Proyecto de fin de
Máster en Ingeniería
de Computadores
Curso 2006-2007

Algoritmos a nivel de bit para reducir el consumo en la SAN

Alberto Antonio Del Barrio García

Dirigido por:
Profesora: MªCarmen Molina Prego
Departamento: Arquitectura de Computadores y Automática

Facultad de Informática
Universidad Complutense de Madrid

INDICE

1. Introducción 3
2. Análisis preliminar de la potencia 5
2.1. Síntesis de alto nivel con ligaduras de consumo de energía 8
3. Motivación 10
4. Representación de los datos 13
4.1. Casos más comunes 13
4.2. Patrones más comunes 14
4.3. Alternativa a las simulaciones del circuito 15
5. Técnicas para reducir el consumo de los circuitos 20
5.1. Propiedad conmutativa 20
5.2. Fragmentación de operaciones 22
5.3. Inhabilitación de los operadores ociosos 25
5.4. La distancia de Hamming 25
6. Algoritmo de SAN basado en la fragmentación de operaciones dirigida por el consumo estático 27
6.1. Algoritmo propuesto 27
6.2. Resultados experimentales 35
6.3. Conclusiones 39
7. Algoritmo de SAN basado en la fragmentación de operaciones guiada por patrones de comportamiento 41
7.1. Algoritmo propuesto 41
7.2. Resultados experimentales 49
7.3. Conclusiones 51
8. Conclusiones 53
9. Referencias 54
10. Apéndice A: Álgebra para 4 patrones 56
11. Apéndice B: Distancia de Hamming para 4 patrones 57
12. Apéndice C: Fragmento de código del decodificador del adpcm 58

1. Introducción

Históricamente, la potencia consumida por un circuito no ha sido un factor de diseño crucial en la industria de los semiconductores. Los diseñadores han intentado reducir el área y la latencia (λ) de los circuitos propuestos. Actualmente, el área ha sido reducida gracias a los altos niveles integración alcanzados, y el rendimiento se ha mejorado con frecuencias de reloj más altas. En este marco, el consumo del circuito ha adquirido mayor importancia, convirtiéndose en un parámetro tan importante como el área ó el retardo, ya que tanto la alta densidad de los circuitos, como las elevadas frecuencias producen un consumo mucho mayor.

Las necesidades energéticas van unidas a la evolución. Cada vez queremos aplicaciones más potentes, sobre plataformas más rápidas y que ocupen y consuman menos, y esto no sólo en sistemas portátiles como móviles, PDA´s, etc. Un ejemplo lo tenemos en el desarrollo de procesadores de propósito general, donde el consumo se está convirtiendo en una restricción muy a tener en cuenta. Tanto que, hasta el diseño de los mismos se ha replanteado y actualmente se focaliza en la reducción del consumo cuando antes lo único importante era el rendimiento. La Ley de Moore afirma que la frecuencia y el número de transistores se duplican cada 2 años, lo que lleva a un consumo exponencial, lo cual no es sostenible en el desarrollo de procesadores.

La figura 1.1 muestra el ritmo de disipación de potencia en los distintos procesadores. Como podemos ver superamos ampliamente el calor generado por un plato caliente, y nos acercamos al calor disipado por un reactor nuclear. Nótese que quizá esta observación es algo exagerada, pues la escala es logarítmica, pero de seguir a este ritmo sería así.

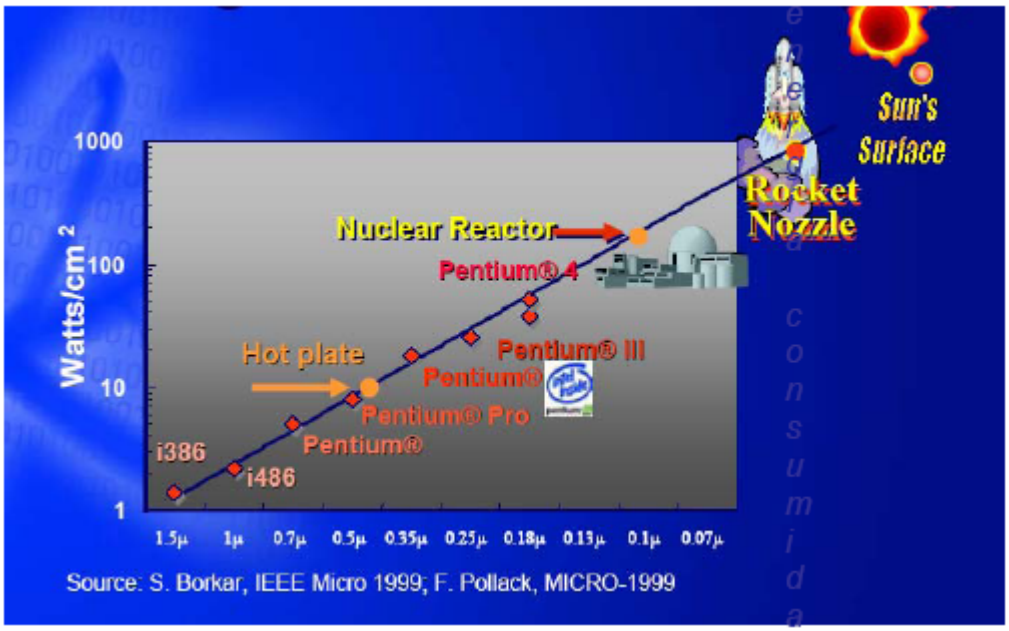


Figura 1.1. Evolución del consumo de los procesadores, según la tecnología del diseño.

2. Análisis preliminar de la potencia

El problema del ahorro de energía tiene su origen en el ahorro de potencia. La potencia es la energía por unidad de tiempo

P = E/t

Por tanto, para disminuir el consumo de potencia basta con aumentar el tiempo de ejecución (véase la figura 2.1). Sin embargo eso supondría una pérdida de rendimiento por lo que, nos centraremos en disminuir el consumo de potencia manteniendo, al menos, ó mejorando también el tiempo de ejecución.

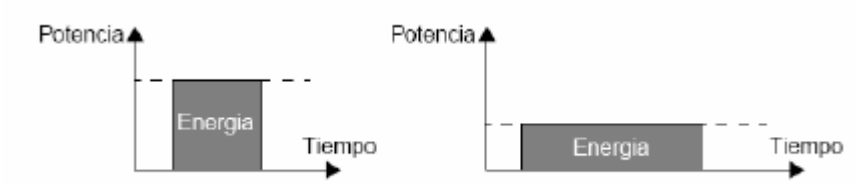


Figura 2.1. Dos formas de repartir la energía en el tiempo

La potencia disipada por un circuito puede tratarse desde los distintos niveles de abstracción del diseño, obteniéndose grandes reducciones en los niveles más altos [ChBr95]. El estudio que se realiza en este proyecto se encuentra dentro de la Síntesis de Alto Nivel (SAN).

La potencia total disipada en un circuito CMOS consta básicamente de dos componentes: una componente estática causada por las corrientes de fuga (*leakage*), y otra dinámica debida principalmente a las capacidades parásitas.

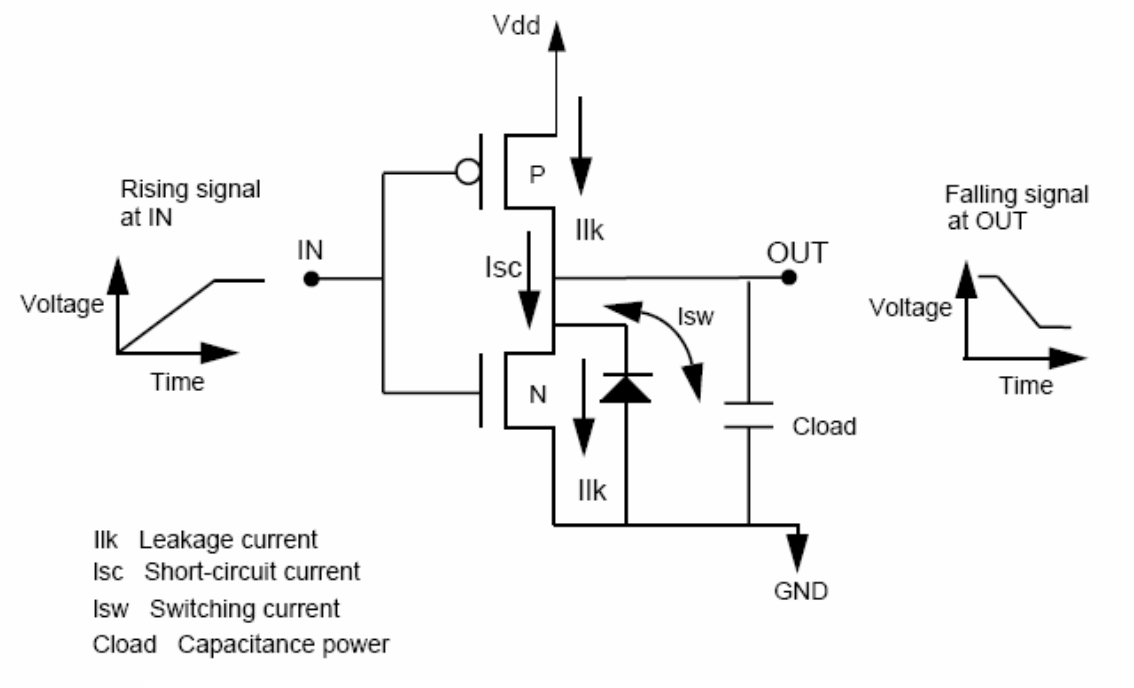


Figura 2.2. Corrientes dentro de una celda

A su vez el consumo dinámico puede subdividirse en (véase figura 2.2):

a) Consumo interno (Cell Internal Power), causado fundamentalmente por el cortocircuito momentáneo que se produce entre los transistores p y n, mientras que están pasando de corte a saturación, o viceversa, ya que ambos funcionan en su región lineal durante un breve periodo de tiempo. Este consumo como máximo supone el 30% del consumo dinámico total (véase figura 2.3).

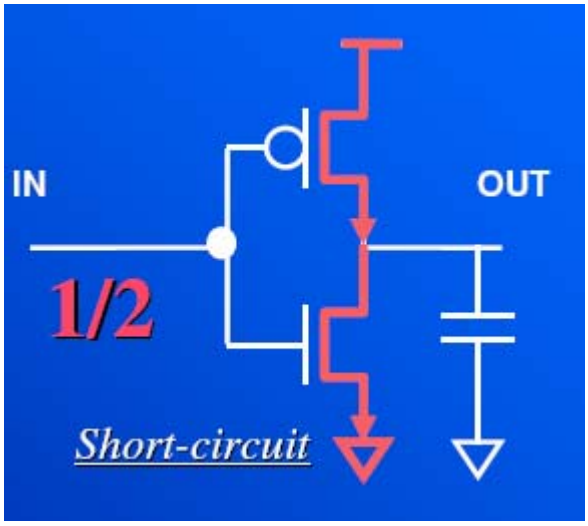


Figura 2.3. Corriente de cortocircuito

b) Consumo debido a las conmutaciones de los valores de entrada en las puertas (Net Switching Power), causado por la carga y descarga de la capacidad de salida de la celda. Se distinguen dos tipos de transiciones: de 0 a 1 y de 1 a 0 pero sólo las de 0 a 1 producen consumo de este tipo, mientras que las de 1 a 0 son las culpables del consumo interno (véase figura 2.4).

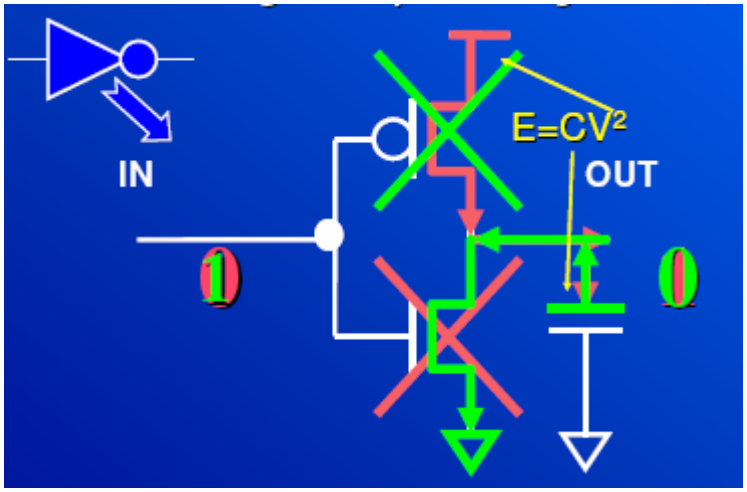


Figura 2.4. Carga y descarga de la capacidad.
Transición de 0->1 en un inversor

Este consumo supone entre un 70% y un 90% del consumo dinámico, y puede modelarse mediante la siguiente fórmula:

$P_{ns} = C*V^2*f_t/2$

donde C es la capacidad, V el voltaje y f_t la frecuencia de conmutación de los valores. Teniendo en cuenta que C y V dependen de factores tecnológicos fundamentalmente, nos centraremos únicamente en disminuir f_t.

Tal y como vemos en la figura 2.5, el consumo dinámico es el predominante actualmente.

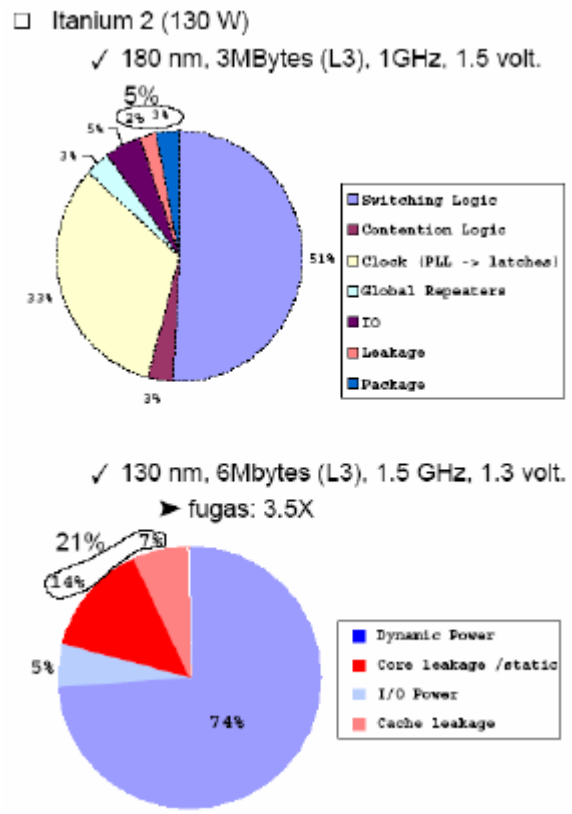


Figura 2.5. Reparto del consumo de potencia en el Itanium 2

2.1. Síntesis de alto nivel con ligaduras de consumo de energía

Existen muchas técnicas orientadas a disminuir el consumo de potencia dinámica. El uso de múltiples voltajes [YaDu05], [ShCh00], [MuRS05] permite funcionar a los módulos del camino crítico a mayores voltajes, mientras que el resto de módulos trabajan con voltajes pequeños. De esta manera, el consumo dinámico puede reducirse al mismo tiempo que se satisfacen las restricciones de tiempo. Por otro lado están las técnicas de reloj dinámico [MoRa05], [OcAK07], que usan diferentes tiempos de ciclo en instantes distintos, o lo que es lo mismo, relojes de distinta frecuencia. Sin embargo, este tipo de técnicas imponen restricciones en el rendimiento e implementación del circuito. Por contra, la disminución del número de conmutaciones no impone ninguna limitación en los parámetros del circuito.

En la SAN la mayoría de las investigaciones se han centrado en reducir

el consumo dinámico disminuyendo el número de transiciones en las unidades funcionales (UFs), registros, multiplexores y buses [ErKP99], [LCBK01], [ZhHC02], [BeBM00], [MuCo99], [KiCh97], [ShCh97], [GSJM97], [ChJC00]. Un método muy efectivo para reducir estas transiciones es incrementar la correlación de los datos de entrada, que puede realizarse modificando la planificación de operaciones [LCBK01], [ZhHC02], la asignación [MuCo99], [ShCh97], la segmentación de bucles [KiCh97], el intercambio de bucles, el reordenamiento de operandos [ShCh97], y compartición de los mismos. El algoritmo de asignación presentado en [ChJC00] divide algunas UFs en 2 nuevas UFs, y usa sólo una de ellas cuando las anchuras de los operandos de entrada así lo permiten. Esta estrategia de diseño, la ejecución de una operación sobre un conjunto de UFs encadenadas, se ha usado también recientemente en algunos algoritmos de SAN para minimizar el área de la ruta de datos incrementando la reutilización de las UFs [MRMH06], [MoMH03].

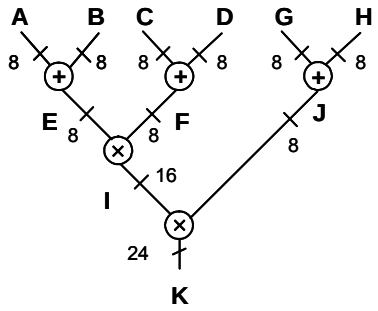
Por otro lado, existe el problema de dirigir el flujo de diseño mientras comprobamos el impacto de las decisiones tomadas sobre la switching activity. En la SAN fundamentalmente se han usado dos técnicas para este propósito. La primera de ellas se basa en el uso de simuladores que ejecutan un modelo del diseño para tomar los datos de la switching activity [GrKu98]. La información proporcionada por los simuladores será la que guíe el proceso de síntesis. La segunda técnica usa información de floorplanning para estimar el consumo [SHSS03], [KSCS01], [ZhJh02], para lo cual se necesita información a nivel RT de cada recurso. Sin embargo, ambas técnicas se utilizan en los algoritmos actuales sólo a nivel de operación por lo que, ambos métodos nos llevan a soluciones subóptimas si las comparamos con las obtenidas tras medir la switching activity a nivel de bit.

En este trabajo se proponen dos algoritmos de SAN que trabajan a nivel de bit y realizan la planificación y asignación de especificaciones con el objeto de obtener circuitos de bajo consumo.

