

Diseño Digital Avanzado

Unidad 4 - Bloques en FPGA y VLSI

Dr. Ariel L. Pola

apola@fundacionfulgor.org.ar

October 11, 2022

Tabla de Contenidos

1. Contenidos Temáticos
2. Proc. Emb. y Unid. aritméticas en FPGAs
3. Instanciación de bloques embebidos
4. Sumadores
5. División con Barrel Shifter
6. Sumadores Carry Save y Compresores
7. Multiplicadores Paralelos
8. Mult. Signado en Completo a Dos
9. Reducción del Árbol de Suma
10. Trasformaciones de Algoritmos para CSA
11. Multiplicación por Constantes
12. Arquitecturas Dedicadas de Filtros FIR
13. Aritmética Distribuida

Contenidos Temáticos



Presentación del Curso

Contenidos Temáticos

Unidad 4 Bloques Básicos de Diseño en FPGA y VLSI

- Procesadores embebidos y unidades aritméticas en FPGAs.
- Instanciación de los mismos.
- Mapeo óptimo para una tecnología dada.
- Bloques básicos de diseño en ASICs: Sumadores básicos. Half y Full adder. Ripple carry adder.
- Sumadores rápidos: Carry Look-ahead Adder, Hybrid Ripple Carry and Carry Look-ahead Adder, Binary Carry Look-ahead Adder, Carry Skip Adder, Conditional Sum Adder, Carry Select Adder, Hybrid Adders.
- División con Barrel Shifters.
- Sumadores Carry Save (CSA) y Compresores.
- Multiplicadores paralelos.
- Generación de productos parciales.
- Reducción de productos parciales.
- Multiplicadores seccionados.
- Optimización de Compresores.
- Contadores de uno o múltiples columnas.
- Multiplicadores signados en complemento a dos.
- Eliminación de extensión de signo.

Presentación del Curso

Contenidos Temáticos

Unidad 4 Bloques Básicos de Diseño en FPGA y VLSI

- Propiedad de cadena.
- Multiplicador modificado de Booth.
- Árboles de compresión para sumas multi-operando.
- Algoritmos para transformar CSA.
- Multiplicación por constantes.
- Representación canónica de dígito signado.
- Arquitecturas dedicadas de filtros FIR en forma directa, transpuesta e híbrida.
- Aritmética distribuida.

Proc. Emb. y Unid. aritméticas en FPGAs



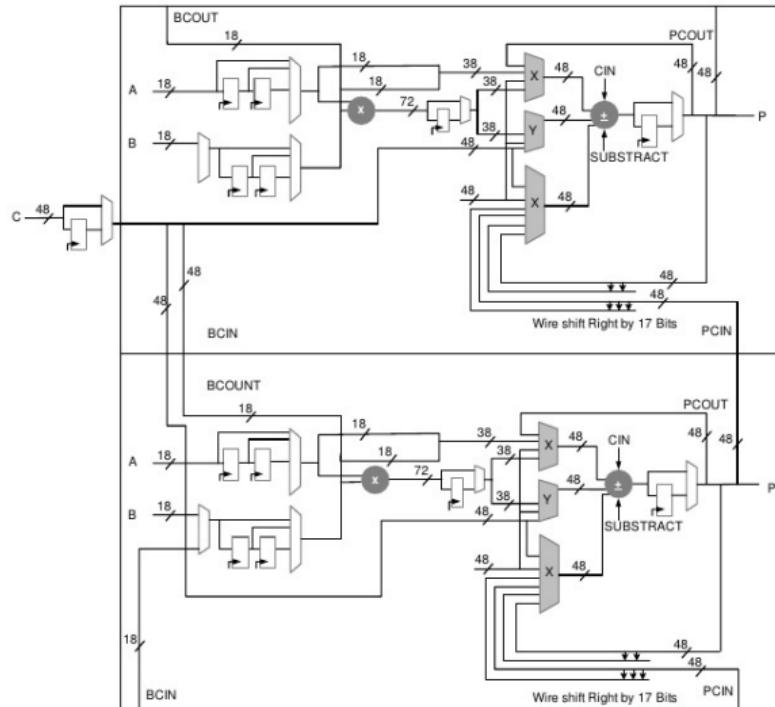
Introducción

- Las FPGAs han surgido como dispositivos que permiten desarrollar aplicaciones de procesamiento de señales de alto rendimiento.
- En este campo de aplicación, los FPGA han superado a la tradicional tecnología de procesadores de señal *DSP*. Sin importar cuantos MACs pueda colocar el proveedor de DSP en un chip, esto no compite contra los cientos de estas unidades que pueden ser colocadas en un dispositivo FPGA de alto rendimiento.
- En la actualidad, los FPGA cuentan con procesadores incorporados, interfaces estándar y bloques de procesamiento de señales constituidos por multiplicadores, sumadores, registros y multiplexores. Diferentes dispositivos de una misma familia poseen un gran número de estas unidades embebidas, y con el tiempo se espera que el número de las mismas en un mismo dispositivo siga creciendo.

Herramientas para el diseño en FPGA

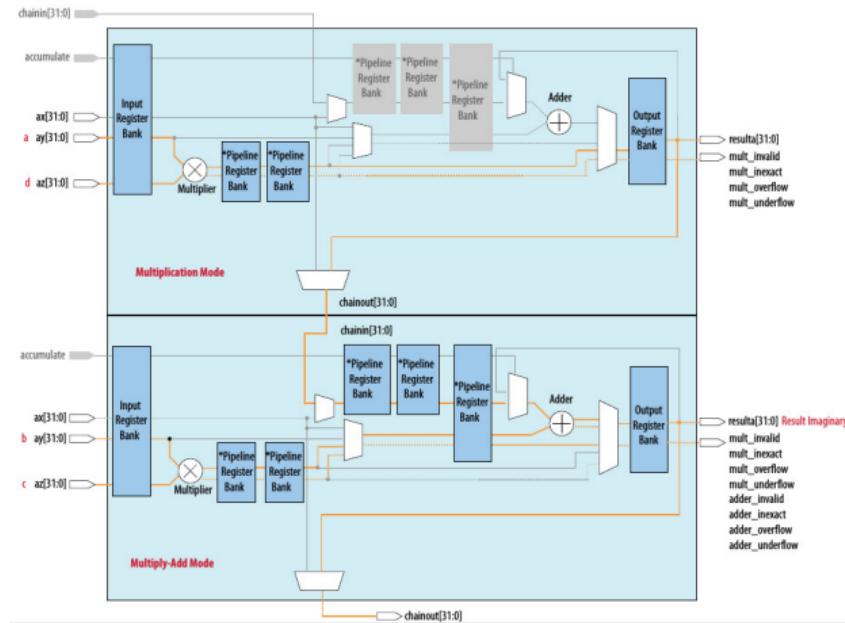
- La mayoría de las herramientas de síntesis disponibles instancian bloques embebidos a partir del código HDL (*lenguaje de descripción de hardware*) que contiene operadores matemáticos y lógicos relacionados a estos bloques.
- El usuario puede configurar distintas opciones de síntesis para el diseño. Las herramientas cuentan con bloques básicos de construcción. En casos en donde se tienen operaciones de orden superior, la herramienta hará múltiples copias y combinaciones de estos bloques básicos para lograr la funcionalidad deseada.
- Por ejemplo si se diseña un multiplicador de 32 bits por 32 bits, la herramienta colocará cuatro multiplicadores de 18 bits por 18 bits para lograr satisfacer el diseño deseado.

Proc. Emb. y Unid. aritméticas en FPGAs



Bloques DSP en FPGAs Xilinx.

Proc. Emb. y Unid. aritméticas en FPGAs



Bloques DSP en FPGAs Altera.

Herramientas para el diseño en FPGA

- El usuario también podrá instanciar estos bloques básicos de forma explícita si el diseño así lo requiriera.
- El fabricante provee en la herramienta de diseño plantillas que contienen estos bloques para que puedan ser utilizados por el diseñador.
- Estos bloques básicos son bloques embebidos en el dispositivo.
- En caso que el FPGA se quedara sin recursos embebidos construirá los bloques utilizando componentes de lógica genérica contenidos en el dispositivo FPGA.
- Estos bloques generados funcionan con menos eficiencia que los bloques embebidos.
- Estos recursos embebidos han generado un gran salto en el rendimiento de las implementaciones.

Instanciación de bloques embebidos



Instanciación de bloques embebidos

Implementación Filtro IIR

Ejemplo: Filtro IIR de segundo orden

- Para demostrar la efectividad de los bloques embebidos en el diseño, se muestra la realización de un filtro de segundo orden con respuesta infinita al impulso (*Filtro IIR*).
- La ecuación en diferencia del filtro es

$$w[n] = a_1 w[n-1] + a_2 w[n-2] + x[n] \quad (1)$$

$$y[n] = b_0 w[n] + b_1 w[n-1] + b_2 w[n-2] \quad (2)$$

Instanciación de bloques embebidos

Filtro IIR

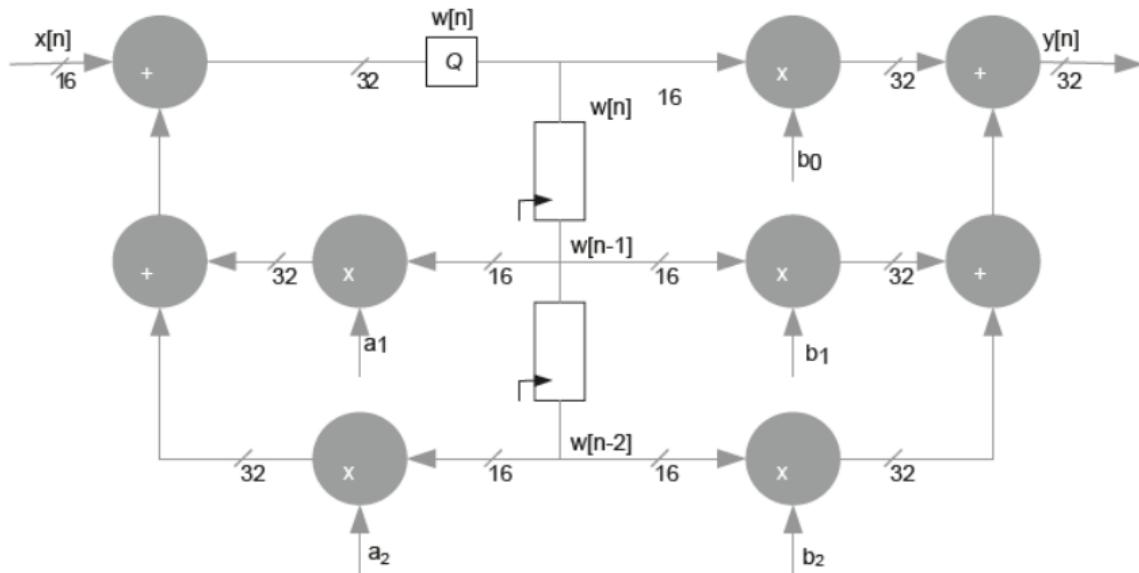


Diagrama en bloque realizado en forma directa

Instanciación de bloques embebidos

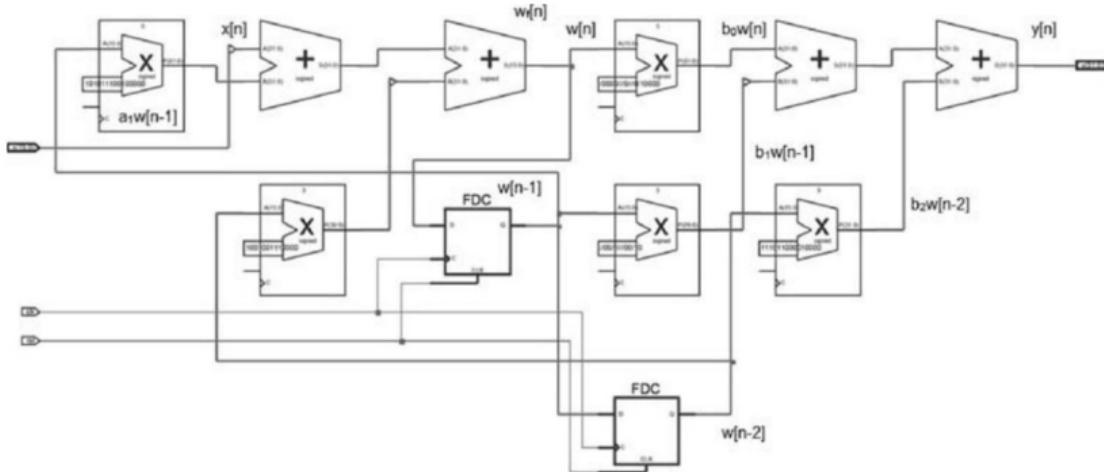
Comparación entre dos tecnologías

Spartan 3

- Para demostrar la utilidad del uso de bloques embebidos en FPGA, el código se sintetizó para una FPGA Spartan 3 y para una Virtex 4.
- Los dispositivos Spartan 3 vienen con bloques multiplicadores embebidos de 18 bits por 18 bits. En las opciones de síntesis se ha seleccionado utilizar los bloques multiplicadores embebidos. En el reporte correspondiente a esta síntesis se puede ver que la herramienta ha utilizado los bloques multiplicadores.
- Como el dispositivo Spartan 3 no posee bloques sumadores incorporados, la herramienta crea un sumador de 32 bits utilizando la lógica de acarreo rápido provista en la LUT.

Instanciación de bloques

Comparación entre dos tecnologías



Esquemático RTL - Spartan3.

Instanciación de bloques embebidos

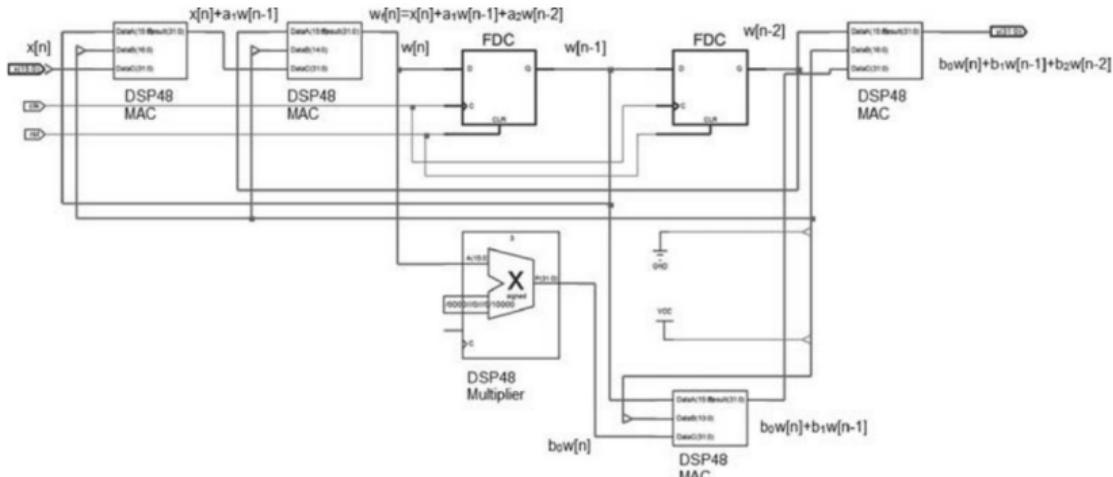
Comparación entre dos tecnologías

Virtex 4

- El diseño fue sintetizado nuevamente para poder observar la eficacia de los bloques DSP48 de la FPGA Virtex 4.
- Como la herramienta esta configurada para utilizar éstos bloques, en el reporte de síntesis se puede observar que se utilizan 5 bloques DSP48 para el diseño.
- La tecnología utilizada por el dispositivo Virtex 4 es claramente superior y provee un mejor timing para un mismo diseño.
- Debe considerarse que el timing de síntesis dado en los reportes anteriores corresponde a un análisis post-síntesis y a pesar de que es una buena estimación, el verdadero timing del diseño proviene de el análisis post place&root.
- Un diseño digital y su implementación RTL-Verilog deberían ser independientes de la tecnología utilizada. Sin embargo en muchos casos es importante conocer la tecnología que se está utilizando, para poder hacer uso de los bloques embebidos con los que cada dispositivo cuenta y de esta manera optimizar la implementación. Podemos notar esto analizando los resultados arrojados por las síntesis usando FPGAs de diferente tecnología.

Instanciación de bloques

Comparación entre dos tecnologías



Esquemático RTL - Virtex4.

Instanciación de bloques

Comparación entre dos tecnologías

Selected device: 3s400pq208-5

Minimum period: 10.917 ns (maximum frequency: 91.597 MHz)

Number of slices:	58 out of 3584	1%
Number of slice flip-flops:	32 out of 7168	0%
Number of 4-input LUTs:	109 out of 7168	1%
Number of IOs:	50	
Number of bonded IOBs:	50 out of 141	35%
Number of multi 18 × 18s:	5 out of 16	31%
Number of GCLKs:	1 out of 8	12%

Reporte de Síntesis - Spartan3

Selected device: 4vlx15sf363-12

Minimum period: 7.664 ns (maximum frequency: 130.484 MHz)

Number of slices:	17 out of 6144	0%
Number of slice flip-flops:	32 out of 12288	0%
Number of 4-input LUTs:	16 out of 12288	0%
Number of IOs:	50	
Number of bonded IOBs:	50 out of 240	20%
Number of GCLKs:	1 out of 32	3%
Number of DSP48s:	5 out of 32	15%

Reporte de Síntesis - Virtex4.

Introducción a los bloques básicos de diseño

Bloques Básicos de Diseño

- Luego de haber analizado el uso de multiplicadores dedicados y bloques MAC (*bloques DSP48*) es importante hacer incapié en los bloques básicos de diseño.
- El objetivo es que el diseñador conozca las diferentes opciones de diseño para algunas de las operaciones matemáticas básicas y pueda comparar y evaluar las ventajas y desventajas de cada una de ellas.
- No importa que tan simple sea el diseño, es muy importante que el diseñador siempre considere varias alternativas y utilice los bloques dedicados.
- Se pueden encontrar muchas opciones de arquitectura para la implementación de operaciones como adición, multiplicación y desplazamiento.

Sumadores



Ripple-carry Adder (RCA)

- Los sumadores se utilizan en adición, substracción, multiplicación y división.
- La velocidad de cualquier diseño digital de un sistema de procesamiento de señales o de comunicaciones depende en gran parte de estas unidades funcionales.
- Los sumadores ripple-carry (*RCA*) son los mas lentos en la familia de los sumadores, implementan el modo tradicional de suma entre dos números.
- En los RCA se suman dos bits los cuales generan un acarreo, este acarreo se sumará a los bits de la siguiente posición (*propagación del acarreo*) los últimos dos bits que se suman generarán un acarreo de salida.
- Si bien el proceso es más lento, su simplicidad permite el uso del menor número de compuertas.

Sumador con acarreo hacia adelante (look-ahead adder)

- Como la propagación del acarreo es lenta, se diseñan sumadores rápidos para poder lograr que el proceso de propagación del acarreo sea más rápida y eficiente.
- Por ejemplo en un sumador con acarreo hacia adelante (*look-ahead adder*) los acarreos de entrada para todas las posiciones de bits son generados simultáneamente por una lógica generadora de acarreo hacia adelante.
- Esto permite al realizar una suma, calcular el acarreo en paralelo sin tener que esperar la propagación del mismo.
- Este tipo de sumador nos permite tener un tiempo de suma constante independientemente de la longitud del sumador.
- Al aumentar la longitud de palabra con que el sumador trabaja, la lógica generadora de acarreo aumenta su complejidad.
- Esto lleva a que los sumadores de grandes longitudes se implementen como varias etapas de sumadores con acarreo hacia adelante de menor complejidad.

Carry Select Adder (CSA)

- Otro dispositivo rápido es el sumador con selección de acarreo (CSA).
- Éste partitiona la suma en K grupos, para ganar velocidad la lógica es replicada y para cada grupo la suma asume acarreo de entrada (*el cual puede ser 0 o 1*).
- La suma y acarreo de salida correctos para cada grupo son seleccionados por el acarreo de salida del grupo previo. El acarreo de salida seleccionado, a su vez, es utilizado para seleccionar la suma y acarreo de salida correctos del siguiente grupo adyacente.
- Para sumar número grandes se utilizan CSA jerárquicos, los cuales dividen la suma en múltiples niveles.

Sumador de suma condicional

- Un sumador de suma condicional puede ser considerado como un CSA con el máximo número posible de niveles.
- La operación de selección de acarreo en el primer nivel es realizada por grupos de 1 bit. En el siguiente nivel dos grupos adyacentes se fusionan para dar el resultado de una operación de selección de acarreo de 2 bits. Esta fusión de dos grupos es repetida hasta que los últimos dos grupos se fusionan para generar la suma final y el acarreo de salida.
- Este sumador es el más rápido de la familia.

Sumadores

Introducción

Resumen

- Es importante destacar que el mapeo de sumadores en FPGA puede no producir los resultados esperados de optimización ya que los dispositivos FPGA poseen bloques embebidos que favorecen el uso de algunos diseños sobre otros.
- Por ejemplo en muchas familias de FPGA una cadena de acarreo rápido ayuda a un RCA, con lo cual en estas FPGA el uso de un RCA será el mejor diseño en cuanto a optimización de área y tiempos.

Half Adders

- Un half adder (*HA*) es un circuito combinacional usado para sumar dos bits, a_i y b_i , sin un acarreo de entrada. La suma s_i y el acarreo de salida c_i están dados por:

$$s_i = a_i \oplus b_i \tag{3}$$

$$c_i = a_i b_i \tag{4}$$

- El path crítico posee una latencia de 1, y corresponde a la longitud de cualquiera de los dos paths (*de cualquiera de las dos operaciones*).

Full Adders

- Un full adder (*FA*) suma 3 bits.
- Un sumador de 3 bits es también llamado compresor 3 : 2.
- Una de las formas de implementar un full adder es usando las siguientes ecuaciones:

$$s_i = a_i \oplus b_i \oplus c_i \quad (5)$$

$$c_{i+1} = (a_i \oplus b_i)c_i + a_i b_i \quad (6)$$

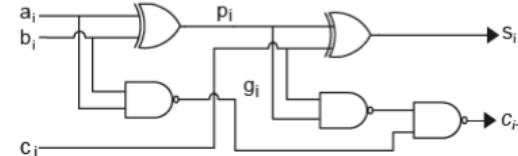
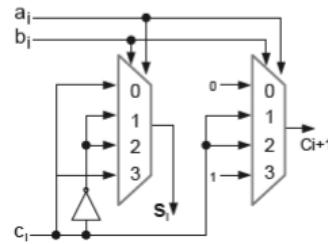
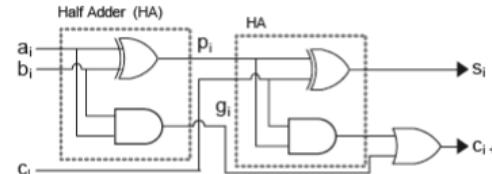
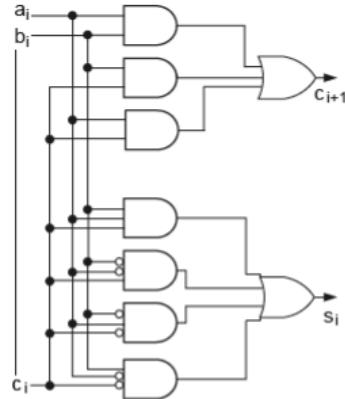
- Existen varios diseños a nivel de compuerta para implementar un full adder.

Full Adders

- Estas opciones para implementar simplemente una suma de 3 bits remarca la idea de que el diseñador debe evitar el modelado a nivel de compuerta o el modelado utilizando operadores bit a bit.
- No es posible que el diseñador conozca cual es el diseño más óptimo si no conoce las librerías de la tecnología que está utilizando.
- En muchos casos es preferible que el diseño sea hecho a nivel RTL independientemente de la tecnología.
- En el caso que el diseñador conozca las librerías de la tecnología que utiliza, el mismo podrá utilizar componentes optimizados por la misma para lograr diseños más eficientes.

