito propuesto para el cálculo de *PSB,* notar que *conv_int* es una función que transforma un vector a un número entero.

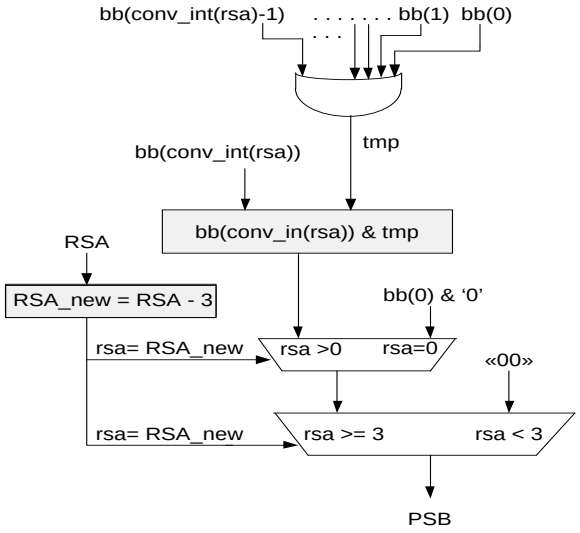

Figura 3.9 Circuito para generar la señal *Predicted Sticky-bit* (*PSB*)

Los módulos *swapping* y *shifting right* realizan el alineamiento de ambas mantisas, considerando dígitos adicionales para procesos posteriores. En la figura 3.10 observamos el resultado de esta etapa.

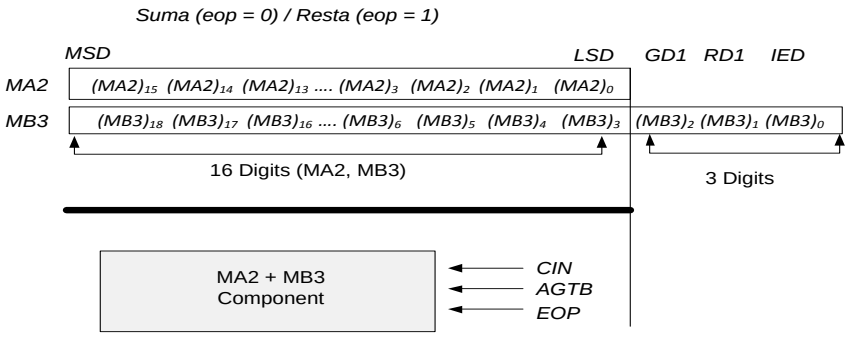

Figura 3.10 Ejemplo del alineamiento de las mantisas a sumar.

3.5 Módulo Pre-Signal Generation

Después de procesar la etapa previa, esta etapa intenta predefinir señales que serán utilizadas en los módulos suma/resta y *post-correction*. El módulo se basa en un circuito netamente combinacional y es descrito como:

Entradas

MA2[4.*p*-1 .. 0], mantisa de 16 dígitos, directamente a la operación suma/resta.

MB3[4.*(p+3)*-1 .. 0], versión desplazada de *MB2*, para la suma/resta solo consideramos los 16 *MSD*, los 3 últimos dígitos son considerados en el cálculo del dígito de guarda, del dígito de redondeo y para el dígito extra respectivamente.

EOP, operación efectiva.

PSB[1 .. 0], predice las señales *sticky-bit* que pueden ocurrir en el proceso.

Salidas

CIN, representa el acarreo de entrada a la suma/resta.

AGTB, verifica si *MA2*<*MA3*[*4.(p+3)-1 .. 12*], señal muy utilizada cuando *EOP*=1.

GD1[3 .. 0], valor predefinido para *DG*.

RD1[3 .. 0], valor predefinido para *RG*.

IED[3 .. 0], valor predefinido para *ED*.

La operación suma/resta opera solo con 16 dígitos por tal motivo los 3 últimos dígitos *LSD* de *MB3* son procesados para generar señales como el acarreo de entrada a la suma/resta y la señal *AGTB* (utilizado en la resta) que verifica si *MA2* es mayor que *MB3*[4.(*p*+3) .. 12]. Se menciona que la señal *MB3*[11 .. 8], *MB3*[7 .. 4] y *MB3*[3 .. 0] son considerados como los valores iniciales para predecir la señal *GD*, *RD* y *IED*, estas señales serán corregidas bajo ciertas condiciones que se explican posteriormente. En paralelo a este módulo se lleva a cabo la etapa de suma/resta que será explicada en el siguiente apartado. El objetivo de esta etapa es calcular las señales *CIN*, *AGTB*, y los valores corregidos para dígito de guarda (*GD1*), dígito de redondeo (*RD1*) y el dígito extra (*ED*).

El algoritmo utilizado para esta etapa es:

Algorithm 5 Generación del GD1, RD1, ED, AGTB, CIN.

Entradas*: B3, MA2,* $IED=(MB3)_0$*, PSB*

EOP = 0:

$GD1=(MB3)_2$

$RD1=(MB3)_1$

AGTB=0

$ED=IED=(MB3)_0$

-- Donde el Sticky-bit=PSB(0) or PSB(1)

EOP=1:

if RSA>0 then

CIN=not(PSB(1)or PSB(0)or((MB3)$_1$ >0)or((MB3)$_2$ >0)

AGTB=1

if (PSB(1) or PSB(0))=1 then

GD1=9–(MB3)$_2$

RD1=9–(MB3)$_1$

else if (MB3)$_1$=0 then RD1=0

if (MB3)$_2$=0 then GD1=0

else GD1=10-(MB3)$_2$

end if

else GD1=9-(MB3)$_2$

RD1=10-(MB3)$_1$

end if

if (IED=0) then

ED=PSB(0)&'0'&'0'&PSB(0)

else ED=9-IED+not(PSB(0))

end if

else -- RSA=0

if (MA2>MB3) then

CIN=not(PSB(1) or ((MB3)$_2$>0)or ((MB3)$_1$>0)))

AGTB=1

else CIN=PSB(1) or ((MB3)$_2$>0) or ((MB3)$_1$>0))

AGTB=0

end if

GD1=0, RD1=0, ED=0

end if

3.6 Operación suma/resta decimal de 16 dígitos BCD

Este módulo realiza una suma/resta decimal de 16 dígitos, el diseño es basado en números *BCD´s* de complemento 10, y el acarreo de salida es llevada a cabo utilizando la tecnología *carry-chain*. El módulo está compuesto por las siguientes entradas: *MA2*[*4.p-1 .. 0*], *MB3*[*4.(p+3)-1 .. 12*], *AGTB, EOP* y *CIN*. Las salidas del módulo son: *COUT* (acarreo de salida) y una suma parcial *S1*[*4.p-1 .. 0*].

Primero debemos corregir el valor de *MB3* de acuerdo a la señal *EOP*.

if EOP=1 then (MB3$_{corrected}$)$_i$=9-(MB3)$_i$ *(8)*

else (MB3$_{corrected}$)$_i$=(MB3)$_i$

end if

El objetivo es obtener la suma parcial *S1=MA2+MB3*[*4.(p+3)-1 .. 12*].La técnica *carry-chain* consiste en calcular de antemano el todos los acarreos mediante condiciones de propagación y generación a fin de reducir el tiempo computacional del sistema. Consideramos para el diseño la buena performance del sumador propuesto en [28]. A continuación se explica en forma general la tecnología *carry-chain*.

3.6.1 Algoritmo Carry-chain

Primero consideramos dos operandos representados en base 10 (por simplicidad denominamos a *MA2* y *MB3* como *x, y* respectivamente) como $x= \Sigma_i x(i).10^i$ y $y= \Sigma_i y(i).10^i$. Para generalizar definimos dos funciones binarias llamadas propagación *(P)* y generación *(G)* que evaluarán las variables *x(i)* y *y(i)* como se muestra en el siguiente ecuaciones:

$$P(i) = P(x(i),y(i)) = 1 \text{ if } x(i) + y(i) = 9 \quad (9)$$

$$P(i) = P(x(i),y(i)) = 0 \text{ otherwise;}$$

$$G(i) = G(x(i),y(i)) = 1 \text{ if } x(i) + y(i) > 9$$

$$G(i) = G(x(i),y(i)) = 0 \text{ otherwise.}$$

El acarreo c(i+1) puede ser calculado como:

$$\begin{array}{ll} \textit{if } P(i)=1 \textit{ the} & c(i+1)=c(i) \\ \textit{else} & c(i+1)=G(i) \\ \textit{end if} & \end{array} \quad (10)$$

El siguiente algoritmo implementa el sumador *Carry-chain*

Algoritmo 6 Sumador carry-chain (cálculo de las funciones generación y propagación (P/G))

```
for i in 0 .. n-1 loop
        G(i)=G(x(i), y(i))
        P(i)=P(x(i), y(i))
end loop
c(0)=c_in                --Cálculo del acarreo
for i in 0 .. n-1 loop
        if P(i)=1 then c(i+1)=c(i)
        else c(i+1)=G(i)                                   (11)
        end if
end loop
for i in 0 .. n-1 loop   --Cálculo de la suma:
        z(i)=(x(i)+y(i)+c(i)) mod 10
end loop
z(n)=c(n)
```

Consideraciones adicionales

1.-La sentencia *(11)* es equivalente a la ecuación booleana:

$$c(i+1)=P(i).c(i) \text{ or } not(P(i)).G(i) \qquad (12)$$

si la relación anterior es usada, las funciones de generación y propagación pueden ser modificadas como:

$$G(i)=1 \text{ if } x(i)+y(i)>9 \qquad (13)$$

$$G(i)=0 \text{ if } x(i)+y(i)<9$$

$$G(i)=0 \text{ or } 1 \text{ (don't care) otherwise.}$$

2.-Otra ecuación booleana equivalente a *(12)* es:

$$c(i+1) = G(i) \text{ or } P(i).c(i) \qquad (14)$$

si la relación anterior es usada, entonces la función de propagación puede ser definida como:

$$P(i)=1 \text{ if } x(i)+y(i)=9 \qquad (15)$$

$$P(i)=0 \text{ if } x(i)+y(i)<9$$

$$P(i)=0 \text{ or } 1 \text{ (don't care) otherwise}$$

La estructura de un sumador de *N*-dígitos basado en un *carry-chain* de base 10 es mostrado en la figura 3.11. La celda *G/P* calcula las funciones de generación y propagación de las ecuaciones. El camino crítico lo muestran los bloques sombreados.

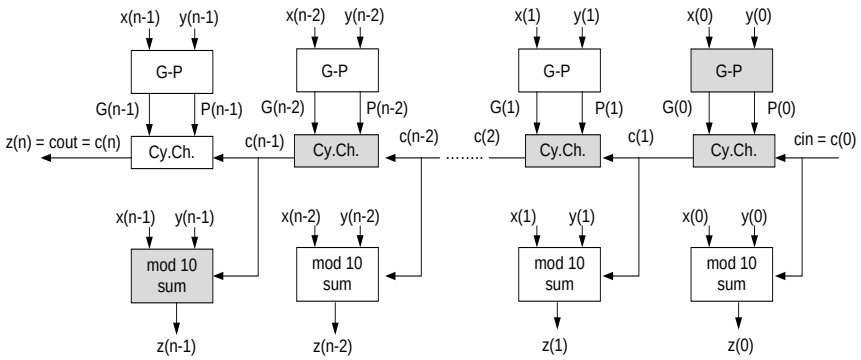

Figura 3.11 Sumador carry-chain en base 10

3.- Las celdas *carry-chain* son circuitos binarios, mientras las funciones generación-propagación y el *mod 10 sum* son circuitos decimales.

La ecuación *(12)* puede ser implementado por un multiplexor 2-1 (figura 3.12.a), mientras que la ecuación *(14)* por 2 compuertas (figura 3.12.b).

