Proyecto Fin de Máster en Ingeniería de Computadores

Máster en Investigación en Informática, Facultad de Informática, Universidad Complutense de Madrid

Reducción de la actividad de conmutación a nivel de subpalabra en síntesis de alto nivel

***Autor**: Diego González Rodríguez*

***Directora**: Maria de Carmen Molina Prego*

***Colaborador externo**: Guillermo Botella Juan*

***Curso académico**: 2007 - 2008*

Contenido

ABSTRACT

A power-aware high-level synthesis algorithm specially suited to reduce dynamic dissipation in data-dominated applications is presented.

It overcomes the limitations of conventional low-power algorithms, as it deals with switching activity information at the subword level.

Our algorithm uses a novel pattern-based representation of the switching activity information to capture the values of input operand bits along the simulation time.

The proposed algorithm performs the scheduling and binding operations in a pattern-matching basis taking advantage of the data-level parallelism.

In order to minimize the switching activity in functional units, it allows the partial application of arithmetic operation properties, the distributed execution of operations over different functional units, and the exploration of different bit alignments in the execution of operation fragments.

These design techniques enlarge the solution space explored in comparison to previous approaches, resulting in datapaths with smaller number of commutations and significant savings of power consumption.

RESUMEN

Presentamos un algoritmo de síntesis de alto nivel basado en la potencia, hecho para reducir la disipación dinámica en aplicaciones dominadas por datos.

Este algoritmo supera las limitaciones convencionales de los algoritmos de ahorro de potencia, ya que trabaja con la información de la actividad de conmutación a nivel de subpalabra.

Nuestro algoritmo usa una nueva representación de la información de la actividad de conmutación basada en patrones para capturar los valores de los bits de los operandos de entrada a lo largo del tiempo de simulación.

El algoritmo propuesto lleva a cabo la planificación y asignación de operaciones sobre una base de coincidencia de patrones tomando como ventaja el paralelismo a nivel de datos.

Para minimizarla actividad de conmutaciones en las unidades funcionales, permite la aplicación parcial de propiedades de las operaciones aritméticas, la ejecución distribuida de las operaciones sobre diferentes unidades funcionales, y la exploración de diferentes alineaciones de bits en la ejecución de los fragmentos de las operaciones.

Estas técnicas de diseño agrandan el espacio de soluciones explorado en comparación con enfoques anteriores, dando lugar a rutas de datos con un menor número de conmutaciones y cantidades de ahorro significantes en el consumo de potencia.

KEY WORDS

Scheduling

Binding

Subword

Fragment

Operation

Functional unit

Save

Consumption

Algorithm

PALABRAS CLAVE

Planificación

Asignación

Subpalabra

Fragmentar

Operación

Unidad funcional

Ahorro

Consumo

Algoritmo

INTRODUCCIÓN

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.

A medida que la tecnología avanza con nodos de 90 y 65 nanómetros, el rendimiento y la densidad alcanzan nuevos niveles, las pérdidas de potencia debidas a las conmutaciones y a las corrientes de fuga hacen el diseño con este tipo de dispositivos un reto mayor.

La reducción de las corrientes de fuga resulta esencial para sustentar la escalada tecnológica en el diseño CMOS. Las pérdidas en nodos de 90 nanómetros y menores pueden significar más de la mitad de la potencia perdida en un procesador.

Así pues, el consumo tanto estático como dinámico deben de tenerse en cuenta a fin de diseñar con éxito un sistema eficiente.

La potencia disipada por un circuito puede ser considerada desde cualquier nivel de abstracción del diseño, alcanzando mayores reducciones en los niveles más altos.

En la síntesis de alto nivel (SAN) se han propuesto muchas técnicas diferentes para optimizar las pérdidas y el consumo de potencia dinámica. En esta memoria proponemos un algoritmo de alto nivel que optimiza el consumo de potencia dinámica reduciendo las conmutaciones producidas a nivel de subpalabra.

El algoritmo propuesto toma como entradas los valores a nivel de bit de los operandos a lo largo del tiempo y una representación basada en patrones que captura las conmutaciones ocurridas.

La planificación y la asignación de operaciones se basan en la identificación y explotación de subpatrones comunes permitiendo la aplicación parcial de algunas propiedades de las operaciones aritméticas (cierta propiedad aritmética se aplica a únicamente un parte de la operación), la ejecución distribuida de operaciones sobre diferentes unidades funcionales, y la posibilidad de diferentes alineaciones de los fragmentos de las operaciones.