Sumadores

Sumadores Básicos



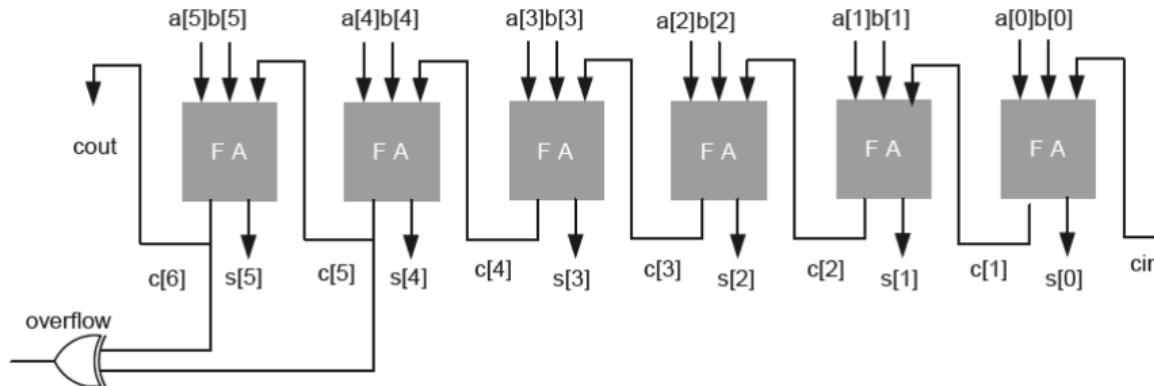
Ejemplos de implementación de Full Adders

Ripple Carry Adders

- Un Ripple Carry Adder (*RCA*) es considerado como el sumador más lento.
- Esto puede NO ser así en el caso que el diseño sea mapeado en un FPGA con lógica de cadena de acarreo embebida.
- Un RCA presenta un área mínima y una estructura regular.
- A los RCA se les puede agregar pipes para mejorar la velocidad de los mismos.
- Un RCA que suma dos operandos de N -bits necesita N full adders.
- La velocidad varía linealmente con la longitud de los operandos.
- Se implementa la forma tradicional de sumar dos números. Los operandos son sumados bit a bit desde los bits menos significativos hasta los bits más significativos, sumando en cada etapa el acarreo proveniente de la etapa anterior. De esta forma, el acarreo de salida de el full adder de la etapa i entra en el full adder de la etapa $(i+1)$, y de esta forma el acarreo se propaga desde los bits menos significativos hacia los bits más significativos (*de aquí el nombre*).

Sumadores

Sumadores Básicos



RCA de 6 bits

Ripple Carry Adders

- La condición de overflow es fácilmente calculada a través de una operación XOR entre los acarreos de salida de los últimos dos full adders, esta condición es mostrada también en la figura.
- El path crítico de retardo de un RCA puede ser calculado con la siguiente ecuación:

$$T_{RCA} = (N - 1)T_{FA} + T_m \quad (7)$$

- Donde T_{RCA} es el retardo del RCA, T_{FA} es el retardo del FA y T_m es la lógica de generación del acarreo.
- De la anterior ecuación se puede ver que el desempeño de un RCA depende de la propagación del acarreo desde el primer full adder hasta el último.

Ripple Carry Adders

- Para mejorar la velocidad muchas FPGAs están cuentan con propagación de acarreo rápida y lógicas de generación de suma.
- Las lógicas de propagación de acarreo permiten caminos rápidos para que el acarreo se propague de un bloque a otro.
- El usuario puede implementar sumadores RCA en paralelo, pero su mapeo en FPGA puede no poseer la velocidad esperada en comparación con el desempeño de un sólo RCA.
- Sin embargo en muchos diseños el usuario encuentra al RCA como el sumador más rápido comparado con otros sumadores en paralelo.

Sumadores

Sumadores Básicos

Implementación en Verilog de un RCA de 16 bits

```
1 module rca
2 #(parameter W=16)
3   (input          clk,
4    input [W-1:0]  a, b,
5    input          cin,
6    output reg [W-1:0] s_r,
7    output reg      cout_r);
8
9   wire [W-1:0]      s;
10  wire              cout;
11  reg  [W-1:0]      a_r, b_r;
12  reg              cin_r;
13  assign {cout,s} = a_r + b_r + cin_r;
14
15 always@(posedge clk)
16 begin
17   a_r    <= a;
18   b_r    <= b;
19   cin_r <= cin;
20   s_r    <= s;
21   cout_r <= cout;
22 end
23 endmodule
```

Sumadores

Sumadores Básicos

Ripple Carry Adders

- Si el código es sintetizado en una FPGA que posee lógica de cadena de acarreo rápido (*como una Virtex II pro*), la herramienta de síntesis inferirá esta lógica para una propagación rápida del acarreo con un sumador RCA.
- Esta FPGA consiste en un número de bloques de lógica configurable (*CLBs*), cada uno de éstos posee cuatro 'slices'. Cada uno de estos slices posee dos tablas (*LUTs*) y lógica dedicada para calcular funciones de generación (g) y propagación (p) del acarreo).
- Estas funciones son utilizadas por la herramienta de síntesis para inferir la lógica de acarreo e implementar un RCA rápido.
- Las ecuaciones de g_i y p_i y la lógica utilizada para generar rápidamente el acarreo de salida c_{i+1} es la siguiente:

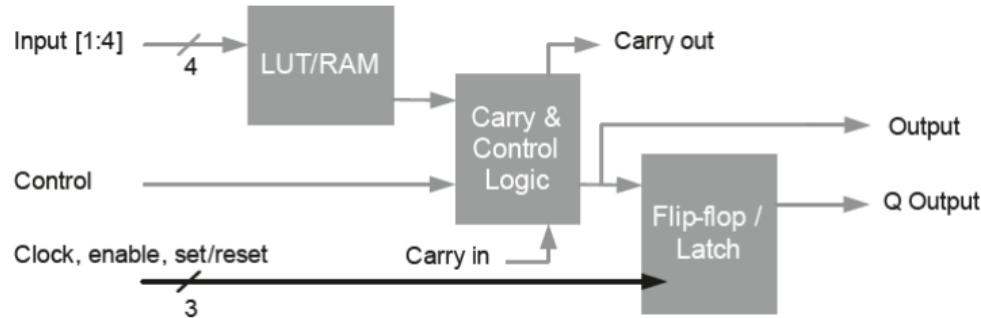
$$c_{i+1} = g_i + p_i c_i \quad (8)$$

$$p_i = a_i \oplus b_i \quad (9)$$

$$g_i = a_i b_i \quad (10)$$

Sumadores

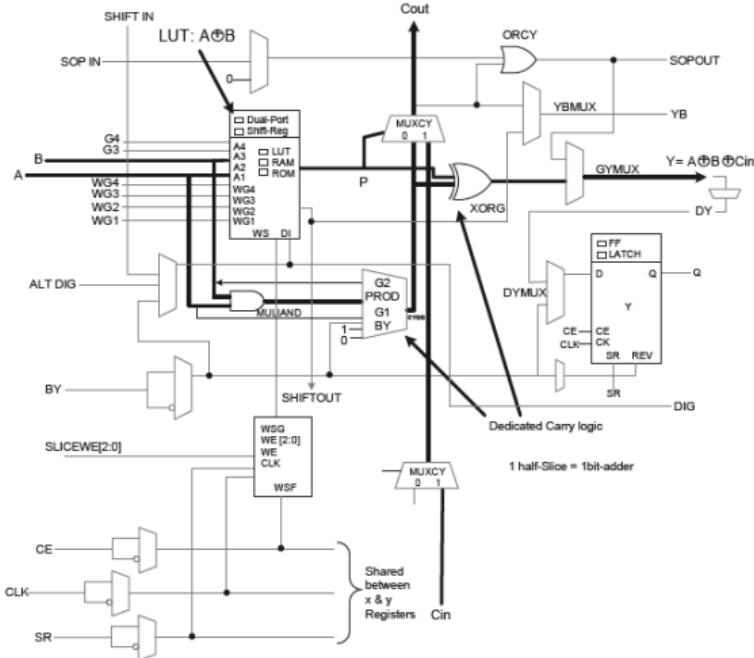
Virtex II pro



Bloques lógicos de acarreo rápido

Sumadores

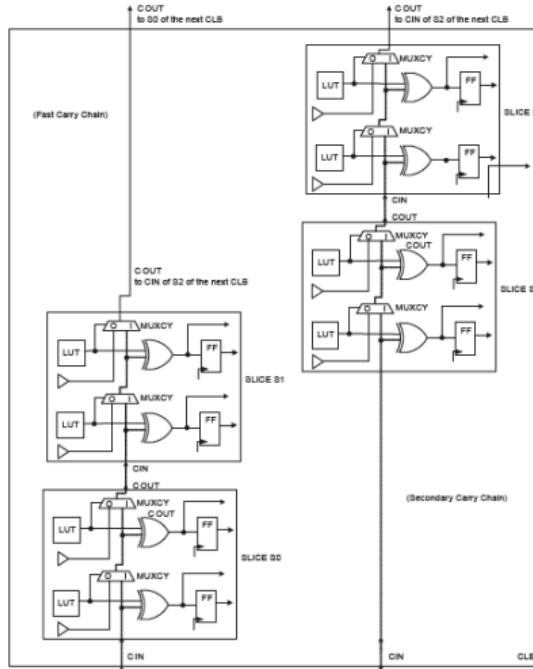
Virtex II pro



Lógica de acarreo rápido en un slice de la FPGA

Sumadores

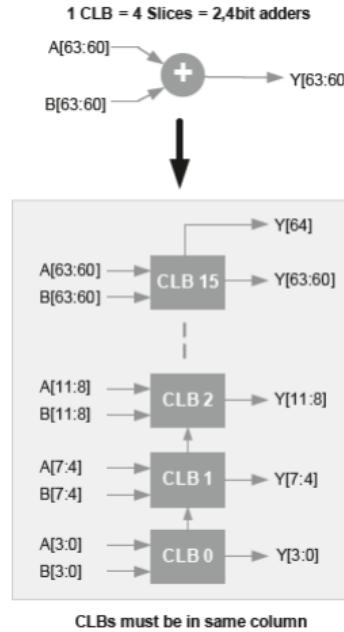
Virtex II pro



CLBs para implementar un sumador de 16 bits

Sumadores

Virtex II pro



RCA de 64 bits usando acarreo rápido

Introducción

- Si un RCA no se mapea en una FPGA con lógica de cadena de acarreo rápida, el mismo suele ser el sumador más lento ya que cada Full Adder requiere para su ejecución el resultado del Full Adder anterior.
- Es por esto que se han desarrollado varias arquitecturas alternativas, las cuales aceleran la generación del acarreo para cada etapa.
- Esta aceleración requiere lógica adicional por lo que el diseñador deberá seleccionar cuidadosamente la arquitectura a utilizar, ya que no todas pueden significar una optimización al ser implementadas en una FPGA.

Carry Look-Ahead Adder

- Haciendo un análisis más profundo de la generación de acarreo se puede llegar a que la misma no tiene que depender de los acarreos anteriores.
- En un carry look-ahead adder (*CLA*) el acarreo de los bits de todas las posiciones del sumador es generado en forma simultánea. Esto significa, que el cálculo del acarreo se realiza en paralelo al cálculo de la suma.
- Esto genera que el tiempo que demora la suma en ejecutarse sea independiente de la longitud del sumador.
- No obstante, a medida que el tamaño de palabra aumenta la distribución del hardware para realizar la adición se hace más compleja.
- Esto nos lleva a que para los sumadores de un gran tamaño sea necesario utilizar dos o tres niveles de bloques CLA.

Carry Look-Ahead Adder

- Una simple consideración de la lógica del full adder muestra que el acarreo c_{i+1} es generado si $a_i = b_i = 1$, y se propaga si a_i o b_i es 1.
- Esto puede observarse en las siguientes ecuaciones:

$$g_i = a_i b_i$$

$$p_i = a_i \oplus b_i$$

$$c_{i+1} = g_i + p_i c_i$$

$$s_i = c_i \oplus p_i$$

- Con esto, una determinada etapa genera acarreo si g_i es verdadero y propaga un acarreo de entrada hacia la siguiente etapa si p_i es verdadero. Con estas ecuaciones el acarreo se puede calcular en paralelo de la siguiente forma:

$$c_1 = g_0 + p_0 c_0$$

$$c_2 = g_1 + p_1 c_1$$

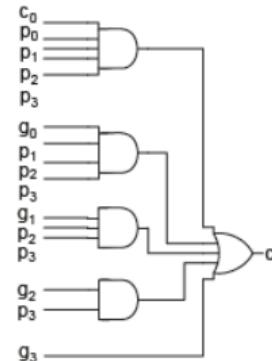
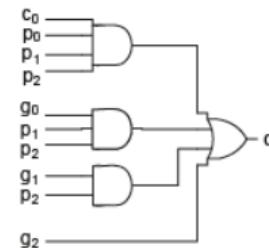
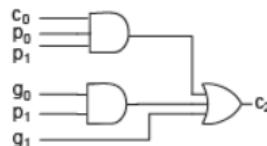
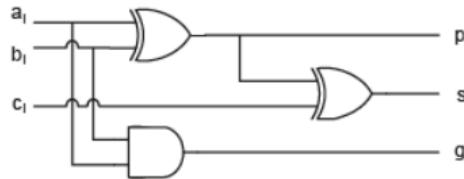
$$c_2 = g_1 + p_1(g_0 + p_0 c_0)$$

$$c_2 = g_1 + p_1 g_0 + p_0 p_1 c_0$$

$$c_3 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$$

Sumadores

Ejemplo de lógica generadora de acarreo



En la figura se puede observar que todos los acarreos pueden calcularse con una latencia de 2 para diferentes arquitecturas del CLA

Carry Look-Ahead Adder

- Podemos generalizar las ecuaciones del acarreo antes vistas de la siguiente forma:

$$c_i = g_{i-1} + \sum_{j=0}^{i-2} \left(\prod_{k=j+1}^{i-1} p_j \right) g_j + \prod_{j=0}^{i-1} p_j c_0$$

- Esto requiere de $i + 1$ compuertas con un fan-in de $i + 1$. Por lo tanto si se incrementa la cantidad de bits se incrementará en gran medida el fan-in de las compuertas.

- Una práctica industrial es usar bloques de 4 bits. Esto limita a que se calcule el acarreo hasta c_3 y c_4 no es calculado. Los 4 primeros términos en c_4 son agrupados como G_0 y el producto $p_3 p_2 p_1 p_0$ en el último término son llamados P_0 .

- Se puede ver la ecuación de c_4 completa y con los términos agrupados a continuación:

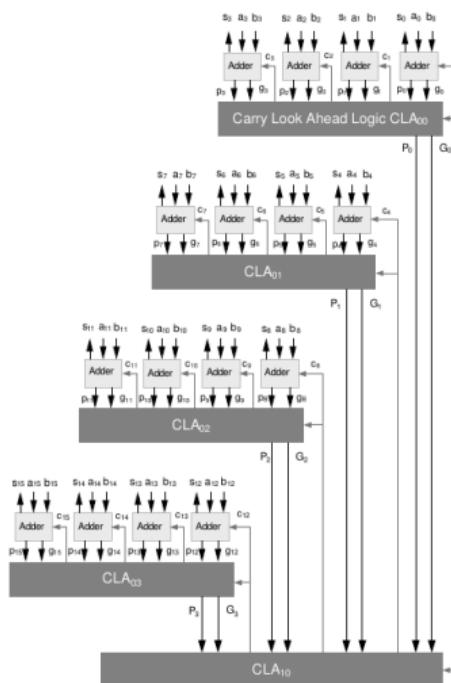
$$c_4 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0$$

$$c_4 = G_0 + P_0 c_0$$

- De manera similar, se agrupan los bits 4 al 7 y se generan c_5 , c_6 y c_7 en el primer nivel del CLA usando c_4 del segundo nivel de la lógica del CLA. A su vez el primer nivel del CLA para éstos bits generará G_1 y P_1 .

Sumadores

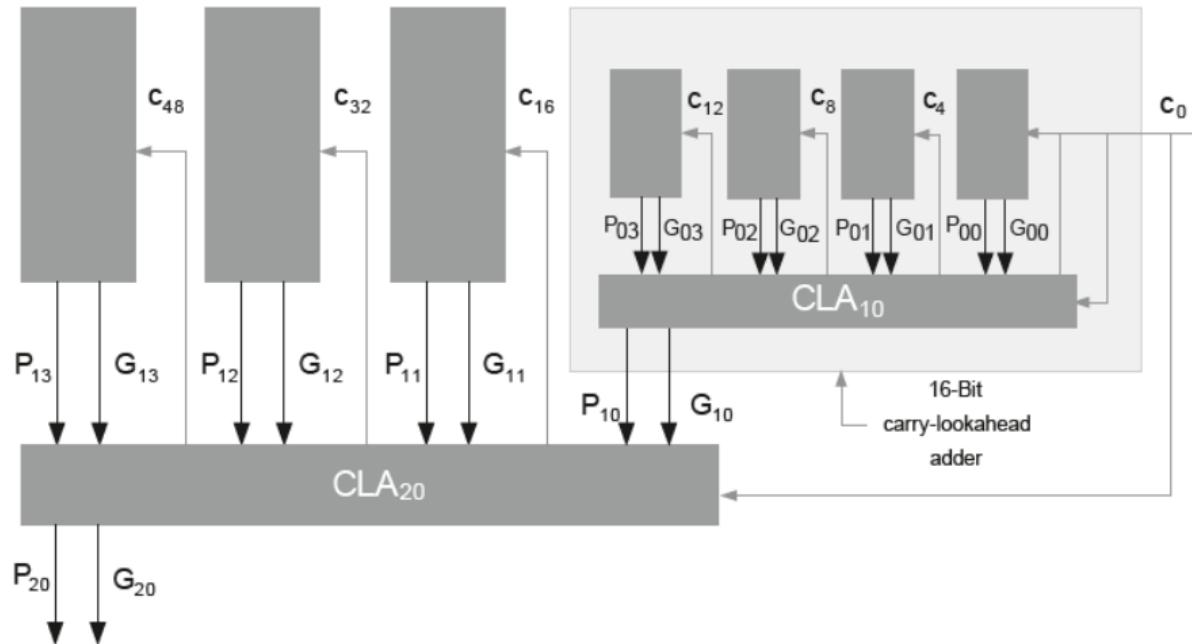
Ejemplo de Carry Look-Ahead Adder



Sumador de 16 bits con 2 niveles de lógica CLA

Sumadores

Ejemplo de Carry Look-Ahead Adder



Sumador de 64 bits con 3 niveles de lógica CLA

Hybrid Ripple Carry y Carry Look-Ahead Adder

- En lugar de construir un gran sumador CLA usando múltiples niveles jerárquicos de lógica CLA, el acarreo puede simplemente propagarse entre los bloques.
- Este tipo de sumador será más rápido que un RCA y ocupará menos área que un sumador jerárquico CLA.
- Esto es debido a que tenemos un sólo nivel de lógica CLA.

Sumadores

Hybrid Ripple Carry y Carry Look-Ahead Adder

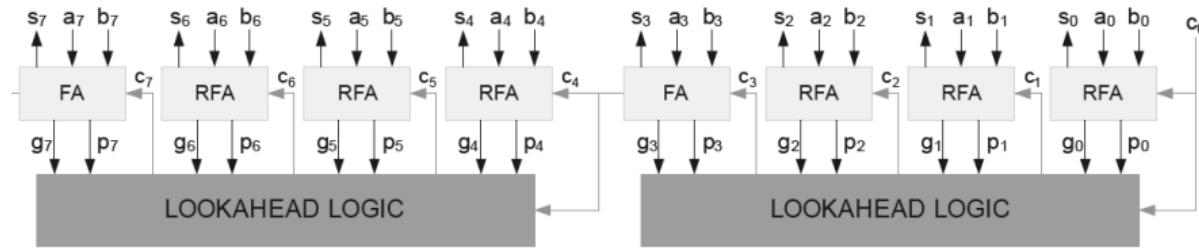


Diagrama en bloque del sumador Híbrido

Binary Carry Look-Ahead Adder

- Un BCLA trabaja con un grupo de dos bits adyacentes, desde el LSB hasta el MSB combinando los bits en grupos de a dos para formar un nuevo grupo de bits y su correspondiente acarreo.

- La lógica de generación de acarreo en un sumador de N bits es:

$$g_i = a_i b_i$$

$$p_i = a_i \oplus b_i$$

$$(G_i, P_i) = (g_i, p_i). (g_{i-1}, p_{i-1}) \dots (g_1, p_1). (g_0, p_0)$$

- El Problema puede ser resuelto en forma recursiva de la siguiente manera:

$$(G_0, P_0) = (g_0, p_0)$$

Para $i = 1$ hasta $N - 1$

$$(G_i, P_i) = (g_i, p_i). (G_{i-1}, P_{i-1})$$

$$c_i = G_i + P_i c_0$$

Binary Carry Look-Ahead Adder

- El operador “.” es interpretado como:

$$(G_i, P_i) = (g_i, p_i) \cdot (G_{i-1}, P_{i-1}) = (g_i + p_i G_{i-1}, p_i P_{i-1})$$

- Hay muchos caminos para implementar estas ecuaciones. El objetivo es realizar esta implementación para cada posición de bit moviéndose desde los LSB a los MSB, esto requerirá una sucesiva aplicación del operador * en las posiciones de dos bits adyacentes.
- Para un sumador de N bits serán necesarios N-1 etapas de implementación de los operadores.