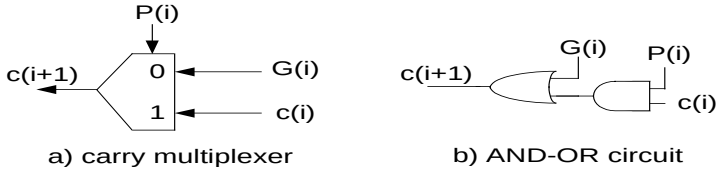

Figura 3.12 Celdas básicas para un sumador Carry-chain

3.6.2 Sumador BCD carry-chain

La celda básica i^{th} para el sumado *BCD carry-chain* es mostrado en la figura 3.12, la celda recibe los 2 dígitos a sumar *x(i), y(i*) y el acarreo de entrada *c(i)* y está formada por tres módulos: un sumador *mod 16*, una celda *carry-chain* manejada por las funciones *P/G* y una etapa de corrección. A continuación se explica cada etapa: luego de capturar dos dígitos *BCD*, dichos dígitos son sumados (*s(i)=x(i)+y(i)*) en *modulo 16*; como paso siguiente se generan las funciones *P/G* y por ende el acarreo de salida (*c(i+1)*); la última etapa corrige la suma *s(i)* sumándole la señal concatenada *c(i+1)&c(i+1)&c(i)* . Las adiciones a *s(i)* dependen de ambos acarreos y pueden ser los siguientes valores (Tabla 3.3):

C(i+1)	C(i)	Corrección
0	0	0
0	1	1
1	0	6
1	1	7

Tabla 3.3 Posibles correcciones en la suma *mod 16*

El diseño toma como acarreo de inicio la señal *CIN (c(0)).*

La figura 3.13 exhibe la manera de generar las funciones *P/G*, que es utilizada como celda básica en la figura 3.14.

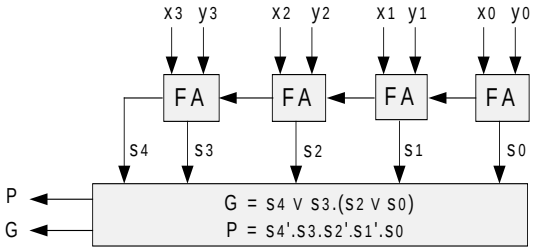

Figura 3.13 Celda *G/P* para sumadores *BCD*

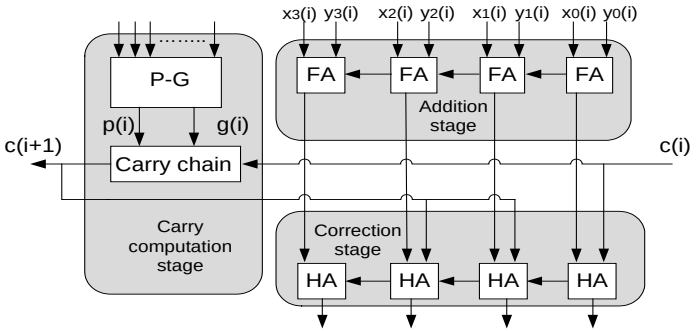

Figura 3.14 i[th] celda básica para sumadores *BCD*

Si queremos mejora en la latencia pero penalizando parcialmente el hardware podemos usar las siguientes formulas:

$$x(i)=(x_3\ x_2\ x_1\ x_0)(i),\ y(i)=(y_3\ y_2\ y_1\ y_0)(i) \qquad (16)$$

$P(i)=p0.[k1.(p3.k2$ or $k3.g2)$ or $g1.k3.p2]$

$G(i)=g0.[p3$ or $g2$ or $p2.g1]$ or $g3$ or $p3.p2$ or $p3.p1$ or $g2.p1$ or $g2.g1$

donde: $pj=xj \oplus yj$; $gj=xj.yj$ and $kj=xj'.yj'$ *(xj' is equal to not xj)*

Asumiendo valores para *MA2* y *MB3* representaremos las posibles operaciones suma/resta que puede ocurrir. Por facilidad de cálculo para el ejemplo, *MA2* y *MB3* serán nombrados como *A* y *B* y representados en *decimal32* o sea vectores de 7 dígitos (*A(i)* y *B(i)* donde *i=0,1,2..6*). La relación *C=(A+B) mod 16*, representa la suma de cada dígito de ambas mantisas en base 16. Se muestra también las funciones *Generation/Propagation* generadas, así como los respectivos acarreos formados y finalmente sumaremos las correcciones producidas a *C*. Se asume un acarreo inicial (*CIN*) igual a cero.

- ***MA2, MB3 y EOP=0***

La figura 3.15 presenta la suma cada dígito en *mod 16*, se calculan las funciones *P/G* de cada dígito así como el acarreo (las flechas indican la generación del acarreo). Las funciones *P/G* se calculan de *(9)*, se calcula luego el acarreo *(c)* y los dígitos de corrección provienen de la señal concatenada *c(i+1)&c(i+1)&c(i).* El resultado final es observado en la figura siguiente.

A =	2	5	6	8	9	1	8	eop=0
B =	3	5	3	1	8	0	9	
C = (A + B) mod 16	5	10	9	9	1	1	1	
Propagación =	0	0	1	1	0	0	0	
Generación =	0	1	0	0	1	0	1	
Acarreo =	0	1	1	1	1	0	1	0 ← CIN
Corrección =	1	7	7	7	6	1	6	
Resultado final = (C + Corrección) mod 16	6	1	0	0	7	2	7	

Figura 3.15 Operación suma para el caso *A>B y EOP=0*

- ***MA2, MB3 y EOP=1***

Cuando MA2 >= MB3

Es el mismo proceso de lo anterior con la diferencia que la mantisa *B* es complementada en base 9, y se adiciona un 1 a resultado final. El resultado final puede observarse en la figura 3.16. Recordar que la señal *AGTB* verifica si *MA2>=MB3*.

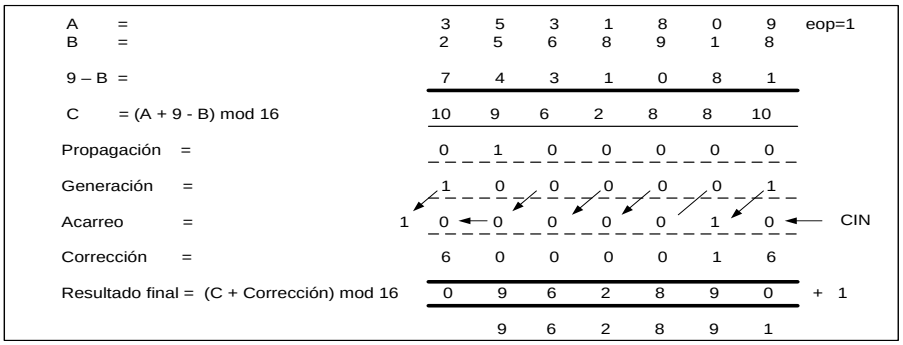

Figura 3.16 Operación resta para el caso *A>B* y *EOP=1*

Cuando MA2 < MB3

Se realiza el mismo proceso explicado anteriormente con la diferencia que el resultado final es complementado en base 9 como se observa en la figura 3.17. El resultado es negativo.

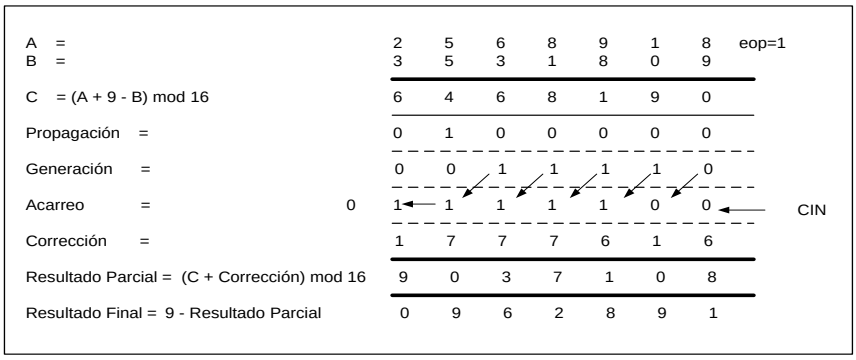

Figura 3.17 Operación resta para el caso *A<B* y *EOP=1*

Para minimizar el consumo de tiempo y hardware las implementaciones de *P(i)* y *G(i) de (16)* pueden ser calculadas como:

$$P(i)=(x0(i)\oplus y0(i)).pp(i) \qquad (17)$$

$$G(i)=gg(i) \text{ or } (pp(i).x0(i).y0(i))$$

Donde *pp* es una función de 6 entradas, tal que *pp(i)=1* si la suma de los primeros 3 bits de *x(i)+y(i)*=4. La función *gg* de 6 entradas es 1 si la suma de los 3 primeros bits *x(i)+y(i)*>=5.

3.6.3 Arquitectura propuesta para el sumador carry-chain

El objetivo es implementar las funciones *P/G* basadas en las formulas anteriores, las operaciones son controladas por *EOP*, recordemos la suma es realizada cuando *EOP=0* caso contrario se procesa la resta. Se debe definir de antemano las funciones las *ppa, gga* para el caso de la suma y *pps, ggs* para la resta, estas funciones se analizaron anteriormente. La arquitectura se observa en la figura 3.18.

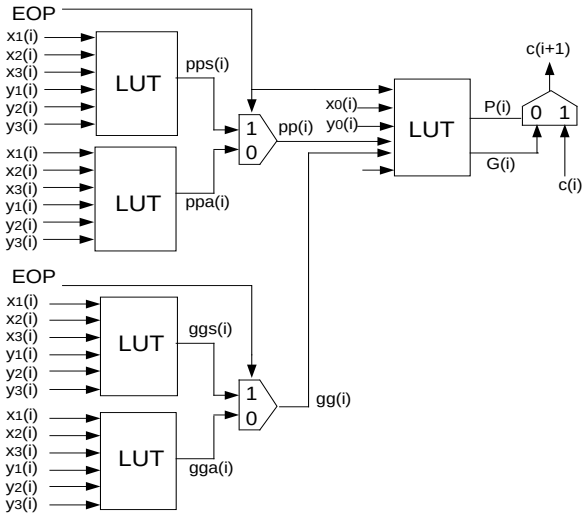

Figura 3.18 Circuito propuesto para un sumador *Carry-chain* sobre *FPGA*

El resultado final es conocido como suma parcial *S1=MA2+MB3*[*4.(p+3) .. 12*], y debe ser corregida cuando ocurre la siguiente condición:

$$(S1_{corrected})_i = 9-(S1)_i \text{ if } EOP=1 \text{ and } AGTB=0, \text{ otherwise } (S1_{corrected})_i = (S1)_i \qquad (18)$$

3.7 Módulo post-correction

El objetivo es generar los valores correctos que son usados por la etapa de redondeo, a continuación se describe las entradas y salidas:

Entradas

S1[*4.p-1* .. 0], suma parcial que proviene de la etapa anterior.

E2[*e_lim-1* .. 0], exponente parcial, el máximo de ambos exponentes normalizados.

COUT, acarreo de salida de la etapa suma decimal.

ED[3 .. 0], dígito extra usado para condiciones de *overflow*.

RD1[3 .. 0], dígito de redondeo parcial.

GD2[3 .. 0], dígito de guarda parcial.

PSB, señal que predice los *sticky-bit´s* correctos.

Salidas

S2 [*4.p-1* .. 0], suma actualizada y enviada a la etapa de redondeo.

E3, exponente corregido.

RD2[3 .. 0], dígito de redondeo final.

GD2[3 .. 0], dígito de guarda final.

FSB, Sticky-bit final.

Las correcciones aplicadas a las entradas son realizadas de acuerdo a las siguientes condiciones:

- ***Si EOP=1 y COUT=1***

Condición forzada cuando se desarrolla una operación suma y se genera un acarreo de salida, en consecuencia una corrección es aplicada a la suma parcial *S1*. Asumimos *COUT* como el *MSD* de *S1* por lo tanto para mantener la precisión de 16 dígitos se descarta el *LSD* de *S1*, este cambio es almacenado en una señal *S2*. También se realizan nuevas correcciones como: el dígito de guarda *GD2* queda definido por el *LSD* de *S1*, el dígito de redondeo es ahora la entrada *GD1*, el exponente final, designado por *E3*, posee el valor de *E2+1* y el valor final del *sticky-bit* (*FSB*) está dado por la operación *OR*: *PSB(0) or PSB(1) or RD1*.

- ***Si EOP=1 y MSD de S1=0 y GD1>0***

Está condición se cumple cuando se realiza una operación resta, el *MSD* de *S1* es igual a cero y el dígito de guarda *GD1* es mayor que cero. Entonces *S2* está formada por los 15 *LSD's* de *S1* concatenado con el *GD1* (*GD1* será el nuevo *LSD* de *S2*). El dígito final de guarda *GD2* es determinado por *RD1*, el dígito de redondeo final *RD2* es igual al dígito extra *ED*, el exponente final *E3* es actualizado a *E2-1* y el *sticky-bit* final (*FSB*) es actualizado a *PSB(0).*

- ***Si EOP=1 y MSD de S1=0 y GD1=0***

Condición forzada cuando se desarrolla una resta, el *MSD* de *S1* es igual a cero y dígito guarda de entrada es igual a cero. El resultado *S2* es igual a *S1*, el digito de guarda final *GD2* es igual a *GD1*, el dígito de redondeo final *RD2* es igual a *RD1*, el exponente final es igual a *E2* y el *sticky-bit* final (*FSB*) equivale a la operación *OR*: *PSB(0) or PSB(1).*

En la figura 3.19 observamos la arquitectura para esta etapa.