La aplicación de estas técnicas produce la fragmentación de algunas operaciones en varios fragmentos más pequeños con una alta probabilidad de incrementar la correlación de datos en unidades funcionales, de almacenamiento y de enrutado.

De esta forma, el número de conmutaciones se reduce de forma significativa en comparación con aproximaciones anteriores. Hasta donde llega nuestro conocimiento, éste es el primer intento con éxito que considera la correlación de datos a nivel de subpalabra.

La memoria se encuentra estructurada de la siguiente manera. La sección II resume el trabajo relacionado. La sección III profundiza en la representación de la actividad de conmutación basada en patrones. La sección IV muestra el uso de la técnica propuesta con un ejemplo. El algoritmo propuesto para explotar la información de conmutaciones a nivel de subpalabra se explica en la sección V. Por último, el trabajo experimental y algunas conclusiones se exponen en las secciones VI y VII respectivamente.

TRABAJO PREVIO

La tecnología actual permite que diferentes tipos de dispositivos convivan en el mismo chip. De esta forma distintas instancias pueden tener la misma estructura a nivel de puertas lógicas e interconexiones, pero ser diferentes en área, tiempo de ejecución o consumo de potencia.

El espacio de diseño ha sido recientemente extendido con el uso de librerías de diseño que contienen diferentes implementaciones para el mismo tipo de módulo utilizando múltiples voltajes.

Técnicas de doble y triple voltaje se pueden usar para mejorar las pérdidas de consumo de energía en grafos de flujo de datos. También pueden usarse múltiples voltajes para permitir que los módulos en el camino crítico puedan operar a mayor voltaje.

El consumo dinámico ha sido atenuado mediante el uso de técnicas como la denominada “clock gating”, que consiste en desactivar el reloj en las celdas lógicas donde no hay actividad, y “dynamic clock” que se usan en síntesis de alto nivel para permitir la existencia de diferentes longitudes de ciclo de reloj en diferentes pasos de control.

Otras técnicas de bajo consumo han contribuido también a reducir de forma significativa la pérdida de potencia, como por ejemplo métodos de reducción de pico de temperatura. Sin embargo, la mayoría de estas técnicas imponen restricciones en el rendimiento y la implementación del circuito. Por el contrario, la optimización de la actividad de conmutación no impone restricciones en los parámetros del circuito.

Ya que el consumo de potencia dinámica esta en relación directa con la actividad de conmutación del circuito y la capacitancia, muchas investigaciones se han centrado en reducir la disipación dinámica decrementando las transiciones en las unidades funcionales, registros, multiplexores y buses.

Como método efectivo para reducir las transiciones se utiliza el incremento de la correlación de los datos de entrada, que puede alcanzarse modificando la planificación de operaciones, la asignación de operaciones, el desarrollo de las iteraciones, el intercambio de iteraciones, la reordenación de operandos y la compartición de operandos.

La conmutaciones y por tanto el consumo de potencia, es una función de las señales de entrada y la correlación, y por lo tanto no es determinista.

La potencia disipada por un circuito puede verse fuertemente afectada si las probabilidades de señales de entrada primarias son alteradas, de forma que un

circuito simple puede ser sintetizado óptimamente de diferentes formas, si las aplicaciones requieren distintos tipos de entradas.

Normalmente, son dos métodos los más usados en la de síntesis de alto nivel para dirigir el flujo de diseño teniendo en cuenta el impacto producido por las conmutaciones. Muchos algoritmos de baja potencia usan simuladores que realizan ejecuciones del modelo para recoger la actividad de conmutación. La información que proporciona el simulador sirve de entrada para la siguiente fase del proceso de síntesis. La segunda opción es usar la información que proporciona el floorplanning para estimar el consumo de potencia para ciertos flujos de datos en las entradas. Aquí, se hace necesario el uso de modelos que describen el consumo de potencia y el área de cada recurso individual al nivel de transferencia de registro. Sin embargo, ambos métodos de diseño solo se utilizan a nivel de operación por los algoritmos existentes, dando lugar a soluciones subóptimas dados los beneficios de computar la actividad de conmutación o el consumo de potencial a nivel de subpalabra.