Sumadores

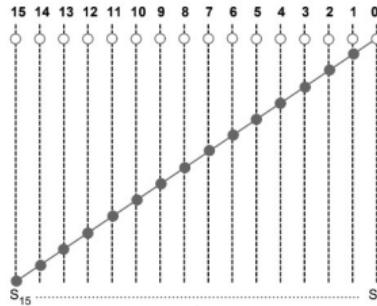
Binary Carry Look-Ahead Adder de 16 Bits

Implementación en Verilog

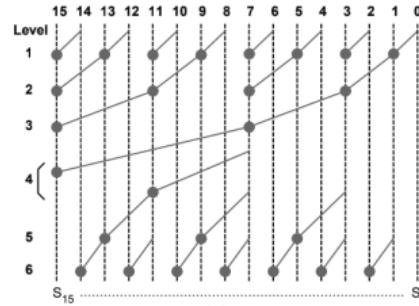
```
1 module bcla
2 # (parameter N = 16)
3   (input      [N-1:0] a,b,
4    input      c_in,  clk,
5    output reg [N-1:0] sum_r,
6    output reg      c_out_r);
7   reg [N-1:0]      a_r,b_r;
8   reg            c_in_r;
9   reg [N-1:0]      p, g, P, G;
10  reg [N:0]        c;
11  reg [N-1:0]      sum;
12  reg            c_out;
13  integer         i;
14  always@(posedge clk) begin
15    c_in_r <= c_in;
16    a_r     <= a;
17    b_r     <= b;
18    c_out_r <= c_out;
19    sum_r   <= sum;
20  end
21  always@(*) begin
22    for (i=0;i<N;i=i+1) begin
23      // Generate all ps and gs
24      p[i]= a_r[i] ^ b_r[i];
25      g[i]= a_r[i] & b_r[i];
26    end
27  end
1   always@(*) begin
2    // Linearly apply dot operators
3    P[0] = p[0];
4    G[0] = g[0];
5    for (i=1; i<N; i=i+1) begin
6      P[i]= p[i] & P[i-1];
7      G[i]= g[i] | (p[i] & G[i-1]);
8    end
9  end
10 always@(*) begin
11   //Generate all carries and sum
12   c[0]=c_in_r;
13   for(i=0;i<N;i=i+1)
14     begin
15       c[i+1] = G[i] | (P[i] & c[0]);
16       sum[i] = p[i] ^ c[i];
17     end
18   c_out = c[N];
19 end
20 endmodule
```

Sumadores

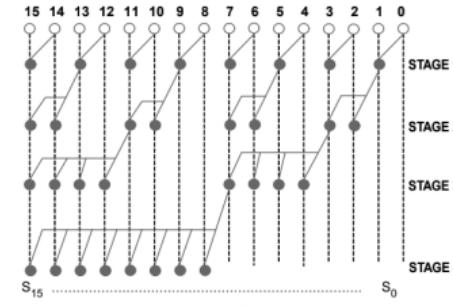
BCLA-Lógica generadora de Carry



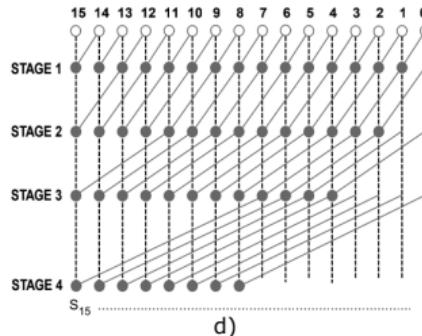
a)



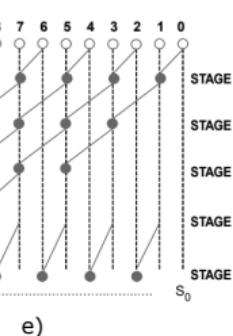
b)



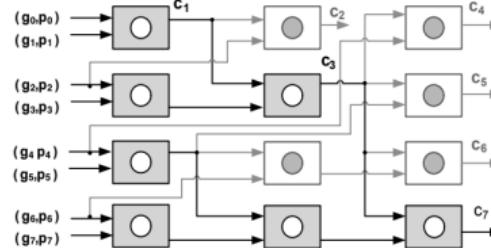
c)



d)



e)



f)

- a) Implementación Serie; b) Sumador de Brent-Kung; c) Sumador de prefijo paralelo de Ladner-Fischer; d) Sumador de prefijo paralelo de Kogge-Stone; e) Sumador de prefijo paralelo de Han-Carlson; f) Layout de un sumador de Brent-Kung

Carry Skip Adder

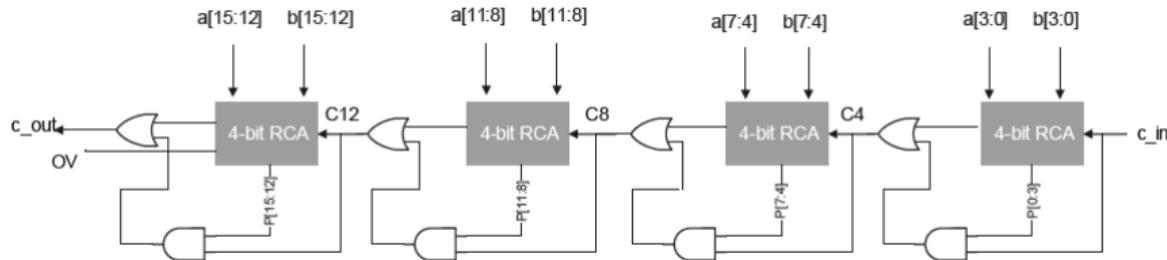
- En un carry skip adder de N bits, estos bits son divididos en grupos de k bits.
- El sumador propaga todos los acarreos en forma simultánea a través de estos grupos.
- Cada grupo i calcula el grupo P_i usando la siguiente relación:
$$P_i = p_i p_{i+1} p_{i+2} \dots p_{i+k-1}$$
- P_i es calculada para cada posición de bit de la siguiente manera:
$$p_i = a_i \oplus b_i$$
- Si cada grupo genera acarreo, lo pasa al siguiente grupo.
- En caso que el grupo no genere acarreo debido a la distribución de los bits en el bloque, entonces simplemente transfiere el acarreo del grupo anterior al grupo siguiente (*bypass del acarreo*).
- Este bypass del acarreo es manejado por P_i .
- El carry skip adder puede ser también diseñado con grupos de distinto número de bits.

Carry Skip Adder

- Supongamos un sumador de 16 bits dividido en grupos de 4 bits.
- Y tomemos el caso cuando el primer grupo genera acarreo de salida y los dos grupos subsecuentes no lo hacen, simplemente se transfiere el acarreo del primer grupo al último grupo.
- El último grupo toma este acarreo y genera luego su propio acarreo de salida.
- Este es el caso que presenta el mayor delay en la generación del acarreo para este ejemplo, este delay es menor al correspondiente en un RCA.

Sumadores

Carry Skip Adder



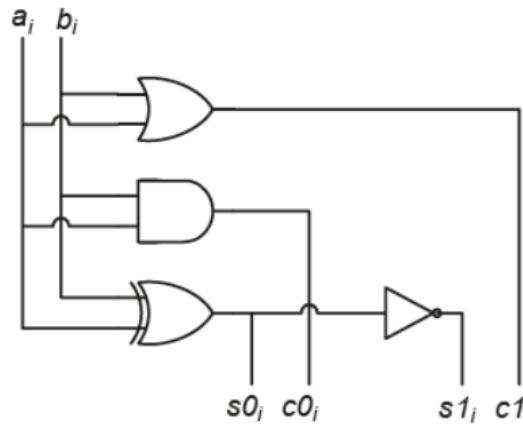
Sumador de 16 bits dividido en grupos iguales

Conditional Sum Adder

- Este sumador es implementado en múltiples niveles.
- En el primer nivel los bits de la suma y el acarreo, para bits de acarreo de entrada 1 y 0, son calculados de la siguiente manera:
$$s0_i = a_i \oplus b_i$$
$$s1_i = \sim(a_i \oplus b_i)$$
$$c0_i = a_i b_i$$
$$c1_i = a_i + b_i$$
- En las ecuaciones anteriores a_i y b_i corresponden a los bits de la posición i de los operandos a y b , $s0_i$ y $c0_i$ son la suma y el acarreo de salida calculados asumiendo un acarreo de entrada igual a 0. $s1_i$ y $c1_i$ corresponden a los resultados asumiendo un acarreo de entrada de 1.
- Estas ecuaciones son implementadas por una celda lógica llamada *conditional cell (CC)*.

Sumadores

Conditional Sum Adder



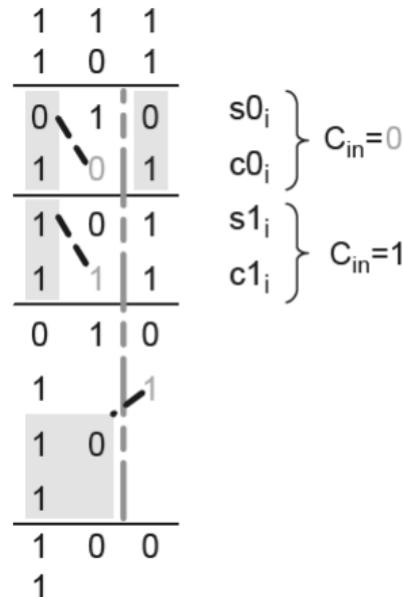
Conditional cell (CC)

Conditional Sum Adder

- En el nivel dos, los resultados del nivel uno se fusionan. Esa fusión se realiza a través del emparejamiento de las columnas consecutivas del nivel 1. Para un sumador de N bits, las columnas son emparejadas de la forma $(i, i+1)$ para i que va desde 0 hasta $N-2$.
- Los bits menos significativos del acarreo (*LSC*) en una determinada posición i (*por ejemplo* c_{0i}, c_{1i}) de cada par seleccionan los bits de suma y de acarreo de la posición $i+1$ para el siguiente nivel de procesamiento.
- Si el least significant carry *LSC* está en 0 los bits calculados para el acarreo de entrada 0 serán seleccionados, en caso contrario se seleccionarán los bits correspondientes al acarreo de entrada 1.
- De esta manera dos bits serán sumados asumiendo un determinado valor para el acarreo de entrada.

Sumadores

Conditional Sum Adder



Suma de números de 3 bits utilizando un conditional sum adder

Conditional Sum Adder

- En el ejemplo anterior podemos ver el sumador CSA sumando números de 3 bits.
- En el primer nivel cada grupo consiste de un bit. El nivel añade los bits de la posición i suponiendo el acarreo de entrada en 0 y en 1.
- En el siguiente nivel las tres columnas se dividen en dos grupos (*como muestra la línea punteada*). La columna en la posición del bit 0 forma el primer grupo y las otras dos columnas el segundo grupo. La selección apropiada en cada grupo por parte del LSC se muestra con la línea diagonal.
- El LSC de cada grupo determina cual de los dos bits de la siguiente columna se reducirá al siguiente nivel de tratamiento.
- Si el LSC es 0 se seleccionan los bits superiores de la siguiente columna, de lo contrario se seleccionan los dos bits inferiores de la siguiente columna.
- Para el segundo grupo el bit de suma de la primer columna también baja al segundo nivel. Los LSC en el primer nivel están resaltados en el gráfico en negrita.
- Finalmente en el siguiente nivel los dos grupos formados en el nivel anterior se combinan y el LSC selecciona uno de los dos grupos de 3 bits para pasar al siguiente nivel. Como el LSC es 1, selecciona el grupo inferior.

Sumadores

Conditional Sum Adder

a_i	1	0	0	1	1	0	0	1	1	1	1	0	0	1	1	1	0	1
b_i	0	0	1	1	0	1	1	0	1	1	0	1	0	1	0	1	1	0
Group width	Group carry-in	Group sum and block carry out																
1	0	1	0	1	0	1	1	1	1	0	0	0	0	1	1	0	1	1
	1	0	1	0	1	0	0	0	0	1	1	1	1	0	0	1	0	0
2	0	1	0	0	0	1	1	1	1	1	0	0	1	0	0	0	1	1
	1	1	1	0	1	0	0	0	0	1	1	1	1	0	0	1	1	1
4	0	1	1	0	0	1	1	1	1	1	0	0	0	1	0	0	1	1
	1	1	1	0	1	0	0	0	0	1	1	1	1	0	0	1	1	1
8	0	1	1	0	0	1	1	1	1	1	0	0	0	1	0	0	1	1
	1	1	1	0	1	0	0	0	0	1	0	0	1	0	0	1	1	1
16	0	1	1	0	1	0	0	0	0	1	0	1	0	0	0	1	1	1
	1																	

a_i

b_i

i

$s0_i$

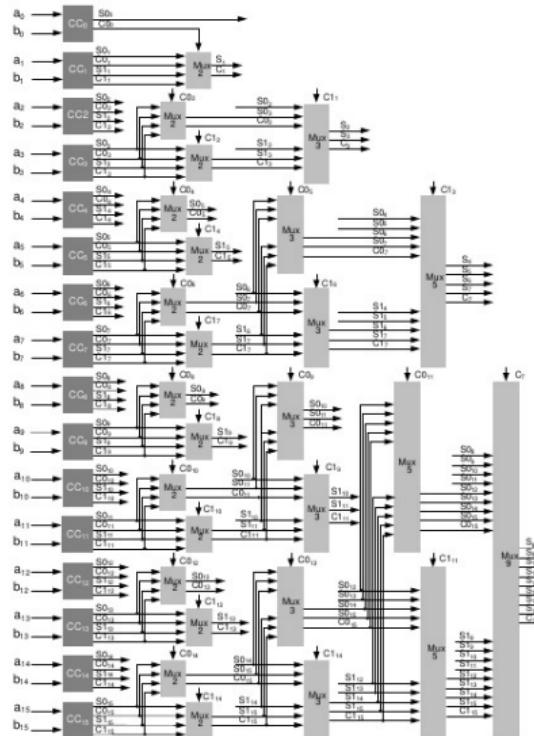
$c0_i$

$s1_i$

$c1_i$

Sumadores

Conditional Sum Adder



Sumadores

Conditional Sum Adder

Implementación en Verilog (8-bits)

```
1 module conditional_sum_adder
2 #(parameter W = 8)
3   (input [W-1:0]      a, b,
4    input              cin, clk,
5    output reg [W-1:0] sum_r,
6    output reg          cout_r);
7   wire s1_0, c2_0, s2_0, c3_0, s3_0, c4_0, s4_0, c5_0, s5_0,
8     c6_0, s6_0, c7_0, s7_0, c8_0;
9   wire s1_1, c2_1, s2_1, c3_1, s3_1, c4_1, s4_1, c5_1, s5_1,
10    c6_1, s6_1, c7_1, s7_1, c8_1;
11  reg fcout;
12  reg s3_level_1_0, s3_level_1_1, s5_level_1_0, s5_level_1_1
13    , s7_level_1_0, s7_level_1_1;
14  reg c4_level_1_0, c4_level_1_1, c6_level_1_0, c6_level_1_1
15    , c8_level_1_0, c8_level_1_1;
16  reg c2_level_1;
17  reg c4_level_2;
18  reg s6_level_2_0, s6_level_2_1, s7_level_2_0, s7_level_2_1
19    , c8_level_2_0, c8_level_2_1;
20  reg [W-1:0] a_r,b_r;
21  reg          cin_r;
22  reg [W-1:0] sum;
23  reg          cout;
```

```
1 always@(posedge clk) begin
2   sum_r <= sum;
3   cout_r <= cout;
4   a_r    <= a;
5   b_r    <= b;
6   cin_r  <= cin;
7 end
8 // Level 0
9 always @* {fcout,sum[0]} = a_r[0] + b_r[0] + cin_r;
10 // Conditional cells instantiation
11 conditional_cell c1( a_r[1], b_r[1], s1_0, s1_1, c2_0, c2_1
12   );
13 conditional_cell c2( a_r[2], b_r[2], s2_0, s2_1, c3_0, c3_1
14   );
15 conditional_cell c3( a_r[3], b_r[3], s3_0, s3_1, c4_0, c4_1
16   );
17 conditional_cell c4( a_r[4], b_r[4], s4_0, s4_1, c5_0, c5_1
18   );
19 conditional_cell c5( a_r[5], b_r[5], s5_0, s5_1, c6_0, c6_1
20   );
21 conditional_cell c6( a_r[6], b_r[6], s6_0, s6_1, c7_0, c7_1
22   );
23 conditional_cell c7( a_r[7], b_r[7], s7_0, s7_1, c8_0, c8_1
24   );
```

Sumadores

Conditional Sum Adder

Implementación en Verilog (8-bits)

```
1 // Level 1 muxes
2 always @(*)
3   case(fcout) // For first mux
4     1'b0: {c2_level_1, sum[1]} = {c2_0, s1_0};
5     1'b1: {c2_level_1, sum[1]} = {c2_1, s1_1};
6   endcase
7 always @(*) // For 2nd mux
8   case(c3_0)
9     1'b0: {c4_level_1_0, s3_level_1_0} = {c4_0, s3_0};
10    1'b1: {c4_level_1_0, s3_level_1_0} = {c4_1, s3_1};
11  endcase
12 always @* // For 3rd mux
13   case(c3_1)
14     1'b0: {c4_level_1_1, s3_level_1_1} = {c4_0, s3_0};
15     1'b1: {c4_level_1_1, s3_level_1_1} = {c4_1, s3_1};
16  endcase
17 always @* // For 4th mux
18   case(c5_0)
19     1'b0: {c6_level_1_0, s5_level_1_0} = {c6_0, s5_0};
20     1'b1: {c6_level_1_0, s5_level_1_0} = {c6_1, s5_1};
21  endcase
```

```
1 always @* // For 5th mux
2   case(c5_1)
3     1'b0: {c6_level_1_1, s5_level_1_1} = {c6_0, s5_0};
4     1'b1: {c6_level_1_1, s5_level_1_1} = {c6_1, s5_1};
5   endcase
6 always @* // For 6th mux
7   case(c7_0)
8     1'b0: {c8_level_1_0, s7_level_1_0} = {c8_0, s7_0};
9     1'b1: {c8_level_1_0, s7_level_1_0} = {c8_1, s7_1};
10    endcase
11 always @* // For 7th mux
12   case(c7_1)
13     1'b0: {c8_level_1_1, s7_level_1_1} = {c8_0, s7_0};
14     1'b1: {c8_level_1_1, s7_level_1_1} = {c8_1, s7_1};
15    endcase
16 // Level 2 muxes
17 always @* // First mux of level2
18   case(c2_level_1)
19     1'b0: {c4_level_2, sum[3], sum[2]} = {c4_level_1_0,
20       s3_level_1_0,s2_0};
21     1'b1: {c4_level_2, sum[3], sum[2]} = {c4_level_1_1,
22       s3_level_1_1,s2_1};
23   endcase
```

Sumadores

Conditional Sum Adder

Implementación en Verilog (8-bits)

```
1  always @* // 2nd mux of level2
2    case(c6_level_1_0)
3      1'b0: {c8_level_2_0, s7_level_2_0, s6_level_2_0}
4        ={c8_level_1_0, s7_level_1_0, s6_0};
5      1'b1: {c8_level_2_0, s7_level_2_0, s6_level_2_0}
6        ={c8_level_1_1, s7_level_1_1, s6_1};
7    endcase
8  always @* // 3rd mux of level2
9    case(c6_level_1_1)
10     1'b0: {c8_level_2_1, s7_level_2_1, s6_level_2_1}
11       ={c8_level_1_0, s7_level_1_0, s6_0};
12     1'b1: {c8_level_2_1, s7_level_2_1, s6_level_2_1}
13       ={c8_level_1_1, s7_level_1_1, s6_1};
14   endcase
```

```
1  // Level 3 mux
2  always @*
3    case(c4_level_2)
4      1'b0: {cout,sum[7:4]} = {c8_level_2_0, s7_level_2_0,
5                                s6_level_2_0, s5_level_1_0,
6                                s4_0};
7      1'b1: {cout,sum[7:4]} = {c8_level_2_1, s7_level_2_1,
8                                s6_level_2_1, s5_level_1_1,
9                                s4_1};
10   endcase
11 endmodule
12 // Module for conditional cell
13 module conditional_cell(a, b, s_0, s_1, c_0, c_1);
14   input a,b;
15   output s_0, c_0, s_1, c_1;
16   assign s_0 = a^b;
17   assign c_0 = a&b;
18   assign s_1 = ~s_0;
19   assign c_1 = a | b;
20 endmodule
```

Carry Select Adder

- El Carry Select Adder (CSA) no es lo suficientemente rápido como el carry look-ahead adder y requiere considerablemente más hardware si se utiliza con diseños personalizados, pero posee un diseño favorable para el uso en FPGA's con lógica de cadena de acarreo rápida.
- El CSA partitiona un sumador de N-Bits en K grupos, en donde se tiene:

$$k = 0, 1, 2, \dots, K - 1$$

$$n_0 + n_1 + \dots + n_{K-1} = N$$

$$n_0 \leq n_1 \leq \dots \leq n_{K-1}$$

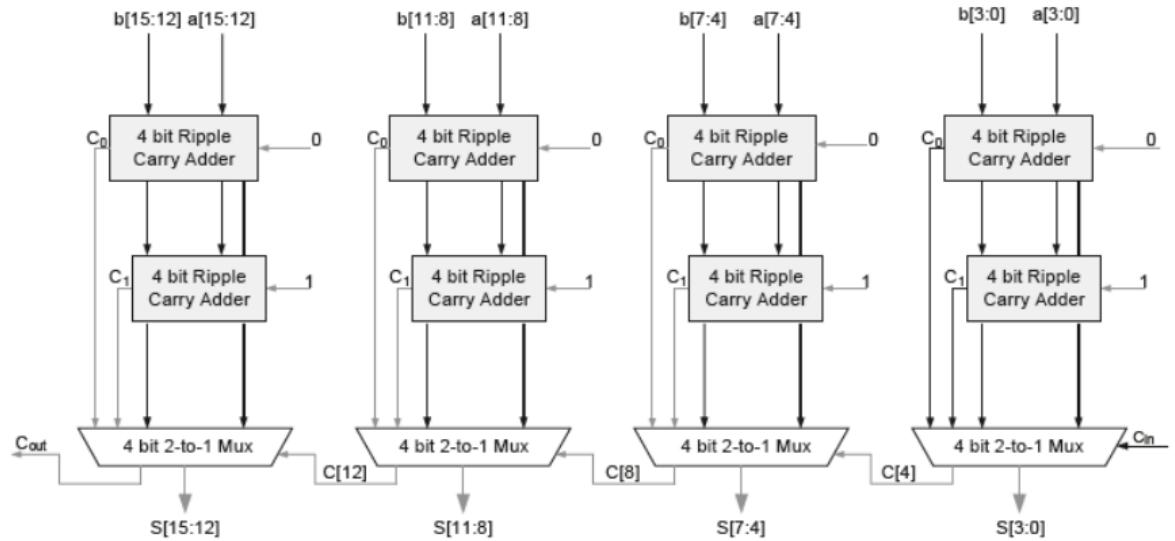
- Donde n_k representa el número de bits en el grupo K. La idea básica es colocar dos sumadores de n_k bits en cada etapa K.
- Un conjunto de sumadores calcula la suma suponiendo un acarreo de entrada igual a 1 y el otro un acarreo de entrada igual a 0.
- La suma real y el acarreo se seleccionan usando un MUX 2 a 1 basado en el acarreo del grupo anterior.

Carry Select Adder

- En la figura que se mostrará a continuación se puede observar un CSA de 16 bits. El mismo, se divide en cuatro grupos de 4 bits cada uno.
- Como cada bloque posee el mismo ancho, sus salidas estarán listas al mismo tiempo.
- En un CSA con bloques de diferente ancho el tamaño del bloque en cualquier etapa del sumador se fijará mayor que el tamaño del bloque en su etapa menos significativa.
- Esto ayuda a reducir el retraso ya que las transferencias en etapas menos significativas están listas para seleccionar la suma y llevarla a las respectivas etapas siguientes.

Sumadores

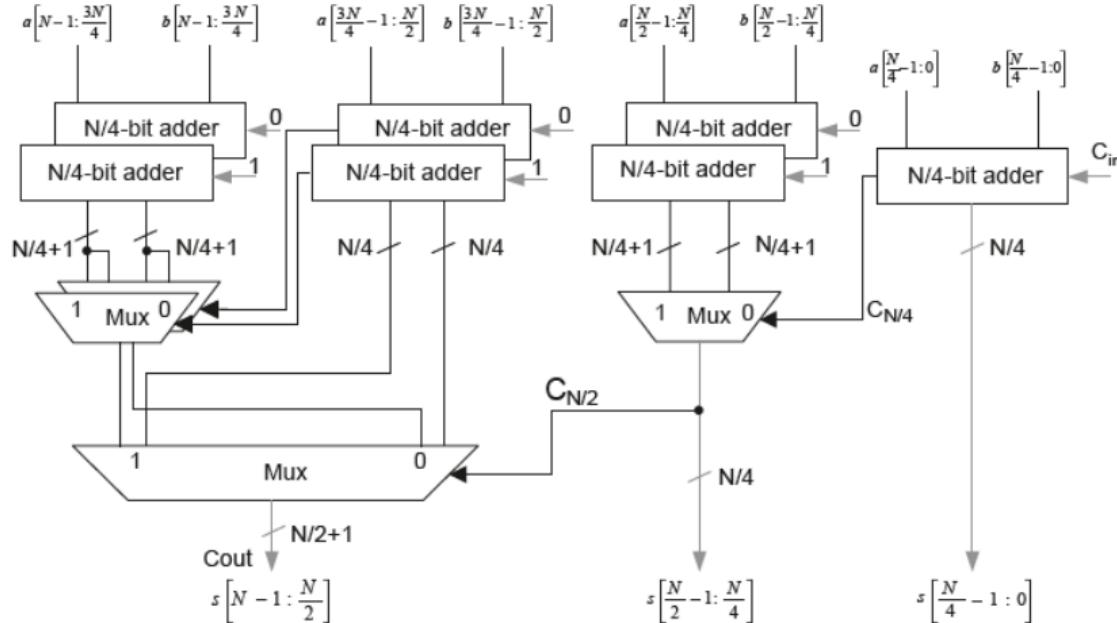
Carry Select Adder



Sumador de 16 Bits

Sumadores

Carry Select Adder



Sumador CSA de dos etapas

Carry Select Adder

- El CSA como muestra la figura anterior puede dividirse en más de una etapa.
- En el caso anterior el sumador de N-Bits se divide en dos grupos de $N/2$ bits. A su vez cada etapa se divide en dos subgrupos de $N/4$ bits.
- En una primera etapa cada subgrupo calcula sumas y acarreos de salida para acarreos de entrada iguales a 1 y 0, dentro de un mismo grupo dos subgrupos se combinan.
- Luego en una segunda etapa se combinan los dos grupos generando la suma y el acarreo final de salida.
- Si se dividieran los grupos hasta llegar a subgrupos de 1 bit cada uno, la arquitectura del sumador sería la misma del conditional sum adder por lo que consideramos a este un caso particular de CSA.