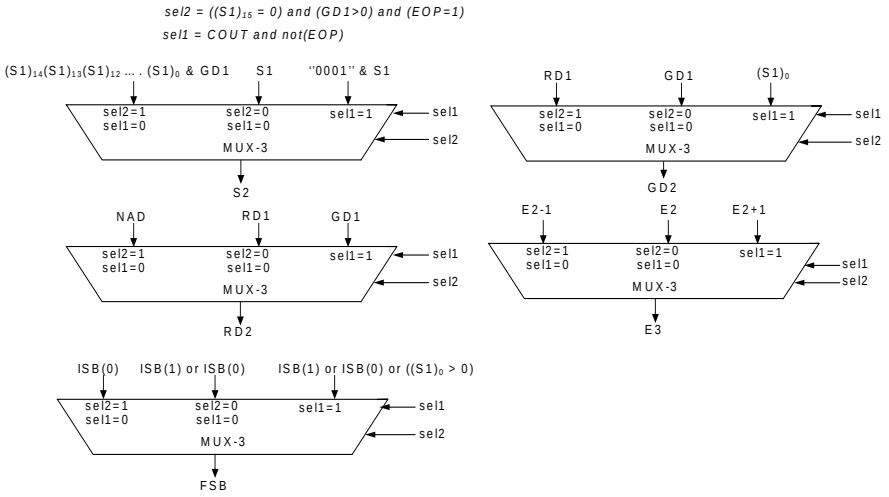

Figura 3.19 Circuito propuesto para la etapa *post-correction*

3.8 Módulo Rounding

Este módulo recibe como entrada señales verificadas las cuáles son: la suma/resta corregida *S2*, el exponente actualizado *E3*, los dígitos finales para el *GD2* y *RD2*, y el *Final Sticky-bit (FSB)*. Luego se desarrolla el redondeo seleccionado, en la tabla 3.4 vemos las alternativas de esta operación. El redondeo es desarrollado cuando no hay presencia de *overflow* ni *underflow* y cuando la precisión de dígitos de *S2* es mayor que *p* (para *Decimal64 p*=16). Las señales *FSB, GD2* y *RD2* son procesadas de acuerdo a las condiciones de la tabla 3.4. Un algoritmo general de *rounding* es:

Algorithm 7 Algoritmo general de redondeo

```
GD=GD2, SB=FSB              -- guard digit y sticky bit.
if GD<5 then add_one='0'
elsif GD>5 then add_one='1'
else                        -- GD=5
        if SB='0' then add_one='0'
        else add_one='1'
        end if
end if;
```

Las salidas de este proceso son la suma final *S3* y el exponente *E4*.

Existe un caso especial cuando la suma *S2* es 999…999 y el redondeo afirma que se debe agregar 1 a *S2*, entonces se adiciona un dígito a *S2* cuyo resultado *S3* sería 10…0, y el exponente final *E4* sería *E3 +1*. Esta etapa detecta los casos de *underflow* y *overflow*. Por último, los resultados obtenidos en esta etapa (*S3* y *E4*) y el signo final *FS* se envían al codificador el cual genera la suma en formato *IEEE 754-2008*.

El signo final *FS* se calcula como:

$$FS = (SA \text{ and not}(EOP)) \text{ or } (EOP \text{ and } (AGTB \text{ xnor } (SA \text{ xor } SWAP))) \quad (19)$$

Modelo de Redondeo	***Proceso***
Round Ties to Even	*(GD > 5) v ((GD == 5) ∧((RD > 0) v SB))*
Round Ties to Away	*GD >= 5*
Round Towars Positive	*!SF ∧ ((GD >0) v (RD >= 0) v SB)*
Round Towars Negative	*SF ∧ ((GD >0) v (RD >= 0) v SB)*
Round Towars Zero	*none*

Tabla 3.4 Modelos de redondeo

3.9 Implementación en FPGA y resultados

Los circuitos fueron descritos en *VHDL* e implementados sobre la *FPGA Xilinx Virtex-5 [34]*, dispositivo *XC25VTX240T*, con *speed-grade*-2. Para la síntesis y la implementación usamos las herramientas *XST [33]* y *Xilinx ISE 12*. [32]. Los módulos diseños son completamente combinacionales, en algunos casos se utilizan las funciones primitivas que provee la *Virtex-5*. Para mayor aceleración el sumador es diseñado en 8 etapas de pipeline, aplicando *timing constraints* llegamos a probar que el diseño propuesto puede trabajar hasta una frecuencia de

200 Mhz. Las aplicaciones de sumadores en *FPGA* son limitadas, a nuestro criterio el diseño del sumador *DFP* presentado es uno de los primeros sobre *FPGA.* Existe una propuesta en [9] basada en un sumador de codificación *BID* (*Binary Integer Decimal*), los resultados serán usados para comparaciones.

Components		***Cycles***	***Delay Breakdown***	
			LUT´s	***%***
Decoder		*1*	*128*	*5.6%*
BCD Addition	*Leading Zero Detector*	*1*	*18*	*0.8%*
	Swapping	*1*	*16*	*0.7%*
	Pre-signal generation	*1*	*97*	*4.2%*
	Carry-chain Adder	*1*	*375*	*16.4%*
	Post-Correction	*1*	*91*	*4.0%*
	Rounding (overflow, underflow)	*1*	*110*	*4.8%*
	Further combinational circuits	*0*	*1416*	*59.9%*
Special Cases		*0*	*4*	*0.2%*
Further combinational circuits		*0*	*48*	*1.0%*
Coder		*1*	*83*	*3.6%*
Entire Design		***8***	***2390***	***100%***

Tabla 3.5 Contribución en área de los módulos involucrados en el diseño

La tabla 3.5 muestra el coste de área de los módulos involucrados. El sumador *carry-chain* está basado en el presentado en [28], se realizó un *rescheduling* y se obtuvo un incremento de velocidad del 13%. El módulo *LZD* propuesto es comparado con el utilizado en el multiplicador *DFP* implementado en [18], se observó que el módulo propuesto presenta 18 % mayor rapidez. En la tabla 3.6 observamos el diseño completo de nuestro sumador considerando mínimo delay.

DFP adder/subtractor	***#slices***	***# FF***	***# LUT***	***# Cy***	***Period minimum (ns)***	***Latency (ns)***
Minimun delay	*935*	*935*	*2390*	*8*	*5*	*40*

Tabla 3.6 Resultados del sumador sintetizado/implementado en *Virtex-5*

Otra manera de evaluar los resultados fue realizar lo siguiente: El sumador fue implementado sobre un *soft-processor Microblaze* obteniendo 32 ciclos para una solución *HW* y 822 ciclos para una solución *SW*, de la misma manera las *The DecNumber Library* [14] fueron evaluadas para comparar la latencia de un sumador *DFP* sobre distintos procesadores. En la Tabla 3.7 se muestra algunas implementaciones (software y hardware) en diferentes plataformas.

Application	Adder	Clock Speed (GHz)	Cycles	Delay		Area		Mop /sec
				#FO4	ns	NANDs	mm2	
SW	Itanium2 [6]	1.4	219	-	156.4	-	-	6.4
	Xeon5100[5]	3	133	-	44.3	-	-	22.6
	Xeon [6]	3.2	249	-	77.8	-	-	12.9
	Pentium M Intel[14]	1.5	848	-	565.3	-	-	1.8
	Power6[31]	5	17	-	3.4	-	-	294.1
	Z10[31]	4.4	12	-	2.7	-	-	366.7
HW	BID Adder on 65nm[9]	1.3	13	-	10	-	-	100
	Adder on 0.11um [30]	-	-	50.2	2.76	22085.5	0.14	362.3
	Adder on 0.11um [25]	-	-	63.6	3.498	22443.3	0.15	285.9
	BID Adder on 0.11um [26]	-	-	44	2.42	68459.0	0.44	413.2
	Adder FPGA proposed	0.002	8		40	-	-	200

Tabla 3.7 Latencias de un sumador *DFP* implementado en diferentes plataformas

Los datos de la Tabla 3.7 fueron obtenidos de [9], se observa que la librería *DFP* ha sido evaluada en varios procesadores comprobando una performance que presenta una peor latencia de 848 ciclos y una mejorada de 12 ciclos. Esto nos permite argumentar una ligera mejora de la propuesta, pues solo necesitamos 8 ciclos con una eficiencia de 200 *Mops/sec*.

- ***Verificación***

La validación del diseño consistió en generar, mediante *SW* y en base a las librerías *DFP*, gran cantidad de vectores de números *DFP* que fueron aplicados en *Testbench* usando la herramienta *ModelSim*. Para el diseño se utilizó más de 40,000 operaciones.

- ***Comparaciones***

En la Tabla 3.8 mostramos la síntesis del sumador/restador comparado con el sumador *BID* que propone [9], el diseño propuesto reporta menos ciclos de trabajo, una performance de 22% más rápido pero un coste de área superior en 10% con respecto al sumador *BID*. Podemos argumentar que el trabajo presente es una gran alternativa de diseño y uno de los primeros aportes para sumadores *DFP* sobre *FPGA*.

Adder DFP	LUTs	FF	DSPs	BRAM	Freq (MHz)	Cycles	Latency (ns)	Mops/sec
Adder en [7]	2171	1392	12	1	163.9	13 / 18	79.3 / 109.8	12.6 / 9.1
Adder proposed	2390	935	-	-	200	8	40.0	200

Tabla 3.8 Resultados del sumador *BID* en [7] y el sumador propuesto

4 DISEÑO DE LA MULTIPLICACIÓN PUNTO FLOTANTE DECIMAL

Este capítulo detalla el proceso de multiplicación de números decimales en punto flotante. El circuito al igual que la suma/resta considera los casos especiales, diferentes técnicas de redondeo y manejo de excepciones. El esquema propuesto es mostrado en la figura 4.1. Los operandos (en formato *IEEE 754-2008*) considerados se representan como cadenas de bits. El diseño presenta las siguientes etapas: decodificación, detección de ceros principales (*LZD*), multiplicador de 16 dígitos en punto fijo, redondeo y codificación. Se explicará solo la etapa de multiplicación pues las demás se explicaron en el sumador.

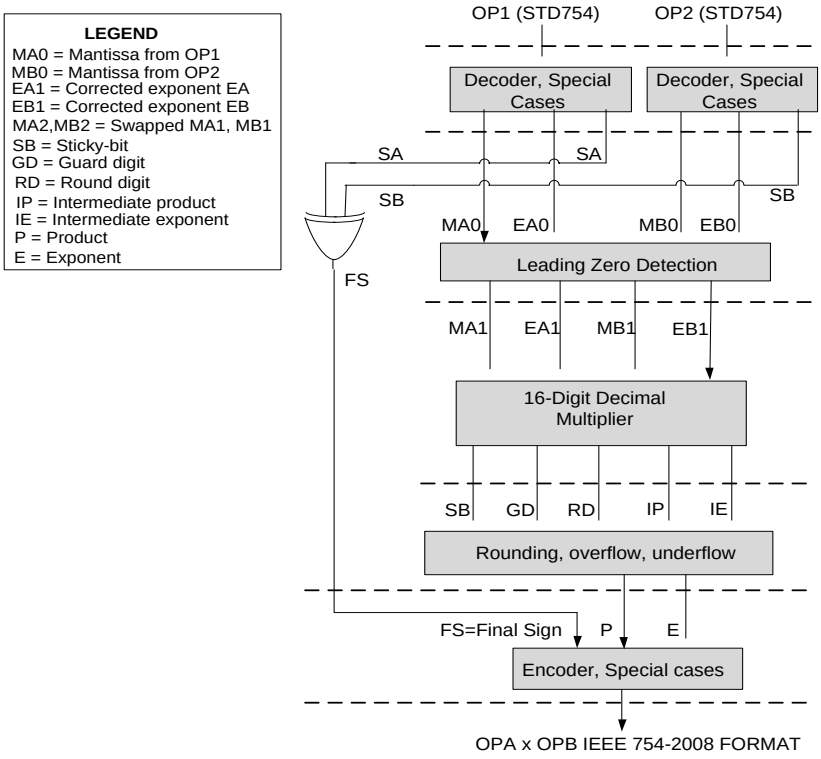

Figura 4.1 Circuito propuesto para la multiplicación DFP

4.1 Módulo Multiplicador de 16-digitos BCD

El diseño es basado en la publicación [24] en donde varias técnicas son propuestas. Seleccionamos un multiplicador secuencial basado en *BRAM* debido a que trabajamos con operandos con aceptable cantidad de dígitos, esto reduce el área de circuito pero penaliza ligeramente la latencia [24]. El diseño consta de una etapa basada en memorias *BRAM*, una etapa multiplicador *BCD* de *Nx1* dígitos y por último la etapa multiplicador *NxM* dígitos *BCD*.