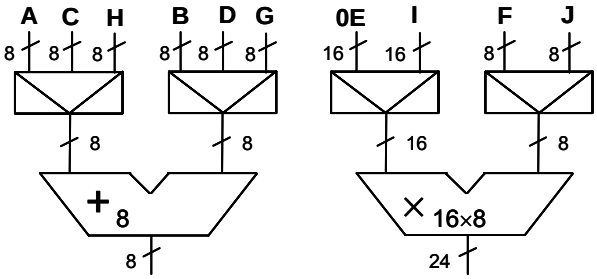
3. Motivación

Para explicar mejor el problema que supone el consumo dinámico, veremos un sencillo ejemplo basado en el GFD de la figura 3.1.a). Debido a que la mayor parte del consumo dinámico se debe al *switching power*, es decir, a los cambios en los bits de los operandos de entrada, cuando hablemos de consumo dinámico nos referiremos a estos cambios.

a)



b)



c)

	# Transiciones 0→1/1→0	
A+B → C+D	3/3	3/3
A+B → G+H	4/1	5/2
C+D → A+B	3/3	3/3
C+D → G+H	6/3	4/1
0E×F → I×J	5/3	6/4
E0×F → I×J	5/3	7/5

d)

	# Transiciones 0→1/1→0
A+B → C+D	1/1
A+B → G+H	3/0
C+D → A+B	1/1
C+D → G+H	3/0

e)

	# Transiciones 0→1/1→0	
E×F→ I(15..8)×J	1/3	3/5
E×F→ I(7..0)×J	2/3	3/4

f)

	Número de conmutaciones en el bit i-ésimo 0→1/1→0															
	7		6		5		4		3		2		1		0	
A+B→C+D	0/0	0/0	0/0	0/0	1/1	0/0	0/0	1/1	0/1	0/1	1/1	0/0	1/0	1/0	0/0	1/1
A+B→G+H	0/0	0/0	0/0	0/0	1/0	1/0	0/0	1/1	1/0	1/0	0/0	1/1	1/0	1/0	1/1	0/0
C+D→A+B	0/0	0/0	0/0	0/0	1/1	0/0	0/0	1/1	1/0	1/0	1/1	0/0	0/1	0/1	0/0	1/1
C+D→G+H	0/0	0/0	0/0	0/0	1/0	1/0	0/0	1/1	2/0	2/0	1/1	0/0	1/1	0/0	1/1	0/0

g)

	Sumador (8 bits)	**Multiplicador 1 (8×8)**	**Multiplicador 2 (8×8)**
Ciclo 1	A[7..6]B[5]A[4..3]B[2]A[1..0] + B[7..6]A[5]B[4..3]A[2]B[1..0]		
Ciclo 2	C+D	E×F	
Ciclo 3	G[7..3]H[2..0] + H[7..3]G[2..0]	I[15..8] ×J	I[7..0] ×J

Figura 23.a) GFD de una descripción conductual, b) implementación convencional de bajo consumo, c) switching activity, d) número mínimo de transiciones usando la propiedad conmutativa a nivel de bit, e) número de conmutaciones después de deshabilitar un multiplicador, f) switching activity a nivel de bit, g) implementación más eficiente en términos de consumo de potencia

Después de realizar varias simulaciones de los valores de entrada, proporcionando datos representativos, obtenemos la información de la *switching activity*, que podemos ver resumida en la figura 3.1.c), donde se muestra el número de transiciones de 0 a 1 y de 1 a 0 sucedidas en una UF durante la ejecución de dos operaciones en ciclos consecutivos.

Además de las transiciones de 0 a 1, que tienen una influencia directa en la *switching activity*, se han medido las transiciones de 1 a 0, debido a su efecto en el consumo interno (*cell internal power*).

La primera columna de la tabla muestra los pares de operaciones ejecutados en ciclos consecutivos sobre la misma UF. La segunda columna indica el número de transiciones y la tercera el número de conmutaciones que habría si aplicáramos la propiedad conmutativa en la primera operación. Nótese que la aplicación de la propiedad conmutativa a la segunda operación produce el mismo número de transiciones.

Por ejemplo, la ejecución en el mismo sumador de A+B y G+H produce 4 transiciones de 0 a 1 (0->1) y una de 1 a 0 (1->0), y cinco transiciones 0->1 y dos 1->0 si aplicamos la propiedad conmutativa para calcular B+A en lugar de la operación original.

Los algoritmos convencionales de bajo consumo utilizan la información de la *switching activity* para minimizar el número de conmutaciones al realizar la planificación y la asignación. Si implementamos el circuito con un sumador y un multiplicador, el mínimo número de conmutaciones se produce cuando las operaciones se ejecutan según se muestra en la figura 3.1.b). Nótese que la ejecución de H+G en lugar de G+H reduce el número de transiciones. Sin embargo, la multiplicación ExF es asignada a un multiplicador de tamaño 16x8, por lo que el operando E es extendido con varios ceros por la izquierda. El número total de conmutaciones es en total 12 0->1 y 7 1->0.

Una implementación más eficiente, en términos del consumo de potencia, puede lograrse si aplicamos la propiedad conmutativa a nivel de bit. La figura 3.1.f) muestra el número de conmutaciones producidas en cada bit de una UF que ejecuta las sumas del ejemplo. Por ejemplo, la ejecución en el mismo sumador de A+B seguida de C+D no produce ningún cambio en los dos bits más significativos. Sin embargo, el bit 5 produce una 1->0 en el primer operando y una 0->1 en el segundo, mientras que si aplicamos la propiedad

conmutativa en ese bit, desaparecen las transiciones. Igualmente sucede con el bit 2. De esta manera, aplicando la propiedad conmutativa tan sólo a algunos bits de los operandos de entrada podríamos reducir el número de transiciones. En la figura 3.1.f) tenemos dos columnas para cada bit, la primera de ellas indica el número de transiciones 0->1 y 1->0 sin aplicar la propiedad conmutativa en ese bit, y la segunda muestra dichas transiciones aplicando la propiedad conmutativa. Las casillas sombreadas indican la asignación elegida para cada bit (con propiedad conmutativa o sin ella), que se corresponde con la que menos transiciones provoca.

En el ejemplo, si aplicamos la propiedad conmutativa a nivel de bit, la ejecución de A+B seguida de C+D se realiza de la siguiente manera: A[7]A[6]B[5]A[4]A[3]B[2]A[1]A[0] + B[7]B[6]A[5]B[4]B[3]A[2]B[1]B[0], con lo que se reduce en 2/3 del número de conmutaciones. La figura 3.1.d) muestra la switching activity mínima producida por la ejecución consecutiva de dos sumas en la misma UF, debida a la aplicación de la propiedad conmutativa a nivel de bit.

En cuanto a las multiplicaciones, la asignación de ExF (8x8) a un multiplicador de 16x8, requiere la extensión de uno de los operandos de entrada, lo que produce conmutaciones innecesarias en la UF. Para evitar estas transiciones no deseadas (suponiendo que latcheamos los operandos de entrada del multiplicador que queda libre), el multiplicador 16x8 puede descomponerse en dos multiplicadores de 8x8, y ExF puede ejecutarse sólo en uno de ellos. De esta manera se reduce la *switching activity*. Además, IxJ tiene que fragmentarse y distribuirse entre los dos multiplicadores.

La figura 3.1.e) muestra el número de conmutaciones producidas si sustituimos el multiplicador 16x8 por dos 8x8, y además uno de ellos es deshabilitado cuando en el otro se ejecuta ExF

La figura 3.1.g) muestra la nueva implementación, con las transiciones asociadas en las figuras 3.1.d) y 3.1.e). El número de 0->1 se reduce alrededor de un 64% en esta implementación, gracias al estudio de la *switching activity* a nivel de bit, lo que nos permite aplicar la propiedad conmutativa adecuadamente en las sumas, y también a la inhabilitación parcial de UFs usadas para realizar operaciones más pequeñas.

4. Representación de los datos

Los dos algoritmos de síntesis propuestos toman como entrada una especificación y un conjunto de estadísticas de los valores de los datos de entrada, obtenidas de la simulación de cada operación de la especificación conductual.

El resultado de ambos algoritmos es una implementación RT de la conducta especificada en la que se ha pretendido minimizar el consumo de energía.

En ambos casos, los algoritmos propuestos realizan en primer lugar una caracterización de los datos a partir de las estadísticas dadas, que guiará las posteriores decisiones de planificación y selección y asignación de recursos. A continuación se presentan las dos aproximaciones propuestas para caracterizar los datos de entrada de las operaciones de la especificación.

4.1. Casos más comunes

El *caso más común* (*cmc*) es el conjunto de bits que tiene la probabilidad más alta de ser entrada de una operación. Por tanto, el cmc de un bit será ‘1’ si la probabilidad de que aparezca un 1 es mayor ó igual que 0.5 y ‘0’ en caso contrario.

Por ejemplo, supongamos que tenemos una operación de 4 bits y que los datos de entrada a lo largo del tiempo son los indicados por la tabla 4.1.

Tabla 4.1. Datos de entrada

Primer operando	Segundo operando
0110	1111
0101	1110
0101	1100
1001	1011
0111	1010

Tabla 4.2. Probabilidades y bits más comunes

Bit	1er op	Prob	2º op	Prob
3	0	4/5	1	1
2	1	4/5	1	3/5
1	0	3/5	1	3/5
0	1	4/5	0	3/5

Tal y como vemos en la tabla 4.2, el caso más común sería “0101” para el primer operando y “1110” para el segundo.

Ésta será la representación de datos que utilizaremos en el primer algoritmo para guiar las fases de planificación y selección y asignación de recursos HW con el objeto de reducir el consumo de energía de los circuitos.

4.2. Patrones más comunes

La idea de patrón más común surge para superar una limitación de los casos más comunes, recogiendo no solo los valores más probables de los datos, sino también sus variaciones a lo largo del tiempo.

Si nos fijamos en el bit 2 del segundo operando lo veremos claramente. En los tres primeros instantes de tiempo su valor es ‘1’, mientras que en los dos siguientes es ‘0’. Sería interesante poder capturar este comportamiento de alguna manera, dando así lugar a los patrones más comunes.

Denotaremos con un carácter un patrón de bits en el tiempo, y la concatenación de estos caracteres constituirá los patrones más comunes. Para ello, dividimos el tiempo en *slots*. En el ejemplo, dividiremos los ciclos de ejecución en dos mitades: la primera del 1 al 3 y la segunda del 4 al 5.

Dentro de cada slot calcularemos el caso más común. La concatenación de casos más comunes se corresponderá con el carácter que represente su patrón.

En el ejemplo supongamos que tenemos 4 caracteres para representar 4 patrones o comportamientos a lo largo del tiempo. (Véase tabla 4.3)

Tabla 4.3. Definición de patrones con el tiempo dividido en dos *slots* o ranuras de tiempo

	1ªmitad	2ªmitad
A	0	0
B	0	1
C	1	0
D	1	1

Tabla 4.4. Probabilidades y patrones más comunes con los casos más comunes por slots

	Primer operando					Segundo operando				
Bit	1er slot	Prob	2º slot	Prob	cmc	1er slot	Prob	2º slot	Prob	cmc
3	0	1	1	1/2	**B**	1	1	1	1	**D**
2	1	1	1	1/2	**D**	1	1	0	1	**D**
1	0	2/3	1	1/2	**B**	1	2/3	1	1	**D**
0	1	2/3	1	1	**D**	0	2/3	1	1/2	**B**

Tal y como vemos en la tabla 4.4, el patrón más común sería “BDBD” para el primer operando y “DDDB” para el segundo.

En el segundo algoritmo seguiremos este esquema.

4.3. Alternativa a las simulaciones del circuito

Una alternativa a las simulaciones, que reduce el tiempo empleado en las mismas, podría ser la estimación de los valores de los operandos de entrada de cada operación de la especificación, a partir de los valores de los puertos de entrada del circuito. A continuación se explica cómo pueden estimarse los valores de los datos para las representaciones presentadas anteriormente.

Supongamos que tenemos el GFD de la figura 4.1.

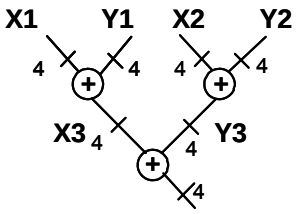


Figura 4.1. Grafo de flujo de datos de tres sumas de 4 bits con dependencias de datos

Los datos de entrada X1, Y1, X2 e Y2 evidentemente son conocidos, pero el problema se presenta con X3 e Y3, que pueden conocerse mediante valores estadísticos o bien calculando el resultado de X1+Y1 y X2+Y2.

Con este ejemplo vemos que la forma de calcular estos valores es sencilla, pero pensemos en una multiplicación. ¿Cómo evaluar el número de cambios que se producirá en una multiplicación? Además de los que hay entre los operandos de entrada y el resultado de la misma, una estimación más precisa sería considerar los cambios que se producen en la matriz intermedia de multiplicaciones. El problema es que las sumas que internamente realiza el multiplicador no pueden ser obtenidas por profiling y deben ser obtenidas a raíz de los casos ó patrones más comunes de entrada. En este caso transformamos la multiplicación en las sumas equivalentes (véase la figura 4.2) y calculamos los cambios entre las distintas sumas.

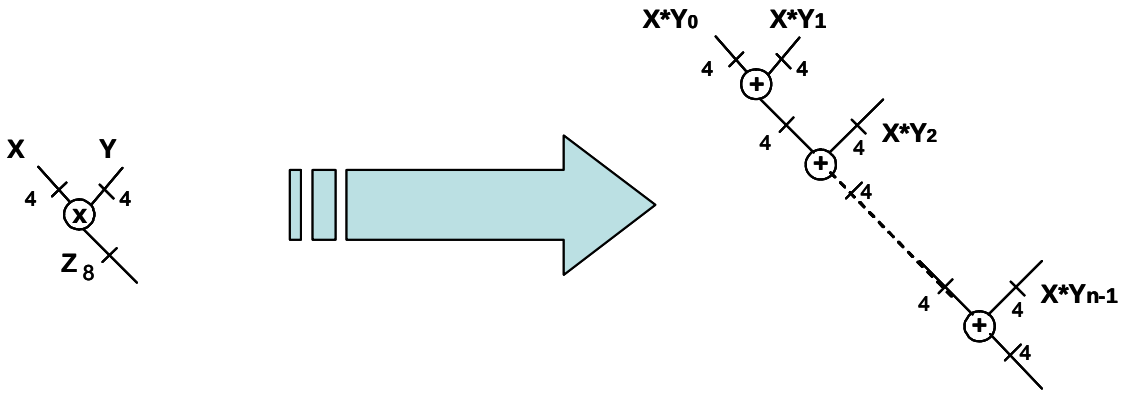


Figura 4.2. Transformación de una multiplicación en sumas

Nótese que en esta transformación el acarreo de entrada de una suma siempre es el acarreo producido por un bit resultado de la multiplicación.

Ejemplo:

Z2 = X2*Y0 + X1*Y1 + X0*Y2 + carry(Z1)

Z1 = X1*Y0 + X0*Y1

Es decir, Z2 necesita el carry producido por el cálculo de Z1

Y por otro lado, el bit más significativo del operando izquierdo de las sumas siempre será el acarreo de salida de la suma anterior. Veámoslo mejor con la figura 4.3.

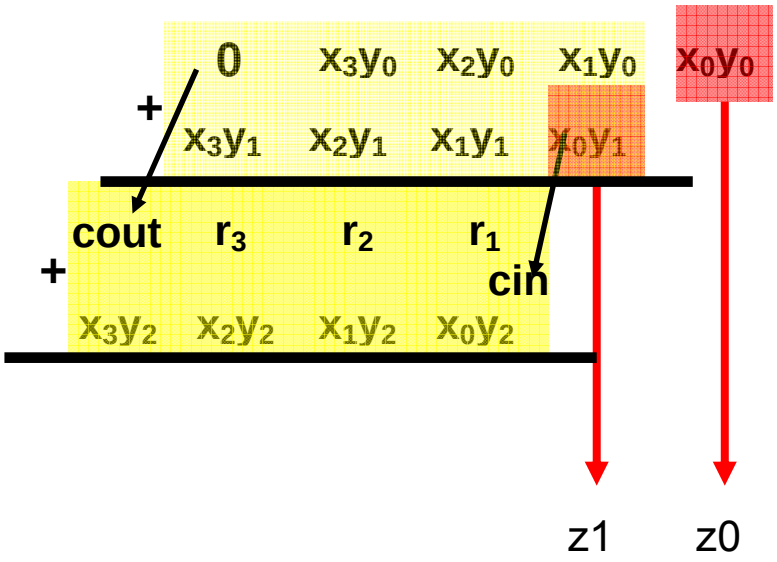


Figura 4.3. Transformación de multiplicación a sumas a nivel de bit

4.3.1. Casos más comunes

Con esta representación es bastante trivial. Supongamos que los casos más comunes de X e Y son “1010” y “1110” respectivamente. Por tanto, en la primera suma los operandos serían “0000” y “1010”, en la segunda “0101” y “1010” etc. (véase la tabla 4.5)

Tabla 4.5. Operandos izquierdo y derecho de las sumas correspondientes a la transformación de la multiplicación 1010 x 1110

1er op	2º op	Cin	Resultado	Cout
0000	1010	0	1010	0
0101	1010	0	1111	0
0111	1010	0	0001	1

```
      1 0 1 0
    x 1 1 1 0
  ------------
      0 0 0 0
    1 0 1 0
  1 0 1 0
1 0 1 0
-----------------
1 0 0 0 1 1 0 0
```

4.3.2. Patrones más comunes

Con los patrones más comunes este proceso no es tan trivial e, incluso, el primer ejemplo con las 3 sumas que presentaban dependencias puede resultar algo complicado.

Si intentamos repetir el proceso de la transformación de multiplicaciones a sumas, la pregunta es, ¿cuánto vale A+A? Es decir, lo que tenemos que hacer es definir un álgebra para nuestros símbolos.

Sin embargo no es tan sencillo. En el caso de definir un álgebra teniendo en cuenta las sumas, debemos considerar también los acarreos. Por ello, el mejor modo de hacerlo, tanto en este ejemplo como en otros, es definir el álgebra para un conjunto universal de puertas y después aplicar las mismas ecuaciones booleanas que tenemos para los valores lógicos “0” y “1”.

En nuestro caso utilizaremos el conjunto universal {AND, OR, NOT}. Para definir cada uno de los valores, hemos de tener en cuenta el patrón que representa cada carácter, y aplicar bit a bit las operaciones lógicas del conjunto universal. Es decir, (NOT A) = D, porque A=”00” y D=”11” , B AND C = “00” porque B = “01” y C = “10”, A OR C = “10”, etc. (ver álgebra para patrones A, B, C, D en el apéndice).

De este modo, el proceso es análogo al caso de los casos más comunes, pues nos basta con tener en cuenta las ecuaciones de la suma (véase figura 4.4):

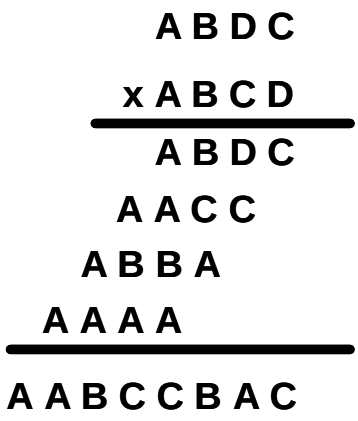
$$z_i = x_i \oplus y_i$$
$$c_{i+1} = x_i c_i + y_i c_i + x_i y_i$$

Figura 4.4. Ecuaciones de la celda básica de un sumador

Por tanto, si suponemos que los patrones más comunes de X e Y son “ABDC” y “ABCD”, respectivamente, obtendremos los resultados que muestra la tabla 4.6:

Tabla 4.6. Operandos izquierdo y derecho de las sumas correspondientes a la transformación de la multiplicación de patrones más comunes ABCD x ABDC

1er op	2º op	Cin	Resultado	Cout
AABD	AACC	A	ADDA	A
AADD	ABBA	C	BCAB	A
ABCA	AAAA	A	ABCC	A



5. Técnicas para reducir el consumo de los circuitos

En este capítulo veremos un conjunto de técnicas que pueden contribuir a la minimización del consumo dinámico, la mayoría de las cuales están sustentadas en ideas sencillas de Aritmética.