Sumadores

Carry Select Adder

Implementación en Verilog (16-bits)

```
1 module hierarchicalcsa(a, b, cin, sum_r, c_out_r, clk);
2   input [15:0] a,b;
3   input cin,clk;
4   output reg c_out_r; output reg [15:0] sum_r;
5   wire      c4,c8,c8_0,c8_1,c12_0,c12_1,c16_0, c16_1, c16L2_0, c16L2_1;
6   wire [15:4] sumL1_0, sumL1_1; wire [15:12] sumL2_0, sumL2_1;
7   reg [15:0]  a_r,b_r; reg cin_r; wire c_out; wire [15:0] sum;
8   always@(posedge clk)begin
9     a_r <= a; b_r <= b; cin_r <= cin; sum_r <= sum; c_out_r <= c_out; end
10 // Level one of hierarchical CSA
11 assign {c4,sum[3:0]} = a_r[3:0] + b_r[3:0] + cin_r;
12 assign {c8_0, sumL1_0[7:4]}= a_r[7:4] + b_r[7:4] + 1'b0;
13 assign {c8_1, sumL1_1[7:4]}= a_r[7:4] + b_r[7:4] + 1'b1;
14 assign {c12_0,sumL1_0[11:8]}= a_r[11:8] + b_r[11:8] + 1'b0;
15 assign {c12_1,sumL1_1[11:8]}= a_r[11:8] + b_r[11:8] + 1'b1;
16 assign {c16_0, sumL1_0[15:12]}= a_r[15:12] + b_r[15:12] + 1'b0;
17 assign {c16_1, sumL1_1[15:12]}= a_r[15:12] + b_r[15:12] + 1'b1;
18 // Level two of hierarchical CSA
19 assign c8 = c4 ? c8_1 : c8_0;
20 assign sum[7:4] = c4 ? sumL1_1[7:4]: sumL1_0[7:4];
21 // Selecting sum and carry within a group
22 assign c16L2_0 = c12_0 ? c16_1 : c16_0;
23 assign sumL2_0 [15:12] = c12_0? sumL1_1[15:12] : sumL1_0[15:12];
24 assign c16L2_1 = c12_1 ? c16_1 : c16_0;
25 assign sumL2_1 [15:12] = c12_1? sumL1_1[15:12]: sumL1_0[15:12];
26 // Level three selecting the final outputs
27 assign c_out = c8 ? c16L2_1 : c16L2_0;
28 assign sum[15:8]=c8?{sumL2_1[15:12],sumL1_1[11:8]}:{sumL2_0[15:12],sumL1_0[11:8]};
29 endmodule
```

Utilizando Sumadores Híbridos

- Un diseñador digital debe siempre buscar la mejor opción para optimizar área, potencia y tiempo.
- El diseñador puede encontrar apropiado dividir el sumador en múltiples grupos y utilizar diferentes arquitecturas para cada uno.
- Esta práctica puede ayudarle al diseñador a encontrar una arquitectura óptima para el diseño.

División con Barrel Shifter



Introducción

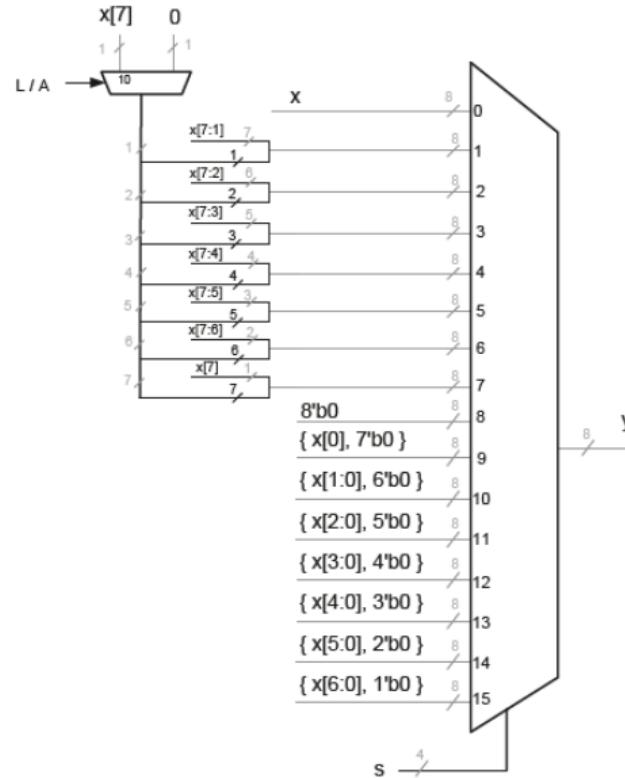
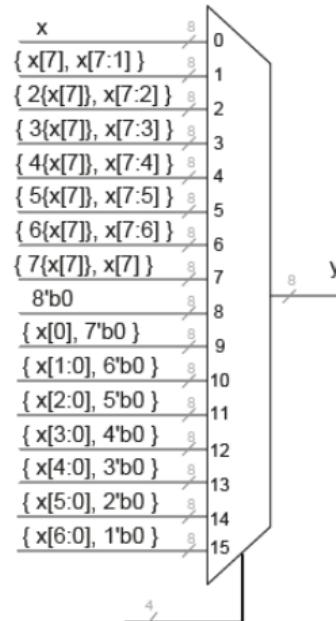
- Un desplazador lógico de N bits implementa la operación $x >> s$, donde s es un número entero signado.
- Se implementa a través del conexionado de todos los desplazamientos posibles como entrada de un multiplexor y luego se selecciona la salida apropiada usando el número s .
- Por ejemplo si $s = -2$ implica un desplazamiento de 2 hacia la izquierda.

Ejemplos de diseño de desplazadores

- En la imagen de la izquierda se puede observar el diseño del desplazador, donde x es el operador de entrada y todos los posibles desplazamientos son realizados previamente e ingresados en el multiplexor en el cual el número s es usado como selector.
- Para un número s negativo el desplazador tomará el valor positivo y realizará el desplazamiento hacia la izquierda.
- El diseño puede ser fácilmente extendido para encargarse de desplazamientos aritméticos y lógicos, esto lo podemos observar en la imagen derecha.
- Para esto primero se debe seleccionar ya sea el bit de signo o 0 para anexar apropiadamente a la izquierda del operador para lograr una operación de desplazamiento a la derecha.
- Para una operación de desplazamiento a la izquierda el diseño para ambas, desplazamiento aritmético y lógico es el mismo.
- Cuando no hay suficientes bits de signo redundantes, el desplazamiento hacia la izquierda provocará overflow.

Barrel Shifter

Ejemplo de diseño de desplazadores



Barrel Shifter jerárquico

- En lugar de usar un multiplexor con múltiples entradas, un barrel shifter puede también ser construido en forma jerárquica.
- Se pueden implementar fácilmente pipelines en este diseño. La técnica puede funcionar tanto para el desplazamiento hacia la derecha como hacia la izquierda.
- Para $x >> s$, la técnica trabaja considerando s como un número de complemento a dos con signo donde el bit de signo tiene peso negativo y el resto de los bits poseen peso positivo.
- Pasando de los bits más significativos a los menos significativos, cada etapa del barrel shifter solo abastece un bit y realiza el desplazamiento requerido igual al peso de el bit.

Barrel Shifter

Ejemplo Desplazador de 16 bits

- En la imagen de la siguiente diapositiva se presenta un barrel shifter capaz de desplazar un número x de 16 bits por un número s de 5 bits signado. Con lo cual el desplazador puede realizar corrimientos del 0 al 15 hacia la derecha y del 1 al 16 hacia la izquierda en 5 etapas o niveles.
- Primero el desplazador chequea si es requerido un desplazamiento lógico o aritmético y selecciona apropiadamente 0 o el bit de signo del operando para luego agregarlo para la operación de desplazamiento hacia la derecha.
- Luego el desplazador chequea el bit más significativo de s , ya que este bit posee peso negativo. Si $s[4] = 1$, se realiza un desplazamiento a la izquierda de 16 y mantiene el resultado como un número de 31 bits.
- En el caso que $s[4] = 0$, el número es apropiadamente extendido a un número de 31 bits para el siguiente nivel para lograr desplazamientos apropiados.
- Para el resto de los bits la lógica realiza un desplazamiento a la derecha igual al peso de el bit bajo consideración, y el diseño sigue reduciendo el número de bits al ancho requerido a la salida de cada etapa.

Barrel Shifter

Ejemplo de desplazador de 16 bits

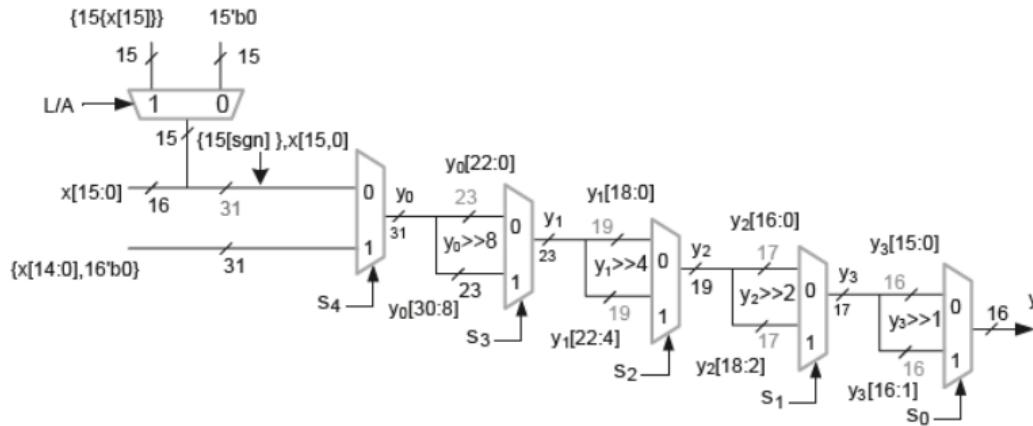


Diagrama en Bloque.

Barrel Shifter

Ejemplo de desplazador de 16 bits-Pipelined

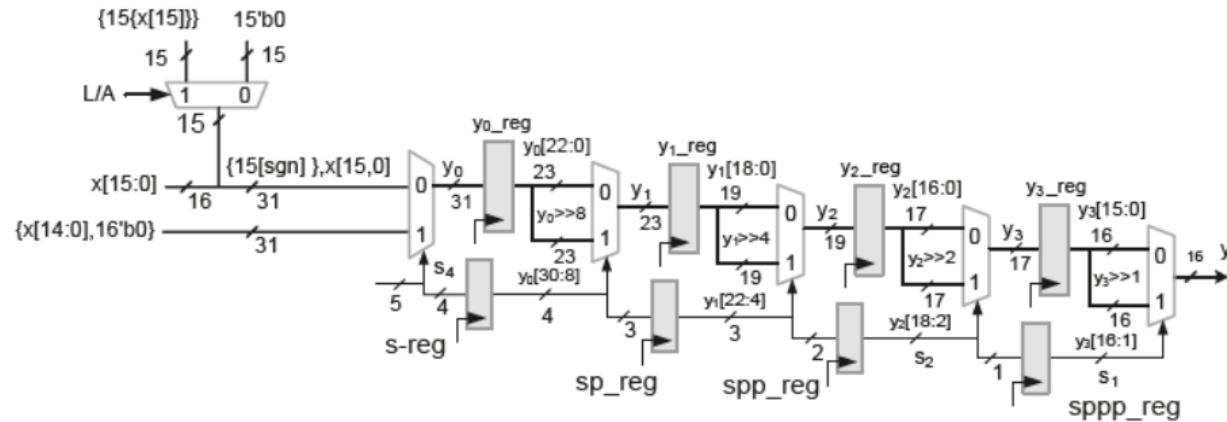


Diagrama en Bloque incluyendo pipeline.

Barrel Shifter

Ejemplo de desplazador de 16 bits

Verilog con y sin Pipeline

```
1 module barrelShifter
2 (
3     input [15:0]      x,
4     input signed [4:0] s,
5     input             A_L,
6     output reg [15:0] y);
7
8     reg [30:0] y0; reg [22:0] y1;
9     reg [18:0] y2; reg [16:0] y3;
10    reg [14:0] sgn;
11    always @(*) begin
12        sgn = (A_L) ? {15{x[15]}} : 15'b0;
13        y0 = (s[4]) ? {x[14:0],16'b0} : {sgn[14:0], x
14            [15:0]};
15        y1 = (s[3]) ? y0[30:8] : y0[22:0];
16        y2 = (s[2]) ? y1[22:4] : y1[18:0];
17        y3 = (s[1]) ? y2[18:2] : y2[16:0];
18        y  = (s[0]) ? y3[16:1] : y3[15:0];
19    end
20 endmodule
```

```
1 module barrelShifterPipelined
2 (
3     input             clk ,
4     input [15:0]      x,
5     input signed [4:0] s,
6     input             A_L,
7     output reg [15:0] y);
8     reg [30:0]y0_r; reg [22:0]y1_r;
9     reg [18:0]y2_r; reg [16:0]y3_r;
10    reg [14:0] sgn;  reg [3:0] s_r;
11    reg [2:0]  sp_r; reg [1:0] spp_r;
12    reg         sppp_r;
13    always @(*) begin
14        sgn=(A_L)?{15{x[15]}:15'b0;
15        y0=(s[4])?{x[14:0],16'b0}:{sgn[14:0],x[15:0]};
16        y1=(s_r[3])?y0_r[30:8]:y0_r[22:0];
17        y2=(sp_r[2])?y1_r[22:4]:y1_r[18:0];
18        y3=(spp_r[1])?y2_r[18:2]:y2_r[16:0];
19        y =(sppp_r)?y3_r[16:1]:y3_r[15:0];
20    end
21    always @ (posedge clk) begin
22        y0_r <= y0;      y1_r <= y1;
23        y2_r <= y2;      y3_r <= y3;
24        s_r  <= s[3:0]; sp_r <= s_r[2:0];
25        spp_r <= sp_r[1:0];
26        sppp_r <= spp_r[0];
27    end
28 endmodule
```

Barrel Shifter como multiplicador

- Un barrel shifter puede también usarse como un multiplicador dedicado en las FPGAs.
- Un desplazamiento por s hacia la izquierda significa una multiplicación por 2^s .
- Un desplazamiento a la derecha por s equivale a una multiplicación por 2^{-s} .
- Para realizar esta operación el número por el que se desea multiplicar debe ser potencia de 2.

Sumadores Carry Save y Compresores



CSA y Compresores

Carry Save adders

CSA

- Se han presentado sumadores que suman dos operandos mientras propagan el acarreo de una posición de bit a la siguiente para calcular luego el resultado final. Estos son en conjunto llamados sumadores propagadores de acarreo (*CPA*).
- Aunque se pueden sumar 3 números en un ciclo usando un CPA, una mejor opción es usar un CSA que primero reduce los tres números a dos y luego se usa un CPA para calcular la suma final.
- En cuanto a timing y área, el CSA es uno de los sumadores más eficientes y utilizados para acelerar los diseños digitales de sistemas de procesamiento de señales que se ocupan de múltiples operandos para la adición y la multiplicación.
- Cuando un CSA reduce tres operandos a dos, no propaga el acarreo. Más bien guarda el acarreo en la siguiente posición de bit significativo. Esto significa que la adición reducirá tres operandos a dos sin tener un retardo por propagación de acarreo.
- A este sumador, por su forma de operar, también se lo denomina compresor 3:2.

Sumador Carry-Save

a0 =	0	0	1	0	1	1
a1 =	0	1	0	1	0	1
a2 =	1	1	1	1	0	1
s =	1	0	0	0	1	1
c =	0	1	1	1	0	1



■ Diagrama en bloques Compresor 3:2

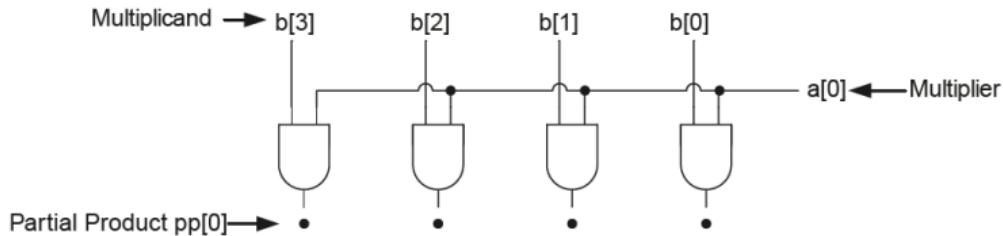
Árboles de Compresión

- Hay muchas técnicas que utilizan CSA para sumar más de tres operandos. Estas técnicas son muy utilizadas para reducir los productos parciales en el diseño de multiplicadores.
- A lo largo del curso se verá la aplicación de estas técnicas en arquitecturas de multiplicadores y en la optimización de otras arquitecturas y aplicaciones de procesamiento de señales.

Notación de Puntos

- La notación de puntos se utiliza para explicar diferentes técnicas de reducción.
- En esta notación cada bit en la suma de dos operandos está representado por un punto.
- Cuatro puntos representan cuatro bits de productos parciales de un multiplicador 4×4 . Los puntos que aparecen en una columna se deben agregar al otro operando para calcular el producto final.
- Una técnica de reducción 3:2 reduce tres capas de productos parciales a dos.

Compresores

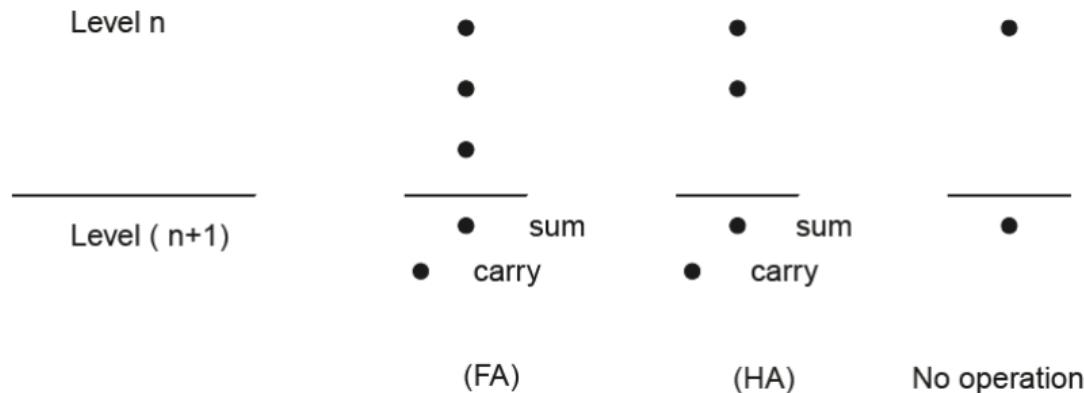


Puntos usados para representar cada bit de un producto parcial

Notación de Puntos

- Esta técnica considera el número de puntos en cada columna, en caso que aparezca un punto aislado en una columna simplemente se reduce al siguiente nivel de lógica.
- Cuando se tienen dos puntos en una sola columna se agregan usando un half adder, el punto para la suma se deja caer en la misma columna y el punto para el acarreo se coloca en la siguiente columna significativa. El uso de un half adder para esta operación también se conoce como reducción 2:2.
- Los tres puntos en una columna se reducen a dos utilizando un full adder. En este caso, el punto para la suma se coloca en la misma ubicación de bit y el punto para el acarreo se coloca en la siguiente posición de bit significativo.

Compresores



Reduciendo los números de puntos en una columna

Multiplicadores Paralelos

Multiplicadores Paralelos

Introducción

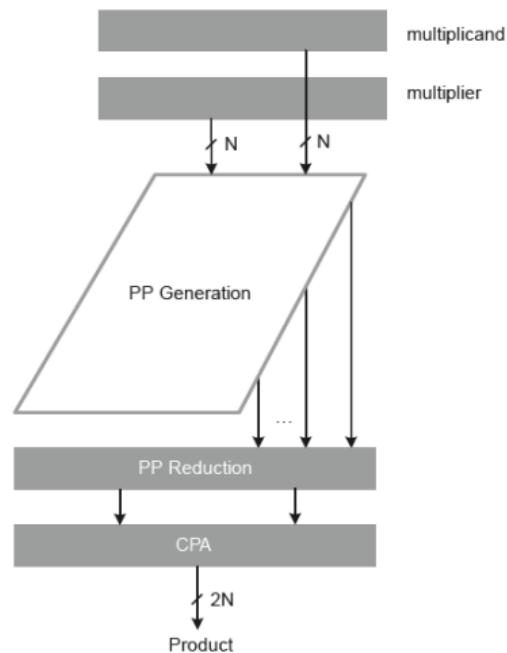
- La mayoría de los algoritmos de procesamiento de señales utilizan multiplicadores.
- Es por esto que es muy importante entender las técnicas que se utilizan para optimizar la implementación de los multiplicadores.
- Debido a su importancia, los fabricantes de FPGA están implementando multiplicadores dedicados en sus dispositivos.
- Si bien se pueden diseñar multiplicadores secuenciales que demoran varios ciclos para calcular el resultado, los diseñadores de sistemas de alto rendimiento se interesan por desarrollar multiplicadores que calculen el producto en un sólo ciclo.
- Para lograr esto se diseñan arquitecturas de multiplicadores paralelos.
- Un CSA es un bloque fundamental en estas arquitecturas.

Multiplicadores Paralelos

Introducción

- Los productos parciales se reducen primero a dos números utilizando un árbol CSA.
- Estos dos números son luego sumados para obtener el producto final.
- En la actualidad las FPGAs poseen un gran número de sumadores dedicados. Estos sumadores pueden sumar tres operandos. Los árboles reductores pueden reducir el número de productos parciales a tres en lugar de dos para hacer uso completo de estos bloques.
- Cualquier arquitectura multiplicadora paralela consta de tres operaciones básicas: generación de producto parcial, reducción del producto parcial y cálculo de la suma final usando un CPA (Carry Propagation Adder).

Multiplicadores Paralelos



Tres componentes de un Multiplicador

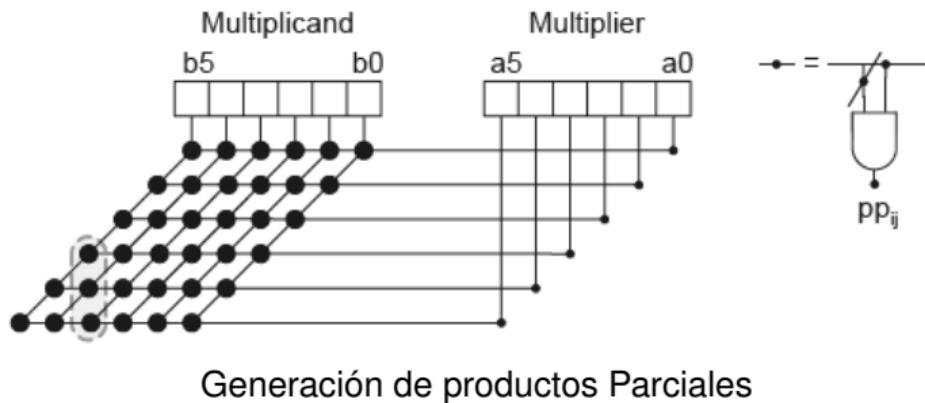
Multiplicadores Paralelos

Generación de Productos Parciales

- Mientras se multiplican dos números de N bits a y b , se generan los productos parciales. Los mismos se pueden generar ya sea utilizando el método de ANDing o implementando un algoritmo de recodificación de Booth.
- El primer método genera un producto parcial PP_i a través de una operación AND de cada bit a_i del multiplicador con todos los bits del multiplicando b .
- En la imagen de la siguiente diapositiva se puede observar la generación de productos parciales para un multiplicador de 6 bits por 6 bits.
- Cada PP_i es desplazado a la izquierda i posiciones antes de que los productos parciales sean sumados por columnas para producir el resultado final.

Multiplicadores Paralelos

Multiplicador de 6*6 bits



Generación de Productos Parciales

- La implementación solo destaca la generación del producto parcial y no utiliza ninguna técnica de reducción del mismo.
- Estos productos parciales son correctamente desplazados utilizando el operador de concatenación en Verilog y luego son sumados para completar la función del modulo multiplicador.