El diseño utiliza las *BRAM´s* presentes en los dispositivos *Xilinx*, en las cuáles almacenamos todas las posibles multiplicaciones de 2 dígitos *BCD*. El total de combinaciones ocupa 100x8 bits *ROM* que pude ser mapeado como 2^8 *x8-bit ROM*. En [24] se argumenta que diseños basado en *BRAM* no son muy recomendables para implementaciones combinacionales, pero si para diseños secuenciales y pipeline. El módulo multiplicador *Nx4*, calcula el producto de 4 dígitos con los *N* dígitos de otro operando. Los resultados demuestran que el módulo *Nx4* genera una mejor performance y un aceptable *trade-off* latencia/área. El circuito multiplicador secuencial *NxM* implementa *M/4* veces el módulo multiplicador *Nx4* lo que genera *M/4* productos parciales, una manera de acelerar el sistema es obtener los productos parciales en paralelo pero cada uno empezando un ciclo después. El módulo multiplicador *NxM* dígitos *BCD* usa como celda básica el módulo *Nx4*, y los resultados demuestran una aceptable velocidad de procesamiento.

El diseño implementa un multiplicador 16x4, dígitos entonces un multiplicador 16x16 posee como celda básica 4 veces el multiplicador 16x4, generando 4 producto parciales. La publicación [24] indica que un multiplicador 16x4 otorga una latencia de 5 ciclos, y como los productos parciales restantes se realizan en forma paralela un ciclo después, entonces la latencia de un multiplicador 16x16 es 8 ciclos. La sumas parciales se basan en sumadores eficientes como el *carry-chain* utilizado en la operación suma/resta. En la figura 4.2 mostramos el esquema de un multiplicador *Nx1* dígitos, multiplicamos un vector de *N* dígitos *a=a(n)a(n-1)a(n-2)….a(1)a(0)* con un dígito *b*. El módulo *1x1 BCD multiplier* multiplica dígito por dígito y el resultado es un número de la forma *DC(n)* con respecto a un dígito genérico *n*, *D* sería la decena y *C* la unidad, luego construimos vectores con las unidades y decenas como se aprecia en la figura 4.2. Se suman ambos vectores pero considerando un desplazamiento de 1 dígito a la derecha de *C* sobre *D*.

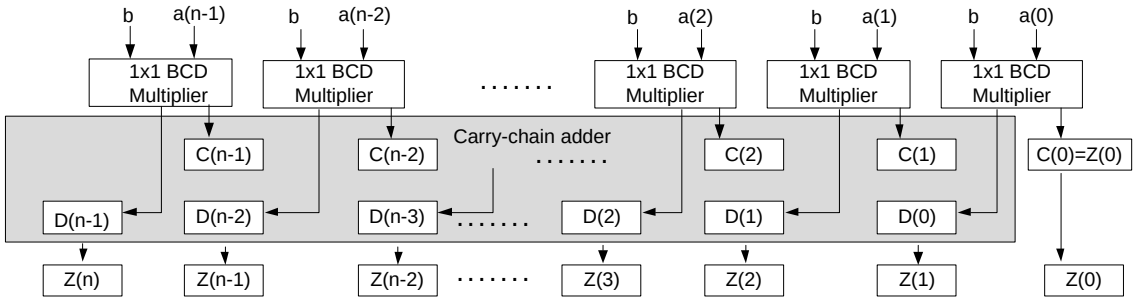

Figura 4.2 Esquema de un multiplicador *BCD Nx1*

En el siguiente ejemplo vemos un el proceso para un multiplicador *NxM*. Para simplicidad se usa operandos en formato *decimal32*.

Algorithm 8 Ejemplo de una multiplicación Decimal Punto Flotante IEEE 754-2008

MA0[27 .. 0] = 0005676, EA0 = -5, SA = 0
MB0[27 .. 0] = 0000435, EB0 = -7, SB = 1

SF = SA xor SB = 1

Etapa *Leading zero detection*

LZDA = LZD(MA0)=3 y EA1 = EA0 - 3 = -8
LZDB = LZD(MB0)=4 y EB1 = EB0 - 4 = -11
MA1 = 5676000
MB1 = 4350000

Etapa Multiplicación

MB2 = 5 6 7 6 0 0 0 x
MB3 = 4 3 5 0 0 0 0

El algoritmo de la multiplicación consiste en:

5 6 7 6 x
4 3 5

Multiplicando 4x1 dígitos
Multiplicamos dígito x dígito

```
          5   6   7   6  x
                      5
          ---------------
                  3   0
              3   5
          3   0
      2   5
     ------------------------
```

```
  2   3   3   3
  0   5   0   5   0   +
  -----------------
  2   8   3   8   0
```

Multiplicamos dígito x dígito

```
          5   6   7   6  x
                      3
          ---------------
                  1   8
              2   1
          1   8
      1   5
     ------------------------
```

```
  1   1   2   1
  0   5   8   1   8   +
  -----------------
  1   7   0   2   8
```

```
          2   8   3   8   0 +
      1   7   0   2   8
  2   2   7   0   4
  ---------------------------
  2   4   6   9   0   6   0
```

Multiplicamos dígito x dígito

```
          5   6   7   6  x
                      4
          ---------------
                  2   4
              2   8
          2   4
      2   0
     ------------------------
```

```
  2   2   2   2
  0   0   4   8   4   +
  -----------------
  2   2   7   0   4
```

Redondeo

PI[55 .. 0] = 24690600000000
GD = PI[27 .. 24] = 0
RD = PI[23 .. 20] = 0
Sticky-bit = Operación OR de los bits restantes = 0

Resultado

Multiplicación final[27 .. 0] = 2469060,
Exponente final = E + 7 = -12,
Signo final=1

4.2 Implementación en FPGA y resultados

Los circuitos fueron descritos en *VHDL* y fueron implementados sobre la *FPGA Xilinx Virtex-5* [34], dispositivo *XC25VTX240T,* con *speed-grade*-2. Para la síntesis y la implementación usamos las herramientas *XST* [33] y *Xilinx ISE 12*. [32]. Como en el caso de la suma, hay pocas propuestas sobre multiplicadores *DFP* ya que la mayoría se basa en *ASIC´s* y una comparación *FPGA´s/ASIC´s* no sería justo. Hay una propuesta de diseño sobre *FPGA* en [18] que será de base para las comparaciones. Todos los componentes son diseños combinacionales excepto el *módulo 16-Digit Decimal Multiplier* que es secuencial y tiene 8 ciclos de reloj. Para aumentar la velocidad el multiplicador presenta 5 etapas de segmentación, aplicando *timing constraints* llegamos a probar que el diseño propuesto puede trabajar hasta una frecuencia de 161.3 Mhz. En la Tabla 4.1 presentamos la contribución de área de cada módulo, así como los ciclos de trabajo.

Components		***Cycles***	***Delay Breakdown***	
			LUT´s	***%***
Decoder		*1*	*128*	*2.6%*
BCD Multiplier	*Leading zero Detector*	*1*	*24*	*0.5%*
	16x16 Multiplier	*8*	*3768*	*77.2%*
	Rounding	*1*	*103*	*2.1%*
	Combinational circuits	*0*	*656*	*13.4%*
Coder		*1*	*83*	*1.7%*
Combinational circuits		*0*	*116*	*2.4%*
Entire Design		*12*	*4878*	*100.0%*

Tabla 4.1 Coste de área en un multiplicador *DFP*

- ***Verificación***

La validación del diseño consistió en generar, mediante *SW* y en base a las librerías *DFP*, gran cantidad de vectores numéricos *DFP* que fueron aplicados a *Testbench* usando la herramienta *ModelSim*. Para nuestro diseño usamos más de 30,000 operaciones.

- ***Comparaciones***

Una manera de comparar los resultados es implementar multiplicaciones en punto flotante binario a partir de los *core* que ofrece *Xilinx* [18] usando el formato *Binary64.* Se desarrolló tal operación sobre una *Virtex 4-12* y se consideraron dos soluciones:

Multiplicador diseñado con *DSPs* que reportó 24 ciclos a 454 Mhz, 17 bloques *DSPs*, 580 *slices* (*768 FF, 524 LUTs*) y una latencia de 50.6ns.

Multiplicador diseñado sin *DSPs* que reportó 9 ciclos a 243 Mhz, 1381 *slices* (*1381 FF, 2308 LUTs)* y una latencia de 36.9 ns. Estos resultados confirman la mejor performance de operaciones en binario.

Otra manera de comparación consistió en hacer un *rescheduling* al multiplicador propuesto en [18]. Este multiplicador realiza el proceso de normalización (instanciación del módulo *leading zero detection*) al finalizar la multiplicación de las mantisas y previo al proceso de redondeo, mientras el diseño propuesto normaliza las mantisas antes de ser multiplicadas y considera

simples arquitecturas para el resto de módulos. Este *rescheduling* reporta una mejor performance: 17% más rápida y 43% menos en área que el multiplicador original.

En la tabla 4.2 se observa la síntesis de ambos multiplicadores.

Multiplier DFP	***Slice***	***LUTs***	***FF***	***T (ns)***	***Cycles***	***Latency (ns)***	***Mops/sec***
Multiplier [16]	*1595*	*3606*	*1595*	*7.3*	*12*	*87.6*	*11.4*
Multiplier proposed	*2068*	*2068*	*4878*	*6.2*	*12*	*74.4*	*13.4*

Tabla 4.2 Comparación de multiplicadores *DFP*

Podemos argumentar que el multiplicador presenta menos coste de área que el sumador, éste último es más complejo pues emplea más módulos en su diseño.

5 PRUEBAS Y RESULTADOS

Para poder utilizar los módulos diseñados, analizar los tiempos entre señales de entrada salida y presentar comparaciones de soluciones *HW/SW* se implementa una aplicación de tarifación de una compañía de teléfono (*telephone company billing*). Este algoritmo se basa en operaciones de suma/multiplicación por lo tanto no intenta cubrir todo el amplio espectro de operaciones en punto decimal.