5.1. Propiedad conmutativa

En el caso de operaciones de los tipos suma y multiplicación, que son las más habituales en las especificaciones utilizadas en la SAN, es posible utilizar la propiedad conmutativa para minimizar el número de cambios de bits de un ciclo al siguiente.

Supongamos que en el ciclo actual las entradas son “0110” y “0101”, y que en el ciclo siguiente son “0101” y “0110”. Si pudiéramos permutar el orden de los operandos, es evidente que no tendríamos ningún cambio de bit.

Esta idea es extensible al concepto de caso y patrón más común. Sin embargo, la propiedad conmutativa tal cual presenta una limitación: el tamaño de los operandos. Para vencer esta limitación aplicaremos la propiedad conmutativa de forma parcial.

5.1.1 Propiedad conmutativa parcial

La propiedad conmutativa parcial consiste en aplicar la propiedad conmutativa a sólo algunos fragmentos de los operandos. La única restricción de esta propiedad, es que sólo lo podemos aplicar a sumas.

Por ejemplo, supongamos ahora que los operandos de entrada en el ciclo actual son “000110” y “110101” y en el ciclo siguiente “000101” y “110110”.

Tanto si aplicamos la propiedad conmutativa, como si no aplicamos ninguna propiedad, tendremos 4 cambios de bits. Sin embargo, si nos fijamos bien veremos que los cuatro bits más significativos de ambos operandos en ambos ciclos coinciden, mientras que los dos bits menos significativos son

iguales pero situados en entradas distintas en cada uno de los ciclos. Por tanto, lo ideal sería aplicar la propiedad conmutativa sólo a los dos bits menos significativos.

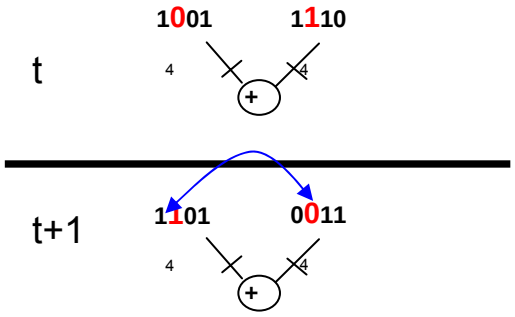
En la siguiente tabla podemos ver cómo quedarían las entradas en los 3 casos:

Tabla 5.1. Número de cambios sin aplicar ninguna propiedad, aplicando la propiedad conmutativa (PC), y aplicando la propiedad conmutativa parcial (PCP)

	1er op	1er op sig	2º op	2º op sig	Cambios
	000110	000101	110101	110110	4
PC	000110	110110	110101	000101	4
PCP	000110	000110	110101	110101	**0**

Esta propiedad llevada al extremo puede entenderse como propiedad conmutativa a nivel de bit, es decir, lo que permutamos no son fragmentos propiamente dichos, sino bits (o fragmentos de tamaño 1).

Por ejemplo, supongamos que en el ciclo actual tenemos por entradas “1001” y “1110” y en el siguiente “1101” y “0011”. En estas condiciones tenemos cuatro cambios, pero si intercambiamos el bit 2 del primer operando en el ciclo actual con el del segundo operando en el ciclo siguiente (y viceversa) bajaremos a 2 cambios únicamente (véase la figura 5.1).



$a_i(t+1) \neq b_i(t+1) \wedge$
$a_i(t) \neq a_i(t+1) \wedge$
$b_i(t) \neq b_i(t+1)$

Condición necesaria y suficiente

Figura 5.1. Flujo de datos. En rojo los bits involucrados en la propiedad conmutativa parcial

Si llamamos *a* al primer operando y *b* al segundo, la condición de intercambio consiste en que el i-ésimo bit en el instante siguiente (*t+1*) sea distinto para ambos operandos y que en ambos operandos se produzca un

cambio del instante actual (*t*) al siguiente (*t+1*).

Esta transformación no conlleva un coste adicional, pues simplemente consiste en cambiar las entradas de los multiplexores. Para ello, hay que ver los multiplexores vectoriales como un array de multiplexores de un bit (véase la figura 5.2).

Sean A y B los operandos del instante actual y C y D los operandos del instante siguiente del ejemplo:

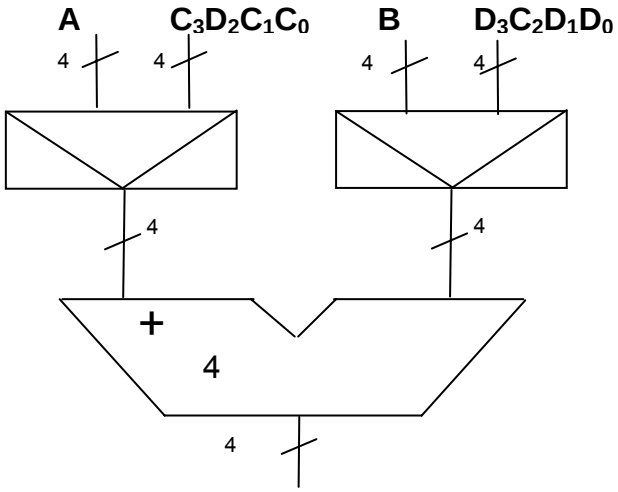


Figura 5.2. Implementación de sumas con propiedad conmutativa parcial

5.2. Fragmentación de operaciones

La fragmentación de operaciones consiste en la descomposición de una operación, en varias operaciones que pueden ejecutarse en la misma o distintas unidades funcionales.

En el caso de las sumas lo único que hay que hacer es propagar el acarreo. (Véase la figura 5.3)

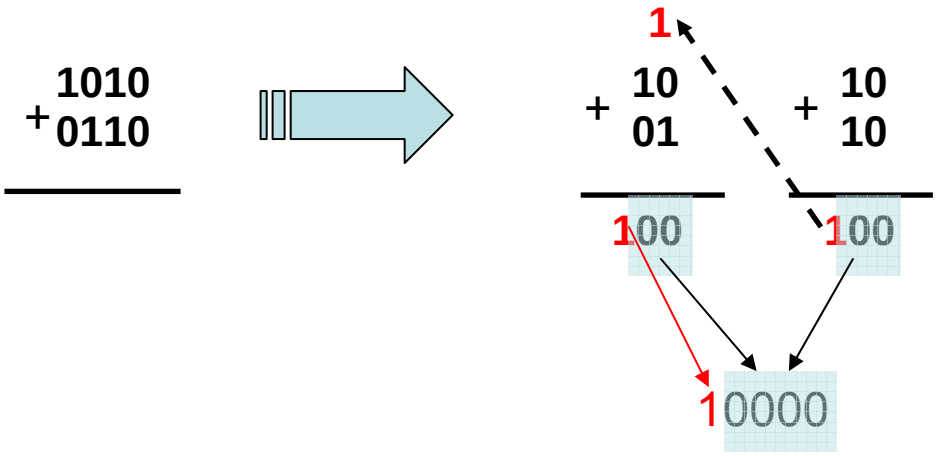


Figura 5.3. Transformación para fragmentación de sumadores

Tal y como podemos ver en la figura 5.3, propagamos el acarreo y obtenemos un resultado que es concatenación de los resultados anteriores y cuyo acarreo es el acarreo de la operación con fragmentos más significativos. Nótese que la fragmentación en el caso de las sumas se basa en el hecho de que los operandos tienen una notación aditiva, es decir "1010 + 0110" es $[(2^3 + 2^1) + (2^2 + 2^1)]$, que es lo mismo que sumar $[(2^3 + 2^2) + (2^1 + 2^1)]$.

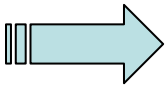
En el caso de las multiplicaciones, el proceso es más complejo. Hay que realizar sumas entre algunos fragmentos que antes pertenecían a las sumas internas del multiplicador.

En la figura 5.4 tenemos la forma de realizar la fragmentación. En el caso peor obtendríamos 5 fragmentos, y en el mejor tan solo 2. (Véase la figura 5.5).

Esta transformación también está basada en una propiedad matemática:

U = **a+b**

V = **c+d**



U*V = (a+b)*(c+d) = a x c + a x d + b x c + b x d

Por lo que podemos ver la fragmentación o bien a nivel de bit o bien a nivel de subproductos.

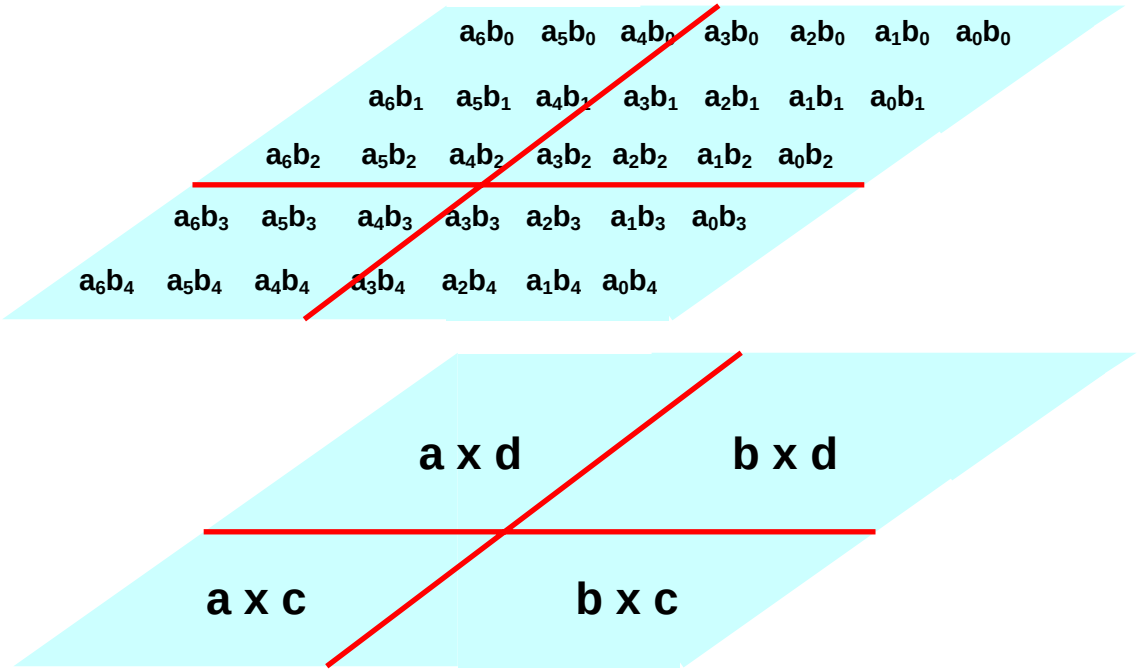
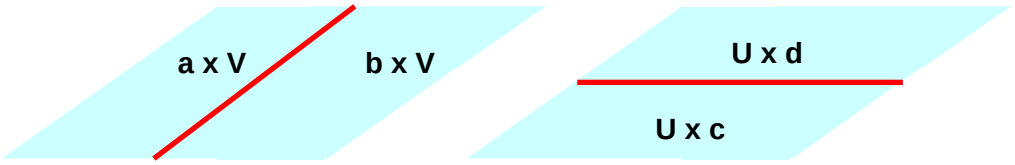


Figura 5.4. Transformación para la fragmentación de multiplicaciones



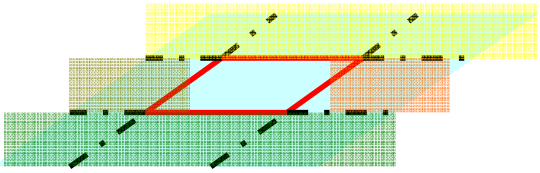


Figura 5.5. Casos mejores (2 fragmentos) y caso peor (5 fragmentos) en una multiplicación

Es decir, si *U*=”1000”, *V*=”1101”, entonces, por ejemplo, *a*=”10”, *b*=”00”, *c*=”11”, *d*=”01”.

En el caso peor, por supuesto que la fórmula matemática sigue siendo válida. Lo único que hay que tener en cuenta es que la forma de elegir *a, b, c, d* es distinta.

5.3. Inhabilitación de los operadores ociosos

Esta técnica consiste en deshabilitar los operadores durante los ciclos en los cuales no realizan un cálculo efectivo, es decir ahorramos la potencia inútilmente gastada. Esto se logra utilizando latches para estabilizar los operandos de entrada en las UFs y ciclos en que no se calculan operaciones de la especificación.

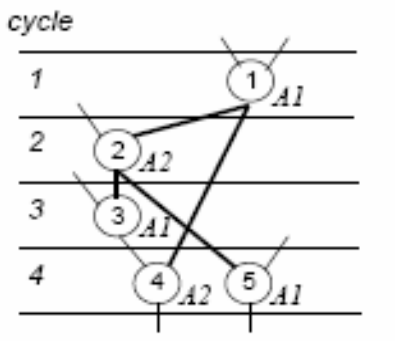


Figura 5.6. GFD con asignación de operaciones en 2 sumadores: A1 y A2

Tal y como podemos ver en el ejemplo de la figura 5.6, durante el ciclo 2 el sumador A1 está realizando un cálculo que ya ha realizado previamente, así como el sumador A2 calcula inútilmente en los ciclos 1 y 3.

5.4. La distancia de Hamming

Para minimizar el número de cambios entre bits de un ciclo al siguiente, vamos a basar nuestros algoritmos en la distancia de Hamming (*dh*),

$d(x,x)=0$

$d(x,y)=1 \qquad x \neq y$, siendo *x* e *y* un bit.

$dh(X,Y)=\sum_{i=0}^{n-1} d(Xi,Yi)$, siendo *X* e *Y* cadenas de bits y *Xi*, *Yi* los bits i-ésimos de la cadena.

de tal forma que siempre que podamos escoger entre dos posibles asignaciones, elegiremos la que produzca una menor *dh*.

Por tanto, hemos de definir la distancia de Hamming (*DH*) para una asignación de operandos a un operador.

Si # = $\{+,*\}$, entonces definimos *DH* como

$$DH(x1\#y1, x2\#y2) = dh(x1, x2) + dh(y1, y2)$$

Por ejemplo,

DH(“101+001”,”111+000”) = *dh*(“101”,”111”) + *dh*(“001”,”000”) = [*d*(1,1) + *d*(0,1) + *d*(1,1)] + [*d*(0,0) + *d*(0,0) + *d*(1,0)] = [0 + 1 + 0] + [0 + 0 + 1] = 1 + 1 = 2

Esta es la definición natural de distancia de Hamming, y funciona perfectamente si utilizamos los casos más comunes como modelo de representación de datos, ya que los *cmc* no son más que una cadena de bits.

En el caso de los patrones más comunes tenemos que redefinir la distancia de Hamming entre dos símbolos básicos (*d*), en este caso los patrones. Sin embargo la relación entre *d*, *dh* y *DH* sigue siendo la misma.

Veámoslo con un ejemplo. Supongamos que tenemos la definición de distancia de Hamming dada en el apéndice B (para cuatro patrones), entonces:

DH(“ABCD+AABB”,”CCDD+DACD”) = *dh*(“ABCD”,”CCDD”) + *dh*(“AABB”,”DACD”) = [*d*(A,C) + *d*(B,C) + *d*(C,D) + *d*(D,D)] + [*d*(A,D) + *d*(A,A) + *d*(B,C) + *d*(B,D)] = [1 + 2 + 1 + 0] + [2 + 0 + 2 + 1] = 4 + 5 = 9

Por último, para comprobar la importancia de la distancia de Hamming en la asignación de operandos, veamos la figura 5.7, en la cual se relacionan las distancias de Hamming de los operandos y el consumo de potencia.

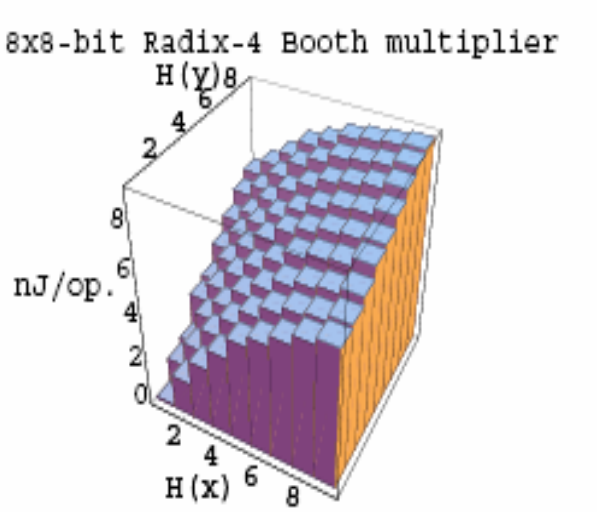


Figura 5.7. Consumo en función de la distancia de Hamming (*H*) entre operandos en un multiplicador según el algoritmo de Booth

Tal y como observamos en la figura, a mayor distancia de Hamming mayor es el consumo.

6. Algoritmo de SAN basado en la fragmentación de operaciones dirigida por el consumo estático

En este capítulo propondremos un algoritmo [BaMoMe06] para disminuir el consumo en la síntesis de circuitos con sumadores. La idea básica consistirá en disminuir el área para minimizar el consumo estático y utilizar la técnica de fragmentación, prediciendo los acarreos entre los fragmentos generados. De esta forma, los fragmentos dejan de tener dependencias LDE entre ellos y pueden considerarse independientes por lo que, podemos reordenarlos de muchas más formas y, a priori, podremos llegar a mejores soluciones basándonos en la heurística de la distancia de Hamming.

Nótese que en este primer algoritmo trabajaremos con los casos más comunes (*cmcs*).

6.1. Algoritmo propuesto

El algoritmo se encarga de la planificación y asignación de operandos. Consta de seis fases. En la primera fase calcula los cmcs, que serán usados únicamente en las etapas 5 y 6 para optimizar los resultados obtenidos en las fases 2, 3 y 4. Es decir, las etapas 2, 3 y 4 son independientes de los datos de entrada, mientras que la 5 y la 6 se encargarán de disminuir el consumo dinámico dependiendo de los cmcs.

Para facilitar el entendimiento de cada una de las fases, utilizaremos un sencillo ejemplo.

6.1.1. Cálculo de los casos más comunes.

En esta fase calculamos los casos más comunes a partir de las estadísticas de entrada que, nos indican para cada bit si es más probable que sea un ‘0’ ó un ‘1’.

Este cálculo se realiza de forma similar que en el apartado 3.1., por lo

que en nuestro ejemplo supondremos que los cmcs ya vienen dados.