Multiplicadores Paralelos

Multiplicador de 6*6 bits

Implementación en Verilog

```
1 module multiplier
2 (
3     input [5:0]    a,b,
4     output [11:0] prod);
5     integer         i;
6     reg [5:0]      pp [0:5]; //6 partial products
7     always@(*)
8         begin
9             for(i=0; i<6; i=i+1)
10                 begin
11                     pp[i] = b & {6{a[i]}};
12                 end
13             end
14         assign prod = pp[0]+{pp[1],1'b0}+{pp[2],2'b0}+
15                         {pp[3],3'b0}+{pp[4],4'b0}+{pp[5],5'b0};
16     endmodule
```

Multiplicadores Paralelos

Reducción de Productos Parciales

- Cuando multiplicamos un número “a” de N_1 bits por otro “b” de N_2 bits, N_1 productos parciales son producidos al aplicar el operador ANDing entre cada bit a_i con todos los bits de b y desplazando el producto parcial PP_i a la izquierda i posiciones.
- Usando la notación de puntos, todos los productos parciales forman un arreglo de puntos en paralelogramo. Los puntos de cada columna se añadirán para calcular el resultado final.
- En general, para un multiplicador de N_1 bits por N_2 bits, se utilizan las siguientes cuatro técnicas para reducir N_1 capas de productos parciales a dos capas para su adición final usando cualquier CPA:
 - Carry Save Reducción.
 - Dual Carry Save Reducción.
 - Wallace Tree Reduction.
 - Dadda Tree Reduction.
- Las técnicas se describen para compresores 3:2 pero pueden extenderse fácilmente a otros compresores.

Multiplicadores Paralelos

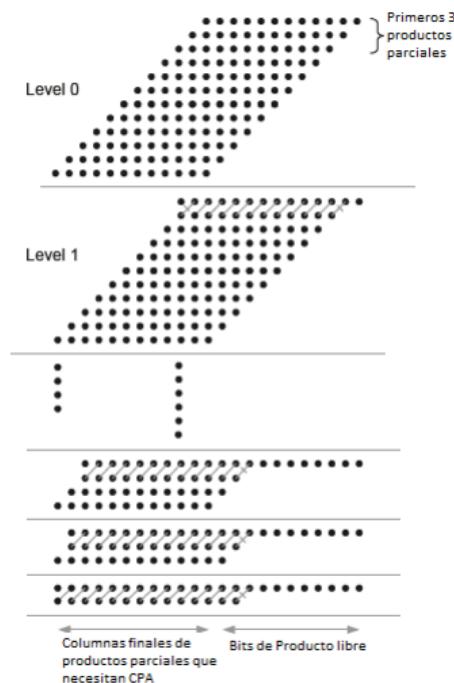
Reducción de Productos Parciales

Carry Save Reduction

- Se reducen las tres primeras capas de los productos parciales usando CSA.
- En las tres capas seleccionadas, los bits aislados en una columna simplemente caen a la misma columna, las columnas con dos bits se reducen a dos bits usando half adders y las columnas con tres bits se reducen a dos bits usando full adders.
- Cuando se suma utilizando las HA y FA, el punto que representa el bit de suma se deja caer en la misma columna mientras que el punto del acarreo se coloca en la siguiente columna de bit más significativo.
- Una vez que los primeros tres productos parciales se reducen a dos capas, el cuarto producto parcial de la composición original se agrupa con las dos capas anteriores para formar un nuevo grupo de tres capas. Estas tres capas se reducen de nuevo a dos utilizando la técnica CSA.
- Este proceso se repite hasta que todo el conjunto se reduce a dos capas de números.
- Como resultado se obtiene unos bits de producto menos significativos, denominados bits de producto libre, y el resto de los bits aparecen en dos capas las cuales se agregan utilizando cualquier CPA.

Multiplicadores Paralelos

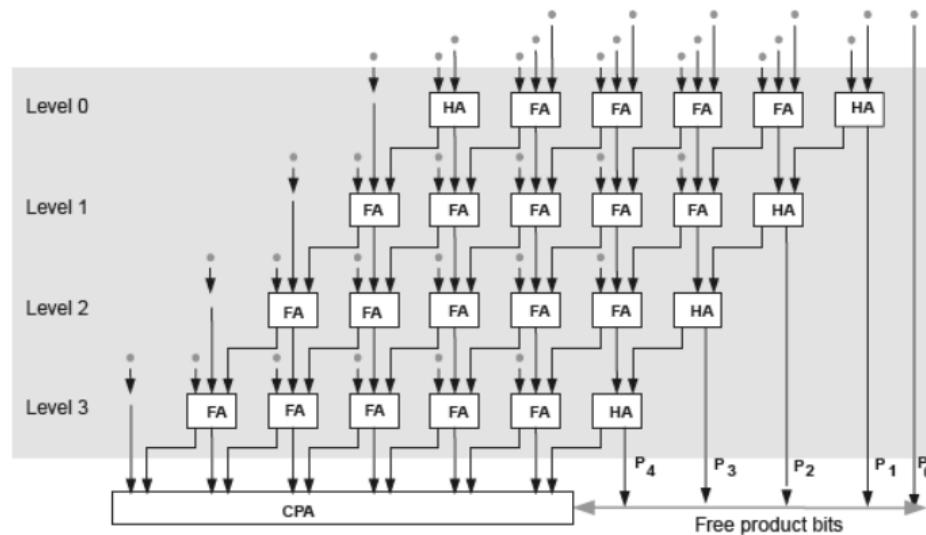
Multiplicador de 12*12 bits



Esquema de Carry Save Reduction

Multiplicadores Paralelos

Multiplicador de 6*6 bits



Esquema Layout de Carry Save Reduction

Multiplicadores Paralelos

Reducción de Productos Parciales

Dual Carry Save Reduction

- Los productos parciales son divididos en dos grupos de igual tamaño.
- El esquema de Carry Save Reduction es aplicado en ambos sub grupos simultáneamente.
- Como resultado se obtienen dos conjuntos de capas de productos parciales en cada sub grupo.
- La técnica finalmente resulta en cuatro capas de productos parciales.
- Estas capaz son reducidas luego a un grupo de tres y finalmente a un grupo de dos capas.

Multiplicadores Paralelos

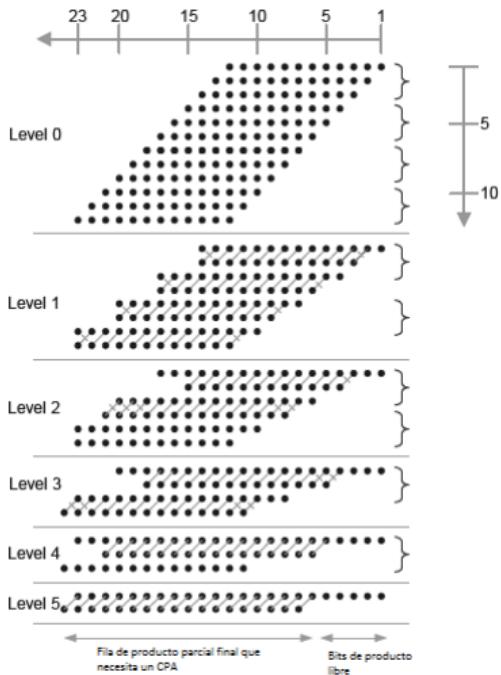
Reducción de Productos Parciales

Wallace Tree Reduction

- Los productos parciales son divididos en grupos de tres productos parciales cada uno.
- A diferencia de la reducción de tiempos de los dos métodos anteriores, estos grupos de productos parciales son reducidos simultáneamente usando CSAs.
- Cada capa de CSA comprime tres capas a dos capas. Estas dos capas de cada grupo se reagrupan en conjuntos de tres capas.
- En el siguiente nivel nuevamente se reducen tres capas a dos capas, este proceso continuará hasta que queden solo dos filas.
- Por último cualquier CPA puede utilizarse para calcular el producto final.

Multiplicadores Paralelos

Wallace Tree Reduction



Reducción aplicada a 12 productos parciales

Multiplicadores Paralelos

Reducción de Productos Parciales

Wallace Tree Reduction

- Esta reducción es uno de los esquemas más usados en arquitecturas de multiplicadores.
- La reducción se realiza en paralelo en grupos de tres.
- A medida que el número de productos parciales aumenta, tendremos un incremento logarítmico en el número de niveles del sumador.
- El número de niveles del sumador representa el retardo de la trayectoria crítica.
- Cada nivel de sumador posee un retardo de FA en su ruta.

Multiplicadores Paralelos

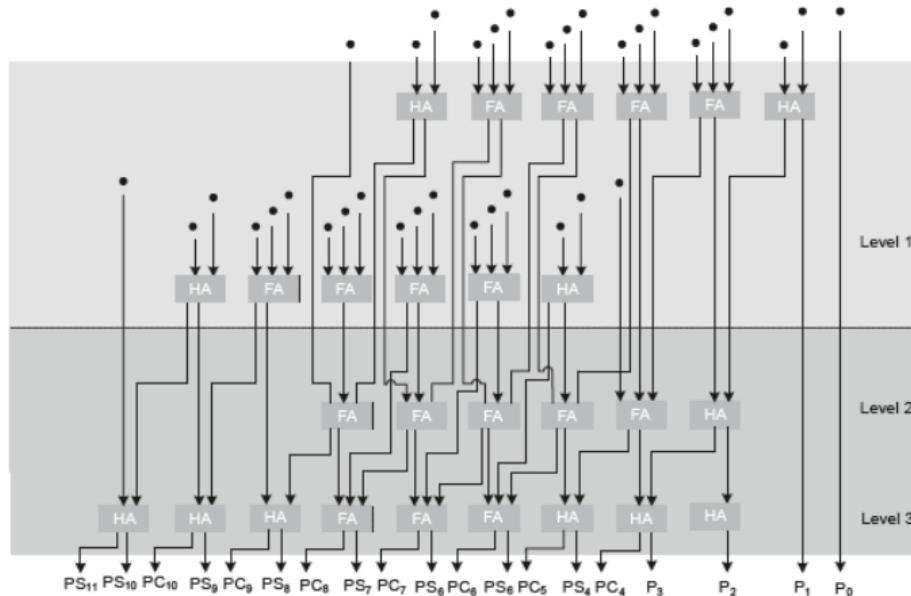
Wallace Tree Reduction

Number of PPs	Number of full-adder delays
3	1
4	2
$5 \leq n \leq 6$	3
$7 \leq n \leq 9$	4
$10 \leq n \leq 13$	5
$14 \leq n \leq 19$	6
$20 \leq n \leq 28$	7
$29 \leq n \leq 42$	8
$43 \leq n \leq 63$	9

Incremento logarítmico en los niveles del sumador

Multiplicadores Paralelos

Wallace Tree Reduction



Layout para un arreglo de productos parciales de 6×6

Multiplicadores Paralelos

Reducción de Productos Parciales

Dadda Tree Reduction

- Los árboles Dadda requieren el mismo número de niveles de sumador que los árboles Wallace, con lo que el path crítico de timing es el mismo.
- La técnica es muy usada porque minimiza el número de HA y FA en cada nivel de lógica.
- Si se observa la tabla anterior de la reducción de Wallace, los límites superiores de la columna de números de productos parciales son los siguientes:
2, 3, 4, 6, 9, 13, 19, 28, ...
- Cada número representa el número máximo de productos parciales en cada nivel que a su vez requiere un número fijo de niveles de sumador.
- La secuencia también muestra que se pueden obtener dos productos parciales a partir de como máximo tres, tres se pueden obtener de cuatro, cuatro de seis y así sucesivamente.

Multiplicadores Paralelos

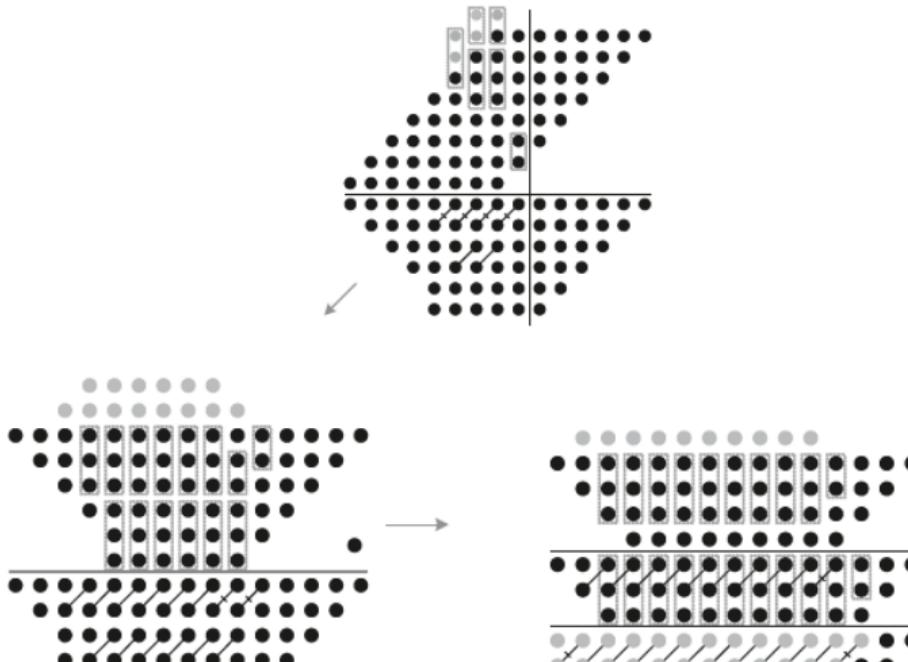
Reducción de Productos Parciales

Dadda Tree Reduction

- Dadda Tree Reduction considera cada columna por separado y reduce el número de niveles lógicos de una columna al máximo número de capas en el siguiente nivel.
- Por ejemplo para reducir un multiplicador de 12×12 bits, la reducción de Wallace reduce de 12 productos a 8 mientras que el esquema Dadda primero los reduce al rango máximo en el siguiente grupo y esto es nueve. Esta acción requerirá el mismo número de niveles de lógica pero resultará en menos hardware.
- En ésta reducción si el número de puntos en una columna es menor que el número máximo de productos parciales que se quiere reducir en el nivel actual, ellos simplemente se pasan al siguiente nivel sin ningún procesamiento.
- Las columnas que tienen más puntos que los puntos requeridos para el siguiente nivel se reducen para tomar las capas máximas en el siguiente nivel.

Multiplicadores Paralelos

Dadda Tree Reduction



Reducción de 8 productos parciales a 2

Multiplicadores Paralelos

Multiplicadores Seccionados

Descripción y ecuaciones

- Una multiplicación puede ser seccionada en un número de multiplicaciones más pequeñas.
- Por ejemplo, la multiplicación de 16 bits se puede realizar considerando los dos operando a y b como cuatro operandos de 8 bits, tomando los bits más significativos por un lado y los menos por otro obtendríamos a_H , a_L , b_H y b_L .
- La descomposición matemática de la operación se da de la siguiente manera:

$$a_L = a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0$$

$$a_H = a_{15} a_{14} a_{13} a_{12} a_{11} a_{10} a_9 a_8$$

$$b_L = b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$$

$$b = b_{15} b_{14} b_{13} b_{12} b_{11} b_{10} b_9 b_8$$

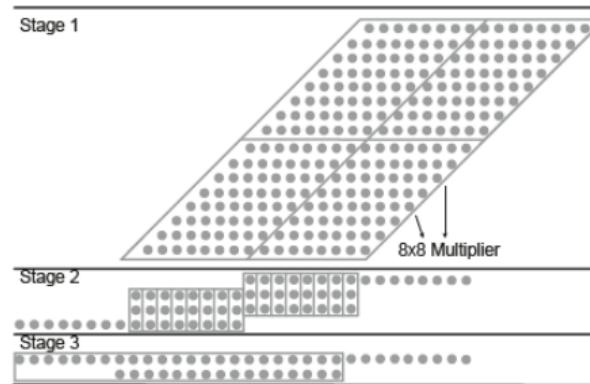
$$(a_L + 2^8 a_H) \times (b_L + 2^8 b_H) = a_L x b_L + a_L x b_H 2^8 + a_H x b_L 2^8 + a_H x b_H 2^{16}$$

- Se pueden realizar estas cuatro multiplicaciones de 8*8 bits en paralelo para luego obtener el resultado final del multiplicador de 16*16 bits.

Multiplicadores Paralelos

Multiplicadores Seccionados

		$a_L \times b_L$ 16-Bits
	$a_L \times b_H$ 16-Bits	
	$a_H \times b_L$ 16-Bits	
$a_H \times b_H$ 16-Bits		
32-Bits		



Esquema de multiplicador de 16*16 bits

Multiplicadores Paralelos

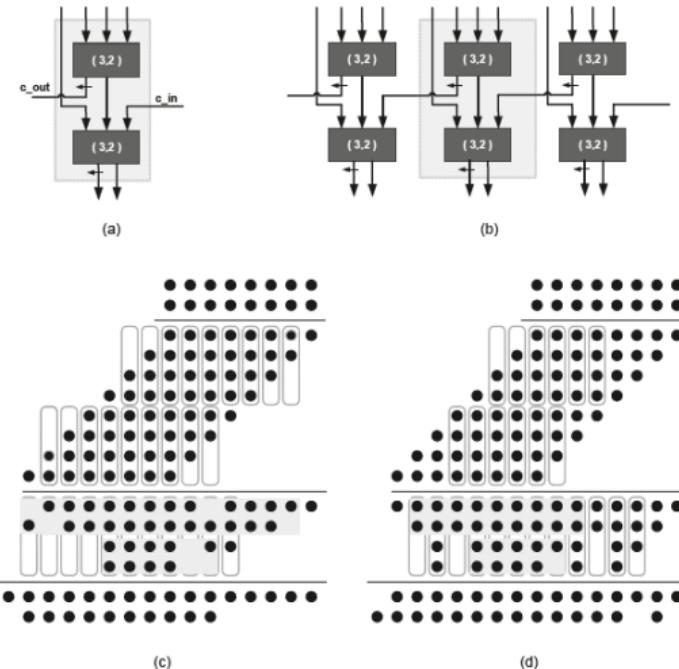
Optimización de Compresores

Descripción

- Basándose en el concepto de CSA, pueden desarrollarse otros bloques de compresores.
- Por ejemplo un compresor 4:2 toma cuatro operandos y un bit para el acarreo de entrada y los reduce a dos bits.
- Comprimiendo múltiples operandos el compresor trabaja en cascada.
- Si se implementan estos compresores utilizando la *lógica de cadena de acarreo* se pueden obtener mejores resultados de timing y de área que utilizando CSA.

Multiplicadores Paralelos

Optimización de Compresores



Diferentes tipos de compresores

Multiplicadores Paralelos

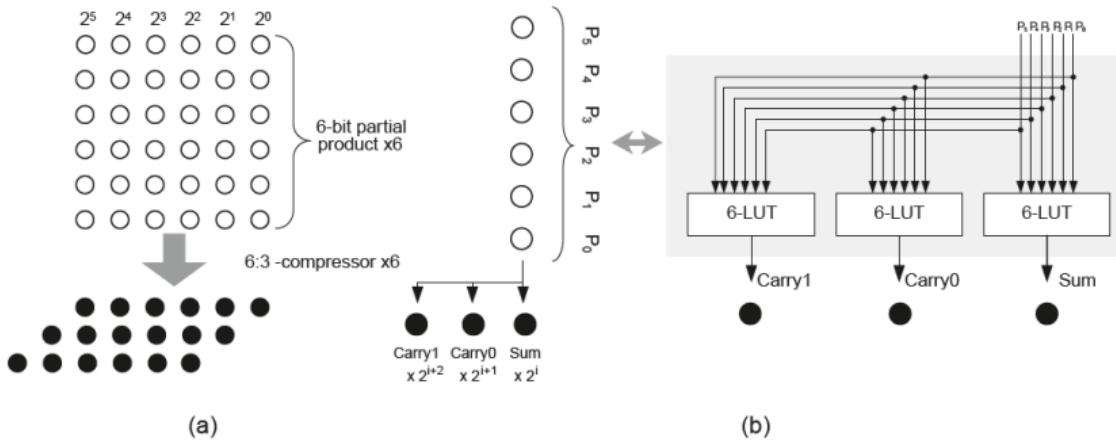
Contadores de una o múltiples columnas

Descripción

- Las sumas multi operando en ASIC (*circuito integrado de aplicación específica*) son generalmente implementados usando CSA basados en arboles de reducción Wallace y Dadda.
- Con LUTs y cadenas de acarreo en la FPGA, la implementación de un contador ofrece una mejor alternativa a la suma de múltiples operandos basados en CSA.
- Estos contadores añaden todos los bits en una o varias columnas para utilizar mejor los recursos de FPGA.
- Un contador $N : n$ de una columna suma todos los Nbits en una columna y devuelve un número de n bits, donde $n = \log_2(N + 1)$
- Las primeras arquitecturas de FPGA eran diseños basados en LUT de 4 entradas. Los LUTs de 6 entradas con lógica de cadena de transporte rápida han aparecido en las familias Vertix - 4 y Virtex - 5.
- Un diseño que efectivamente utiliza estas características es más eficiente que los demás.

Multiplicadores Paralelos

Contadores de una o múltiples columnas



Contador de una sola columna (a) Un contador 6: 3 que reduce seis capas de múltiples operandos a tres. (B) Un contador 6: 3 se mapea en tres LUTs de 6 entradas para generar la suma, llevar 0 y llevar 1 salida.

Multiplicadores Paralelos

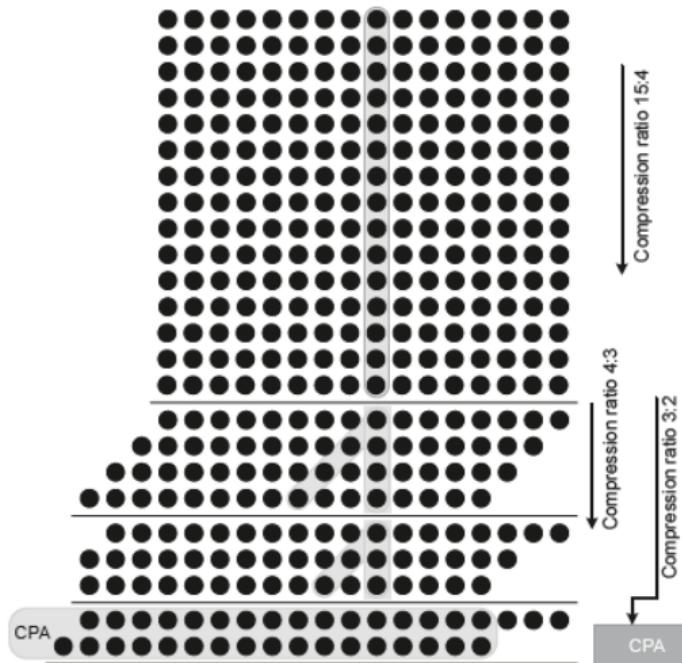
Contadores de una o múltiples columnas

Descripción

- Cada LUT calcula la suma respectiva, bits de carry0 y carry1 del compresor.
- También se pueden construir contadores de diferentes dimensiones, y una mezcla de éstos se puede utilizar para reducir varios operandos a dos.
- En la siguiente figura se pueden ver contadores 15:4, 4:3 y 3:2 que trabajan en cascada para comprimir una matriz 15 x 15.

Multiplicadores Paralelos

Contadores de una o múltiples columnas



Contador de compresión de una matriz 15×15 .

Multiplicadores Paralelos

Contadores de una o múltiples columnas

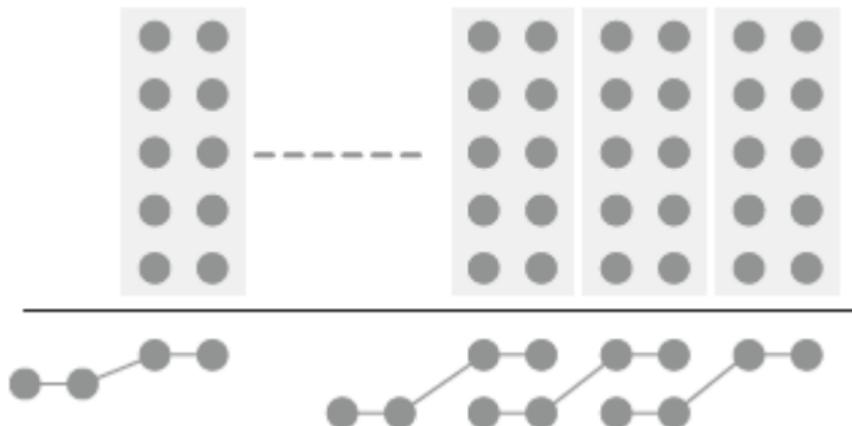
Descripción

- En una operación de suma de varios operandos, un contador paralelo generalizado (GPC) añade número de bits en varias columnas adyacentes.
- Un GPC de columna K suma N_0, N_1, \dots, N_{K-1} bits, en el menos significativo 0 a la columna más significativa $K - 1$, respectivamente, y produce un número de n bits, donde:

$$N = \sum_{i=0}^{K-1} (N_i * 2^i)$$
$$n = \lceil \log_2(N+1) \rceil$$

Multiplicadores Paralelos

Contadores de una o múltiples columnas



Síntesis de árboles compresores mediante compresión de 2 columnas de 5 bits cada una en GPC de 4 bits (5,5:4).