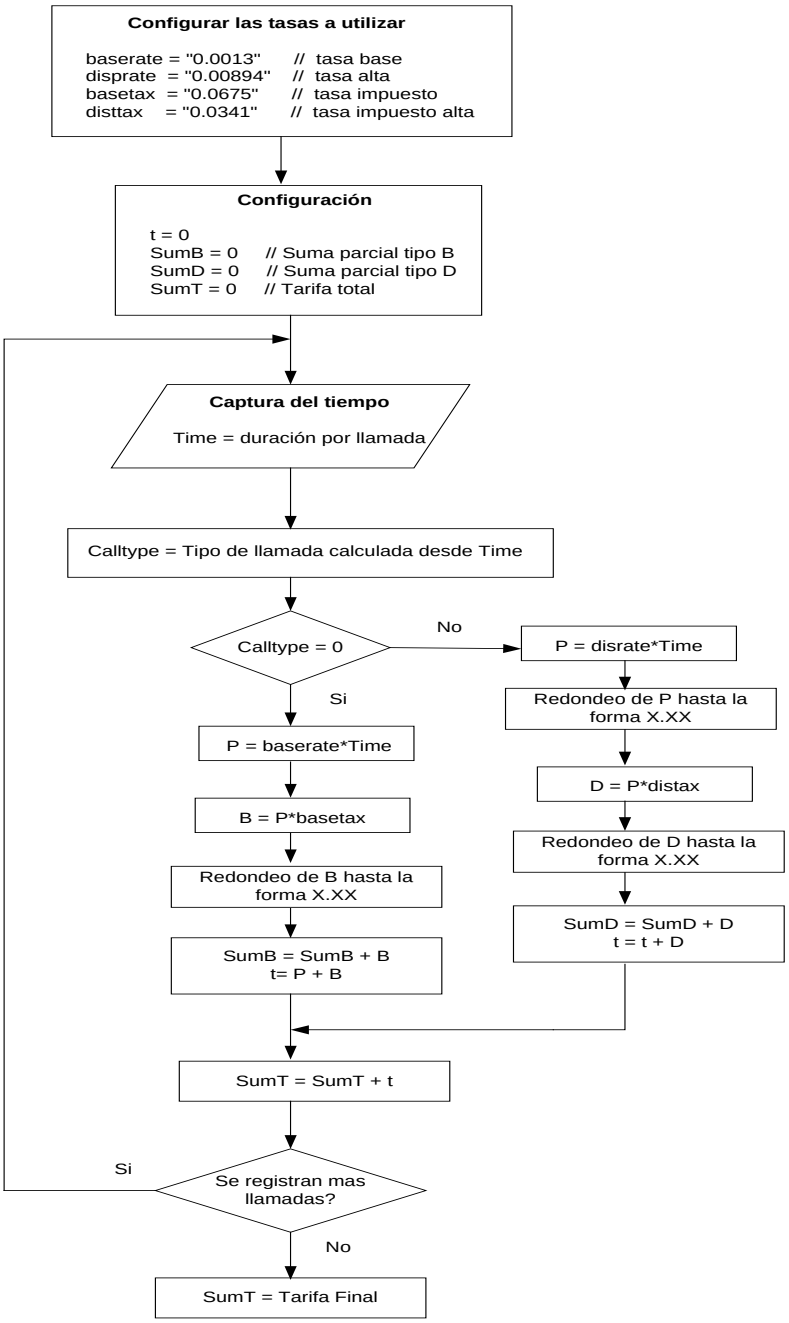

Figura 5.1 Diagrama de flujo de la aplicación *telephone company billing*

La aplicación consiste en capturar desde un archivo de texto una gran cantidad de datos que representan las duraciones telefónicas en segundos que un usuario realiza. Para cada llamada se aplica una serie de tasas de costo y diferentes impuestos, también se aplican redondeos para casos particulares. Estas operaciones procesan todos los tiempos y generan como resultado la tarifa telefónica, esto se aplica para cientos de llamadas. El software implementado en C++ se basó en las funciones de aritmética decimal propuestas por la *The DecNumber Library* [14]. Las operaciones aritméticas propuestas por la librería fueron rediseñadas debido a la gran complejidad de codificación que muestran y sobre todo por la necesidad de probar parte del diseño sobre memorias locales (*BRAM*) de limitada capacidad. El diagrama de flujo de la figura 5.1 muestra los pasos para el desarrollo de la aplicación.

La implementación final se lleva a cabo mediante técnicas de *HW/SW* y mediante una plataforma reconfigurable *Soft Core Microblaze* para el desarrollo de las operaciones suma, resta y multiplicación.
La solución *SW* de las operaciones suma y multiplicación se basaron en la librería de números en punto flotantes *The DecNumber Library*, la solución *HW* de las mismas operaciones se implementó en *VHDL* y fueron explicadas en los capítulos anteriores. Mencionaremos algunas especificaciones que posee el procesador *Microblaze*, que es utilizado como herramienta para las pruebas finales.

5.1 Microblaze Soft Processor

Xilinx proporciona a los diseñadores, mediante su procesador embebido *Microblaze*, la forma de aprovechar el potencial de sus *FPGAs*, *Spartan o Virtex*. Se trata de un "*soft processor*" de 32 bits tipo RISC, con arquitectura *Hardvard* y posibilidad de usar memorias *on-chip* y memorias externas. Está optimizado para ser implementado en las *FPGA's* de *Xilinx*, su propietario. Existe gran cantidad de periféricos y coprocesadores en el mercado para aumentar el rendimiento de este procesador. Principalmente los que tienen interfaz para el bus *OPB*, al ser éste compatible con *PowerPC*. Permite el uso también del bus *FSL (Fast Simplex Link*), un bus unidireccional y dedicado que garantiza la mayor velocidad de transmisión.

5.2 Xilinx EDK

Xilinx, mediante el paquete *Embedded Development Kit (EDK)*, proporciona las herramientas necesarias para el completo diseño de los sistemas embebidos, tanto para la creación de los componentes hardware y software, como para la verificación y simulación de los mismos. Para la creación y edición de las especificaciones hardware de sistemas embebidos que usen los procesadores *MicroBlaze* de *Xilinx* o *PowerPC* de *AIM, EDK* contiene la aplicación *Xilinx Platform Studio (XPS),* que automatiza el proceso de creación del hardware mediante la aplicación *Base System Builder (BSB) wizard* [32]. Para la verificación de los proyectos, *EDK* permite la intrusión en el sistema tanto de *Debug* software como hardware, haciendo posible su estudio para la posterior optimización. En el caso de necesitar una funcionalidad no implementada en las librerías de periféricos de *EDK* existe la posibilidad de añadir al proyecto *cores* específicos. La herramienta *Create and Import Peripheral Wizard* ayuda al diseñador a crear su propio hardware e importarlo a *EDK*.

5.3 Bus FSL

El bus *FSL* contiene 8 entrada y 8 salidas para interfaces. *FSL* es un canal unidireccional punto a punto para recibir o trasmitir palabrada de 32 bits. La performance de una interfaz *FSL* puede alcanzar los 300 MB/sec. El *FSL* bus es manejado por *drivers* de tipo *Máster* y *Esclavo*, y los datos de entrada/salida basados en comunicación *FIFO*. *Xilinx EDK* provee un conjunto de macros para las operaciones de escritura y lectura del bus *FSL*. Este bus es un canal de comunicación rápida por tal motivo acelera el envío y recibo de datos. El bus presenta señales de control para escritura, lectura, verificación de memoria vacía o llena, y existencia de datos. En la figura 5.2 se observan las señales de control, la letra *M* representa al driver *Master* (*Microblaze*) y *S* al driver *slave* (*Coprocesador*).

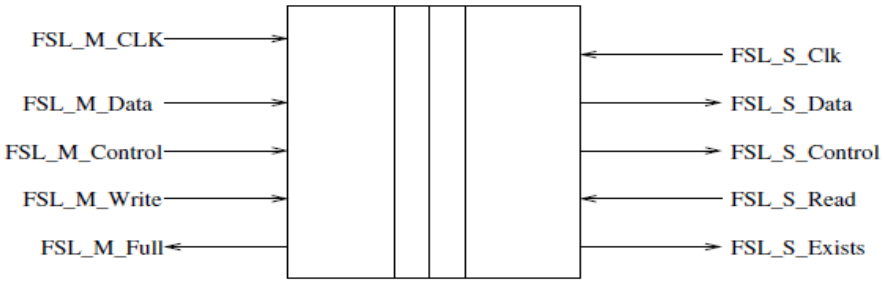

Figura 5.2 Señales de control del *BUS FSL*

Por las definiciones anteriores, parte del trabajo es la aceleración del algoritmo *telephone billing* por eso la idea es generar coprocesadores para la suma y multiplicación y conectarlos al *Microblaze* mediante la interfaz *FSL*. La figura 5.3 muestra la idea del sistema propuesto *SW/HW*.

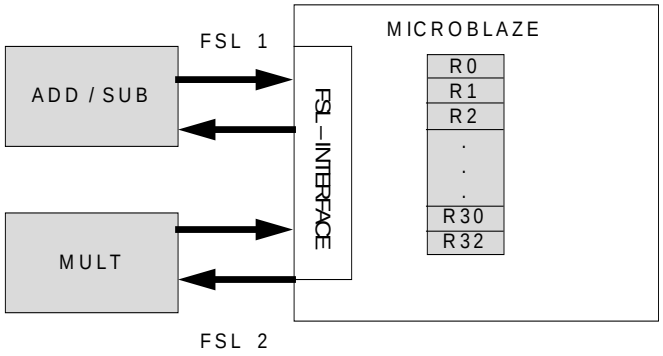

Figura 5.3 Diagrama general del diseño en base a coprocesadores

5.4 Implementación SW/HW sobre Microblaze

La primera solución fue implementar las operaciones suma y multiplicación sobre el procesador, se hizo un *rescheduling* de las librerías debido a la complejidad de programación que poseen y debido a que gran parte de nuestras pruebas debían hacerse sobre *BRAM*. La plataforma usada fue la interfaz gráfica *Xilinx Platform Studio (EDK 12.1)* [32], tarjeta de evaluación *AES-V5FXT-EVL30* de *Avnet*, un procesador simple *Microblaze @ 125Mhz*, *BRAM*

de 64 K, bus locales de datos e instrucciones de *64K*, comunicación *UART* y una interrupción de *timer*. Para la solución *HW* se diseñaron coprocesadores usando como interfaz de comunicación el *BUS FSL* (figura 5.3). El diagrama de bloques generado por la herramienta gráfica *Xilinx Platform Studio* es observada en la figura 5.4.

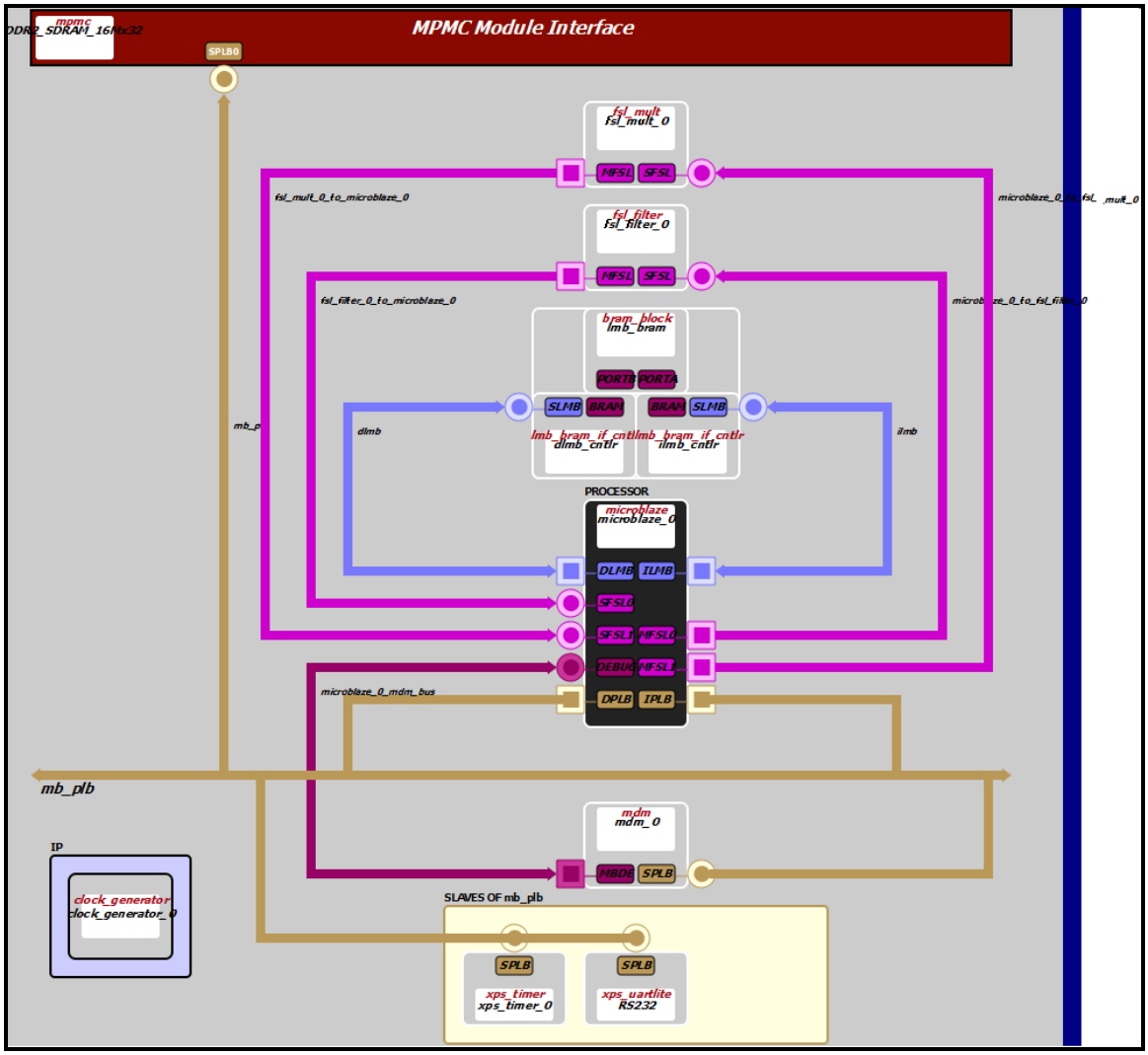

Figura 5.4 Diagrama de bloques propuesto por la herramienta *Xilinx Platform Studio*

Las pruebas se basan en la generación a gran escala de datos en formato *decimal64* que representan los tiempos de duración de las llamadas telefónicas, desarrollo de las operaciones aritméticas en *SW* y luego en *HW*, por último la comparación de los valores obtenidos en *SW* y *HW*.

Primero se implementa los coprocesadores para la suma y multiplicación, detallaremos en tablas los resultados obtenidos respectivamente. Finalmente se desarrolla una solución *HW/SW* para la aplicación *telephone billing* detallando también los resultados generados.