Sean A, B, C, D, E, F y G el conjunto de sumas correspondientes al caso más común de los operandos de entrada, obtenidos de la forma descrita anteriormente. Las anchuras se especifican entre paréntesis, y la latencia del circuito se fija a 4 ciclos (λ=4). (Véase la figura 6.1)

(A,3) = 101 + 010
(B,3) = 000 + 111
(C,4) = 1010 + 1110
(D,4) = 0011 + 0101
(E,12) = 111000110001 + 010101001110
(F,12) = 101010010101 + 111000111000
(G,12) = 001100110011 + 011001100110

Figura 6.1. Ejemplo de casos más comunes. El tamaño de las operaciones viene indicado entre paréntesis

6.1.2. Cálculo de la anchura media

Para minimizar el área de los sumadores, el algoritmo calcula en esta primera fase la media aritmética de las anchuras de las sumas. Está demostrado que el área mínima se alcanza equilibrando el número de bits sumados en cada ciclo, por lo que nuestro propósito será sumar un número de bits por ciclo tan próximo a la media como sea posible. Al disminuir el área reducimos por un lado el consumo estático de energía, ya que disminuimos el máximo número de bits sumados por ciclo. Y en la mayoría de los casos también se reduce el consumo dinámico, al producirse menos cambios de bits en los recursos HW. Además, reduciendo las anchuras de los sumadores es más probable disminuir el tiempo de ciclo, por lo que las operaciones se ejecutan más rápido y el circuito está activo menos tiempo.

En el ejemplo, obtendríamos el siguiente resultado:

Número de bits sumados: 3+3+4+4+12+12+12 = 50

Media aritmética: $M = 50/4 = 12.5 \rightarrow M = 13$

Nótese que usamos la media aritmética (otras medias también son admisibles), y redondeamos el resultado al entero mayor más cercano.

6.1.3. Reparto del número de bits sumados por ciclo

En esta fase, el algoritmo intenta equilibrar el número de bits sumados por ciclo, planificando en cada ciclo un conjunto de operaciones aditivas cuya suma de anchuras sea lo más próxima a la media. Esto lo realizamos en tres etapas.

1) Ordenación de las sumas, de la mayor a la de menor anchura. En el ejemplo:

(E,12), (F,12), (G,12), (C,4), (D,4), (A,3), (B,3)

2) Cálculo de la *anchura de ciclo* (número de bits sumados por ciclo), sin exceder la media. En el ejemplo:

Ciclo 1 → (E,12)
Ciclo 2 → (F,12)
Ciclo 3 → (G,12)
Ciclo 4 → (C,4),(D,4),(A,3)

3) Planificación. Si todavía quedan sumas sin planificar, el algoritmo las asigna al ciclo tal que la nueva *anchura de ciclo* sea mínima. En el ejemplo:

Ciclo 1 → (E,12)
(12+3 = 15)
Ciclo 2 → (F,12)
(12+3 = 15)
Ciclo 3 → (G,12)
(12+3 = 15)
Ciclo 4 → (C,4), (D,4), (A,3), (B,3)
(11+3 = 14 es menor que 15)

6.1.4. Asignación de sumadores

En esta etapa se decide el número y las anchuras de los sumadores del circuito. Usamos la técnica de fragmentación, para minimizar la anchura de los sumadores y el hardware desaprovechado. Esta fase se divide a su vez en varias etapas:

1) El algoritmo considera en primer lugar el ciclo con la mayor suma

de anchuras, y asigna un sumador por cada suma ejecutada en ese ciclo (nótese que la anchura de los sumadores coincide con la de las operaciones). En caso de que haya varios ciclos con la máxima suma de anchuras, el algoritmo escoge el ciclo con el mayor número de sumas. En nuestro ejemplo:

Ciclo 4 → (C,4), (D,4), (A,3), (B,3)

Sumadores = {+4, +4, +3, +3}

2) En esta etapa cada suma no asignada, se asigna a un conjunto de sumadores disponibles cuya suma de anchuras sea igual a la anchura de la operación. En algunos casos esta asignación resulta imposible y se soluciona sustituyendo uno o más sumadores por un conjunto de sumadores más pequeños cuya suma de anchuras coincida con la de los sumadores sustituidos.

El algoritmo procesa los ciclos con sumas no asignadas desde el primero al último, aunque cualquier orden obtendría resultados similares. Cuando es necesario actualizar la tabla de sumadores (un sumador es sustituido por un conjunto de sumadores más pequeños), las operaciones ya asignadas son automáticamente asignadas al nuevo conjunto de unidades funcionales (*UFs*). En el ejemplo:

Asignación de operaciones planificadas en el ciclo 1:

Ciclo 1 = (E,12) → (E1,4), (E2,4), (E3,3), (E4,1)

La suma E se fragmenta en las sumas E1, E2, E3 y E4.

El sumador +3 se sustituye por un sumador +2 y un sumador +1.

Sumadores = {+4, +4, +3, +2, +1}

Ciclo 4 actualizado → (C,4), (D,4), (A,3), (B1,2), (B2,1)

Asignación de operaciones planificadas en el ciclo 2:

Ciclo 2 = (F,12) → (F1,4), (F2,4), (F3,3), (F4,1)

La suma F se fragmenta en las sumas F1, F2, F3, F4.

No es necesario actualizar.

Sumadores = {+4, +4, +3, +2, +1}

Asignación de las operaciones planificadas en el ciclo 3:

Ciclo 3 = (G,12) → (G1,4), (G2,4), (G3,3), (G4,1)

La suma G se fragmenta en las sumas G1, G2, G3, G4.

No es necesario actualizar.

Sumadores = {+4, +4, +3, +2, +1}

Una optimización en esta fase consiste en desacoplar la fragmentación de la reasignación de operaciones de tal forma que, después de realizar una nueva fragmentación no haya que reasignar los ciclos previos. Es decir, simplemente actualizamos la tabla de sumadores y cuando tengamos el resultado final de la fragmentación procesamos todos los ciclos de nuevo para asignar las operaciones. De esta manera las operaciones se asignan una única vez.

En el ejemplo obtendríamos primero el resultado de la fragmentación

Sumadores = {+4, +4, +3, +2, +1}

y después realizaríamos la asignación de operaciones sobre esos sumadores, obteniéndose el mismo resultado pero más rápido.

6.1.5 Precálculo de los acarreos de salida de los nuevos fragmentos

Esta fase es crítica de cara a la siguiente, de tal manera que aumenta el número de posibles ordenaciones de sumas dentro de cada sumador, lo que potencialmente puede llevar a reducir el número de cambios de bit en cada UF. El principal objetivo de esta fase es eliminar las dependencias que aparecen cuando partimos una suma en otras más pequeñas (es decir, el fragmento menos significativo no puede ejecutarse después de otro fragmento que pertenezca a la misma suma original). Al eliminar las dependencias, los nuevos fragmentos pueden considerarse como operaciones independientes entre sí. En esta versión del algoritmo solo consideramos los casos más comunes calculados en la fase 1. Por tanto, es necesario validar cada decisión tomada en esta fase antes de continuar con el proceso de síntesis. La eliminación de las dependencias de datos entre fragmentos de operación se realiza mediante simulaciones de un gran número de posibles valores de entrada. Nótese que estas simulaciones sólo son necesarias para el conjunto de operaciones fragmentadas, y no para todas las operaciones de la especificación. Además, dichas simulaciones se detienen si el acarreo de

salida calculado en la simulación es distinto del computado por el algoritmo.

En nuestro ejemplo, el algoritmo calcula los siguientes valores de los acarreos:

Sea *S* una suma, y S_i el *i-ésimo* fragmento. S_i es menos significativo que S_{i+1}. Y sea CS_i el acarreo de entrada de la suma S_i. Nótese que para calcular CS_{i+1} es necesario conocer CS_i, y que CS_{i+1} = *Cout* procedente de S_i.

$CE_1 = 0$, $CE_2 = 0$, $CE_3 = 0$, $CE_4 = 1$

$CF_1 = 0$, $CF_2 = 0$, $CF_3 = 0$, $CF_4 = 1$

$CG_1 = 0$, $CG_2 = 0$, $CG_3 = 0$, $CG_4 = 1$

$CB_1 = 0$, $CB_2 = 0$

6.1.6. Ordenación de las sumas y fragmentos

Esta fase del algoritmo pretende reducir el consumo dinámico del circuito. Para cada sumador que sobrevive a la última etapa, el algoritmo aplica la heurística de la distancia de Hamming (*DH*) para planificar las sumas minimizando los cambios de bits de un modo voraz.

La ordenación de las sumas y fragmentos tiene lugar durante esta última fase. La etapa previa nos permite reducir el conjunto de dependencias de datos entre operaciones. Por tanto, esta fase consigue mayor independencia del estilo descriptivo utilizado en la especificación original, siendo capaz de planificar algunos de los fragmentos de operación fuera de orden (el cálculo de los bits más significativos del resultado puede realizarse antes del cómputo de los bits menos significativos). En concreto, esto puede realizarse sólo para las operaciones cuyas simulaciones hayan verificado las decisiones de diseño adoptadas en la etapa previa.

Esta fase comprende, para cada conjunto de operaciones asignadas a la misma UF (sin dependencias de datos), las siguientes etapas:

1) La primera suma es escogida aleatoriamente, ya que todas las sumas asignadas a un sumador tienen la misma anchura.

2) Para cada sumador, el algoritmo busca la operación no planificada con el menor valor de la heurística. Este paso se repite hasta que no quedan sumas sin planificar.

En el paso 2) aplicamos la *DH* a cada suma entre los operandos

procesados en el ciclo actual y los operandos planificados en el ciclo anterior. Cada suma tiene dos posibles ubicaciones en el sumador: la de la especificación y la resultante de aplicar la propiedad conmutativa. Debido a la fragmentación, la anchura de los sumadores coincide con las anchuras de los fragmentos que le han sido asignados. Por lo tanto, en esta etapa aplicaremos la *DH* a dos casos distintos por cada suma no planificada.

En el caso de dependencias de datos entre operaciones, el algoritmo procede de una forma similar, restringiendo el conjunto de ciclos considerados a aquellos incluidos entre la planificación ASAP (As Soon As Possible) y ALAP (As Late As Possible) de cada operación.

Con la *DH* obtenemos un mínimo local, es decir, no podemos garantizar alcanzar el mínimo absoluto. Este mínimo global sólo se podría obtener explorando todas las posibles ordenaciones de las sumas, con lo que el coste del algoritmo sería exponencial con respecto al número de operaciones.

Consideremos nuestro ejemplo. En primer lugar introduciremos una notación especial: el carácter ’ indica el uso de la propiedad conmutativa. Por ejemplo, para la suma original X → 100 + 010 asignada a un sumador de anchura 3, escribiríamos:

X → 100 + 010 y X’ → 010 + 100

En esta fase tenemos que considerar para cada sumador el conjunto de sumas asignadas, en lugar de los ciclos, que es lo que se hizo en las fases anteriores. Para los valores de los acarreos de salida calculados en la fase previa (que suponemos han sido validados mediante simulaciones) el proceso de síntesis de nuestro ejemplo termina de la siguiente manera:

+4_1 = {C, E1, F1, G1}
+4_2 = {D, E2, F2, G2}
+3 = {A, E3, F3, G3}
+2 = {B1}
+1 = {B2, E4, F4, G4}

En la etapa 1) elegimos la primera suma mientras que el paso 2) será aplicado tres veces para obtener la siguiente planificación:

6.1.7. Conclusiones del ejemplo.

Tabla 6.1. Planificación y asignación de sumadores

Ciclo	+4_1	+4_2	+3	+2	+1
1	C	D	A	B1	B2
2	E1	E2	E3'		E4'
3	G1	G2	G3		F4
4	F1	F2	F3		G4

La tabla 6.1 muestra el resultado final de la SAN para la especificación conductual de nuestro ejemplo. Nótese que las nuevas técnicas de bajo consumo implementadas en este algoritmo han ampliado el espacio de diseño a explorar, traspasando las limitaciones impuestas por la descripción conductual. En concreto, el circuito sintetizado presenta un conjunto de características que no podrían haberse logrado con cualquier otra técnica de bajo consumo conocida:

1) Las operaciones G y F se ejecutan en varios ciclos. En concreto, la mayor parte de los bits de G son calculados en el ciclo 3, y sólo un bit del resultado se calcula en el ciclo 4. Algo similar ocurre con la operación F en los ciclos 3 y 4.

2) Las operaciones B, E, F y G se ejecutan distribuidas en varias *UF*s.

3) La propiedad conmutativa se ha aplicado a los bits más significativos de la operación E, pero no al resto. La aplicación parcial de dicha propiedad ha contribuido a disminuir el número de cambios en los sumadores del circuito.

Estas tres características tienen un impacto directo en la reducción de la energía consumida por el circuito sintetizado para nuestro ejemplo:

1) El número de bits sumados en cada ciclo es menor en nuestra implementación que en la convencional (14 vs 15), lo que implica directamente un área menor. Pero también implica una reducción del 7% en el consumo estático de las UFs. De hecho, esta reducción además afecta a

los consumos estáticos debidos a la interconexión y almacenamiento de datos. Por tanto, podemos concluir que el consumo estático global se reduce alrededor del 7%.

2) La anchura máxima de los sumadores es bastante menor en nuestra implementación que en la convencional (4 vs 12), lo que provoca una considerable reducción del tiempo de ciclo. El tiempo que el circuito estará encendido será 3 veces menor con nuestra implementación, lo que implica una reducción adicional del consumo estático de alrededor del 67%.

3) El consumo dinámico también es menor debido a las características anteriores, pero se reduce incluso más gracias al reordenamiento de operaciones logrado por la eliminación de dependencias de datos entre operaciones y fragmentos de operación, y a la aplicación parcial de la propiedad conmutativa. El número de cambios de bits se ha reducido alrededor de un 55% con nuestra implementación, lo que corresponde a una reducción del mismo orden en el consumo dinámico.

6.2. Resultados experimentales

Para medir la calidad del algoritmo propuesto, hemos comparado nuestras implementaciones con aquéllas realizadas por los algoritmos convencionales de SAN para bajo consumo implementados en las herramientas comerciales *Synopsys Behavioral Compiler* y *Design Compiler*.

Las especificaciones sintetizadas han sido generadas aleatoriamente, para partir de un conjunto de experimentos cuya distribución de bits no beneficie su alineamiento. La anchura de las sumas oscila entre 3 y la anchura máxima especificada en la tabla 6.2. Después de generar estas especificaciones sintéticas, aplicamos nuestro algoritmo y creamos una descripción en VHDL del circuito sintetizado. A continuación, usamos *Modelsim* para generar los estímulos de ambos circuitos: el obtenido por nuestro algoritmo y el sintetizado por *Synopsys*. Y finalmente usamos *Synopsys Design Analyzer* y *Design Compiler* para estimar el consumo de potencia. La librería utilizada en ambos casos es *VTVTLIB25* de *Virginia Tech*. basada en una tecnología de 0,25µm *TSMC*.

Tabla 6.2. Resultados experimentales

Especificación			Consumo de potencia (uW)								
#Sumas	Anchura máxima	Latencia	Syn CI	Ntro CI	FCI	Syn NS	Ntro NS	FNS	Total Syn	Total Ntro	Total F
10	14	3	9,8177	5,1606	1,9024	17,1584	14,4329	1,1888	26,9761	19,5935	1,3768
20	18	5	54,1506	26,8212	2,0189	29,0063	16,3279	1,7765	83,1569	43,1491	1,9272
35	20	7	95,8885	37,6086	2,5496	39,3953	22,6038	1,7429	135,284	60,2124	2,2468
50	25	12	137,451	9,9303	13,8416	48,5197	13,0261	3,7248	185,971	22,9564	8,1011
70	30	16	203,975	22,8277	8,9354	60,2987	15,1763	3,9732	264,273	38,004	6,9538
85	35	18	231,783	20,4429	11,3381	79,6377	15,9549	4,9914	311,421	36,3978	8,556
100	40	20	285,98	89,6243	3,1909	96,0659	32,8949	2,9204	382,046	122,519	3,1183
52,8571	26	11,5714	145,578	30,3451	**6,2539**	52,8689	18,631	**2,9026**	198,447	48,9761	**4,6114**

La Tabla 6.2 muestra las principales características de algunos de los circuitos sintetizados (número de sumas, anchura máxima de los operandos, y latencia), las estimaciones del consumo de potencia de las implementaciones de ambos algoritmos, y el beneficio obtenido (columnas fracción). En la tabla, CI significa Cell Internal Power (consumo estático) y NS Net Switching (consumo dinámico). FX es la fracción obtenida al dividir Syn X entre Ntro X, donde X es CI o NS. En realidad, FX es el beneficio obtenido por nuestro algoritmo. La última celda de cada columna representa la media de todos los valores mostrados en la misma.

Con los resultados de la tabla 6.2, podemos concluir que el porcentaje de energía ahorrada por nuestro algoritmo alcanza el 88% y es en promedio el 65%.

La figura 6.2 compara el consumo total de los circuitos sintetizados por la herramienta comercial y por nuestro algoritmo. El consumo de los circuitos sintetizados por nuestro algoritmo es en promedio 4 veces menor que el de las implementaciones convencionales. Esta reducción crece en general con la latencia y con el número de operaciones, debido a que estas condiciones permiten al algoritmo obtener circuitos con un mayor equilibrio de la energía estática consumida por ciclo, e incrementan el número de posibles asignaciones de operaciones con cadenas de bits similares a la misma UF.

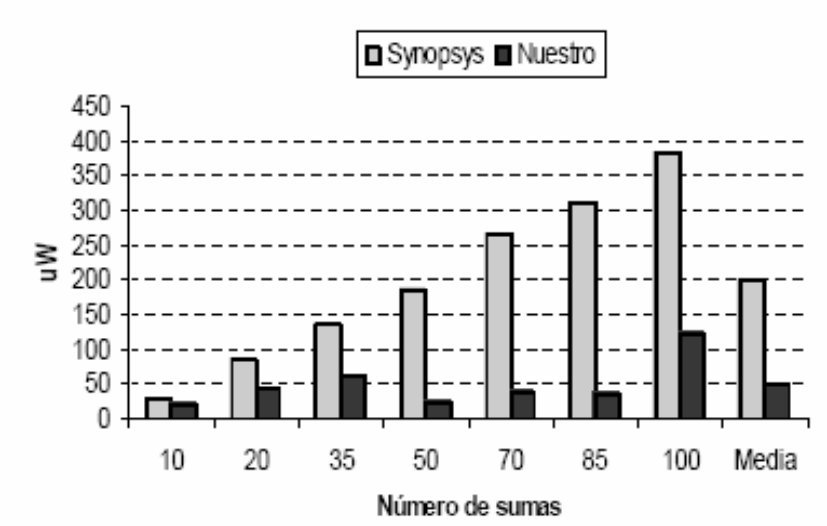


Figura 6.2.Consumo de potencia de los circuitos

Para ilustrar la dependencia de la cantidad de potencia consumida con la latencia del circuito, hemos sintetizado el ejemplo de las 50 sumas, descrito en la tabla 6.2, usando diferentes valores de latencia. La figura 6.3 muestra el porcentaje de potencia ahorrada por nuestro algoritmo. Dicho porcentaje oscila entre el 50% y el 90%, y crece con la latencia hasta que se alcanza un máximo (después de ello el porcentaje es menor debido al consumo estático). Los circuitos consumen energía desde dos puntos de vista: por un lado, cuanto mayor es la latencia menor es el consumo dinámico, pero por otro lado, cuanto mayor es la latencia mayor es el consumo debido a que los sumadores están más tiempo encendidos. Por tanto, el ahorro máximo se obtiene en un punto de equilibrio entre estos dos tipos de consumo.