Multiplicadores Paralelos

Contadores de una o múltiples columnas

Descripción

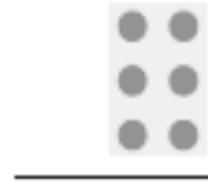
- GPC ofrece flexibilidad una vez asignada a una FPGA.
- El problema de configurar dimensiones de GPC para mapear en FPGA para la suma de múltiples operandos, es un problema complejo - NP.
- El problema se resuelve con la "programación entera", y se informa que el método supera a la aplicación basada en árboles sumadores desde el área y las perspectivas de tiempo.
- Las FPGAs son las más adecuadas para los contadores y los árboles de compresión basados en GPC.
- Para utilizar completamente las FPGAs basadas en 6- LUT, es mejor que cada contador o GPC tenga 6 bits de entrada y 3 (o 4 mejor) bits de salida, como se ve en las siguientes imágenes.
- Los cuatro bits de salida son favorecidos como los LUTs en muchos FPGAs vienen en grupos de dos con entrada de 6 bits compartida, y un 6: 3 GPC desperdiciaría un LUT en cada compresor, como se muestra en la en la segunda imagen.

Multiplicadores Paralelos

Contadores de una o múltiples columnas



(a)

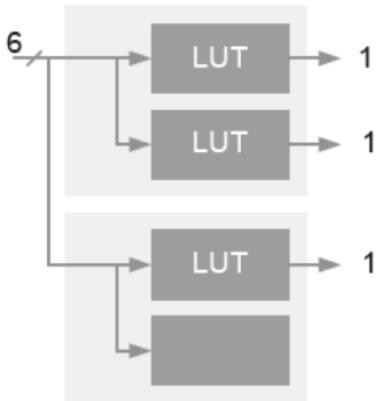


(b)

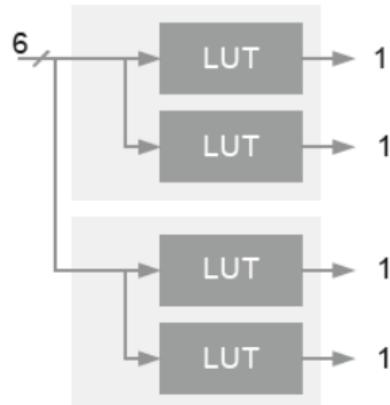
Mapeo de árboles compresores por (a) contadores 3: 2, y (b) a (3,3; 4) GPC.

Multiplicadores Paralelos

Contadores de una o múltiples columnas



(a)



(b)

(a) El módulo de lógica adaptativa (ALM) de Altera FPGA contiene dos 6-LUTs con entradas compartidas; 6-input 3-output GPC tiene 3/4 utilización lógica (b) Un 6-input 4-output GPC tiene plena utilización lógica.

Mult. Signado en Completo a Dos



Multiplicador Signado en Completo a Dos

Conceptos

- En esta sección vamos a ver las arquitecturas que implementan las multiplicaciones de números con signos.
- Un número "x" con signo en N bits se expresa como:
$$x = -x_{n-1}2^{N-1} + \sum_{i=0}^{N-2}(x_i2^i)$$
Dónde x_{n-1} es el MSB, que lleva el signo negativo.
- Cuando multiplicamos un número a de N_1 -bit con un número b con N_2 -bit, obtenemos N_1 productos parciales.
- Para los primeros $N_1 - 1$ productos parciales, el $PP[i]$ se obtiene haciendo una ANDing del bit a_i de a con b y desplazando el resultado i posiciones a la izquierda, esto implementa la multiplicación de b con a_i2^i :
$$PP[i] = (a_i2^i)(-b_{n-1}2^{N_2-1} + \sum_{i=0}^{N_2-2}(b_i2^i))$$
para $i = 0, 1, \dots, N_1 - 2$
- El $PP[i]$ en la expresión anterior es sólo el multiplicador signado que se desplaza por i a la izquierda, por lo que el MSBs de todos los productos parciales tiene peso negativos.

Multiplicador Signado en Completo a Dos

Concepto

- Debido al cambio de i , todos los PPs son números con signos de distinto ancho.
- Todos estos números son necesarios para ser alineados a la izquierda por la lógica de extensión de signo antes de que se agregan para calcular el producto final.
- Como el MSB de a tiene peso negativo, la multiplicación de este bit resulta en un PP que es el complemento de 2 del multiplicando.
- El $pp[N_1 - i]$ se calcula como:
$$PP[N_1] = (-a_{N_1-1}2^{N_1-1}).(-b_{n-1}2^{N_2-1} + \sum_{i=0}^{N_2-2}(b_i2^i))$$
- Se extiende el signo de los N_1 productos parciales y se suman para obtener el producto final.

Multiplicador Signado en Completo a Dos

Lo esencial

Ejemplo

- La siguiente figura muestra una multiplicación signada de $4 * 4$ -bits.
- Los bits de signo de los 3 primeros PPs se extienden y muestran en negrita.
- El complemento a 2 del último PP se toma para satisfacer el peso negativo del MSB del multiplicador.
- Si el multiplicador signado se implementa como en este ejemplo, la lógica de extensión de signo tomará un área significativa del árbol de compresión.
- Se desea eliminar de alguna manera esta lógica del multiplicador.

Multiplicador Signado en Completo a Dos

Ejemplo

sign
extension
logic

	1	1	0	1			
	1	1	0	1			
<hr/>							
1	1	1	1	1	1	0	1
0	0	0	0	0	0	0	X
1	1	1	1	0	1	X	X
0	0	0	1	1	X	X	X
<hr/>							
0	0	0	0	1	0	0	1

Multiplicación signada de 4 * 4-bits.

Multiplicador Signado en Completo a Dos

Eliminación de extensión de signo

- Una observación simple en un número extendido de signos nos lleva a una técnica eficaz para la eliminación de la lógica de extensión de signos.
- Un equivalente del número extendido de signo, se calcula invirtiendo el bit de signo y sumando un 1 en la ubicación del bit de signo, y extendiendo el número con todos los 1s.
- La siguiente imagen explica el cálculo en un número positivo.

Multiplicador Signado en Completo a Dos

Eliminación de extensión de signo

(a)

✓ extend all 1s

- flip the sign bit

$$\begin{array}{r}
 B = 0 \ 0 \ 0 \ 0 \ 0 \ 0.1 \ 1 \ 0 \ 1 \ 0 \ 1 \ 1 \\
 B = 1 \ 1 \ 1 \ 1 \ 1 \ 0.1 \ 1 \ 0 \ 1 \ 0 \ 1 \ 1 \\
 + \qquad \qquad \qquad 1 \rightarrow \text{add 1 at the location of sign bit} \\
 \hline
 0 \ 0 \ 0 \ 0 \ 0 \ 0.1 \ 1 \ 0 \ 1 \ 0 \ 1 \ 1
 \end{array}$$

(b)

- extend all 1s

flip the sign bit

$B = 1\ 1\ 1\ 1\ 1\ 1\ .\ 1\ 1\ 0\ 1\ 0\ 1\ 1$
 $B = 1\ 1\ 1\ 1\ 1\ 1\ .\ 1\ 1\ 0\ 1\ 0\ 1\ 1$
 + 1 → add 1 at the location of sign bit
1\ 1\ 1\ 1\ 1\ 1\ 1\ .\ 1\ 1\ 0\ 1\ 0\ 1\ 1

Ejemplo de eliminación de extensión de signo

Multiplicador Signado en Completo a Dos

Eliminación de extensión de signo

- El bit de signo 0 es invertido a 1, se suma un 1 en la posición del bit de signo y los bits extendidos se reemplazan por todos los 1s.
- Esta técnica que trabaja de forma equivalente en números negativos, se observa en la imagen (b) anterior.
- El bit de signo 1 se invierte a 0, se suma un 1 a la posición del bit de signo y los bits extendidos son todos 1s.
- Con lo que la técnica hace que todos los bits extendidos sean 1, independientemente del signo del número.
- Para eliminar la lógica de extensión de signo, todos los 1s se suman fuera de línea para formar un vector de corrección.

Multiplicador Signado en Completo a Dos

Eliminación de extensión de signo

- En la siguiente imagen se ven los pasos necesarios en la lógica de eliminación de signo de la extensión de signo en un multiplicador signado de $11 * 6$ bit.
- Primero, el MSB de todos los PPs, excepto el último, se da vuelta, suma un 1 en la ubicación del bit de signo, y el número se extiende por todos los 1s.
- Para el último PP, el complemento a 2 se calcula invirtiendo todos los bits y sumando 1 a la posición LSB.
- El MSB del último PP se invierte y suma 1 a esta ubicación de bit para la extensión de signo.
- Todos estos 1s son sumados para obtener un *vector de corrección* (CV).
- Todos los 1s se quitan, el CV simplemente se suma y se encarga de la lógica de la extensión de signo.

Multiplicador Signado en Completo a Dos

Eliminación de extensión de signo

111111	1 1 1 1 1 1	1
11111 SXXXXXXXXXX	1 1 1 1 1	
1111 SXXXXXXXXXX	1 1 1 1	
111 SXXXXXXXXXX	1 1 1	
11 SXXXXXXXXXX	1 1	
1 SXXXXXXXXXX	1	
SXXXXXXXXXX	1's complement	
1	and adding 1 at LSB	
	<hr/>	
	0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0	

Eliminación de extensión de signo y formulación de CV para multiplicación signada

Multiplicador Signado en Completo a Dos

Eliminación de extensión de signo

Ejemplo

- En el siguiente ejemplo se encuentra el CV para un multiplicador signado de 4x4 bits y se lo usa para multiplicar 2 números: 0011 y 1101.
- En la figura "a" de la siguiente imagen, todos los 1s para la extensión de signo y los complemento a 2 se suman , y el $CV = 0001_0000$
- Aplicando la lógica de la eliminación de extensión de signo y sumando el CV a los PPs, la multiplicación se realiza de nuevo dando el mismo resultado, como se puede ver en la figura "b" de la siguiente imagen.
- Como el CV tiene un solo bit distinto de cero, el bit se añade con el primer PP (mostrado en gris).
- Esta técnica ahorra área y por lo tanto es muy eficaz.

Multiplicador Signado en Completo a Dos

Eliminación de extensión de signo

(a)

Negative Number

Correction
Vector

0	0	1	0
1	1	0	1
1	1	1	1
1	1	1	0
1	1	1	0
1	1	0	0
1	1	0	0
1	0	0	1
1	0	0	1
			1

(b)

Correction Vector

0	0	1	0
1	1	0	1
1	1	0	1
1	1	0	1
1	0	0	0
1	0	1	0
0	1	0	1
1	1	1	1
0	1	0	1
1	1	1	1

Multiplicación entre 0011 y 1101.

Multiplicador Signado en Completo a Dos

Propiedades de la cadenas

- Hasta ahora sólo se han representado números en forma de complemento a 2, donde cada bit es 0 o 1.
- También existen otras formas para representar números.
- Algunas pocas son eficaces para el diseño de sistemas de procesamiento de señales, como ser el dígito canónico con signo (CSD).
- En CSD, un dígito puede ser 1, 0 o -1.
- La representación restringe la ocurrencia de 2 dígitos distintos de ceros consecutivos en el número, generando una representación única con un número mínimo de dígitos distintos de cero.
- El CSD de un número se puede calcular utilizando la propiedad de cadena de los números.
- Esta propiedad, mientras pasa de LSB a MSB, encuentra sucesivamente cadenas de 1s y las reemplaza con un valor equivalente, usando 1, 0 o -1.

Multiplicador Signado en Completo a Dos

Propiedad de cadenas

- Considerando el número 7, este puede escribirse como $8 - 1$, o en CSD como:

$$0111 = 1000 - 1 = 100\bar{1}$$

- El bit con una barra tiene peso negativo y los demás tienen peso positivo.
- De forma similar, el 31 puede escribirse como $32 - 1$, o en CSD como:

$$011111 = 100000 - 1 = 10000\bar{1}$$

- Así, de forma equivalente, una cadena de 1s se reemplaza con $\bar{1}$ en el 1 menos significativo de la cadena, y un 1 es colocado después del 1 más significativo de la cadena, muestra que todos los demás bits se llenan de ceros.
- Se puede extender de manera trivial esta transformación a cualquier cadena de 1s en representación binaria de un número.
- La propiedad de cadena se puede aplicar recursivamente en representaciones binarias de un número.
- El número transformado tiene un número mínimo de bits distintos de cero.

Multiplicador Signado en Completo a Dos

Propiedad de cadena



Aplicación de la propiedad de cadena.

Multiplicador Signado en Completo a Dos

Multiplicador Modificado de Booth

Introducción

- Los tres bloques básicos de un multiplicador son la generación de los productos parciales, la reducción de los productos parciales a dos capas, y la suma de estas capas usando CSA.
- Reducir el número de productos parciales es una técnica de optimización que se utiliza en muchos diseños.
- El multiplicador modificado de Booth (*MBR*) es una de estas técnicas.

Multiplicador Signado en Completo a Dos

Multiplicador Modificado de Booth

Descripción

- Cuando multiplicamos dos números de N bits signados a y b, la técnica genera un producto parcial por cada uno a través del emparejamiento de todos los bits de b en grupos de dos bits.
- La técnica, al pasar del LSB al MSB de b, empareja dos bits juntos para hacer un grupo que será recodificado usando el algoritmo MBR.
- Los dos bits de un grupo pueden ser 00, 01, 10, 11 (*binarios*).
- La multiplicación por 00, 01 y 10 simplemente resultan en 0, a y $2a = a << 1$ respectivamente, donde cada producto parcial es calculado como un número.
- La cuarta posibilidad es 11 binario que es igual a 3 decimal, igual a $2 + 1$, y un simple corrimiento podría no generar el producto parcial requerido. Esta multiplicación dará como resultado dos productos parciales que son a y $2a$. Esto significa que en el peor caso el multiplicador tendrá N productos parciales.

Multiplicador Signado en Completo a Dos

Multiplicador Modificado de Booth

Descripción

- El problema del caso 11 para generar dos productos parciales es resuelto usando el algoritmo de recodificación de Booth.
- Este algoritmo trabaja con grupos de dos bits pero recodifica cada grupo para usar uno de los cinco valores equivalentes: 0, 1, 2, -1, -2. La multiplicación por todos estos dígitos resulta en un producto parcial por cada uno.
- Estos valores equivalentes son codificados indexándolos en una LUT. Esta LUT es calculada utilizando la propiedad de cadena de números vista anteriormente.
- Esta propiedad es observada en cada par de bits desde del LSB al MSB.
- Para comprobar la propiedad de cadena, es requerido también el MSB del par anterior junto con los dos bits del par bajo consideración. Para el primer par se añade un cero a la derecha.
- La Tabla siguiente muestra la propiedad de cadena trabajando con todos los números posibles de 3 bits para generar una tabla que es luego usada para recodificar.

Multiplicador Signado en Completo a Dos

Multiplicador Modificado de Booth

$2^1 2^0$	String property implemented	Numeric computations	Recoded value
000	No string	0	0
001	End of string at bit location 0	2^0	1
010	Isolated 1	1	1
011	End of string at bit location 0	2^1	2
100	Start of string at bit location 1	-2^1	-2
101	End and start of string at bit locations 0 and 1, respectively	$-2^0 - 2^1$	-1
110	Start of string at bit location 0	-2^0	-1
111	Middle of string	0	0

Recodificador de Booth modificado usando propiedad de cadena

Multiplicador Signado en Completo a Dos

Multiplicador Modificado de Booth

Ejemplo

- En este ejemplo se multiplican dos números de 8 bits 10101101 y 10001101 usando la técnica antes descripta.
- La técnica primero separa los bits en el multiplicador en grupos de dos bits. Luego el MSB del grupo anterior recodifica cada grupo usando la tabla previa. Se supone un cero para el MSB del grupo menos significativo ya que no tiene grupo anterior.
- Los 8 grupos con el MSB del grupo anterior son:
100001110010
- Se observa en la tabla que los números son recodificados a -2, 1, -1 y 1 y los cuatro productos parciales son generados como se verá en la próxima imagen.
- Para un dígito 1 recodificado en el multiplicador, el multiplicando es simplemente copiado. Como cada producto parcial es generado de un par de 2 bits del multiplicador, el i producto parcial es desplazado $2i$ posiciones hacia la izquierda.
- Para el segundo dígito recodificado del multiplicador que es -1, el complemento a 2 del multiplicando es copiado.

Multiplicador Signado en Completo a Dos

Multiplicador Modificado de Booth

Ejemplo

- Los productos parciales son generados para todos los dígitos recodificados, para el ultimo dígito de -2 el complemento a 2 del producto parcial es desplazado más a la izquierda una posición de bit para realizar la multiplicación por dos.
- Se extiende el signo de los cuatro productos parciales y luego son sumados para obtener el producto final.
- La lógica de eliminación de extensión de signo podría usarse para reducir la lógica de implementación de hardware.

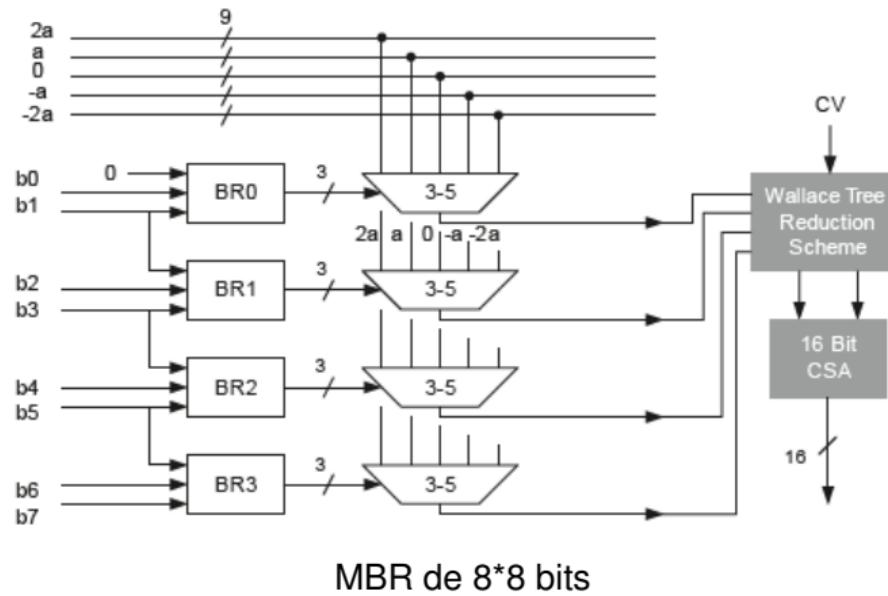
Multiplicador Signado en Completo a Dos

Multiplicador Modificado de Booth

10101101 (-83)	Multiplicador
10001101 (-115)	Binario
<hr/>	
111111110101101	
11111110101101	Productos
111110101101	Parciales
001010011	
<hr/>	
0010010101001001	
10101101 (-83)	Multiplicador de Booth
-2 0 1 0 -1 0 1	
<hr/>	
111111110101101	100011010 (1)
00000001010011	100011010 (-1)
111110101101	100011010 (1)
001010011	100011010 (-2)
<hr/>	
0010010101001001	

Multiplicador Signado en Completo a Dos

Multiplicador Modificado de Booth



Multiplicador Signado en Completo a Dos

Multiplicador 6 x 6-Bits MBR

Implementación en Verilog

```
1 module BOOTH_MULTIPLIER
2   (input [5:0]   multiplier,
3   input [5:0]   multiplicand,
4   output [10:0] product);
5   parameter WIDTH = 6;
6   reg [6:0]     pps [0:2];
7   reg [10:0]    correctionVector;
8   reg [2:0]     recoderOut[2:0];
9   wire [6:0]    a, a_n;
10  wire [6:0]   _2a, _2a_n;
11  assign a_n   = { multiplicand[WIDTH-1], ~multiplicand };
12  assign a     = { ~multiplicand[WIDTH-1], multiplicand };
13  assign _2a_n = {multiplicand[WIDTH-1], ~multiplicand[WIDTH-2:0], 1'b0};
14  assign _2a   = { ~multiplicand[WIDTH-1],multiplicand[WIDTH-2:0], 1'b0};
15  assign product = pps[0] + {pps[1],2'b00} + {pps[2],4'b0000} + correctionVector;
16 always@*
17 begin
18   recoderOut[0] = RECODERfn ({multiplier[1:0],1'b0});
19   recoderOut[1] = RECODERfn (multiplier[3:1]);
20   recoderOut[2] = RECODERfn (multiplier[5:3]);
21   GENERATE_PPtk(recoderOut[0],a,_2a,a_n,_2a_n,pps[0],correctionVector[1:0]);
22   GENERATE_PPtk(recoderOut[1],a,_2a,a_n,_2a_n,pps[1],correctionVector[3:2]);
23   GENERATE_PPtk(recoderOut[2],a,_2a,a_n,_2a_n,pps[2],correctionVector[5:4]);
24   correctionVector[10:6] = 5'b01011;
25 end
```

Multiplicador Signado en Completo a Dos

Multiplicador 6 x 6-Bits MBR

Implementación en Verilog

```
1  task GENERATE_PPtk;
2    input [2:0] recoderOut;
3    input [WIDTH:0] a;      input [WIDTH:0] _2a;
4    input [WIDTH:0] a_n;    input [WIDTH:0] _2a_n;
5    output [WIDTH:0] ppi;   output [1:0]    correctionVector;
6    reg [WIDTH-1:0] zeros;
7    begin
8      zeros = 0;
9      case(recoderOut)
10        3'b000:begin
11          ppi = {1'b1,zeros};  correctionVector = 2'b00;
12        end
13        3'b001:begin
14          ppi = a;            correctionVector = 2'b00;
15        end
16        3'b010:begin
17          ppi = _2a;          correctionVector = 2'b00;
18        end
19        3'b110:begin
20          ppi = _2a_n;        correctionVector = 2'b10;
21        end
22        3'b111:begin
23          ppi = a_n;         correctionVector = 2'b01;
24        end
25        default:begin
26          ppi = 'bx;          correctionVector = 2'bx;
27        end
28      endcase
29    end
30  endtask
```

Multiplicador Signado en Completo a Dos

Multiplicador 6 x 6-Bits MBR

Implementación en Verilog

```
1  function [2:0] RECODERfn;
2    input [2:0] recoderIn;
3    begin
4      case(recoderIn)
5        3'b000: RECODERfn = 3'b000;
6        3'b001: RECODERfn = 3'b001;
7        3'b010: RECODERfn = 3'b001;
8        3'b011: RECODERfn = 3'b010;
9        3'b100: RECODERfn = 3'b110;
10       3'b101: RECODERfn = 3'b111;
11       3'b110: RECODERfn = 3'b111;
12       3'b111: RECODERfn = 3'b000;
13       default: RECODERfn = 3'bx;
14     endcase
15   end
16 endfunction
17 endmodule
```

Reducción del Árbol de Suma



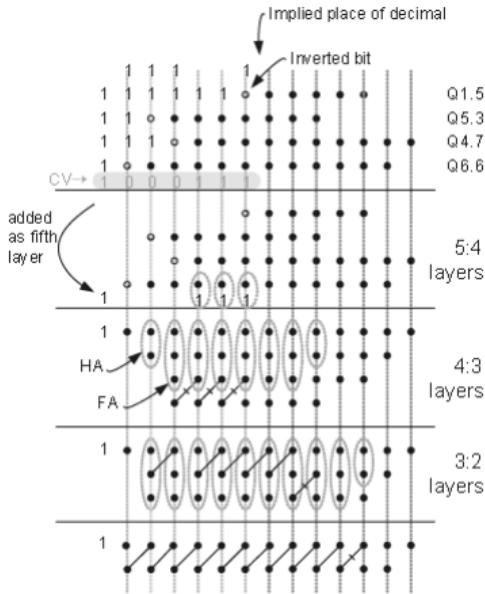
Reducción del Árbol de Suma

Árboles Compresores

- Aunque varios dispositivos de las familias de las FPGA ofrecen multiplicadores embebidos, los árboles compresores todavía son críticos en muchas aplicaciones.
- El árbol de compresión es el primer bloque de construcción en reducir los requerimientos del número de CPAs para la suma de múltiples operadores.
- En el siguiente ejemplo, se agregan 5 operadores signados en los formatos S(6.5), S(8.3), S(11.7) y S(12.6).
- La lógica de extensión de signo que se construye primero, se elimina mediante el cálculo de un vector de corrección y se agrega como la sexta capa en representación de puntos.
- En la figura se muestra que la colocación de los puntos en una cuadrícula, requiere un árbol de compresión para reducir el número de puntos en cada columna a 2. Esto se puede demostrar utilizando cualquier técnica de reducción. Aquí emplearemos la reducción de Dadda.
- Luego, los dos operandos se introducen en un CPA para la suma final.