El programa utilizado para la suma se muestra en la figura 5.5, observar que la simulación en *SW* está representado por la función *decDoubleAdd* (para la suma) y la solución HW está representado por las funciones son *nputfsl* (para escribir sobre el bus) y *ngetfsl* (para leer del

bus) que nos permite comunicarnos con el bus *FSL*. Recordar que las funciones utilizadas se basan en [14].

Para el caso de la multiplicación la implementación *SW* se basa en la función *decMultiplyAdd()*. Como trabajamos con números en formato *Decimal64* debemos escribir 4 números (los dos operandos a sumar o multiplicar) de 32 bits y leer 2 números de 32 bits que representa el resultado final.

```
/*****************************************************************************/
/*********************************Create set of samples*************************/
/*****************************************************************************/
xil_printf("\n\n\rGenerating samples...\n\r");
j = 0;
for (n = -SAMPLES/16; n < SAMPLES/16; n++)
    { for ( m = -SAMPLES/16; m < SAMPLES/16; m++)
          {  RangedRandDemo(16,0,bcdp1);
             RangedRandDemo(16,0,bcdp2);
             sig_A=0; exp_A=m;
             sig_B=0; exp_B=n;
             decDoubleFromBCD(&sample[j],exp_A,bcdp1,sig_A);
             decDoubleFromBCD(&sample[j+1],exp_B,bcdp2,sig_B);
             j=j+2;
          }
    }

/*****************************************************************************/
/*********************************SW IMPLEMENTATION***************************/
/*****************************************************************************/
xil_printf("Executing multiplier in SW...\n\r");
start_time = XTmrCtr_GetTimerCounterReg(XPAR_XPS_TIMER_0_BASEADDR, 0);
for (n=0; n < 2*SAMPLES; n=n+2)   decDoubleAdd(&sw_result[n/2],&sample[n],&sample[n+1],&set);
end_time = XTmrCtr_GetTimerCounterReg(XPAR_XPS_TIMER_0_BASEADDR, 0);
xil_printf("elapsed cycles SW : %d  \n\r",(end_time-start_time) );

/*****************************************************************************/
/*********************************HW IMPLEMENTATION***************************/
/*****************************************************************************/
xil_printf("Executing adder in HW...\n\r");
start_time = XTmrCtr_GetTimerCounterReg(XPAR_XPS_TIMER_0_BASEADDR, 0);
for (n = 0; n < 2*SAMPLES; n=n+2)
          {
                nputfsl((&sample[n])->words[0],0);
                nputfsl((&sample[n])->words[1],0);
                nputfsl((&sample[n+1])->words[0],0);
                nputfsl((&sample[n+1])->words[1],0);
                ngetfsl((&hw_result[n/2])->words[0],0);
                ngetfsl((&hw_result[n/2])->words[1],0);
          }
end_time = XTmrCtr_GetTimerCounterReg(XPAR_XPS_TIMER_0_BASEADDR, 0);
xil_printf("elapsed cycles HW : %d  \n\r", (end_time-start_time));
```

Figura 5.5 Programa utilizado para la operación suma

En la tabla 5.1 y 5.2 observamos la comparación de los ciclos de trabajo para solución *SW* y *HW* de ambas operaciones. Para una medida precisa de los ciclos de consumo de cada operación el diseño se implementó sobre memorias *BRAM´s*. La tabla presenta los ciclos totales en *SW (Cycles SW)*, ciclos totales en *HW (Cycles HW)*, ciclos por operación (*Cycles/operation*), el *speed-up* y el *delay*. El número de operaciones realizadas fueron de 256, 512, 1024, 2048 y 4096 como se observa en la tabla de abajo. El mismo número de operaciones se utilizaron en las demás tablas.

Adder Bram / *# Operations*	*Cycles SW*	*Cycles HW*	*Cycles / operation SW*	*Cycles / operation HW*	*Speed-up*	*Delay (ns) SW*	*Delay (ns) HW*
256	*222222*	*8190*	*868*	*32*	*23.4*	*6944.4*	*255.9*
512	*444430*	*16388*	*868*	*32*	*23.4*	*6944.2*	*256.1*
1024	*888846*	*32780*	*868*	*32*	*23.5*	*6944.1*	*256.1*
2048	*1777678*	*65600*	*868*	*32*	*23.5*	*6944.1*	*256.3*
4096	*3555342*	*131100*	*868*	*32*	*23.5*	*6944.0*	*256.1*

Tabla 5.1 Ciclos de trabajo y *speed-up* de una solución *HW* y *SW* para la suma

Multiplier Bram / *# Operations*	*Cycles SW*	*Cycles HW*	*Cycles / operation SW*	*Cycles / operation HW*	*Speed-up*	*Delay (ns) SW*	*Delay (ns) HW*
256	*404238*	*9486*	*1579*	*37*	*42.6*	*12632.4*	*296.4*
512	*808462*	*18958*	*1579*	*37*	*42.6*	*12632.2*	*296.2*
1024	*1616910*	*37902*	*1579*	*37*	*42.7*	*12632.1*	*296.1*
2048	*3233806*	*75790*	*1579*	*37*	*42.7*	*12632.1*	*296.1*
4096	*6467598*	*151566*	*1579*	*37*	*42.7*	*12632.0*	*296.0*

Tabla 5.2 Ciclos de trabajo y *speed-up* de una solución *HW* y *SW* para la multiplicación

En ambas tablas se confirma la buena performance que presenta el uso de coprocesadores para acelerar sistemas, vemos mayor *speed-up* en la multiplicación que resultaría una herramienta potente cuando se requiere procesar gran cantidad de multiplicaciones. Recordemos que la suma y la multiplicación *HW* analizada en los capítulos anteriores presentan un delay de 8 y 13 ciclos respectivamente.

Observación: La propuesta *HW* del sumador explicado en los anteriores capítulos presenta una latencia de 8 ciclos, pero el reporte del coprocesador establece 32 ciclos de reloj. Nosotros esperábamos alrededor de 21 ó 25 ciclos de reloj por el motivo de considerar el siguiente análisis de tiempos en la tabla 5.3:

Proceso	*Cycles*
Write First Operand nputfsl	*2*
Write Second Operand nputfsl	*2*
Read First Operand from FSL	*2*
Read Second Operand from FSL	*2*
Operation (adder / multiplier)	*N*
Write to FSL	*2*
Exist data in the FIFO	*1*
Read result ngetfsl	*2*

Tabla 5.3 Ciclos de reloj utilizados por el *BUS FSL*

Con este análisis se esperaba un delay de 13+N, en caso de la suma N=8 ciclos, que debería ser 21 ciclos para caso de la suma. Se obtuvo 32 ciclos lo que demuestra un exceso de 11 ciclos, una alternativa para analizar esta situación fue revisar el proceso de *disassembler* de cada instrucción como se muestra en la figura 5.6. Analizando la operación suma, el exceso de ciclos se puede justificar si observamos el *disassembler* de las funciones *ngetfsl()* y *nputfsl()*, éstas funciones generan instrucciones como *nput* (comunicación directa con el *FSL*) y aparte las instrucciones para acceso de memoria *imm* y *lwi*. Las 2 instrucciones últimas son las que producen los ciclos en exceso, una solución sería ejecutar el código en *assembler* e inicializar los registros (instrucción *imm*) al inicio de la programación y no en cada función *ngetfsl* o *nputfsl*.

Para el caso de la multiplicación el análisis es el mismo.

```
78  start_time=XTmrCtr_mGetTimerCounterReg(XPAR_XPS_TIMER_0_BASEADDR, 0);
-   0x2c48    <main+268>:  imm    0
-   0x2c4c    <main+272>:  lbui   r11, r0, -14216     //0xc878 <XTmrCtr_Offsets>
-   0x2c50    <main+276>:  addk   r7, r0, r0
-   0x2c54    <main+280>:  addk   r8, r7, r0
-   0x2c58    <main+284>:  imm    -31808
-   0x2c5c    <main+288>:  addik  r3, r11, 8
-   0x2c60    <main+292>:  lwi    r12, r3, 0
-   0x2c64    <main+296>:  imm    0
-   0x2c68    <main+300>:  addik  r10, r0, -10744     // 0xd608 <hw_result>
-   0x2c6c    <main+304>:  imm    0
-   0x2c70    <main+308>:  addik  r9, r0, -8688       // 0xde10 <sample+8>
79  for (n = 0; n < 2*SAMPLES; n=n+2)
-   0x2cc0    <main+388>:  addik  r3, r7, 2
-   0x2cc4    <main+392>:  imm    0
-   0x2cc8    <main+396>:  andi   r7, r3, -1
-   0x2cd0    <main+404>:  xori   r18, r7, 512
-   0x2cd4    <main+408>:  bneid  r18, -96            // 0x2c74 <main+312>
-   0x2cd8    <main+412>:  addik  r8, r8, 16
80     {
81           nputfsl((&sample[n])->words[0],0);
-   0x2c74    <main+312>:  imm    0
-   0x2c78    <main+316>:  lwi    r3, r8, -8696
-   0x2c7c    <main+320>:  nput   r3, rfsl0
82           nputfsl((&sample[n])->words[1],0);
-   0x2c80    <main+324>:  imm    0
-   0x2c84    <main+328>:  lwi    r4, r8, -8692
-   0x2c88    <main+332>:  nput   r4, rfsl0
83           nputfsl((&sample[n+1])->words[0],0);
```

```
-   0x2c8c    <main+336>:  imm    0
-   0x2c90    <main+340>:  lwi    r3, r8, -8688
-   0x2c94    <main+344>:  nput   r3, rfsl0
84             nputfsl((&sample[n+1])->words[1],0);
-0x2c98       <main+348>:  addk   r5, r8, r9
-0x2c9c       <main+352>:  lwi    r3, r5, 4
-0x2ca0       <main+356>:  nput   r3, rfsl0
85             ngetfsl((&hw_result[n/2])->words[0],0);
-0x2ca4       <main+360>:  srl    r4, r7
-0x2ca8       <main+364>:  muli   r4, r4, 8
-0x2cac       <main+368>:  addk   r6, r4, r10
-0x2cb0       <main+372>:  nget   r3, rfsl0
-0x2cb4       <main+376>:  imm    0
-0x2cb8       <main+380>:  swi    r3, r4, -10744
86             ngetfsl((&hw_result[n/2])->words[1],0);
-0x2cbc       <main+384>:  nget   r5, rfsl0
-0x2ccc       <main+400>:  swi    r5, r6, 4
87             }
88
               end_time=XTmrCtr_mGetTimerCounterReg(XPAR_XPS_TIMER_0_BASEADDR,0)
;
```

Figura 5.6 Proceso de *disassembler* del programa utilizado

Como pruebas adicionales implementamos una arquitectura con memorias *Cache* de datos e instrucciones de 16K y 32K. Tuvimos que ejecutar el sistema sobre una memoria externa, por eso consideramos una *DDRAM* de *64M*, nos interesa ver la aceleración que logramos tanto en las soluciones *HW* como *SW*. Los resultados son mostrados en la tabla 5.4, para las pruebas consideramos 256, 512, 1024, 2048 y 4096 operaciones.

Adder DDram 64M	***No cache I-D***		***Cache I-D 16K***				***Cache I-D 32K***			
# Op.	***Cycles SW***	***Cycles HW***	***Cycles SW***	***Cycles HW***	***Speed-up SW***	***Speed-up HW***	***Cycles SW***	***Cycles HW***	***Speed-up SW***	***Speed-up HW***
256	*4,563,911*	*162,379*	*365,499*	*17,803*	*12.5*	*9.1*	*365,393*	*17,797*	*12.5*	*9.1*
512	*9,135,432*	*324,619*	*725,545*	*35,316*	*12.6*	*9.2*	*725,407*	*35,335*	*12.6*	*9.2*
1024	*18,270,862*	*649,099*	*1,446,588*	*70,390*	*12.6*	*9.2*	*1,445,431*	*70,387*	*12.6*	*9.2*
2048	*36,541,582*	*1,298,044*	*2,888,274*	*140,522*	*12.7*	*9.2*	*2,887,100*	*140,524*	*12.7*	*9.2*
4096	*73,083,034*	*2,595,963*	*5,771,383*	*280,792*	*12.7*	*9.2*	*5,768,899*	*280,787*	*12.7*	*9.2*

Tabla 5.4 Ciclos de trabajo de la suma usando memorias *Cache*

La tabla 5.4 muestra una versión del sumador sobre una arquitectura sin *Cache* y con Cache, para estos diseños mostramos la latencia en ciclos tanto para la solución *SW* como *HW*, la

aceleración en solución *SW* es ligeramente que la obtenida en *HW*, esto confirma el impacto de las *Cache* sobre el *SW*. Las aceleraciones se mantienen al considerar *Cache* de 16K y 32K.