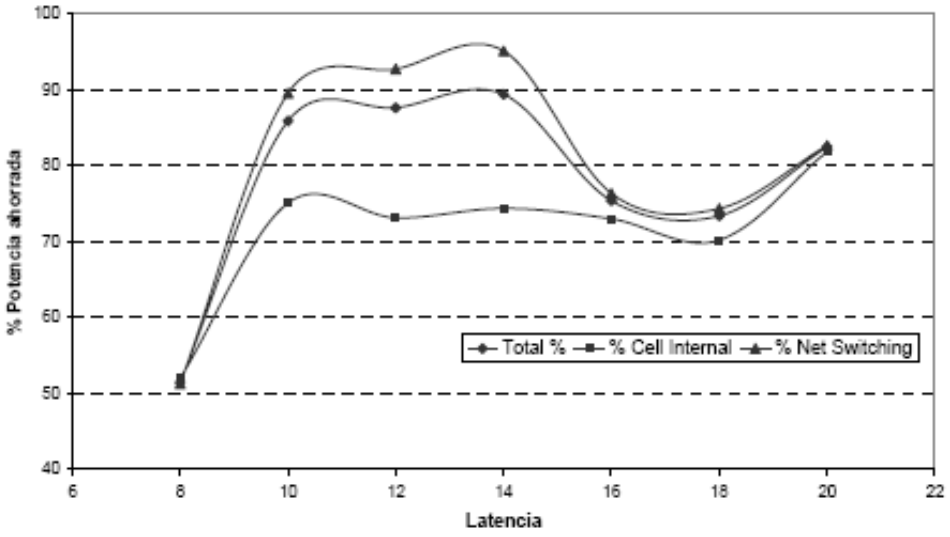


Figura 6.3. Crecimiento del ahorro de potencia con la latencia

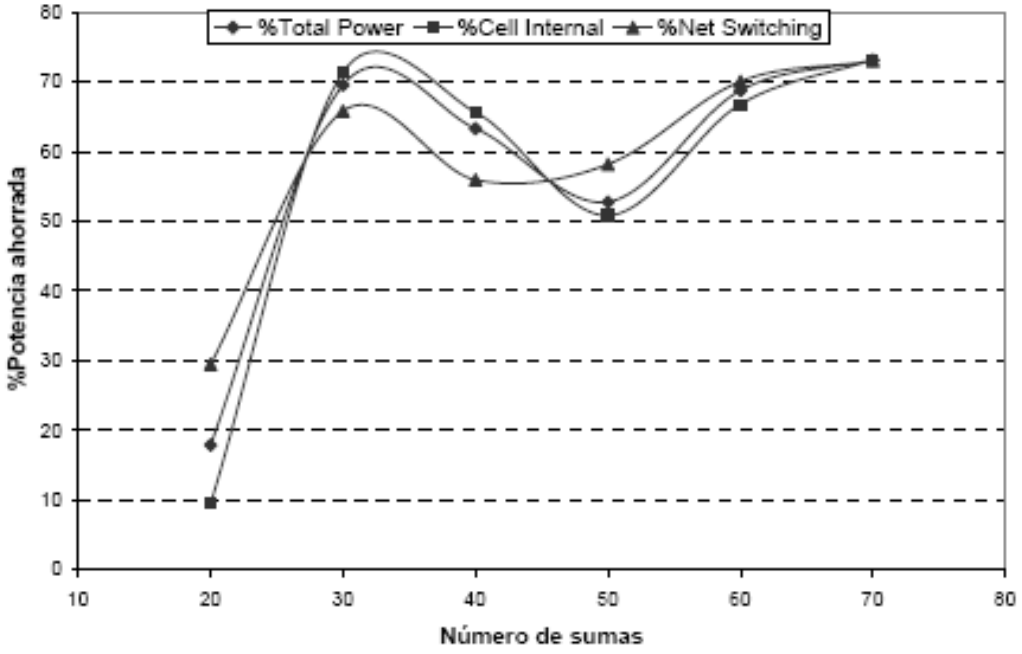


Figura 6.4. Crecimiento del ahorro de potencia con el número de operaciones

La figura 6.4 muestra la dependencia de la cantidad de potencia consumida con el número de sumas. En este experimento, hemos fijado la latencia del circuito, y las especificaciones contienen sumas con una anchura máxima de 16 bits (el tamaño del conjunto de sumas se especifica en el eje X).

En este caso, la cantidad de energía ahorrada oscila entre el 20% y el 80%.

6.3. Conclusiones

El algoritmo propuesto realiza conjuntamente la planificación y la asignación de operaciones con el objeto de reducir tanto el consumo estático como el dinámico. El consumo estático se minimiza reduciendo el área y el tiempo de ciclo de los circuitos sintetizados, lo que se consigue equilibrando el número de bits computados en cada ciclo y permitiendo la ejecución de una operación sobre varias *UF*s encadenadas. Como consecuencia de estas estrategias de diseño, el consumo dinámico también se reduce. Sin embargo, el algoritmo propuesto logra una reducción adicional del consumo dinámico, al eliminar parcialmente las dependencias de datos entre operaciones y fragmentos de operación. Esta técnica extiende considerablemente las posibilidades de asignar operaciones a *UF*s, lo que conlleva un incremento de las posibilidades de reducir el número de cambios de bits en las *UF*s de la ruta de datos.

Los resultados experimentales muestran que las estimaciones del consumo de los circuitos sintetizados mediante nuestro algoritmo son substancialmente más pequeñas que las logradas por los algoritmos convencionales de bajo consumo. En general, la reducción del consumo de potencia crece con la latencia y el número de operaciones de la especificación.

Sin embargo el algoritmo propuesto presenta varias limitaciones:

1) Sólo se tienen en cuenta operaciones aditivas.

2) Las últimas dos fases del algoritmo optimizan el resultado obtenido en las fases previas para reducir el consumo dinámico del circuito en el caso más común, calculado a partir de las estadísticas de los posibles valores de los operandos de entrada. La optimización realizada puede resultar muy particular, por lo que sería interesante ampliar el conjunto de casos más comunes, para optimizar de un modo más general los resultados producidos por las fases previas del algoritmo. Este trabajo supuestamente redundará en mayores ahorros del consumo de potencia dinámico de los circuitos.

3) El problema de la simulación de valores para obtener los acarreos de entrada a los fragmentos, lo que supone un pequeño cuello de botella al algoritmo. La idea es continuar con el concepto de conocer los acarreos de los fragmentos y permitir la predicción de los mismos (estática o dinámica).

7. Algoritmo de SAN basado en la fragmentación de operaciones guiada por patrones de comportamiento

En este capítulo proponemos un algoritmo para disminuir el consumo dinámico en rutas de datos compuestas principalmente por sumadores y multiplicadores.

Este algoritmo supera las limitaciones de los algoritmos de bajo consumo tradicionales y trata la información relativa a la actividad (*switching activity*) a nivel de bit.

Además, introduce una nueva forma de representar los datos basada en patrones, que permite capturar la información de los operandos de entrada a nivel de bit.

El algoritmo propuesto realiza la planificación y asignación de operaciones basándose en los patrones, permitiendo la ejecución de las mismas en varios ciclos y sobre varias UFs enlazadas. Además permite la aplicación parcial de las propiedades de las operaciones y la sustitución de UFs parcialmente desaprovechadas (ejecutan operaciones de menor anchura) por otras más pequeñas, que pueden ser deshabilitadas en algunos ciclos.

Todas estas características permiten reducir el número total de conmutaciones en el circuito que, a la postre serán las que determinen el consumo del mismo.

7.1. Algoritmo propuesto

El algoritmo de SAN propuesto realiza la planificación y la asignación de las operaciones de la especificación, minimizando el consumo dinámico.

Para lograr tal reducción, nos beneficiamos de la similitud de los datos, es decir, las cadenas de bits de los operandos de entrada que difieren poco entre sí, con lo que el número de conmutaciones se reduce.

Para ello utilizamos la técnica de la fragmentación guiada por patrones,

es decir, fragmentamos según los patrones más comunes que encontremos entre los operandos de entrada y no para minimizar el área, que es lo que hacíamos en el algoritmo anterior. En otras palabras, las cadenas de bits similares se asignarán a la misma UF, reduciendo así la *switching activity*.

El algoritmo propuesto toma como entrada la especificación conductual y la información de la *switching activity* obtenida mediante simulaciones del diseño. Estas estadísticas son traducidas a nuestro formato de patrones más comunes (*pmcs*), los cuales capturan la información de de las conmutaciones a lo largo del tiempo.

Después de obtener los *pmcs* realizamos primero la asignación y después la planificación de las operaciones, basándonos en dichos patrones. A lo largo del algoritmo aplicaremos la propiedad conmutativa parcial y permitiremos la deshabilitación parcial de operadores para para disminuir el número de conmutaciones (en realidad se descompone el operador en 2 operadores y se deshabilita el que no se usa).

Para facilitar el entendimiento del algoritmo, explicaremos cada una de las fases a continuación, acompañadas de un sencillo ejemplo.

7.1.1. Asignación de recursos

Una vez que disponemos de la información relativa a los patrones más comunes, identificamos los que más se repiten para realizar la fragmentación de operaciones. Esta fase a su vez queda dividida en 4 etapas:

A) Generación y ordenación de los pmcs.

En esta etapa procesamos la información de los pmcs para seleccionar los mejores, de acuerdo a una serie de heurísticas y parámetros de entrada. Como el número de patrones que podemos encontrar es muy grande, en cada iteración de esta etapa debemos filtrar la lista de patrones basándonos en una serie de parámetros que el algoritmo toma como entrada. Tales parámetros pueden ser: el tamaño mínimo y máximo de los patrones, el valor mínimo de *similitud* que deben alcanzar los patrones, la antigüedad mínima de un patrón para poder aplicarle el filtrado, etc

El objetivo de este filtrado es eliminar los patrones que tienen un bajo índice de repetición en las iteraciones previas, siempre que tengan la suficiente

antigüedad (número de ciclos que han sido tomados en cuenta por el algoritmo).

Para agrupar los patrones similares utilizamos la siguiente heurística de *similitud* S:

S = (tam_op1+tam_op2)/(DH(op1,op2)+1)

donde tam_op*i* es el tamaño del operando *i*, y DH es la distancia de Hamming entre op1 y op2. Nótese que el +1 es para evitar posibles casos de división por 0, ya que DH(x,y) >= 0.

Si el valor de S es mayor que un mínimo ó umbral, consideraremos que el patrón del operador y el nuevo patrón que evaluamos coinciden.

Después de procesar todos los patrones obtenidos a partir de las estadísticas, los ordenamos de acuerdo a otra heurística, R.

R = repeticiones^2*tam_op1*tam_op2/antigüedad

Es decir, tratamos de beneficiar a los fragmentos más repetidos (para disminuir la actividad de conmutación) y a los más grandes (para disminuir la lógica de interconexión). En esta ordenación no se tendrán en cuenta los fragmentos que no lleven cierto tiempo siendo procesados en el algoritmo.

Para ilustrar esta y las sucesivas fases del algoritmo utilizaremos el siguiente ejemplo:

X1→ AABB + DDCC
X2→ AABBAA + DDCCAB
X3→ AAAA + DCCC
X4→ ABBB + CDDD
X5→ ABBB + DCDD
X6→ AABBAABB + DDCCDDCC

Sean X1, …, X6 6 sumas sin dependencias entre ellas. Después de procesarlas y aplicar el filtrado de patrones llegamos a que el único superviviente es el patrón “AABB + DDCC”, el cual aparece camuflado en las sumas X1, X2 y X6 y presenta cierta similitud con X3, X4 y X5.

B) Fragmentación guiada por patrones

Las operaciones que presenten fragmentos con cierta similitud al patrón operador (en el ejemplo “AABB + DDCC”) se fragmentan, siempre siguiendo el orden de la lista ordenada de patrones operador. Es decir, primero intentamos traducir con respecto al primer patrón de la lista, después con el segundo, y así

sucesivamente.

Sin embargo, podrían quedar operaciones (o nuevos fragmentos) que no se parezcan mucho a los patrones seleccionados, por lo que no serán asignados a ningún operador, si no que crearemos un operador nuevo, de tal forma que se minimicen los cambios de bits.

En el ejemplo tenemos que fragmentar las operaciones X2 y X6 en 2 nuevos fragmentos cada una:

X6 → X6[7..4] + X6[3..0] = "AABB + DDCC" + "AABB + DDCC"
X2 → X2[5..2] + X2[1..0] = "AABB + DDCC" + "AA + AB"

En el caso de X6 hemos obtenido dos nuevos fragmentos que se ajustan perfectamente al patrón operador (de hecho ambos patrones coinciden) que teníamos. X2[5..2] también se ajusta a dicho patrón, pero no así X2[1..0], por lo que creamos un nuevo patrón operador para X2[1..0]

C) Selección del mejor par operación-operador.

Definimos un par operando-operador como el conjunto formado por un patrón operador y un patrón de una operación. Nótese que en el ejemplo tenemos 2 patrones operador, a saber: "AABB + DDCC" y "AA + AB".

En esta etapa el algoritmo intenta ajustar lo mejor posible cada 'bit' (cuyo comportamiento viene representado por el carácter asociado) de cada patrón de operación con cada 'bit' de un patrón operador. Es decir, elegimos la asignación que produzca menor DH con respecto a la anterior. De este modo, una operación ya asignada después de fragmentar, puede ser reasignada de su patrón huésped a otro patrón operador al que se ajuste mejor, siempre que sea del mismo tamaño y del mismo tipo, o bien puede permanecer en el mismo patrón huésped pero modificando su asignación.

En esta etapa aplicamos la propiedad conmutativa parcial a nivel de bit, para crear el par que produzca una DH mínima.

En nuestro ejemplo no se produce ningún cambio de asignación, por lo que para ver cómo actuaría esta etapa consideraremos dos casos.

En el primer caso supongamos que "AB+CD" asignado al patrón "DB+BD". Si cambiamos la A por la C, la DH con el patrón operador se reduce.

$$HD(\text{“AB+CD”},\text{”DB+BD”}) = 2 + 0 + 2 + 0 = 4$$
$$HD(\text{“CB+AD”},\text{”DB+BD”}) = 1 + 0 + 1 + 0 = 2$$

En el segundo caso, supongamos que tras la fragmentación tenemos un patrón operador "AA+BB" y otro "AB+BB". Si el procesamiento de un fragmento "AB+BB" se hace antes de crear el patrón "AB+BB", quedará asignado al patrón operador "AA+BB", mientras que la DH con respecto al patrón "AB+BB" es menor. Y más aún, si suponemos que el patrón del fragmento es "BB+AB", tras aplicar la propiedad conmutativa parcial para el patrón "AA+BB" tendríamos que su DH es 1 mientras que si la aplicamos para el patrón "AB+BB" la DH sería 0.

Caso 2:

Patrones operador: "AA+BB", "AB+BB"

Patrón operación: "AB+BB"

DH("AB+BB","AA+BB") = 0 + 1 + 0 + 0 = 1

DH("AB+BB","AA+BB") = 0 + 0 + 0 + 0 = 0

Caso 3:

Patrones operador: "AA+BB", "AB+BB"

Patrón operación: "BB+AB"

Mejor asignación con "AA+BB" ➔ "AB+BB"

DH("AB+BB","AA+BB") = 0 + 1 + 0 + 0 = 1

Mejor asignación con "AB+BB" ➔ "AB+BB"

DH("AB+BB","AA+BB") = 0 + 0 + 0 + 0 = 0

D) Replicación y unificación de operadores

En esta etapa el algoritmo trata de mejorar el rendimiento del circuito. La ejecución de las anteriores etapas nos podría llevar a tener los operadores sobrecargados, es decir, uno ó varios operadores podrían tener muchas operaciones asignadas, por lo que la latencia del circuito sería muy grande. Para solucionar este problema, el algoritmo replica los operadores que tienen una gran carga de trabajo. Repetimos esta técnica de replicación hasta que todos los operadores tienen un número de operaciones asignadas muy parecido.

Sin embargo, la replicación tiene una contrapartida: la creación de nuevos operadores incrementa el número de recursos y, consecuentemente, el

consumo estático y el área. Por ello, para disminuir los efectos de la replicación el algoritmo trata de unificar los operadores que tienen una baja carga de operaciones. En el caso de que estos operadores sean de distinto tamaño, las operaciones del operador más pequeño serán completadas con un “0” para poder guardar el acarreo de salida (una A con el modelo de pmcs), y el resto será igual a la asignación anterior.

De esta forma, la replicación reduce la latencia y las conmutaciones, y la unificación reduce el área, ya que disminuye el número de UFs utilizadas. Nótese que estas dos técnicas hay que aplicarlas en distinta proporción según el objetivo. Puesto que pretendemos reducir el consumo de potencia, la técnica más utilizada debe ser la replicación.

En nuestro ejemplo, tenemos el patrón operador “AABB + DDCC” sobrecargado con siete operaciones. La técnica de replicación nos permite repartir estas siete operaciones en cuatro operadores con el mismo patrón (“AABB + DDCC”) por lo que, la carga de operaciones se equilibra, tal y como vemos en la figura 7.1.

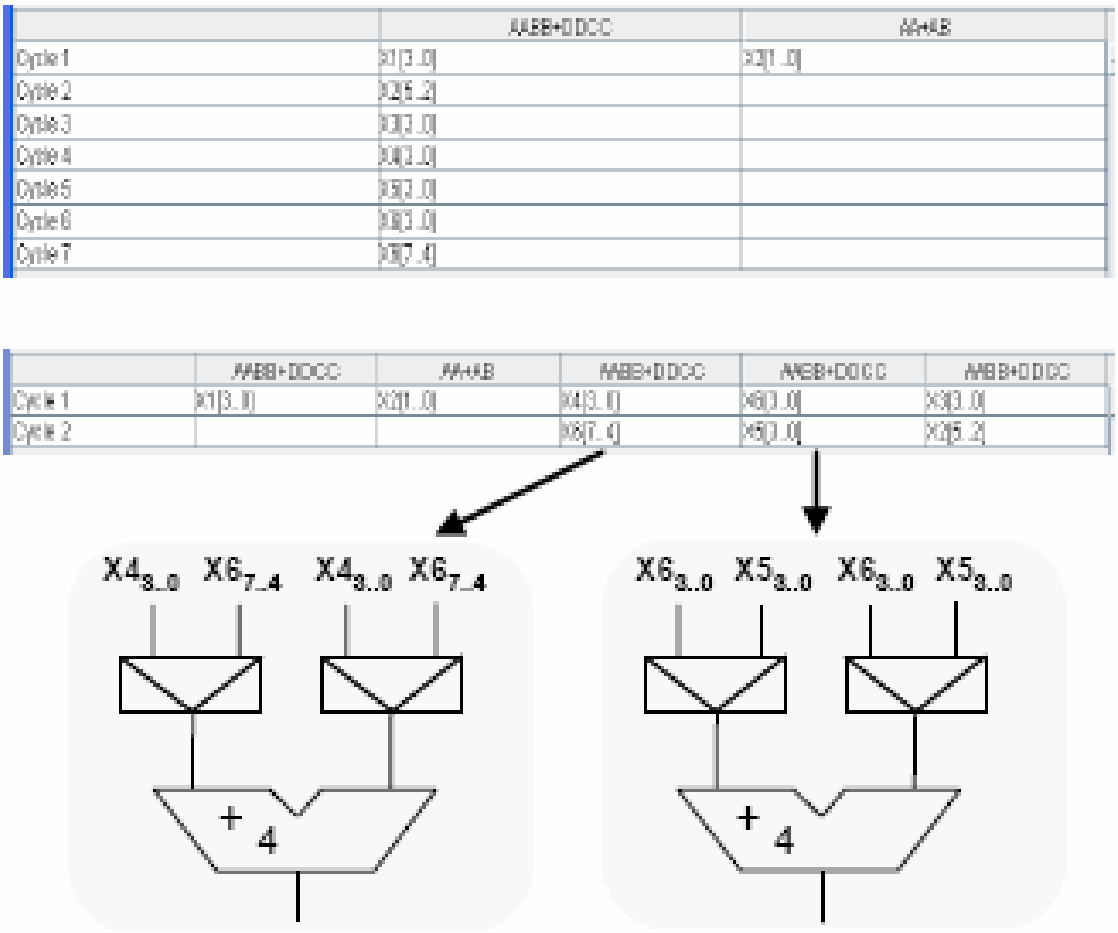


Figura 7.1. Asignación y planificación sin y con replicación, e implementación de la especificación que permite replicación

7.1.2. Planificación

En esta fase del algoritmo ya tenemos asignadas todas las operaciones y fragmentos de operación. Nótese que la fragmentación de operaciones de la fase anterior produce una serie de dependencias entre los fragmentos, además de las que pudiera haber de la especificación original.