Reducción del Árbol de Suma

Árboles Compresores



Uso de un árbol de compresión en adición de múltiples operandos

Trasformaciones de Algoritmos para CSA



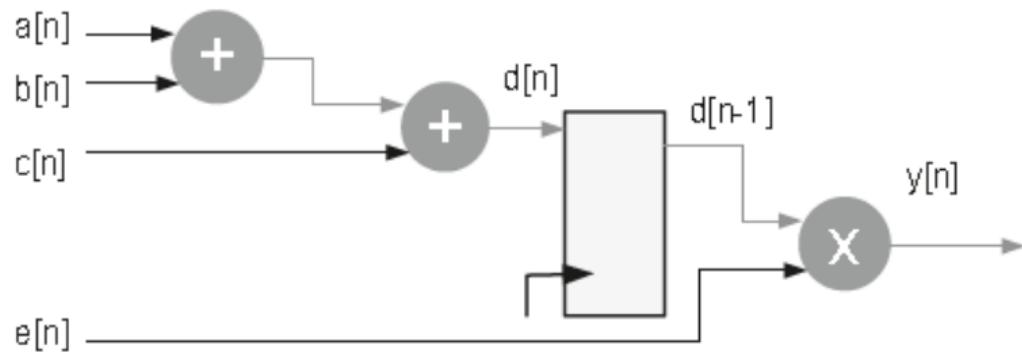
Trasformaciones de Algoritmos para CSA

Transformación de Algoritmos

- El CSA desempeña un papel importante en la implementación de aplicaciones DSP de alto rendimiento en hardware.
- Como primer paso, mientras se asigna un gráfico de flujo de datos a la arquitectura, se observa que el gráfico muestra cualquier uso potencia de CSA y, en consecuencia, el gráfico se modifica.
- Por ejemplo, considere implementar las siguientes ecuaciones:
 $d[n] = a[n] + b[n] + c[n]$
 $y[n] = d[n - 1] * e[n]$
Las ecuaciones son convertidas a DFG en la figura (a).
- Esto se modifica usando CSA para comprimir
 $a[n] + b[n] + c[n]$
en dos números, que luego se suman usando un CPA. Éste DFG transformado se muestra en la figura (b).

Trasformaciones de Algoritmos para CSA

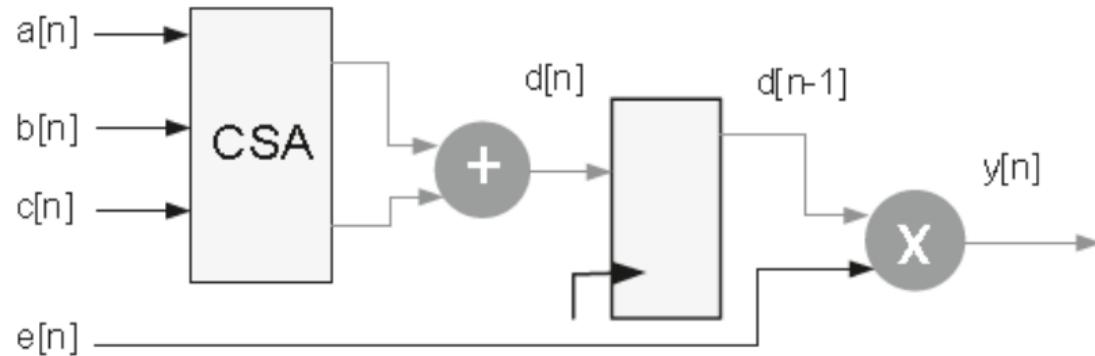
Ejemplo



(a) FSFG con suma de múltiples operandos

Trasformaciones de Algoritmos para CSA

Ejemplo



(b) FSFG modificado reduciendo 3 operandos a 2.

Trasformaciones de Algoritmos para CSA

Transformación de Algoritmos

- Ésta técnica de extracción de suma de múltiples operandos puede extenderse a gráficos de flujo de datos, donde se observa que los gráficos muestran cualquier uso potencial de CSA y árboles compresores.
- Los gráficos se transforman primero para utilizar óptimamente los árboles compresores y luego se mapean en hardware.
- Éstas transformaciones han demostrado mejorar significativamente el diseño de hardware de las aplicaciones de procesamiento de señales.
- Las operaciones de suma múltiple son las más fáciles de todas las transformaciones.

Trasformaciones de Algoritmos para CSA

Transformación de Algoritmos

- El árbol compresor también puede colocarse en la siguiente operación de suma comparación-selección:

$sum1 = op1 + op2;$

$sum2 = op3 + op4;$

if($sum1 > sum2$)

.....*sel* = 0;

else

.....*sel* = 1;

- Para transformar la lógica para el uso óptimo de un árbol compresor, el algoritmo se modifica como sigue:

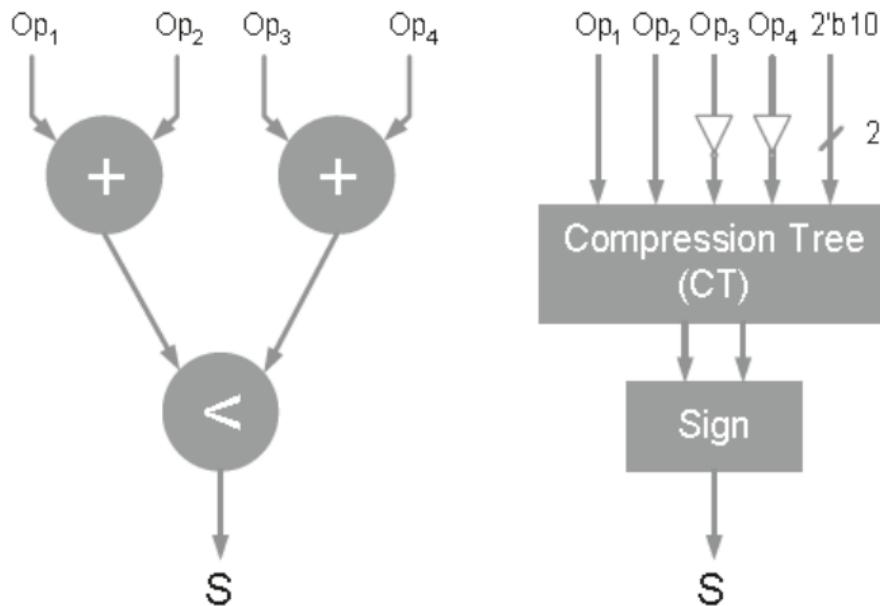
$$\text{sign}(op1 + op2 - (op3 + op4)) = \text{sign}(op1 + op2 - op3 - op4)$$

$$\text{sign}(op1 + op2 + op3' + 1 + op4' + 1) = \text{sign}(op1 + op2 + op3' + op4' + 2)$$

- Esta transformación del árbol compresor en el equivalente DFG se muestra en la siguiente figura.

Trasformaciones de Algoritmos para CSA

Equivalencia del árbol compresor en DFG



Reemplazo del árbol compresor por una operación de suma de comparación y selección

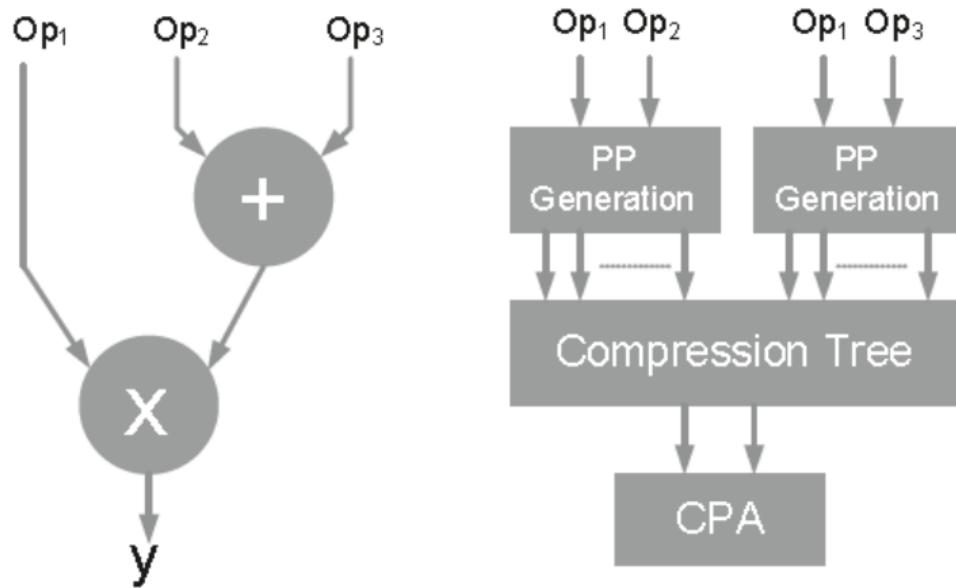
Trasformaciones de Algoritmos para CSA

Transformación de Algoritmos

- De forma similar a la anterior, la siguiente operación de sumar y multiplicar se puede representar con su equivalente DFG:
 $op1 * (op2 + op3)$
- El DFG puede ser transformado para usar eficazmente un árbol compresor.
- Una implementación directa requiere un CPA para realizar $op2 + op3$, y luego el resultado de esta operación es multiplicado por $op1$.
- La arquitectura de un multiplicador comprende un árbol compresor y un CPA.
- Para implementar el cálculo, se requieren 2 CPA.
- Una transformación simple utiliza la propiedad distributiva del operador de multiplicación:
 $op1 * op2 + op1 * op3$
- Esta representación de la expresión ahora requiere un árbol de compresión, que luego es seguido por un CPA para calcular el valor final.
- El DFG asociado y la transformación se ven en la siguiente figura.

Trasformaciones de Algoritmos para CSA

DFG asociado y la transformación



Transformar la operación de sumar y multiplicar para usar un CPA y un árbol de compresión.

Trasformaciones de Algoritmos para CSA

Transformación de Algoritmos

- Ampliar la técnica de generar sumas parciales y acarreo, también puede optimizar la implementación de una cascada de multiplicaciones:

$$\text{prod} = \text{op1} * \text{op2} * \text{op3} * \text{op4}$$

- La transformación primero genera PPs para $\text{op1} * \text{op2}$ y los reduce a los PPs, $s1$ y $c1$:

$$(s1, c1) = \text{op1} * \text{op2}$$

- Estos 2 PPs se multiplican independientemente con op3 , generando 2 conjuntos PPs que se reducen de nuevo a 2 PPs, $s2$ y $c2$, usando un árbol de compresión:

$$(s2, c2) = s1 * \text{op3} + c1 * \text{op3}$$

- Estos dos PPs se multiplican con op4 para generar 2 conjuntos de PPs que se comprimen de nuevo para calcular 2 PPs finales, $s3$ y $c3$. Estos dos PPs se agregan a continuación usando un CPA para calcular el producto final:

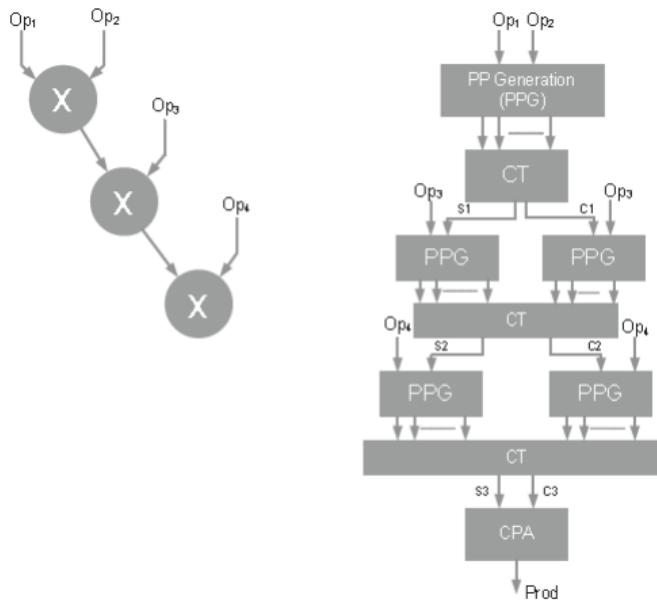
$$(s3, c3) = s2 * \text{op4} + c2 * \text{op4}$$

$$\dots \text{prod} = s3 + c3$$

Esto se ilustra en la siguiente imagen.

Trasformaciones de Algoritmos para CSA

Transformación equivalente en DFG



Transformación para usar árboles de compresión y un solo CPA para implementar una cascada de operaciones de multiplicación.

Trasformaciones de Algoritmos para CSA

Transformación de Algoritmos

- Siguiendo los ejemplos anteriores, se pueden usar varias transformaciones usando propiedades matemáticas básicas para acumular varios operadores para el uso efectivo de árboles de compresión para mapeo de hardware optimizado.

Multiplicación por Constantes



Multiplicación por Constantes

En donde se utiliza?

- En muchos sistemas de procesamiento digitales (DSP) y algoritmos de comunicación, una gran proporción de multiplicaciones son números constantes.
- Por ejemplo:
 - Filtros de respuesta de impulso finito (FIR).
 - Filtros respuesta de impulso infinito (IIR).
 - La transformada de coseno discreta (DCT).
 - La transformada de coseno discreto inverso (IDCT).
 - La transformada rápida de Fourier (FFT).
 - La transformada rápida de Fourier inversa (IFFT).
- Para una arquitectura totalmente dedicada (FDA), donde la multiplicación por una constante se asigna a un multiplicador dedicado, no se requiere la complejidad de un multiplicador de propósito general.
- La representación binaria de una constante muestra claramente los bits distintos de cero que requieren la generación de productos parciales (PP), mientras que los bits que son cero en la representación se pueden ignorar para la operación de generación de PP.
- La representación CSD puede reducir aún más el número de productos parciales.

Multiplicación por Constantes

Dígito Canónico con Signo (CSD)

Concepto

- Es un código de dígito signado radix-2.
- Codifica una constante usando dígitos signados 1, 0 y -1.
- Representación de constante de N-bits:

$$C = \sum_{i=0}^{N-1} s_i 2^i; \quad s_i \in \{-1, 0, 1\} \quad (11)$$

- Propiedades
 - No hay dos bits consecutivos en la representación de CSD de un número que no sean cero.
 - La representación de CSD de un número usa un número mínimo de dígitos distintos de cero.
 - La representación de CSD de un número es única.

Multiplicación por Constantes

Dígito Canónico con Signo (CSD)

Concepto

- La representación de CSD de un número se puede calcular recursivamente usando la propiedad de la cadena.
- El LSB en una cadena de 1s se cambia a $\bar{1}$ que representa -1, y todos los otros 1s en la cadena se reemplazan por ceros, y el 0 que marca el final de la cadena se cambia a 1.
- Despues de reemplazar una cadena por su dígitos CSD equivalentes, el número se observa nuevamente moviéndose desde el dígito codificado al MSB para contener cualquier cadena adicional de 1s.
- El proceso se repite hasta que no se encuentre una cadena de 1 en el número.

Ejemplo

0011111011110111
0011111011111001
0011111100001001
0100000100001001

Arquitecturas Dedicadas de Filtros FIR



Arquitecturas Dedicadas de Filtros FIR

Forma Directa

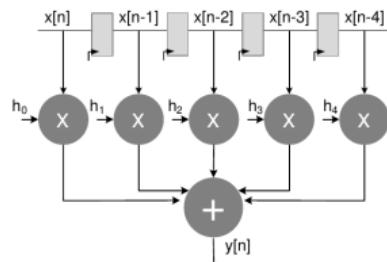
Concepto

- El filtro FIR es muy común en aplicaciones de procesamiento de señal.
- Un filtro FIR se implementa como

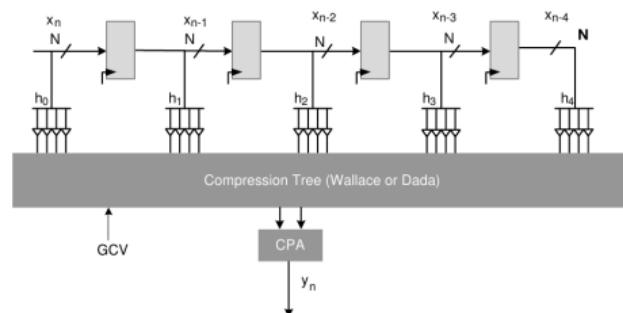
$$y[n] = \sum_{k=0}^{L-1} h[k]x[n-k] \quad (12)$$

- Una implementación del FIR requiere que todas estas multiplicaciones y adiciones se ejecuten simultáneamente, requiriendo L multiplicadores y L-1 sumadores.
- La multiplicación con coeficientes constantes puede explotar la simplicidad del multiplicador CSD.
- Cada uno de estos multiplicadores, en muchas instancias de diseño, se simplifica aún más al restringir a cuatro el número de dígitos CSD distintos de cero en cada coeficiente.
- Un dígito de CSD distinto de cero en un coeficiente contribuye aproximadamente 20 dB de atenuación de banda de corte.
- La atenuación de la banda de corte es una medida de la efectividad de un filtro.

Arquitecturas Dedicadas de Filtros FIR



FIR forma directa.



Implementación.

Arquitecturas Dedicadas de Filtros FIR

FIR con multiplicadores

```
1 module FIRfilter
2 (
3     input signed [15:0]      x,
4     input                  clk,
5     output reg signed [31:0] yn);
6     reg signed [15:0]      xn [0:4];
7     wire signed [31:0]      yn_v;
8 // Coefficients of the filter
9     wire signed [15:0]      h0 = 16'h0325;
10    wire signed [15:0]      h1 = 16'h1e00;
11    wire signed [15:0]      h2 = 16'h3DB6;
12    wire signed [15:0]      h3 = 16'h1e00;
13    wire signed [15:0]      h4 = 16'h0325;
14 // Implementing filters using multiplication and addition operators
15 assign yn_v = (h0*xn[0] + h1*xn[1] + h2*xn[2] + h3*xn[3] + h4*xn[4]);
16 always @ (posedge clk)
17 begin
18     // Tap delay line of the filter
19     xn[0] <= x;
20     xn[1] <= xn[0];
21     xn[2] <= xn[1];
22     xn[3] <= xn[2];
23     xn[4] <= xn[3];
24     // Registering the output
25     yn <= yn_v;
26 end
27 endmodule
```

Arquitecturas Dedicadas de Filtros FIR

FIR usando CSD

```
1 module FIRfilterCSD
2   (input signed [15:0]      x,
3    input                  clk ,
4    output reg signed [31:0] yncsd);
5   reg signed [31:0]         yncsd_v;
6   reg signed [31:0]         xn [0:4];
7   reg signed [31:0]         pp[0:15];
8   always @(posedge clk)
9     begin
10       xn[0] <= {x, 16'h0};
11       xn[1] <= xn[0];
12       xn[2] <= xn[1];
13       xn[3] <= xn[2];
14       xn[4] <= xn[3];
15       yncsd <= yncsd_v;
16     end
17   always @ (*) begin
18     pp[0] = xn[0]>>>5;
19     pp[1] = -xn[0]>>>7;
20     pp[2] = xn[0]>>>10;
21     pp[3] = xn[0]>>>13;
22
23     pp[4] = xn[1]>>>2;
24     pp[5] = - xn[1]>>>6;
25
26     pp[6] = xn[2]>>>1;
27     pp[7] = -xn[2]>>>6;
28     pp[8] = -xn[2]>>>9;
29     pp[9] = -xn[2]>>>12;
30
31     pp[10] = xn[3]>>>2;
32     pp[11] = -xn[3]>>>6;
33
34     pp[12] = xn[4]>>>5;
35     pp[13] = -xn[4]>>>7;
36     pp[14] = xn[4]>>>10;
37     pp[15] = xn[4]>>> 13;
38
39     yncsd_v = pp[0]+pp[1]+pp[2]+pp[3]+pp[4]+pp[5]+
40               pp[6]+pp[7]+pp[8]+pp[9]+pp[10]+pp[11]+
41               pp[12]+pp[13]+pp[14]+pp[15];
42   end
43 endmodule
```

Arquitecturas Dedicadas de Filtros FIR

FIR usando CV

```
1 module FIRfilterCV
2   (input signed [15:0]      x,
3    input                  clk ,
4    output reg signed [31:0] yn
5  );
6   reg signed [31:0] yn_v;
7   reg signed [15:0] xn_0,xn_1,
8           xn_2,xn_3,xn_4;
9   reg signed [31:0] pp[0:15];
10  reg signed [31:0] gcv = 32'h6F701090;
11  always @(posedge clk)
12    begin
13      xn_0 <= x;
14      xn_1 <= xn_0;
15      xn_2 <= xn_1;
16      xn_3 <= xn_2;
17      xn_4 <= xn_3;
18      yn <= yn_v;
19    end
20  always @ (*)begin
21    pp[0]= {5'b0, ~xn_0[15],
22            xn_0[14:0], 11'b0};
23    pp[1] = {7'b0, xn_0[15],
24              ~xn_0[14:0], 9'b0};
25    pp[2] = {10'b0, ~xn_0[15],
26              xn_0[14:0], 6'b0};
27    pp[3] = {13'b0, ~xn_0[15],
28              xn_0[14:0], 3'b0};
29
30    pp[4] = {2'b0, ~xn_1[15],
31              xn_1[14:0], 14'b0};
32    pp[5] = {6'b0, xn_1[15],
33              ~xn_1[14:0], 10'b0};
34    pp[6] = {1'b0, ~xn_2[15],
35              xn_2[14:0], 15'b0};
36    pp[7] = {6'b0, xn_2[15],
37              ~xn_2[14:0], 10'b0};
38    pp[8] = {9'b0, xn_2[15],
39              ~xn_2[14:0], 7'b0};
40    pp[9] = {12'b0, xn_2[15],
41              ~xn_2[14:0], 4'b0};
42    pp[10] = {2'b0, ~xn_3[15],
43               xn_3[14:0], 14'b0};
44    pp[11] = {6'b0, xn_3[15],
45               ~xn_3[14:0], 10'b0};
46    pp[12]= {5'b0, ~xn_4[15],
47               xn_4[14:0], 11'b0};
48    pp[13] = {7'b0, xn_4[15],
49               ~xn_4[14:0], 9'b0};
50    pp[14] = {10'b0, ~xn_4[15],
51               xn_4[14:0], 6'b0};
52    pp[15] = {13'b0, ~xn_4[15],
53               xn_4[14:0], 3'b0};
54
55    yn_v = pp[0]+pp[1]+pp[2]+pp[3]+
56              pp[4]+pp[5]+pp[6]+pp[7]+
57              pp[8]+pp[9]+pp[10]+pp[11]+
58              pp[12]+pp[13]+pp[14]+
59              pp[15]+gcv;
60
61  end
62 endmodule
```

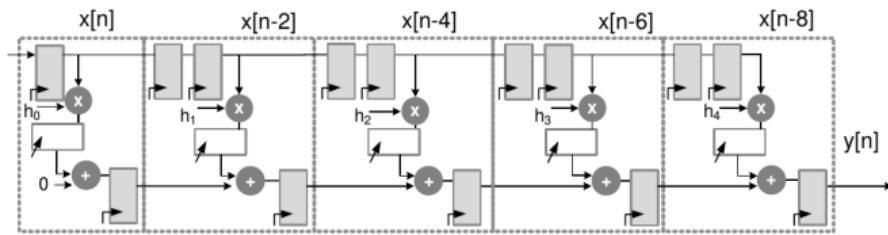
Arquitecturas Dedicadas de Filtros FIR

Forma Directa Transpuesta

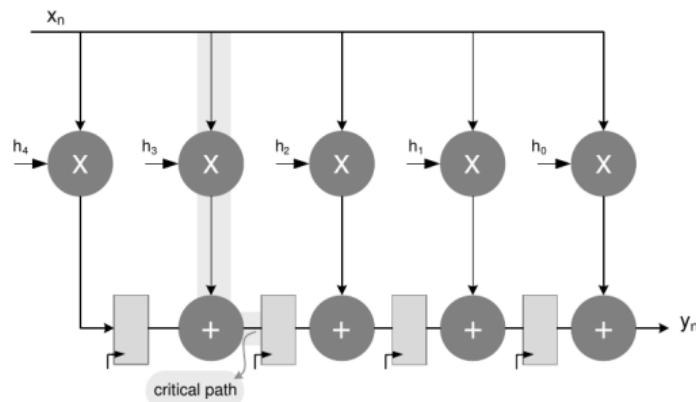
Concepto

- La estructura de filtro FIR de forma directa da como resultado una gran nube combinacional de árbol de reducción y CPA.
- El camino crítica consiste en un multiplicador y un sumador.
- Agregar pipeling para reducir el camino crítico causa latencia y una gran sobrecarga de área en la implementación de registros.
- Retiming es una técnica efectiva para mover sistemáticamente retrasos algorítmicos en un diseño para reducir el camino crítica del circuito lógico.
- La técnica retiming se aplica para transformar el filtro FIR directo en transpuesto.
- Es interesante observar que de esta forma, sin agregar registros de pipeling, los retrasos del algoritmo se mueven sistemáticamente usando la transformación de retiming desde el borde superior del DFG al borde inferior.