En la tabla 5.5 muestra la versión de la multiplicación sobre una arquitectura sin *Cache* y con Cache, se muestra la latencia en ciclos para la solución *SW* como *HW*, el análisis es el mismo del caso anterior y también se confirma el impacto de las *Cache* sobre la solución *SW*. Observamos que la aceleración que ofrece la *Cache* en la multiplicación es mayor a la operación suma y se puede deber a que la multiplicación se basa en sumas y operaciones frecuentes..

Multiplier Ddram 64M	***No cache I-D***		***Cache I-D 16K***				***Cache I-D 32K***			
# Op.	***Cycles SW***	***Cycles HW***	***Cycles SW***	***Cycles HW***	***Speed-up SW***	***Speed-up HW***	***Cycles SW***	***Cycles HW***	***Speed-up SW***	***Speed-up HW***
256	*8,948,303*	*162,370*	*518,012*	*17,769*	*17.3*	*9.1*	*518,082*	*17,781*	*17.3*	*9.1*
512	*17,896,408*	*324,620*	*1,029,125*	*35,314*	*17.4*	*9.2*	*1,029,063*	*35,316*	*17.4*	*9.2*
1024	*35,792,781*	*649,102*	*2,051,826*	*70,399*	*17.4*	*9.2*	*2,051,328*	*70,384*	*17.4*	*9.2*
2048	*71,585,392*	*1,298,057*	*4,096,666*	*140,506*	*17.5*	*9.2*	*4,096,129*	*140,523*	*17.5*	*9.2*
4096	*143,170,701*	*2,595,979*	*8,186,407*	*280,807*	*17.5*	*9.2*	*8,185,207*	*280,798*	*17.5*	*9.2*

Tabla 5.5 Ciclos de trabajo de la multiplicación usando memorias *Cache*

La figura 5.7 muestra la solución *SW* de la aplicación, la cual consta de 4 multiplicaciones y 5 sumas, la variable *Number_of_samples* representa la cantidad de llamadas a procesar. Estos datos son tiempos de duración por llamada y son números generados aleatoriamente en formato *Decimal64*. La multiplicación la realiza la función *decDoubleMultiply* y la suma la *decDoubleAdd*. Consideramos 256, 512, 1024, 22048 y 4096 números de llamadas a procesar. Algo importante de mencionar es la cantidad de operaciones a realizar, esto lo podemos ver en la tabla 5.6

```
/*****************************************************************************/
/*********************************SW IMPLEMENTATION***************************/
/*****************************************************************************/
xil_printf("Executing invoicing in SW...\n\r");
start_time = XTmrCtr_GetTimerCounterReg(XPAR_XPS_TIMER_0_BASEADDR, 0);
for (n=0; n <  NUMBER_OF_SAMPLES ; n++)
 {  if (calltype[n] == 0) decDoubleMultiply( &p, &baserate, &sample[n], &set); // p=r[c]*n
     decDoubleMultiply( &p, &distrate, &sample[n], &set);
     decDoubleMultiply( &b, &p, &basetax, &set);          // b=p*0.0675
     decDoubleAdd( &sumB, &sumB, &b, &set);               // sumB=sumB+b
     decDoubleAdd( &t, &p, &b, &set);                     // t=p+b
    if (calltype[n] != 0)
          {  decDoubleMultiply( &d, &p, &disttax, &set);  // d=p*0.0341
             decDoubleAdd( &sumD, &sumD, &d, &set);       // sumD=sumD+d
             decDoubleAdd( &t, &t, &d, &set);             // t=t+d
          }
    decDoubleAdd( &sumT, &sumT, &t, &set);            // sumT=sumT+t
 }
end_time = XTmrCtr_GetTimerCounterReg(XPAR_XPS_TIMER_0_BASEADDR,0);
xil_printf("elapsed cycles SW:  %d SW\n\r",(end_time-start_time));
```

Figura 5.7 Solución *SW* para la aplicación *Telephone Billing*

La figura 5.8 muestra la solución *HW* de la aplicación, se observa comentadas las funciones en C++ de la solución *SW* las cuáles son reemplazadas por las funciones de escritura y lectura al

bus *FSL*, recordar que trabajamos con operandos de 64 bits por eso enviamos los primeros 32 bits y luego los restantes.

```
  /*************************************************************************
  /*********************************HW IMPLEMENTATION***********************
  /*************************************************************************
  xil_printf("Executing invoicing in HW...\n\r");
  start_time = XTmrCtr_GetTimerCounterReg(XPAR_XPS_TIMER_0_BASEADDR, 0);
  for (n=0; n <  NUMBER_OF_SAMPLES ; n++)
   {  if (calltype[n] == 0)
            {      //decDoubleMultiply( &p, &baserate, &sample[n], &set);
                   nputfsl((&baserate)->words[0],1);
                   nputfsl((&baserate)->words[1],1);
                   nputfsl((&sample[n])->words[0],1);
                   nputfsl((&sample[n])->words[1],1);
                   ngetfsl((&p_hw)->words[0],1);
                   ngetfsl((&p_hw)->words[1],1);
            }
//     decDoubleMultiply( &p, &distrate, &sample[n], &set);
       nputfsl((&distrate)->words[0],1);
       nputfsl((&distrate)->words[1],1);
       nputfsl((&sample[n])->words[0],1);
       nputfsl((&sample[n])->words[1],1);
       ngetfsl((&p_hw)->words[0],1);
       ngetfsl((&p_hw)->words[1],1);
//     decDoubleMultiply( &b, &p, &basetax, &set);           // b=p*0.0675
       nputfsl((&p_hw)->words[0],1);
       nputfsl((&p_hw)->words[1],1);
       nputfsl((&basetax)->words[0],1);
       nputfsl((&basetax)->words[1],1);
       ngetfsl((&b_hw)->words[0],1);
       ngetfsl((&b_hw)->words[1],1);
//     decDoubleAdd( &sumB, &sumB, &b, &set);                // sumB=sumB+b
       nputfsl((&sumB_hw)->words[0],0);
       nputfsl((&sumB_hw)->words[1],0);
       nputfsl((&b_hw)->words[0],0);
       nputfsl((&b_hw)->words[1],0);
       ngetfsl((&sumB_hw)->words[0],0);
       ngetfsl((&sumB_hw)->words[1],0);
//     decDoubleAdd( &t, &p, &b, &set);                      // t=p+b
       nputfsl((&p_hw)->words[0],0);
       nputfsl((&p_hw)->words[1],0);
       nputfsl((&b_hw)->words[0],0);
       nputfsl((&b_hw)->words[1],0);
       ngetfsl((&t_hw)->words[0],0);
       ngetfsl((&t_hw)->words[1],0);

     if (calltype[n] != 0)
         {
//             decDoubleMultiply( &d, &p, &disttax, &set);  // d=p*0.0341
               nputfsl((&p_hw)->words[0],1);
               nputfsl((&p_hw)->words[1],1);
               nputfsl((&disttax)->words[0],1);
               nputfsl((&disttax)->words[1],1);
               ngetfsl((&d_hw)->words[0],1);
               ngetfsl((&d_hw)->words[1],1);
//             decDoubleAdd( &sumD, &sumD, &d, &set);        // sumD=sumD+d
               nputfsl((&sumD_hw)->words[0],0);
               nputfsl((&sumD_hw)->words[1],0);
               nputfsl((&d_hw)->words[0],0);
               nputfsl((&d_hw)->words[1],0);
               ngetfsl((&sumD_hw)->words[0],0);
               ngetfsl((&sumD_hw)->words[1],0);
//             decDoubleAdd( &t, &t, &d, &set);              // t=t+d
               nputfsl((&t_hw)->words[0],0);
               nputfsl((&t_hw)->words[1],0);
               nputfsl((&d_hw)->words[0],0);
               nputfsl((&d_hw)->words[1],0);
               ngetfsl((&t_hw)->words[0],0);
               ngetfsl((&t_hw)->words[1],0);
         }
//       decDoubleAdd( &sumT, &sumT, &t, &set);              // sumT=sumT+t
      nputfsl((&sumT_hw)->words[0],0);
      nputfsl((&sumT_hw)->words[1],0);
      nputfsl((&t_hw)->words[0],0);
      nputfsl((&t_hw)->words[1],0);
      ngetfsl((&sumT_hw)->words[0],0);
      ngetfsl((&sumT_hw)->words[1],0);
   }
```

Figura 5.8 Solución *HW* para la aplicación *Telephone Billing*

# Calling	# Adder's	# Multiplier's	# Total
256	1,280	1,024	2,304
512	2,560	2,048	4,608
1,024	5,120	4,096	9,216
2,048	10,240	8,192	18,432
4,096	20,480	16,384	36,864

Tabla 5.6 Número de multiplicaciones y sumas en las pruebas realizadas

En la Tabla 5.7 se presenta los ciclos de trabajo de la aplicación *Telephone Billing* implementadas sobre *BRAM*, para calcular la tarifa final consideramos números de 256, 512, 1024, 2048 y 4096 llamadas. Observamos que la aceleración es considerable en comparación con las obtenidas al analizar las operaciones independientemente.

Telephone Billing BRAM / # Calling	Cycles SW	Cycles HW	Cycles / operation SW	Cycles / operation HW	Speed-up	Delay (ns) SW	Delay (ns) HW
256	4,158,240	26,651	16,243	104	156.0	129,945.0	832.8
512	8,339,061	53,275	16,287	104	156.5	130,297.8	832.4
1024	16,685,145	106,523	16,294	104	156.6	130,352.7	832.2
2048	33,378,432	213,019	16,298	104	156.7	130,384.5	832.1
4096	66,757,768	426,011	16,298	104	156.7	130,386.3	832.1

Tabla 5.7 Ciclos de trabajo de aplicación *Telephone Billing*

En la Tabla 5.8 presentamos los ciclos de trabajo de la aplicación *Telephone Billing* implementadas sobre *DDRAM* considerando memorias *Cache* de instrucciones y datos de 16K y 32K. La aceleración *SW* sigue siendo mayor que la aceleración *HW* pero en menor magnitud que las operaciones analizadas por separado, lo cual se esperaba debido a la presencia de más operaciones sumas. Recordar los resultados de la memoria *Cache* en la operación suma mostrada en la tabla 5.4.

Billing, Ddram 64M	No cache I-D		Cache I-D 16K				Cache I-D 32K			
# Call	Cycles SW	Cycles HW	Cycles SW	Cycles HW	Speed-up SW	Speed-up HW	Cycles SW	Cycles HW	Speed-up SW	Speed-up HW
256	88,237,730	464,588	5,440,898	30,720	16.2	15.1	5,373,390	30,705	16.4	15.1
512	176,955,459	928,842	10,887,162	60,650	16.3	15.3	10,732,459	60,657	16.5	15.3
1024	354,108,058	1,857,357	21,988,183	120,580	16.1	15.4	21,684,326	120,456	16.3	15.4
2048	708,389,157	3,714,378	43,459,051	240,610	16.3	15.4	42,875,296	240,162	16.5	15.5
4096	1416719981	7,428,428	86,878,180	480,340	16.3	15.5	85,663,490	480,005	16.5	15.5

Tabla 5.8 Ciclos de trabajo de aplicación *Telephone Billing* con memorias *Cache*

Como resumen se muestra en la figura 5.9 el *speed up* al instanciar los coprocesadores en los diseños propuestos de la suma (*Add HW/SW*), la multiplicación (*Mult HW/SW*) y la aplicación *Telephone Billing* (*TB HW/SW*).