Para planificar las operaciones asignadas a cada UF, se ha implementado una variante de bajo consumo del algoritmo ASAP. Su objetivo es minimizar el número de conmutaciones de un ciclo al siguiente en cada UF, al mismo tiempo que cumple las restricciones del algoritmo ASAP tradicional, es decir, una operación se planifica en un ciclo si todas sus predecesoras ya han sido planificadas previamente. La novedad consiste en que si varias operaciones de una misma UF cumplen las restricciones, elegimos aquélla que produzca un menor número de conmutaciones en los patrones o, en otras palabras, la que tenga una menor DH con respecto a la anterior operación asignada.

7.1.3 Conclusiones del ejemplo

Para disminuir el número de conmutaciones, el algoritmo podría dividir algunas operaciones de la especificación en varias más pequeñas, que son asignadas a distintas UFs en distintos ciclos. De esta manera, la operación puede ser ejecutada a lo largo de distintos ciclos no consecutivos y sobre diferentes recursos hardware. Por tanto, la técnica propuesta extiende el espacio de diseño, permitiendo circuitos más eficientes en términos de consumo dinámico.

En el ejemplo, X6 se ejecuta en los ciclos 1 y 2 y sobre 2 sumadores diferentes. En particular, los bits menos significativos se calculan en el mismo sumador que X5 y los más significativos en el mismo sumador que X4, tal y como se puede ver en la figura 21. Estas características contribuyen significativamente a minimizar la *switching activity*, y ninguna de ellas se da en las implementaciones convencionales.

Después de planificar las operaciones de nuestro ejemplo, la implementación resultante tiene las siguientes características:

1) La *switching activity* se reduce de un 0.35 a 0.1 (más del 70%).

2) El tiempo de ejecución es de 46800 ps versus 51060 ps, lo que implica una reducción del 8%.
3) El área se ha incrementado ligeramente de 25442 μm^2 a 28773 (alrededor del 13%).
4) En términos de energía, el consumo se reduce hasta un 70%, comparado con la implementación realizada por un algoritmo convencional de bajo consumo, para una misma latencia fijada de antemano.

7.1.3. Inhabilitación parcial de las UFs en estado ocioso

Los recursos que calculan operaciones más pequeñas que el operador al que están asignadas (resultado de la técnica de unificación), son sustituidos por un conjunto de varias UFs más pequeñas y de área similar.

Esta nueva fragmentación se hace para deshabilitar algunas de las nuevas UFs en los ciclos durante los cuales la UF original era más grande que la operación. De esta forma, se evitan las conmutaciones innecesarias debidas a la extensión de los operandos de entrada.

Cada sumador que ejecuta una suma más pequeña en algún ciclo, se sustituye por dos sumadores más pequeños cuya suma de anchuras es la misma que el tamaño de la UF original. Por tanto, la anchura de los nuevos sumadores coincide con los tamaños de las operaciones asignadas a la UF original.

Nótese que este cambio en la implementación no requiere ninguna modificación en las asignaciones. Sólo implica la fragmentación de algunas de las operaciones asignadas a la UF sustituida. Por tanto, los nuevos fragmentos son ejecutados en UFs distintas. Por otro lado, además de realizar la inhabilitación de los sumadores que no ejecutan sumas en algunos ciclos, sus operandos de entrada son latcheados para guardar los valores previos.

Los multiplicadores que realizan multiplicaciones más pequeñas, en algún ciclo, también son sustituidos por UFs más pequeñas para deshabilitar parte del operador original. El número de nuevos multiplicadores depende de la alineación de las operaciones más pequeñas dentro del multiplicador y tal y como hemos visto en el capítulo 4 necesitaríamos 2 multiplicadores para el caso mejor y 5 para el peor, aunque en la mayoría de los casos nos basta con 3 multiplicadores.

Como en el caso de los sumadores, los multiplicadores que no se utilizan son deshabilitados y sus entradas latcheadas.

7.2. Resultados experimentales

Para medir la calidad del algoritmo, se han realizado varios experimentos, utilizando dos ejemplos muy conocidos: el código AXPY y el adpcm decoder (véase Apéndice C). Los resultados obtenidos los hemos comparado con aquéllos de implementaciones sintetizadas por algoritmos convencionales que utilizan varias técnicas de bajo consumo, como la propiedad conmutativa de operaciones y multiplicaciones, similitud de los operandos de entrada, replicación de UFs o uso de latches que mantienen los operandos estables.

Para cada experimento se comparan diferentes parámetros: energía, tiempo de ejecución y área. La energía se mide en número de conmutaciones, ya que la *switching activity* es proporcional al consumo de potencia y, por lo tanto a la energía. Además, utilizamos la métrica de Energía Por Retardo (*EPR*), la cual captura la relación entre rendimiento y energía. EPR se define como el producto del tiempo de ejecución y de la energía consumida.

El código AXPY (y=a*x+y) se ha sintetizado para tres tamaños de datos: 8, 16 y 32 bits.

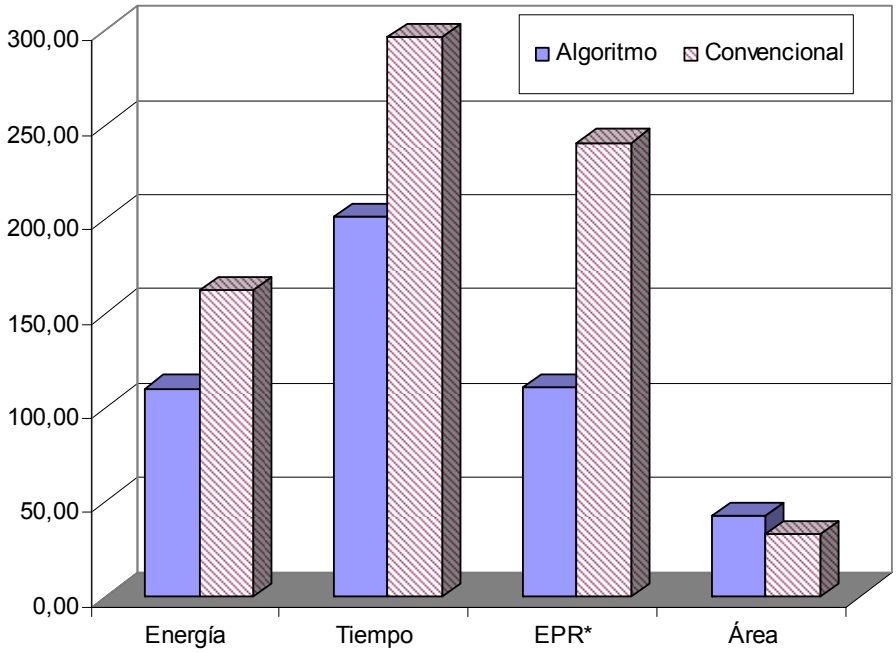


Figura 7.2. Parámetros promedio para el código AXPY

La figura 7.2 muestra la media de los valores obtenidos para cada uno de los parámetros. La energía, y el tiempo de ejecución se han reducido alrededor de un 30%, por lo que la mejora en el EPR es mayor del 50%. Como contrapartida, tenemos un pequeño incremento de área.

Nótese que el parámetro EPR ha sido normalizado para que pueda verse bien en el gráfico.

Por otro lado, hemos implementado dos módulos del adpcm decoder, en concreto: IAQ y OPFC, siguiendo el estándar G.711 de la ITU. Para realizar el *profiling* del adpcm hemos utilizado valores reales del ejemplo de referencia *clinton.adpcm*, y a partir de estas estadísticas hemos generado los patrones más comunes.

Nótese que el módulo OPFC se ha implementado con diferentes valores en los parámetros de entrada del algoritmo. Por ejemplo, el umbral de similitud es de 1,5 en la primera implementación, y de 1,2 en la segunda, mientras que los tamaños permitidos en la selección de patrones se encuentran entre 12 y 8 en la primera implementación y entre 12 y 10 en la segunda, de lo que podemos deducir que el umbral de similitud no es directamente proporcional al ahorro de energía conseguido, lo cual puede deberse a que valores muy

restrictivos de los parámetros pueden descartar patrones en el proceso de selección en un momento dado, que en un futuro pueden volver a aparecer. Lo que sí se mantiene generalmente en todas las implementaciones es que se consume menos cuanto mayores son los fragmentos seleccionados, de ahí que el no permitir fragmentos de tamaño 8 y 9 en la segunda implementación redunde en una mayor reducción del consumo.

Tabla 7.1. Resultados de la síntesis del adpcm decoder

ADPCM	Energía			EPR			Tiempo	Área
Ejemplo	Alg	Conv	%Ahorro	Alg	Conv	%Ahorro	%Ahorro	%Ahorro
IAQ	8	16	**50,00**	384	1216	**68,42**	**36,84**	**21,05**
OPFC	104	152	**31,58**	12480	23104	**45,98**	**21,05**	**-18,26**
OPFC	70	152	**53,95**	8400	23104	**63,64**	**21,05**	**-22,11**
Promedio	60,67	106,67	**45,18**	7088	15808	**59,35**	**26,32**	**-6,44**

Los resultados obtenidos podemos verlos en la tabla 7.1. El consumo de energía se reduce en un 45% en promedio, mientras que EPR alcanza una mejora del 60%.

Los experimentos realizados muestran que la energía y el tiempo ahorrados, y también el incremento de área, aumentan con respecto la anchura de los datos de la especificación. Para ejemplos grandes, podemos lograr grandes ahorros de energía y tiempo de ejecución, pero a cambio de un mayor incremento en el área.

Sin embargo, esta reducción tanto en energía como en tiempo produce excelentes resultados, ya que no podemos olvidar el hecho de que energía y rendimiento están muy relacionados. Por ejemplo, en los procesadores actuales se disminuye el rendimiento para ahorrar energía cuando el tiempo de ejecución no es crítico.

7.3. Conclusiones

En este capítulo se presenta un novedoso algoritmo de SAN, que realiza la planificación y asignación de una especificación conductual para reducir el consumo dinámico. Considera la información a nivel de bit de la *switching*

activity, representada por medio de patrones que capturan los valores más comunes de los operandos a lo largo del tiempo.

El algoritmo propuesto realiza la planificación y asignación de operandos basándose en los patrones proporcionados por las estadísticas, permitiendo la ejecución de operaciones en varios ciclos no consecutivos y sobre varias UFs enlazadas.

Las técnicas de diseño que se utilizan en este algoritmo extienden el espacio de diseño que exploran los algoritmos convencionales de bajo consumo.

Además, se permite la utilización de propiedades aritméticas a nivel de bit y la inhabilitación de algunas fracciones de los recursos funcionales cuando están ejecutando operaciones más pequeñas.

Los resultados obtenidos de la síntesis de circuitos reales muestran grandes mejoras en comparación con los algoritmos convencionales de bajo consumo.

7.3.1. Comparación de los algoritmos

En comparación con el algoritmo del capítulo 6, en este nuevo algoritmo hemos ampliado el ámbito de aplicación de las técnicas propuestas, incluyendo las operaciones del tipo multiplicación, y resolvemos las dependencias de datos sin tener que recurrir a simulaciones, simplemente mediante planificación.

Nótese que al fragmentar, las dependencias de datos (Lectura Después de Escritura, *LDE*) se relajan, de tal manera que podemos empezar a ejecutar fragmentos de una operación que depende de una anterior pero que no ha terminado completamente.

Además, la representación de los patrones más comunes permite capturar mejor que los casos más comunes el comportamiento de las operaciones a lo largo del tiempo.

Estas características, junto con la verificación de los experimentos basados en circuitos reales hacen que el algoritmo sea mucho más robusto que el anterior.

8. Conclusiones

En este trabajo se presentan dos algoritmos cuyo objetivo es minimizar el consumo de energía en los circuitos obtenidos de la síntesis de alto nivel. Para ello tratan de minimizar tanto el consumo de potencia como el tiempo de ejecución.

En ambos algoritmos se combinan una serie de técnicas de diseño basadas en propiedades aritméticas, (Ej. la propiedad conmutativa), arquitectónicas (Ej. inhabilitación de operandos) o mezcla de ambas (Ej. Fragmentación, propiedad conmutativa a nivel de bit), que son novedosas con respecto a los algoritmos convencionales de bajo consumo.

Además de estas nuevas técnicas se introducen dos sistemas de representación de datos para homogeneizar el tratamiento de los mismos, ya que, como hemos visto, el consumo depende en gran medida de la distribución de dichos datos a lo largo de la ejecución de los GFD.

En el primero de ellos la idea es disminuir el hardware del circuito, ya que al minimizar el área tendremos menos elementos que puedan conmutar, lo que además reduce el consumo estático. El consumo dinámico se reduce aumentando las posibilidades de reuso de las UFs mediante la predicción estática de los acarreos intermedios de las operaciones.

En el segundo algoritmo sin embargo, resolvemos las dependencias de datos por planificación y nos basamos en la idea de minimizar al máximo el número de cambios entre los operandos de entrada a las UFs.

De cara al futuro quedan la introducción de nuevas propiedades parciales, como puede ser la propiedad asociativa, y un estudio similar del consumo estático y dinámico de UFs de más de dos operandos y de UFs con modelos de ejecución sin propagación de acarreos, como por ejemplo los carry save adders. De igual manera, la introducción del nuevo sistema de notación basado en patrones nos servirá de base para recoger con mayor exactitud las probabilidades de que un bit valga ‘0’ ó ‘1’ sin más que aumentar el número de comportamientos representados, es decir, utilizar más de los 4 caracteres que hemos usado en los ejemplos.

9. Referencias

[BaMM06] A. Del Barrio. M.C. Molina, J.M. Mendías. “Bit-level power optimization during behavioural synthesis“. In Proceedings of Design of Circuits and Integrated Systems, DCIS 2006.

[BeBM00] L. Benini, A. Bogliolo, and C. De Micheli. “A survey of design techniques for system-level dynamic power management”. IEEE Transactions on VLSI systems, June 2000.

[ChBr95] A. Chandrakasau, and R. Brodersen. “Low Power Digital CMOS Design”. Kluwer Academic Publishers, 1995.

[ChJC00] J. Choi, J. Jeon, and K. Choi. “Power Minimization of Functional Units by Partially Guarded Computation”. In Proceedings of ISLPED, 2000.

[ErKP99] M. Ercegovac, D. Kirovski, and M. Potkonjak. “Low-power behavioural synthesis optimization using multiple precision arithmetic”. In Proceedings of Design Automation Conference, DAC 1999.

[GSJM97] S. Gailhard, O. Sentieys, N. Julien, and E. Martin. “Area/Time/Power Space Exploration in Module Selection for DSP High Level Synthesis”. In Proceedings of PATMOS, 1997.

[GrKu98] F. Gruian and K. Kuchcinski, “A Constraint Logic Programming Based Approach to High-Level Synthesis for Low Power”. Proceedings of Euromicro, 1998.

[KiCh97] D. Kim, and K. Choi. “ Power conscious high level synthesis using loop folding”. In Proceedings of Design Automation Conference, DAC 1997.

[KSCS01] L. Kruse, E. Schmidt, G. v. Cölln, A. Stammermann, A. Schulz, E. Macii, and W. Nebel, “Estimation of lower and upper bounds on the power consumption from scheduled data flow graphs”. IEEE Transactions on VLSI Systems, 2001.

[LCBK01] J. Liu, P.H. Chou, N. Bagherzadeh, F. Kurdahi. “Power-Aware Scheduling under Timing Constraints for Mission-Critical Embedded systems”. In Proceedings of Design Automation Conference, DAC 2001.

[MoRa05] S. P. Mohanty, N. Ranganathan. “Energy-efficient datapath scheduling using multiple voltages and dynamic clocking”. ACM Transactions on Design Automation of Electronic Systems (TODAES), 10(2):330-353, 2005.

[MoMH03] M.C. Molina, J.M. Mendías, R. Hermida. “Behavioural Specifications Allocation to Minimize Bit Level Waste of Functional Units”. In IEE Proceedings: Computers and Digital Techniques, Vol. 150 (5), September 2003.

[MRMH06] M.C. Molina, R. Ruiz-Sautua, J.M. Mendías, R. Hermida. “Bitwise Scheduling to Balance the Computational Cost of Behavioural Specifications”. In IEEE Transactions on Computer Aided Design, Vol. 25 (1), January 2006.

[MuRS05] V. Muthukumar, B. Radhakrishnan and H. Selvaraj. “Multiple voltage and frequency scheduling for power minimization”. Journal of Systems Architecture, 51 (6-7), 2005.

[MuCo99] E. Mussoll, and J. Cortadella. “High-level synthesis techniques for reducing the activity of functional units”. In Proceedings of International Symposium on System Synthesis, ISSS 1999.

[OcAK07] M. A. Ochoa-Montiel, B. M. Al-Hashimi, P. Kollig. “Exploiting Power-Area Tradeoffs in Behavioural Synthesis through Clock and Operations Throughput Selection”. Proceedings of ASP-DAC, 2007.

[ShCh97] D. Shin, and K, Choi, “Lower power high level synthesis by increasing data correlation”. In Proceedings of ISLPED,1997

[ShCh00] W. Shiue and C. Chakrabarti. “Low-power scheduling with resources operating at multiple voltages”. IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing, 47(6):536–543, 2000.

[SHSS03] A. Stammermann, D. Helms, M. Schulte, A. Schulz, W. Nebel. “Binding, Allocation and Floorplanning in Low Power High-Level Synthesis”. Proceedings of ICCAD, 2003.