Arquitecturas Dedicadas de Filtros FIR

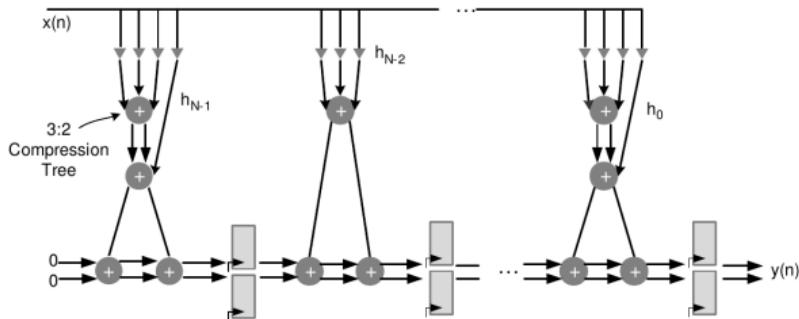


FIR forma directa con pipeline.

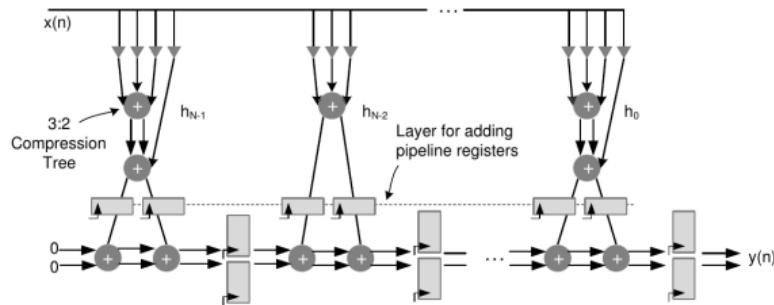


FIR forma directa transpuesta.

Arquitecturas Dedicadas de Filtros FIR

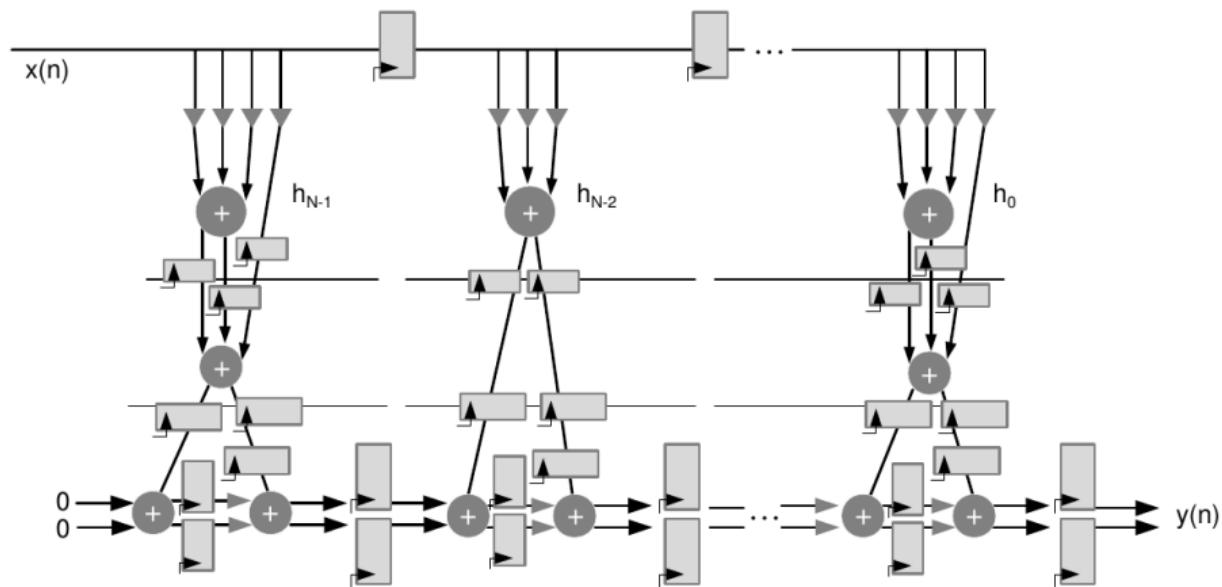


FIR forma directa transpuesta con multiplicadores CSD y sumadores carry save.



FIR forma directa transpuesta con un nivel de pipeling.

Arquitecturas Dedicadas de Filtros FIR



FIR forma directa transpuesta con dos niveles de pipeline.

Arquitecturas Dedicadas de Filtros FIR

TDF

```

1 module FIRfilterTDFComp
2   (input signed [15:0] x,
3    input               clk ,
4    output signed [31:0] yn);
5   integer             i;
6   reg signed [15:0]   xn;
7   reg signed [31:0]   sn_0,sn_1,sn_2,
8                      sn_3,sn_4;
9   reg signed [32:0]   cn_0,cn_1,cn_2,
10                     cn_3,cn_4;
11  reg signed [31:0]   pp_0,pp_1,pp_2,
12    pp_3,pp_4,pp_5,pp_6,pp_7,pp_8,pp_9,
13    pp_10,pp_11,pp_12,pp_13,pp_14,pp_15;
14  reg signed [31:0]   s00,s01,s10,s11,
15    s20,s200,s21,s30,s31,s40,s400,s41,
16    s02,s22,s42;
17  reg signed [32:0]   c00,c01,c10,c11,
18    c20,c200,c21,c30,c31,c40,c400,c41,
19    c02,c22,c42;
20  reg signed [31:0]   gcv = 32'h6F701090;
21  assign yn = cn_4+sn_4;
22  always @ (posedge clk) begin
23    xn <= x; cn_0 <= c02;
24    sn_0 <= s02; cn_1 <= c11;
25    sn_1 <= s11; cn_2 <= c22;
26    sn_2 <= s22; cn_3 <= c31;
27    sn_3 <= s31; cn_4 <= c42;
28    sn_4 <= s42;
29 end

```

```

1 always@(*) begin
2   c00[0]=0;c10[0]=0;c20[0]=0;c200[0]=0;
3   c30[0]=0; c40[0]=0;c400[0]=0;
4     for (i=0; i<32; i=i+1) begin
5       {c00[i+1],s00[i]} =
6         pp_0[i]+pp_1[i]+pp_2[i];
7       {c10[i+1],s10[i]} =
8         pp_4[i]+pp_5[i]+sn_0[i];
9       {c20[i+1],s20[i]} =
10        pp_6[i]+pp_7[i]+pp_8[i];
11      {c200[i+1],s200[i]} =
12        pp_9[i]+sn_1[i]+cn_1[i];
13      {c30[i+1],s30[i]} =
14        pp_10[i]+pp_11[i]+sn_2[i];
15      {c40[i+1],s40[i]} =
16        pp_12[i]+pp_13[i]+pp_14[i];
17      {c400[i+1],s400[i]} =
18        pp_15[i]+sn_3[i]+cn_3[i];
19
20    end
21    c01[0]=0;c11[0]=0;c21[0]=0;
22    c31[0]=0;c41[0]=0;
23  for (i=0; i<32; i=i+1)begin
24    {c01[i+1],s01[i]}=c00[i]+s00[i]+pp_3[i];
25    {c11[i+1],s11[i]}=c10[i]+s10[i]+cn_0[i];
26    {c21[i+1],s21[i]}=c20[i]+s20[i]+c200[i];
27    {c31[i+1],s31[i]}=c30[i]+s30[i]+cn_2[i];
28    {c41[i+1],s41[i]}=c40[i]+s40[i]+c400[i];
29  end

```

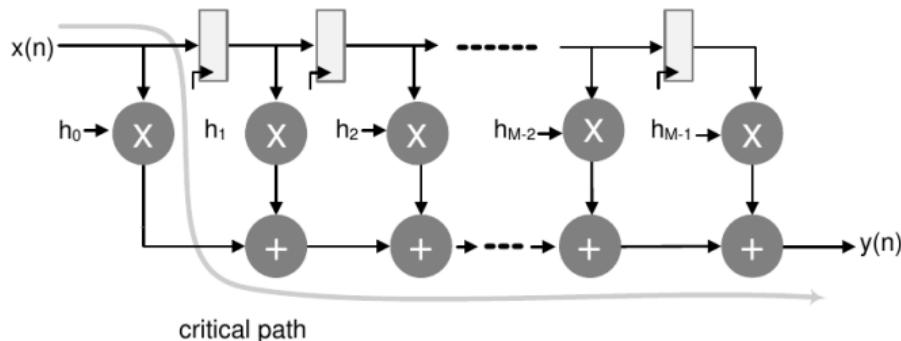
Arquitecturas Dedicadas de Filtros FIR

TDF

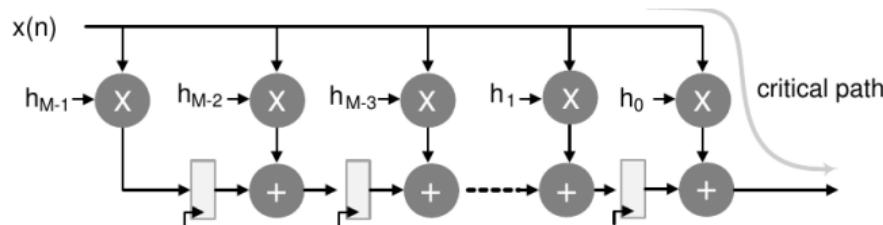
```
1    c02[0]=0; c22[0]=0; c42[0]=0;
2    for (i=0; i<32; i=i+1)
3        begin
4            {c02[i+1],s02[i]} = c01[i]+s01[i]+gcv[i];
5            {c22[i+1],s22[i]} = c21[i]+s21[i]+s200[i];
6            {c42[i+1],s42[i]}= c41[i]+s41[i]+s400[i];
7        end
8    end
9    always @(*) begin
10       pp_0 = {5'b0, ~xn[15], xn[14:0], 11'b0};
11       pp_1 = {7'b0, xn[15], ~xn[14:0], 9'b0};
12       pp_2 = {10'b0, ~ xn[15], xn[14:0],6'b0};
13       pp_3 = {13'b0, ~xn[15], xn[14:0],3'b0};
14       pp_4 = {2'b0, ~xn[15], xn[14:0], 14'b0};
15       pp_5 = {6'b0, xn[15], ~xn[14:0], 10'b0};
16       pp_6 = {1'b0, xn[15], xn[14:0], 15'b0};
17       pp_7 = {6'b0, xn[15], ~xn[14:0], 10'b0};
18       pp_8 = {9'b0, xn[15], ~xn[14:0], 7'b0};
19       pp_9 = {12'b0, xn[15], ~ xn[14:0], 4'b0};
20       pp_10 = {2'b0, ~xn[15], xn[14:0], 14'b0};
21       pp_11 = {6'b0, xn[15], ~xn[14:0], 10'b0};
22       pp_12 = {5'b0, ~xn[15], xn[14:0], 11'b0};
23       pp_13 = {7'b0, xn[15], ~xn[14:0], 9'b0};
24       pp_14 = {10'b0, ~xn[15], xn[14:0],6'b0};
25       pp_15 = {13'b0, ~xn[15], xn[14:0],3'b0};
26   end
27 endmodule
```

Arquitecturas Dedicadas de Filtros FIR

Estructuras FIR híbridas



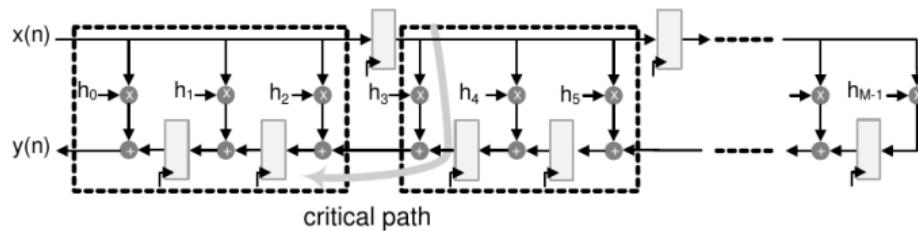
FIR forma directa.



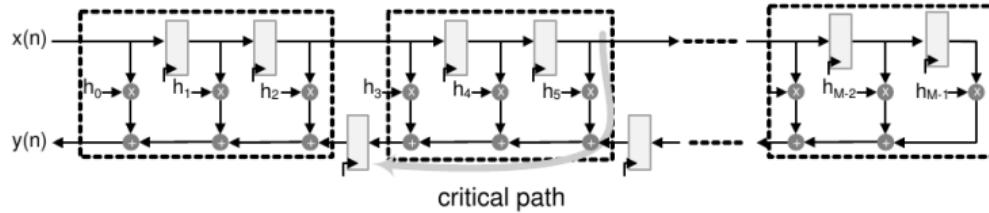
FIR forma directa transpuesta.

Arquitecturas Dedicadas de Filtros FIR

Estructuras FIR híbridas



Estructura Híbrida.



Estructura directa transpuesta híbrida.

Aritmética Distribuida



Aritmética Distribuida

Concepto

- La aritmética distribuida (DA) es otra forma de implementar un producto punto donde una de las matrices tiene elementos constantes.
- El DA se puede utilizar de manera efectiva para implementar algoritmos de tipo FIR, IIR y FFT.
- La lógica DA reemplaza la operación MAC de la suma de convolución en una lectura de tabla de búsqueda en serie de bits y operación de adición.
- Teniendo en cuenta la arquitectura de los FPGA, los diseños efectivos de tiempo/área pueden implementarse usando técnicas DA.
- La lógica de DA funciona expandiendo primero la matriz de números variables en el producto punto como un número binario y luego reorganizando los términos de MAC con respecto al peso de los bits.

$$y = \sum_{k=0}^{K-1} A_k x_k \quad (13)$$

- Donde K , A_k y x_k es la longitud de ambas matrices, los elementos de una matriz de constantes y variables, respectivamente.

Aritmética Distribuida

Concepto

- Considerando que x_k tiene un formato de N-bits S(N,N-1), el producto punto queda definido como

$$y = \sum_{k=0}^{K-1} \left(-x_{k0}2^0 + \sum_{b=1}^{N-1} x_{kb}2^{-b} \right) A_k$$
$$y = -\sum_{k=0}^{K-1} x_{k0}A_k2^0 + \sum_{b=1}^{N-1} 2^{-b} \sum_{k=0}^{K-1} x_{kb}A_k$$

- El tamaño de las palabras almacenadas en la tabla se determinan usando la siguiente expresión

$$P = \left\lceil \log_2 \sum_{k=0}^{K-1} |A_k| \right\rceil + 1 \quad (14)$$

Aritmética Distribuida

x_{2b}	x_{1b}	x_{0b}	Contents of ROM
0	0	0	0
0	0	1	A_0
0	1	0	A_1
0	1	1	$A_1 + A_0$
1	0	0	A_2
1	0	1	$A_2 + A_0$
1	1	0	$A_2 + A_1$
1	1	1	$A_2 + A_1 + A_0$

ROM para Aritmética Distribuida.

Aritmética Distribuida

Ejemplo

Producto y Suma

$$\begin{array}{r} 0101(A0) \\ 0111(x_0) \\ \hline 0101 \end{array} \quad \begin{array}{r} 1011(A1) \\ 1001(x_1) \\ \hline 11111011 \end{array} \quad \begin{array}{r} 0110(A2) \\ 0100(x_2) \\ \hline 0000 \end{array}$$
$$\begin{array}{r} 0101 \\ 0101 \\ 0000 \\ \hline 00100011 \end{array} \quad \begin{array}{r} 0000 \\ 0000 \\ 0110 \\ \hline 0000 \end{array} \quad \begin{array}{r} 00100011 \\ 00100011 \\ 00011000 \\ \hline 0001011110 \end{array}$$

LUT

000000 (0) Aritmética Distribuida

000101 (1)

111011 (2)

1 1 ($x_2 \ x_1 \ x_0$)

000000 (3)

000000 (0 1 1)

000110 (4)

000101 (0 0 1)

001011 (5)

001011 (1 0 1)

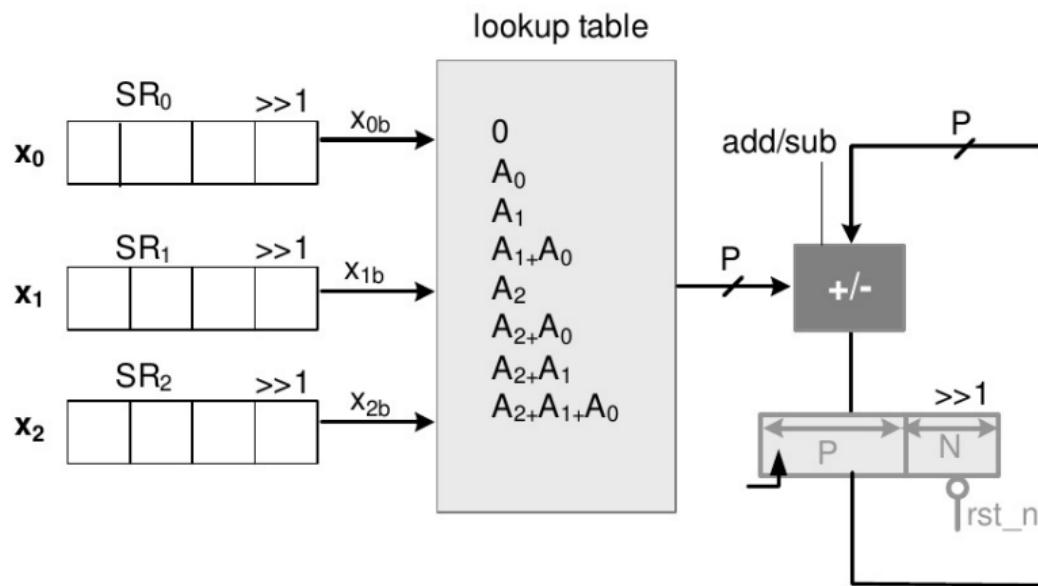
000001 (6)

000101 (0 1 0)

000110 (7)

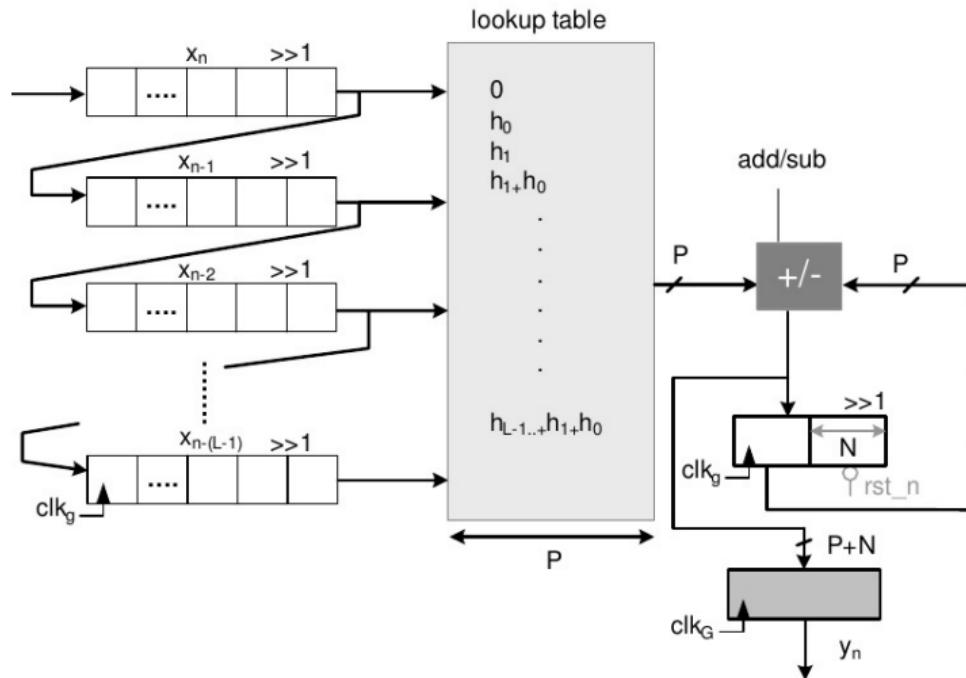
0001011110

Aritmética Distribuida



Implementación de producto punto.

Aritmética Distribuida



Implementación de un filtro FIR.

Aritmética Distribuida

Implementación en Verilog

```
1 module FIRDistributedArithmetics
2   ( input xn_b,clk_g,rst_n,
3     input [3:0]           contr,
4     output reg signed [31:0] yn,
5     output              valid);
6   reg signed [31:0]      acc;
7   reg [15:0]      xn_0, xn_1, xn_2, xn_3;
8   reg [16:0] rom_out; reg [3:0] addr;
9   wire signed [31:0] sum; wire msb;
10  always@(*) begin
11    addr={xn_3[0],xn_2[0],xn_1[0],xn_0[0]};
12    case(addr)
13      4'd0: rom_out=17'b00000000000000000000;
14      4'd1: rom_out=17'b00000001000100100;
15      4'd2: rom_out=17'b00011110111011100;
16      4'd3: rom_out=17'b00100000000000000000;
17      4'd4: rom_out=17'b00011110111011100;
18      4'd5: rom_out=17'b00100000000000000000;
19      4'd6: rom_out=17'b00111101110111000;
20      4'd7: rom_out=17'b00111101110111000;
21      4'd8: rom_out=17'b00000001000100100;
22      4'd9: rom_out=17'b00000010001001000;
23      4'd10: rom_out=17'b00100000000000000000;
24      4'd11: rom_out=17'b00100001000100100;
25      4'd12: rom_out=17'b00100000000000000000;
26      4'd13: rom_out=17'b00100001000100100;
27      4'd14: rom_out=17'b00111101110111000;
28      4'd15: rom_out=17'b01000000000000000000;
29      default: rom_out= 17'bx;
30      endcase
31    end
```

```
1   assign valid = ~ (~ contr);
2   assign msb = contr;
3   assign sum = (acc +
4     {rom_out^{17{msb}}, 16'b0} +
5     {15'b0, msb, 16'b0}) >> 1;
6   always@(posedge clk_g or negedge rst_n)
7     begin
8       if (!rst_n) begin
9         xn_0 <= 0;
10        xn_1 <= 0;
11        xn_2 <= 0;
12        xn_3 <= 0;
13        acc <= 0;
14      end
15      else begin
16        xn_0 <= {xn_b, xn_0[15:1]};
17        xn_1 <= {xn_0[0], xn_1[15:1]};
18        xn_2 <= {xn_1[0], xn_2[15:1]};
19        xn_3 <= {xn_2[0], xn_3[15:1]};
20        if(&contr) begin
21          yn <= sum;
22          acc <= 0;
23        end
24        else
25          acc <= sum;
26      end
27    end
28  endmodule
```