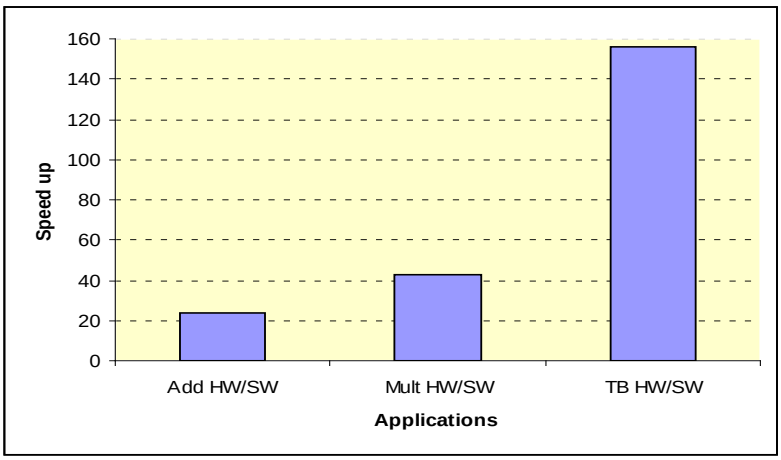

Figura 5.9 *Speed up* de las aplicaciones implementadas en *BRAM*

En la figura 5.9 mostramos el *speed up* en los diseños con memorias *Cache* tanto en las soluciones *HW* como *SW*, la operación suma con *Cache* en *SW* y *HW* se denomina en la gráfica como *Add SW Cache* y *Add HW Cache* respectivamente. Lo mismo ocurre con las operaciones de multiplicación y *Telephone Billing (TB).*

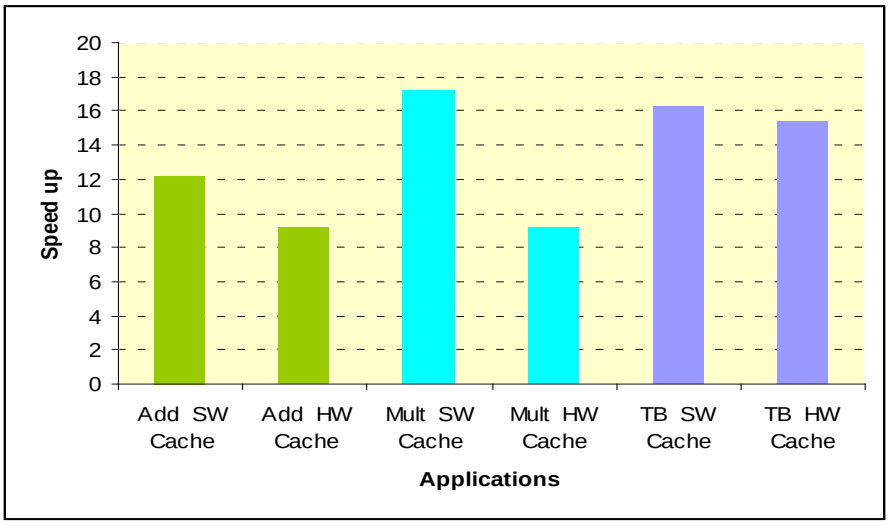

Figura 5.10 *Speed up* de las aplicaciones con *Cache* implementada en *DDRAM*

6 CONCLUSIONES Y TRABAJOS FUTUROS

Como trabajos futuros podemos considerar:

- Estudiar varias alternativas de diseño para mejorar la performance del el diseño, se puede empezar mejorando la performance de los multiplicadores de punto fijo tanto en velocidad como en latencia, esto puede conseguirse adoptando sumadores *carry-save* en lugar que los *carry-chain*.
- Otra manera de mejorar el *speed up* sería modificar el modulo multiplicación en punto fijo reduciendo el números de ciclo de reloj. En cuanto a la operación de suma podemos analizar una posible segmentación en la etapa de suma en punto fijo, lo que podría incrementar la velocidad.
- Desarrollar un hardware reconfigurable con las operaciones aritméticas básicas en punto flotante como la suma, resta, multiplicación y división para propósitos de investigación.
- La solución *HW/SW* de la aplicación de tarifación telefónica puede ser mejorada ejecutando en paralelo operaciones sumas/multiplicaciones.
- Para futuros diseños y según requerimientos de *trade-off delay/area* se pueden considerar arquitecturas con memorias *Cache*, según las tablas mostradas observamos que podemos llegar a una aceleración aceptable.
- Analizar y probar las operaciones diseñadas sobre herramientas matemáticas no convencionales como las utilizadas en procesamiento de señales e imágenes.
- Implementación futura de las alternativas propuestas en tecnología *ASIC´s* para poder comparar la performance con las varias alternativas propuestas en la mayoría de publicaciones.
- Generar un producto final de prestaciones sofisticadas para ser introducido en el mercado.
- Presentación de futuras publicaciones sobre la operación suma/multiplicación y sobre aceleración de algoritmos como el de tarifación telefónica.

Se puede concluir lo siguiente:

- Se confirma la buena performance (*trade-off área/latencia*) del multiplicador al computar la normalización de las mantisas antes de la multiplicación en punto fijo. La comparación se realizó con [18].
- Para obtener una medida aceptable de los ciclos que consume las operaciones *DFP* sobre un *Soft Coprocessor*, fue necesario implementarlos en *BRAM* a pesar de presentar una ligera penalización en latencia. Al ser implementado sobre una memoria externa *DDRAM* surgieron demasiados ciclos innecesarios debido a accesos de memoria

- Podemos argumentar que este trabajo es la primera implementación en *FPGA* de operaciones aritméticas de punto flotante en formato decimal.
- En este trabajo se presentó la investigación y diseños de estructuras novedosas para las operaciones de suma y multiplicación. Las implementaciones pueden considerarse como los primeros aportes en prototipado de diseños sobre *FPGA*, la mayoría de seño propuesto se basan en *ASIC´s*. El diseño del sumador facilitó el desarrollo del multiplicador pues gran parte de componentes fueron reutilizados con pequeñas modificaciones.
- El reporte de la síntesis muestra que el diseño de la operación suma muestra una mayor performance que la multiplicación tanto en área como latencia, y los resultados son comparables con otros diseños.

Aportes

En base a este trabajo se aportan las siguientes investigaciones:

Se realizó la publicación sobre la implementación de una multiplicación en punto flotante decimal, el trabajo fue presentado en el congreso *Reconfigurable Computing and FPGAs 2009.[18]*.

Se ha realizado una segunda publicación sobre la operación suma en punto flotante decimal, la cual está siendo revisada.

Se está escribiendo un tercer trabajo sobre aceleración de operaciones matemáticas de alta precisión basados en los módulos diseñados de suma / multiplicación.

REFERENCIAS

[1] IEEE Standard for Floating-Point Arithmetic, 2008. IEEE Std 754-2008.

[2] Simone Borgio, Davide Bosisio, Fabrizio Ferrandi, Matteo Monchiero, Marco D. Santambrogio, Donatella Sciuto, and Antonino Tumeo. Hardware dwt accelerator for multiprocessor system-on-chip on fpga. In *Proc. Int. Conf. Embedded Computer Systems: Architectures, Modeling and Simulation IC-SAMOS 2006*, pages 107–114, 2006.

[3] F. Y. Busaba, C. A. Krygowski, W. H. Li, E. M. Schwarz, and S. R. Carlough. The IBM z900 decimal arithmetic unit. In *Proc. Conf Signals, Systems and Computers Record of the Thirty-Fifth Asilomar Conf*, volume 2, pages 1335–1339, 2001.

[4] M. S. Cohen, T. E. Hull, and V. C. Hamacher. CADAC: A Controlled-Precision Decimal Arithmetic Unit. (4):370–377, 1983.

[5] M. Cornea, C. Anderson, J. Harrison, P. T. P. Tang, E. Schneider, and C. Tsen. A Software Implementation of the IEEE 754R Decimal Floating-Point Arithmetic Using the Binary Encoding Format, 2007. Computer Arithmetic, 2007. ARITH '07. 18th IEEE Symposium on.

[6] M. Cornea, J. Harrison, C. Anderson, P. Tang, E. Schneider, and E. Gvozdev. A Software Implementation of the IEEE 754R Decimal Floating-Point Arithmetic Using the Binary Encoding Format, 2009. Computers, IEEE Transactions on.

[7] L. Eisen, J. W. Ward, H. W. Tast, N. Mading, J. Leenstra, S. M. Mueller, C. Jacobi, J. Preiss, E. M. Schwarz, and S. R. Carlough. IBM POWER6 accelerators: VMX and DFU. In *IBM Journal of Research and Development*, volume 51, pages 663–683, 2007.

[8] M. A. Erle, B. J. Hickmann, and M. J. Schulte. Decimal Floating-Point Multiplication. 58(7):902–916, 2009.

[9] A. Farmahini-Farahani, C. Tsen, and K. Compton. FPGA implementation of a 64-Bit BID-based decimal floating-point adder/subtractor. In *Proc. Int. Conf. Field-Programmable Technology FPT 2009*, pages 518–521, 2009.

[10] H. H. Goldstine and A. Goldstine. The Electronic Numerical Integrator and Computer (ENIAC). 18(1):10–16, 1996.

[11] K. Goossens, M. Bennebroek, Jae Young Hur, and M. A. Wahlah. Hardwired Networks on Chip in FPGAs to Unify Functional and Configuration Interconnects. In *Proc. Second ACM/IEEE Int. Symp. Networks-on-Chip NoCS 2008*, pages 45–54, 2008.

[12] B. Hickmann, A. Krioukov, M. Schulte, and M. Erle. A parallel IEEE P754 decimal floating-point multiplier, 2007. Computer Design, 2007. ICCD 2007. 25th International Conference on.

[13] Hans Holten-Lund. Embedded 3D Graphics Core for FPGA-based System-on-Chip Applications. september 2005.

[14] IBM UK Laboratories. *The decNumber C library*, v3.68 edition, January 2010.

[15] ISO Standards, JTC 1/SC 22. *ISO 1989:2002 Programming Languages - COBOL*, 2002.

[16] Sait Izmit. Floating-Point Support for Embedded FPGA Platform with 6502 Soft-Processor.

[17] Microsoft Corporation. *Library Visual Basic 6.0 Reference*, Jun 2002.

[18] C. Minchola and G. Sutter. A FPGA IEEE-754-2008 Decimal64 Floating-Point Multiplier. In *Proc. Int. Conf. Reconfigurable Computing and FPGAs ReConFig '09*, pages 59–64, 2009.

[19] H. C. Neto and M. P. Vestias. Decimal multiplier on FPGA using embedded binary multipliers. In *Proc. Int. Conf. Field Programmable Logic and Applications FPL 2008*, pages 197–202, 2008.

[20] R. Raafat, A. M. Abdel-Majeed, R. Samy, T. ElDeeb, Y. Farouk, M. Elkhouly, and H. A. H. Fahmy. A decimal fully parallel and pipelined floating point multiplier. In *Proc. 42nd Asilomar Conf. Signals, Systems and Computers*, pages 1800–1804, 2008.

[21] Hans-Peter Rosinger. Connecting Customized IP to the MicroBlaze Soft Processor Using the Fast Simplex Link (FSL) Channel. May 2004.

[22] Standard ECMA-334. *C lenguaje specification*, v4 edition, Jun 2006.

[23] Sun Corp. on-line documentation. *Class BigDecimal*, v1.4.2 edition.

[24] G. Sutter, E. Todorovich, G. Bioul, M. Vazquez, and J.-P. Deschamps. FPGA Implementations of BCD Multipliers. In *Proc. Int. Conf. Reconfigurable Computing and FPGAs ReConFig '09*, pages 36–41, 2009.

[25] J. Thompson, N. Karra, and M. J. Schulte. A 64-bit decimal floating-point adder. In *Proc. IEEE Computer society Annual Symp. VLSI*, pages 297–298, 2004.

[26] C. Tsen, S. Gonzalez-Navarro, and M. Schulte. Hardware design of a Binary Integer Decimal-based floating-point adder, 2007. Computer Design, 2007. ICCD 2007. 25th International Conference on.

[27] A. Vazquez and E. Antelo. A High-Performance Significand BCD Adder with IEEE 754-2008 Decimal Rounding. In *Proc. 19th IEEE Symp. Computer Arithmetic ARITH 2009*, pages 135–144, 2009.

[28] M. Vazquez, G. Sutter, G. Bioul, and J. P. Deschamps. Decimal Adders/Subtractors in FPGA: Efficient 6-input LUT Implementations. In *Proc. Int. Conf. Reconfigurable Computing and FPGAs ReConFig '09*, pages 42–47, 2009.

[29] W3C Recommendation 28 October 2004. *XML Scheme Part 2: Datatypes Second Edition*, Octuber 2004.

[30] Liang-Kai Wang and M. J. Schulte. Decimal Floating-Point Adder and Multifunction Unit with Injection-Based Rounding. In *Proc. 18th IEEE Symp. Computer Arithmetic ARITH '07*, pages 56–68, 2007.

[31] C. F. Webb. Ibm z10: The next-generation mainframe microprocessor. 28(2):19–29, 2008.

[32] Xilinx Inc. *Xilinx Inc. Xilinx ISE Design Suite 12.1 Software Manuals*, v12.1 edition, June 2009.

[33] Xilinx Inc. *Xilinx Inc. XST User Guide 12.1*, v12.1 edition, June 2009.

[34] Xilinx Inc. *Virtex-5 FPGA Data Sheet: DC and Switching Characteristics (DS202)*, v5.3 edition, May 5 2010.