[YaDu05] H. Yang, and L. Dung. “On multiple-voltage high-level synthesis using algorithmic transformations”. Proceedings of ASP-DAC, 2005.

[ZhHC02] Y. Zhang, X.S. Hu, and D.Z. Chen. “Task Scheduling and Voltage Selection for Energy Minimization”. In Proceedings of Design Automation Conference, DAC 2002.

[ZhJh02] L. Zhong, and N.K. Jha “Interconnect-aware High-level Synthesis for Low Power”. Proceedings of ICCAD, 2002.

10. Apéndice A: Álgebra para 4 patrones

Supongamos que tenemos la siguiente asignación de patrones

	1ªmitad	2ªmitad
A	0	0
B	0	1
C	1	0
D	1	1

Entonces definimos el conjunto universal {AND, OR, NOT} como sigue:

x	NOT x
A	D
B	C
C	B
D	A

Funciones NOT, y AND y OR de 2 entradas

x	y	x AND y	xOR y
A	A	A	A
A	B	A	B
A	C	A	C
A	D	A	D
B	A	A	B
B	B	B	B
B	C	A	D
B	D	B	D
C	A	A	C
C	B	A	D
C	C	C	C
C	D	C	D
D	A	A	D
D	B	B	D
D	C	C	D
D	D	D	D

11. Apéndice B: Distancia de Hamming para 4 patrones

		d
A	A	0
A	B	1
A	C	1
A	D	2
B	A	1
B	B	0
B	C	2
B	D	1
C	A	1
C	B	2
C	C	0
C	D	1
D	A	2
D	B	1
D	C	1
D	D	0

Distancia de Hamming para 4 patrones

Nótese que $d(x,y) = d(y,x)$

12. Apéndice C: Fragmento de código del decodificador del adpcm

El adpcm es un algoritmo de codificación/decodificación de la voz, muy utilizado en telefonía móvil. Por ello, se ha implementado la especificación dada por el estándar G.711 de la ITU (véase el diagrama de bloques de la figura 12.1), con el objetivo de obtener estadísticas de los datos de entrada en cada operación a partir de datos de entrada representativos. En nuestro caso hemos utilizado como entrada el fichero clinton.adpcm.

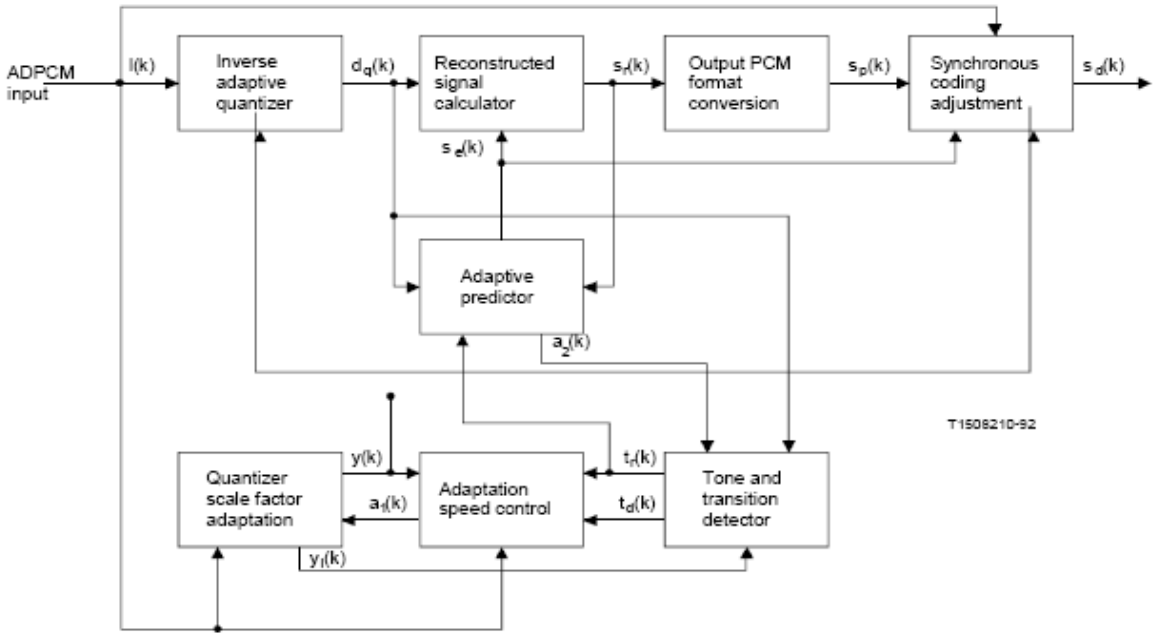


Figura 12.1. Diagrama de bloques del decodificador

Para capturar la información de los datos (enteros), hemos creado una clase envoltorio llamada Integer, modificando la de la librería GMP para enteros de longitud variable.

A continuación se muestran unos fragmentos del código

En Decoder.c tenemos el método main y los dos métodos que implementan el decodificador, uno preparado para leer ficheros de texto y otro para leer ficheros binarios.

Decoder.c

```
#include <stdlib.h>
#include <iostream>
#include <fstream>
#include "Integer_class/Integer.h"
#include "Modulos.h"
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define NSAMPLES 1000

char     abuf[NSAMPLES/2];
short    sbuf[NSAMPLES];

using namespace std;

extern int id_add;

void decoder(Integer &law, char* fich){

        Integer i, y, dq, sr, sp, sd, se, al, tr, td, a2, yl;//Signals that interconnect the
modules
        //Signals for simulating the delays
        Integer yup, yu, ylp, tdp, dms, dmsp, dml, dmlp, ap, apr, sez, pk0, pk1, pk2,
sigpk;
        Integer dq0, dq1, dq2, dq3, dq4, dq5, dq6, b1r, b1, b2r, b2, b3r, b3, b4r, b4, b5r,
b5;
        Integer b6r, b6, sr0, sr1, sr2, a1r, a1, a2r, tdr;

        ifstream in(fich);
        char cadena[64];

        while (!in.eof()){

                in >> cadena;

                //We update the variables for simulating the delays
                //We have to update before calling the module methods because if we
updated them
                //afterwards, we would have an instant with two variables with the same
value. For
                //example, pk0 and pk1.

                //quantizer_scale_factor_adaptation
                yu = yup;
                yl = ylp;

                //adaptation_speed_control
```

```
                        dms = dmsp;
                        dml = dmlp;
                        ap = apr;

                        //adaptive_predictor_and_reconstructed_signal_calculator
                        pk2 = pk1; pk1 = pk0;
                        sr2 = sr1; sr1 = sr0;
                        dq6 = dq5; dq5 = dq4; dq4 = dq3; dq3 =dq2; dq2 = dq1;
                        b1 = b1r; b2 = b2r; b3 = b3r; b4 = b4r; b5 = b5r; b6 = b6r;
                        a2 = a2r; a1 = a1r;

                        //tone_and_transition_detector
                        td = tdr;

                        id_add = 0;

                        dq = Modulos::inverse_adaptative_quantizer(i,y);
                        Modulos::quantizer_scale_factor_adaptation(i,al,yup,yu,ylp,yl,y);

        Modulos::adaptation_speed_control(i,y,tdp,tr,dms,dmsp,dml,dmlp,ap,apr,al);

        Modulos::adaptive_predictor_and_reconstructed_signal_calculator(dq,tr,se,sez,pk0,pk1,
pk2,sigpk,dq0,dq1,dq2,dq3,dq4,dq5,dq6,b1r,b1,b2r,b2,b3r,b3,b4r,b4,b5r,b5,
        b6r,b6,sr,sr0,sr1,sr2,a1r,a1,a2r,a2);
                        Modulos::tone_and_transition_detector(a2,yl,dq,tdp,tr,tdr,td);

        Modulos::output_pcm_format_conversion_and_synchronous_coding_adjustement(sr,la
w,se,y,i,sd);

                        sd.escribe();

                }

                in.close();

        }

        void adpcm_decoder(Integer &law, char* fich){

                Integer i, y, dq, sr, sp, sd, se, al, tr, td, a2, yl;//Signals that interconnect the
modules
                //Signals for simulating the delays
                Integer yup, yu, ylp, tdp, dms, dmsp, dml, dmlp, ap, apr, sez, pk0, pk1, pk2,
sigpk;
                Integer dq0, dq1, dq2, dq3, dq4, dq5, dq6, b1r, b1, b2r, b2, b3r, b3, b4r, b4, b5r,
b5;
                Integer b6r, b6, sr0, sr1, sr2, a1r, a1, a2r, tdr;


                int n;
                signed char* inp;
                int len;
                int bufferstep;
                Integer inputbuffer;

                int fd = open(fich,O_RDONLY);

                while (1){
                        n = read(fd, abuf, NSAMPLES/2);
                        if ( n < 0 ) {
```

```
                    perror("input file");
                    exit(1);
                }
                if ( n == 0 ) break;
                inp = (signed char*)abuf;
                len = 2*n;
                bufferstep=0;
                for ( ; len > 0 ; len-- ) {
                    inputbuffer = *inp++;
                        i = inputbuffer & 0xF;
                        if (i!=0){
                        printf("input-> ");
                        i.escribe();
                        }

                //We update the variables for simulating the delays
                //We have to update before calling the module methods because if we
updated them
                //afterwards, we would have an instant with two variables with the same
value. For
                //example, pk0 and pk1.

                //quantizer_scale_factor_adaptation
                yu = yup;
                yl = ylp;

                //adaptation_speed_control
                dms = dmsp;
                dml = dmlp;
                ap = apr;

                //adaptive_predictor_and_reconstructed_signal_calculator
                pk2 = pk1; pk1 = pk0;
                sr2 = sr1; sr1 = sr0;
                dq6 = dq5; dq5 = dq4; dq4 = dq3; dq3 =dq2; dq2 = dq1;
                b1 = b1r; b2 = b2r; b3 = b3r; b4 = b4r; b5 = b5r; b6 = b6r;
                a2 = a2r; a1 = a1r;

                //tone_and_transition_detector
                td = tdr;

                id_add = 0;

                dq = Modulos::inverse_adaptative_quantizer(i,y);
                Modulos::quantizer_scale_factor_adaptation(i,al,yup,yu,ylp,yl,y);

        Modulos::adaptation_speed_control(i,y,tdp,tr,dms,dmsp,dml,dmlp,ap,apr,al);

        Modulos::adaptive_predictor_and_reconstructed_signal_calculator(dq,tr,se,sez,pk0,pk1,
pk2,sigpk,dq0,dq1,dq2,dq3,dq4,dq5,dq6,b1r,b1,b2r,b2,b3r,b3,b4r,b4,b5r,b5,
        b6r,b6,sr,sr0,sr1,sr2,a1r,a1,a2r,a2);
                Modulos::tone_and_transition_detector(a2,yl,dq,tdp,tr,tdr,td);

        Modulos::output_pcm_format_conversion_and_synchronous_coding_adjustement(sr,la
w,se,y,i,sd);

                }
            }

            close(fd);
```

```
}

int main(int args, char** argv){

        int input2 = atoi(argv[2]);

        Integer i2(input2,"");

        adpcm_decoder(i2,argv[1]);
        //decoder(i2,argv[1]);

        return 0;
}
```

En Modulos.c tenemos todos los módulos que componen el decodificador. Cada uno de ellos está implementado con diferentes submódulos, de los cuales veremos sólo los relativos a OPFC (Output PCM Format Conversion) e IAQ (Inverse Adaptative Quantizer), que son los módulos utilizados en los ejemplos.

Modulos.c

```
#include "Modulos.h"
#include <stdlib.h>

//Delays are implemented out of the modulus, by changing the input/output variables

Integer Modulos::inverse_adaptative_quantizer(Integer &i, Integer &y){
        Integer dqs;
        Integer dqln;
        Integer dql;
        Integer dq;

        Submodulos::reconst(i,dqln,dqs);
        Submodulos::adda(dqln,y,dql);
        Submodulos::antilog(dqs,dql,dq);

        return dq;
}

void Modulos::quantizer_scale_factor_adaptation(Integer &i, Integer &al, Integer &yup,
Integer &yu, Integer &ylp,
        Integer &yl, Integer &y){
        Integer wi;
        Integer yut;

        Submodulos::functw(i,wi);
        Submodulos::filtd(wi,y,yut);
        Submodulos::limb(yut,yup);
        Submodulos::filte(yup,yl,ylp);
        Submodulos::mix(al,yu,yl,y);

}
```

```
	void Modulos::adaptation_speed_control(Integer &i, Integer &y, Integer &tdp, Integer
&tr, Integer &dms, Integer &dmsp, Integer &dml, Integer &dmlp, Integer &ap, Integer &apr,
Integer &al){
		Integer fi;
		Integer ax;
		Integer app;

		Submodulos::functf(i,fi);
		Submodulos::filta(fi,dms,dmsp);
		Submodulos::filtb(fi,dml,dmlp);
		Submodulos::subtc(dmsp,dmlp,tdp,y,ax);
		Submodulos::filtc(ax,ap,app);
		Submodulos::triga(tr,app,apr);
		Submodulos::lima(ap,al);
	}

	void Modulos::adaptive_predictor_and_reconstructed_signal_calculator(Integer &dq,
Integer &tr, Integer &se, Integer &sez, Integer &pk0, Integer &pk1, Integer &pk2, Integer &sigpk,
Integer &dq0, Integer &dq1, Integer &dq2, Integer &dq3, Integer &dq4, Integer &dq5, Integer
&dq6, Integer &b1r, Integer &b1, Integer &b2r, Integer &b2, Integer &b3r, Integer &b3, Integer
&b4r, Integer &b4, Integer &b5r, Integer &b5, Integer &b6r, Integer &b6, Integer &sr, Integer
&sr0, Integer &sr1, Integer &sr2, Integer &a1r, Integer &a1, Integer &a2r, Integer &a2){


		Integer wb1, wb2, wb3, wb4, wb5, wb6;
		Integer wa1, wa2;
		Integer u1, u2, u3, u4, u5, u6;
		Integer b1p, b2p, b3p, b4p, b5p, b6p;
		Integer a1t, a1p, a2t, a2p;

		Submodulos::addc(dq,sez,pk0,sigpk);

		Submodulos::addb(dq,se,sr);
		Submodulos::floatb(sr,sr0);

		Submodulos::floata(dq,dq0);

		//1
		Submodulos::sxor(dq1,dq,u1);
		Submodulos::upb(u1,b1,dq,b1p);
		Submodulos::trigb(tr,b1p,b1r);
		Submodulos::fmult(b1,dq1,wb1);

		//2
		Submodulos::sxor(dq2,dq,u2);
		Submodulos::upb(u2,b2,dq,b2p);
		Submodulos::trigb(tr,b2p,b2r);
		Submodulos::fmult(b2,dq2,wb2);

		//3
		Submodulos::sxor(dq3,dq,u3);
		Submodulos::upb(u3,b3,dq,b3p);
		Submodulos::trigb(tr,b3p,b3r);
		Submodulos::fmult(b3,dq3,wb3);

		//4
		Submodulos::sxor(dq4,dq,u4);
		Submodulos::upb(u4,b4,dq,b4p);
		Submodulos::trigb(tr,b4p,b4r);
		Submodulos::fmult(b4,dq4,wb4);
```

```
		//5
		Submodulos::sxor(dq5,dq,u5);
		Submodulos::upb(u5,b5,dq,b5p);
		Submodulos::trigb(tr,b5p,b5r);
		Submodulos::fmult(b5,dq5,wb5);

		//6
		Submodulos::sxor(dq6,dq,u6);
		Submodulos::upb(u6,b6,dq,b6p);
		Submodulos::trigb(tr,b6p,b6r);
		Submodulos::fmult(b6,dq6,wb6);

		Submodulos::upa2(pk0,pk1,pk2,a1,a2,sigpk,a2t);
		Submodulos::limc(a2t,a2p);
		Submodulos::trigb(tr,a2p,a2r);
		Submodulos::fmult(a2,sr2,wa2);

		Submodulos::upa1(pk0,pk1,a1,sigpk,a1t);
		Submodulos::limd(a1t,a2p,a1p);
		Submodulos::trigb(tr,a1p,a1r);
		Submodulos::fmult(a1,sr1,wa1);

		Submodulos::accum(wa1,wa2,wb1,wb2,wb3,wb4,wb5,wb6,se,sez);

	}

	void Modulos::tone_and_transition_detector(Integer &a2p, Integer &yl, Integer &dq,
Integer &tdp, Integer &tr, Integer &tdr, Integer &td){

		Submodulos::tone(a2p,tdp);
		Submodulos::trigb(tr,tdp,tdr);
		Submodulos::trans(td,yl,dq,tr);

	}

	void
Modulos::output_pcm_format_conversion_and_synchronous_coding_adjustement(Integer &sr,
	Integer &law, Integer &se, Integer &t, Integer &i, Integer &sd){


		Integer sp;
		Integer slx;
		Integer dx;
		Integer dlx,dsx;
		Integer dlnx;

		Submodulos::compress(sr,law,sp);
		Submodulos::expand(sp,law,slx);
		Submodulos::subta(slx,se,dx);
		Submodulos::log(dx,dlx,dsx);
		Submodulos::subtb(dlx,t,dlnx);
		Submodulos::sync(i,sp,dlnx,dsx,law,sd);

	}
```

Submodulos.c

```
	//INVERSE ADAPTATIVE QUANTIZER
```

```
void Submodulos::adda(Integer &dqln, Integer &y, Integer &dql){
//Addition of scale factor to logarithmic version of quantized difference signal
	id_add = 16;
	dql = (dqln + y>>2) & 4095;
	dql.setTipo("12_TC");
}

void Submodulos::antilog(Integer &dqs, Integer &dql, Integer &dq){
//Convert quantized difference signal from the logarithmic to the linear domain

	Integer ds;
	Integer dex;
	Integer dmn;
	Integer dqt;
	Integer dqmag;

	ds = dql >> 11;
	dex = (dql >> 7) & 15;
	dmn = dql & 127;
	id_add = 17;
	dqt = (1 << 7) + dmn;
	if (ds==0){
		id_add = 18;
		dqmag = (dqt << 7) >> (14 + (-dex));
	}
	else{
		dqmag = 0;
	}
	id_add = 19;
	dq = (dqs << 14) + dqmag;
	dq.setTipo("15_SM");

}

void Submodulos::reconst(Integer &i, Integer &dqln, Integer &dqs){
//Reconstruction of quantized difference signal in the logarithmic domain

	dqs = i >> 3;
	dqs.setTipo("1_TC");
	int dqln_int;
	switch((int)i){
		case 0:
			dqln_int = 2048;
			break;
		case 1:
			dqln_int = 4;
			break;
		case 2:
			dqln_int = 135;
			break;
		case 3:
			dqln_int = 213;
			break;
		case 4:
			dqln_int = 273;
			break;
		case 5:
			dqln_int = 323;
			break;
		case 6:
```

```
                        dqln_int = 373;
                        break;
                case 7:
                        dqln_int = 425;
                        break;
                case 8:
                        dqln_int = 425;
                        break;
                case 9:
                        dqln_int = 373;
                        break;
                case 10:
                        dqln_int = 323;
                        break;
                case 11:
                        dqln_int = 273;
                        break;
                case 12:
                        dqln_int = 213;
                        break;
                case 13:
                        dqln_int = 135;
                        break;
                case 14:
                        dqln_int = 4;
                        break;
                case 15:
                        dqln_int = 2048;
                        break;
                default: break;
        }
        dqln = dqln_int;
        dqln.setTipo("12_TC");
}
//OUTPUT PCM FORMAT CONVERSION

void Submodulos::compress(Integer &sr, Integer &law, Integer &sp){
//Convert from uniform PCM to either A-law or u-law PCM

        Integer is;
        Integer im;
        Integer imag;

        is = sr >> 15;
        //Convert two's complement to signed magnitude
        if (is==0){
                im = sr;
        }
        else{//is==1
                id_add = 99;
                im = (65536 + (-sr)) & 32767;
        }

        //im es sólo la magnitud, ya no tiene signo !!!
        if (law==0){//u-law
                imag = im;
        }
        else if (is==0){//A-law
                imag = im >> 1;
        }
```

```
		else{//A-law
			id_add = 100;
			imag = (im + 1) >> 1;
		}

		Integer decision;

		if (is==0 && law==1){
			decision = dameNumeroDeValorDeDecisionTabla1(imag);
			id_add = 101;
			decision = decision + 128;
			sp = invierteBitsPares(decision);
		}
		else if (is==1 && law==1){
			decision = dameNumeroDeValorDeDecisionTabla1(imag);
			sp = invierteBitsPares(decision);
		}
		else if (is==0 && law==0){
			decision = dameNumeroDeValorDeDecisionTabla2(imag);
			id_add = 102;
			sp = 255 + (-decision);
		}
		else{//is==0 && law==1
			decision = dameNumeroDeValorDeDecisionTabla2(imag);
			id_add = 103;
			sp = 127 + (-decision);
		}


		sp = invierteBitsPares(decision);
		sp.setTipo("8");

	}

	void Submodulos::expand(Integer &sp, Integer &law, Integer &slx){
	//Decode PCM code word, SP, according to Recommendation G.711, using character
signals (column 6,
	//before inversion of even bits for A-law) and values at decoder output (column 7). The
values at
	//decoder output, SLX, must be represented in 13-bit signed magnitude form for A-law
PCM and
	//14-bit signed magnitude form for u-Law PCM (the sign bit is equal to one for negative
values)

		Integer sss;
		Integer ssq;
		Integer ssm;

		if (law==0){//u-Law
			sss = sp >> 13;
			ssq = sp & 8191;
		}
		else{//A-law
			sss = sp >> 12;
			ssm = sp & 4095;
			ssq = ssm << 1;
		}

		//Convert signed magnitude to two's complement
		if (sss==0){
```

```
                slx = ssq;
        }
        else{//sss==1
                id_add = 104;
                slx = (16384 + (-ssq)) & 16383;
        }

        slx.setTipo("14_TC");

}

void Submodulos::log(Integer &dx, Integer &dlx, Integer &dsx){
//Convert difference signal from the linear to the logarithmic domain

        Integer ds;
        Integer dqm;
        Integer exp;
        Integer mant;

        ds = dx >> 15;
        //Convert dx from two's complement to signed magnitude
        if (ds==0){
                dqm = dx;
        }
        else{//ds==1
                id_add = 105;
                dqm = (65536 + (-dx)) & 32767;
        }
        //Compute exponent
        exp = flooredLog2(dqm);
        //Compute approximation log2(1+x) = x
        mant = ((dqm << 7) >> exp) & 127;
        //Combine 7 mantissa bits and 4 exponent bits into one 11-bit word
        id_add = 106;
        dlx = (exp << 7) + mant;
        dlx.setTipo("11_SM");
        ds.setTipo("1_TC");

}

void Submodulos::subta(Integer &slx, Integer &se, Integer &dx){
//Compute difference signal by subtracting signal estimate from input signal (or
quantized
//reconstructed signal in decoder)

        Integer sls;
        Integer sli;
        Integer ses;
        Integer sei;

        sls = slx >> 13;
        //Sign extension
        if (sls==0){
                sli = slx;
        }
        else{//sls==1
                id_add = 107;
                sli = 49152 + slx;
        }
        ses = se >> 14;
```

```
	//Sign extension
	if (ses==0){
		sei = se;
	}
	else{//ses==1
		id_add = 108;
		sei = 32768 + se;
	}
	id_add = 109;
	dx = sli + 65536;
	id_add = 110;
	dx = (dx + (-sei)) & 65535;
	dx.setTipo("16_TC");
}

void Submodulos::subtb(Integer &dlx, Integer &y, Integer &dlnx){
//Scale logarithmic version of difference signal by subtracting scale factor

	id_add = 111;
	dlnx = dlx + 4096;
	id_add = 112;
	dlnx = (dlnx + (-(y >> 2))) & 4095;
	dlnx.setTipo("12_TC");

}
```

En Integer.h e Integer.cc tenemos la clase *wrapper* o envoltorio de los enteros, con las cuales iremos obteniendo los datos de las operaciones.

Integer.h

```
/***************************** BEGIN *******************************/

#include <stdlib.h>

#ifndef		FALSE
#define		FALSE 0
#endif // FALSE
#ifndef		TRUE
#define		TRUE 1
#endif // TRUE

#ifndef _INTEGER_CLASS__
#define _INTEGER_CLASS__

#include <iostream>		// for cin & cout istreams & ostreams
#include <time.h>		// use time as seed for random numbers


/***************************** INTEGER *************************************/

#include <fstream>
#include <typeinfo>

using namespace std;

class Integer {
```

```
  private:
          int _x;
          char* tipo;
  public:
//
// Constructors
//
    Integer(void);
    Integer(int a);
    Integer(unsigned int a);
    Integer(const Integer &a);
    Integer(char * a);
    Integer(int a, char* t);
//
// Destructor
//
    ~Integer(void);
//
// Casting          (changing Integer to another type)
//
    operator bool(void);   // (bool) i
    operator int(void);                 // (int)  i
    operator unsigned int(void);// (unsigned int) i
    operator float(void);   // (float) i
    operator double(void);          // (double)i
//
// Assignments  (Setting Integer equal to something else)
//
    void operator =(Integer a);
    void operator =(int a);
    void operator =(unsigned int a);
    void operator =(char *a);
    void operator =(const char *a);
//
// Binary (one sided) operators
//
    Integer operator -(void);                   // c = -a;
    bool operator !(void);              // c = (bool) (!a)
    void operator +=(Integer a);   // c += a;
    void operator +=(int a);                    // c += a;
    void operator +=(unsigned int a);       // c += a;
    void operator -=(Integer a);    // c -= a;
    void operator -=(int a);                    // c -= a;
    void operator -=(unsigned int a);       // c -= a;
    void operator *=(Integer a);    // c *= a;
    void operator *=(int a);                    // c *= a;
    void operator *=(unsigned int a);       // c *= a;
    void operator /=(Integer a);    // c /= a;
    void operator /=(int a);                    // c /= a;
    void operator /=(unsigned int a);       // c /= a;
    void operator %=(Integer a);  // c %= a;
    void operator %=(int a);                    // c %= a;
    void operator ^=(int a);                    // c ^= a;
    void operator <<=(int b);                   // c <<= b;
    void operator >>=(int b);                   // c >>= b;
    void operator &= (Integer a);  // c |= a;
    void operator &=(int a);                    // c |= a;
    void operator &=(unsigned int a);       // c |= a;
    void operator |= (Integer a);   // c |= a;
    void operator |=(int a);                    // c |= a;
```

```
    void operator |=(unsigned int a);         // c |= a;
// WARNING: No ^ with type (Integer) as exponent

    Integer operator ++(void);                // c = a++;
    Integer operator --(void);                // c = a--;
    Integer operator ++(int from); // c = ++a;
    Integer operator --(int from);   // c = --a;

//
// The following operators must be friendly because the are binary
//
    friend Integer operator +(Integer a, Integer b);          // a + b;
    friend Integer operator +(Integer a, int b);       // a + b;
    friend Integer operator +(int a, Integer b);       // a + b;
    friend Integer operator -(Integer a, Integer b); // a - b;
    friend Integer operator -(Integer a, int b);       // a - b;
    friend Integer operator -(int a, Integer b);       // a - b;
    friend Integer operator *(Integer a, Integer b); // a * b;
    friend Integer operator *(Integer a, int b);       // a * b;
    friend Integer operator *(int a, Integer b);       // a * b;
    friend Integer operator /(Integer a, Integer b); // a / b;
    friend Integer operator /(Integer a, int b);       // a / b;
    friend Integer operator /(int a, Integer b);       // a / b;
    friend Integer operator %(Integer a, Integer b);          // a % b;
    friend Integer operator %(Integer a, int b);      // a % b;
    friend Integer operator %(int a, Integer b);      // a % b;
    friend Integer operator ^(Integer a, int b);      // a ^ b;
    friend Integer operator &(Integer a, Integer b);          // a & b;
    friend Integer operator &(Integer a, int b);      // a & b;
    friend Integer operator &(int a, Integer b);      // a & b;
    friend Integer operator |(Integer a, Integer b); // a | b;
    friend Integer operator |(Integer a, int b);       // a | b;
    friend Integer operator |(int a, Integer b);       // a | b;

    friend bool operator ==(Integer a, Integer b);  // (c == a)?
    friend bool operator ==(Integer a, int b);                // (c == a)?
    friend bool operator ==(int a, Integer b);                // (c == a)?
    friend bool operator !=(Integer a, Integer b);   // (c != a)?
    friend bool operator !=(Integer a, int b);                // (c != a)?
    friend bool operator !=(int a, Integer b);                // (c != a)?
    friend bool operator <(Integer a, Integer b);    // (c < a)?
    friend bool operator <(Integer a, int b);                 // (c < a)?
    friend bool operator <(int a, Integer b);                 // (c < a)?
    friend bool operator <=(Integer a, Integer b);  // (c <= a)?
    friend bool operator <=(Integer a, int b);                // (c <= a)?
    friend bool operator <=(int a, Integer b);                // (c <= a)?
    friend bool operator >(Integer a, Integer b);    // (c > a)?
    friend bool operator >(Integer a, int b);                 // (c > a)?
    friend bool operator >(int a, Integer b);                 // (c > a)?
    friend bool operator >=(Integer a, Integer b);  // (c >= a)?
    friend bool operator >=(Integer a, int b);                // (c >= a)?
    friend bool operator >=(int a, Integer b);                // (c >= a)?
    friend Integer& operator <<(Integer a, int b);            // a << b;
    friend Integer& operator >>(Integer a, int b);            // a >> b;


        //Métodos

        void escribe();
        void escribe(char* ruta);
```

```
        void escribe(FILE* f);
        char* getTipo();
        void setTipo(char* t);

};

#endif  //      _INTEGER_CLASS__
```

Integer.cc

```
/***************************** BEGIN *******************************/

#include <stdlib.h>
#include "Integer.h"

static const char* fich = "datos.res";
static FILE* out = fopen(fich,"w");
int id_add = 0;

//
// Constructors
//
Integer::Integer()                  {_x = 0; tipo=0;}
Integer::Integer(int a)             {_x = (int)a; tipo=0;}
Integer::Integer(unsigned int a){_x = (int)a; tipo=0;}
Integer::Integer(const Integer &a) {_x = a._x; tipo=0;}
Integer::Integer(char * a)          {_x = atoi(a); tipo=0;}
Integer::Integer(int a, char* t)    {_x = a; tipo = (char*)malloc((strlen(t)+1)*sizeof(char));
tipo=strcpy(tipo,t);}
//
// Destructor
//
Integer::~Integer()     {}      // releases memory
//
// Casting
//
Integer::operator bool(void)       { return (bool) _x; }
Integer::operator int(void)        { return _x; }
Integer::operator unsigned int(void)       { return (unsigned int)_x; }
Integer::operator float(void)      { return (float)_x; }
Integer::operator double(void)     { return (double)_x; }
//
// Assignments
//
void Integer::operator =(Integer a)        { _x = a._x;}
void Integer::operator =(int a)            { _x = a;}
void Integer::operator =(unsigned int a){ _x = (int)a;}
void Integer::operator =(char *a)          { _x = atoi(a);}
void Integer::operator =(const char *a)    { _x = atoi(a);}
//
// Unary Operators -, !, etc...
//
Integer Integer::operator -(void) {  return -_x; }
bool    Integer::operator !(void) { return (_x==0)? 1 : 0; }
void    Integer::operator +=(Integer a)            { fprintf(out,"%b,%d\n",_x,id_add);
fprintf(out,"%b,%d\n",a,id_add); _x = _x + a._x; }
void    Integer::operator +=(int a)                { fprintf(out,"%b,%d\n",_x,id_add);
fprintf(out,"%b,%d\n",a,id_add); _x = _x + a; }
```

```
        void    Integer::operator +=(unsigned int a)    { fprintf(out,"%b,%d\n",_x,id_add);
fprintf(out,"%b,%d\n",a,id_add); _x = _x + (int)a; }
        void    Integer::operator -=(Integer a)         { _x = _x - a._x; }
        void    Integer::operator -=(int a)             { _x = _x - a; }
        void    Integer::operator -=(unsigned int a)    { _x = _x - (int)a; }
        void    Integer::operator *=(Integer a)         { _x = _x * a._x; }
        void    Integer::operator *=(int a)             { _x = _x * a;}
        void    Integer::operator *=(unsigned int a)    { _x = _x * (int)a; }
        void    Integer::operator /=(Integer a)         { _x = _x / a._x; }
        void    Integer::operator /=(int a)             { _x = _x / a;}
        void    Integer::operator /=(unsigned int a)    { _x = _x / (int)a; }
        void    Integer::operator %=(Integer a)         { _x = _x % a._x; }
        void    Integer::operator %=(int a)             { _x = _x % a; }
        void    Integer::operator ^=(int a)             { _x = _x ^ a; }
        void    Integer::operator <<=(int b) {_x = _x << b;}
        void    Integer::operator >>=(int b) {_x = _x >> b;}
        void    Integer::operator &=(Integer a)         { _x = _x & a._x; }
        void    Integer::operator &=(int a)             { _x = _x & a; }
        void    Integer::operator &=(unsigned int a)    { _x = _x & (int)a; }
        void    Integer::operator |=(Integer a)         { _x = _x | a._x; }
        void    Integer::operator |=(int a)             { _x = _x | a; }
        void    Integer::operator |=(unsigned int a)    { _x = _x | (int)a; }

        //
        // Inc/Dec --  prefix:  ++a
        Integer Integer::operator ++(void) { _x = _x + 1; return _x;}
        Integer Integer::operator --(void) { _x = _x - 1; return _x;}
        //
        // Inc/Dec -- postfix: a++
        Integer Integer::operator ++(int a) {Integer r=_x; _x = _x + 1; return r; }
        Integer Integer::operator --(int a) {Integer r=_x; _x = _x - 1; return r;}
        //
        //  Binary (friendly) operators
        //

        Integer operator +(Integer a, Integer b) {fprintf(out,"%d,%d\n",a._x,id_add);
fprintf(out,"%d,%d\n",b._x,id_add); Integer c; c._x = a._x + b._x; return c;}
        Integer operator +(Integer a, int b) {fprintf(out,"%d,%d\n",a._x,id_add);
fprintf(out,"%d,%d\n",b,id_add); Integer c; c._x = a._x + b; return c;}
        Integer operator +(int a, Integer b) {fprintf(out,"%d,%d\n",a,id_add);
fprintf(out,"%d,%d\n",b._x,id_add); Integer c; c._x = a + b._x; return c; }
        Integer operator -(Integer a, Integer b) {Integer c; c._x = a._x - b._x; return c;}
        Integer operator -(Integer a, int b) { return a + (-b);}
        Integer operator -(int a, Integer b) { return -(b-a); }
        Integer operator *(Integer a, Integer b) {Integer c; c._x = a._x * b._x; return c; }
        Integer operator *(Integer a, int b) {Integer c; c._x = a._x * b; return c; }
        Integer operator *(int a, Integer b) { return b*a; }
        Integer operator /(Integer a, Integer b) {Integer c; c._x = a._x / b._x; return c; }
        Integer operator /(Integer a, int b) {Integer c; c._x = a._x / b; return c; }
        Integer operator /(int a, Integer b) {Integer c; c = a; return c/b; }
        Integer operator %(Integer a, Integer b) {Integer c; c._x = a._x % b._x; return c; }
        Integer operator %(Integer a, int b) {Integer c; c._x = a._x % b; return c; }
        Integer operator %(int a, Integer b) {Integer c; c = a; return c%b; }
        Integer operator ^(Integer a, int b) {Integer c; c._x = a._x ^ b; return c; }
        Integer operator &(Integer a, Integer b) {Integer c; c._x = a._x & b._x; return c; }
        Integer operator &(Integer a, int b) {Integer c; c._x = a._x & b; return c;}
        Integer operator &(int a, Integer b) { return b&a; }
        Integer operator |(Integer a, Integer b) {Integer c; c._x = a._x | b._x; return c; }
        Integer operator |(Integer a, int b) {Integer c; c._x = a._x | b; return c;}
        Integer operator |(int a, Integer b) { return b|a; }
```

```
bool operator ==(Integer a, Integer b) {return a._x == b._x;}
bool operator ==(Integer a, int b) {return a._x == b;}
bool operator ==(int a, Integer b) {return (b==a); }
bool operator !=(Integer a, Integer b) {return a._x != b._x;}
bool operator !=(Integer a, int b) {return a._x != b;}
bool operator !=(int a, Integer b) {return (b!=a); }
bool operator <(Integer a, Integer b) {return a._x < b._x;}
bool operator <(Integer a, int b) {return a._x < b;}
bool operator <(int a, Integer b) {return (b>a); }
bool operator <=(Integer a,Integer b) {return a._x <= b._x;}
bool operator <=(Integer a, int b) {return a._x <= b;}
bool operator <=(int a, Integer b) {return (b>=a); }
bool operator >(Integer a, Integer b) {return a._x > b._x;}
bool operator >(Integer a, int b) {return a._x > b;}
bool operator >(int a, Integer b) {return (b<a); }
bool operator >=(Integer a, Integer b) {return a._x >= b._x;}
bool operator >=(Integer a, int b) {return a._x >= b;}
bool operator >=(int a, Integer b) {return (b<=a); }

Integer& operator <<(Integer a, int b) {
        Integer c; c=a; a._x = c._x << b; return a;
        }
Integer& operator >>(Integer a, int b) {
        Integer c; c=a; a._x = c._x >> b; return a;
        }

//Métodos

void Integer::escribe(){
        if (tipo){
                printf("(%d, %s)\n",_x,tipo);
        }
        else{
                printf("(%d, )\n",_x);
        }
}

void Integer::escribe(char* ruta){
        ofstream out2(ruta);
        out2 << "(" << _x << ", " << tipo << ")" << endl;
        out2.close();
}

void Integer::escribe(FILE* f){
        fprintf(f,"%d",_x);
}

char* Integer::getTipo(){
        return tipo;
}

void Integer::setTipo(char* t){
        tipo = (char*)malloc((strlen(t)+1)*sizeof(char));
        tipo = strcpy(tipo,t);
}
```