La primera aproximación en manejar la actividad de conmutación a nivel de subpalabra se presenta en [22]. Los autores proponen una forma muy simplificada de capturar las conmutaciones. Cada bit del operando se representa por la transición más común ocurrida durante la simulación (cero a cero, cero a uno, uno a cero ó uno a uno). Este método tiende a diferir bastante de las formas simuladas, perdiendo la eficiencia de las técnicas de diseño usadas para reducir conmutaciones. La falta de precisión de este método y la simplicidad de los algoritmos propuestos para desarrollar la planificación y asignación lo hacen poco idóneo para la síntesis de alto nivel de bajo consumo.

El algoritmo propuesto de síntesis de alto nivel en esta memoria palia este problema y recoge los valores más comunes dados en cada bit de cada operando a lo largo del tiempo de la simulación. Usamos diferentes patrones para representar de forma apropiada los valores simulados para cada operando.

También se propone un algoritmo de síntesis de alto nivel basado en patrones para reducir la actividad de conmutación a nivel de subpalabra. Identifica patrones similares dentro de las operaciones, y fragmenta algunas operaciones para así incrementar la correlación de datos dentro de cada unidad funcional.

Los fragmentos de las operaciones son planificados y asignados de forma separada, permitiendo la aplicación de las propiedades aritméticas solo a algunos de los fragmentos extraídos de la operación original. Los distintos fragmentos pueden usar también las mismas unidades funcionales aunque sean de diferentes operaciones, así que el conjunto de operaciones ejecutadas en cada unidad funcional puede presentar alineaciones de bits inalcanzables con las propuestas anteriores.

De esta forma, la mayoría de las implementaciones sintetizadas por nuestra propuesta son inasequibles para los algoritmos existentes.

Nuestra propuesta puede ser también usada en combinación con algoritmos desarrollados para reducir la pérdida de potencia estática ya que ellos se centran en diferentes aspectos del diseño.

Técnicas de múltiples voltajes podrían ser usadas para seleccionar el módulo apropiado, mientras tanto nuestra técnica podría usarse para agrupar el conjunto de operaciones candidatas a compartir un recurso funcional y planificarlas para reducir el número de conmutaciones.

REPRESENTACION DE LA ACTIVIDAD DE CONMUTACION

La información referente a la actividad de conmutación reportada por las simulaciones de alto nivel representando el comportamiento del sistema en modo normal de operación se usa típicamente para reducir el consumo de potencia dinámica.

Nuestro propósito consiste en la traducción de la información de la actividad de conmutación recogida de las simulaciones en una representación más apropiada, que captura el porcentaje de conmutaciones que se dan en cada bit de cada operando a lo largo del tiempo de simulación.

El tiempo de simulación se divide en diferentes ventanas para hacer corresponder cada ventana con una ruta de datos y el número de conmutaciones se tiene en cuenta de forma independiente para cada ventana, recogiendo así el porcentaje de conmutaciones más común para cada ventana de simulación.

Para poder capturar la actividad de conmutación de los circuitos simulados, proponemos una representación basada en patrones de los valores de los operandos de entrada. Este modelo recoge el total del tiempo que cada bit de cada operando es o cero o uno. Se asocia un símbolo a cada bit de cada operando de entrada indicando el porcentaje del tiempo en el que su valor es cero, asumiendo que el tiempo restante del total su valor es uno.

En nuestra formulación hemos usado los caracteres del alfabeto inglés para representar los valores simulados de los operandos de entrada. El carácter A representa que el valor del bit correspondiente del operando ha sido siempre cero, es decir, el bit tuvo un valor de cero durante el 100% de las simulaciones. El carácter B significa que el correspondiente bit del operando ha tenido un valor de cero el 96% del tiempo de las simulaciones y un valor de uno el 4% restante. El significado del resto de los caracteres del alfabeto se calcula restando 4 al porcentaje del carácter anterior. Por último, el carácter Z representa que el valor del bit del operando tuvo un valor de 1 el 100% del tiempo.

La correlación en el significado de dos caracteres consecutivos hace más fácil la identificación de las operaciones que, ejecutadas en la misma unidad funcional, producen el menor número de conmutaciones.

Generación de patrones de operaciones

Cada operación se representa a través de un patrón de actividad de conmutación. Para generar los correspondientes patrones, la simulación de la actividad de conmutación se divide de acuerdo al número y duración de las ventanas de simulación definidas por el diseñador.

Cada bit de cada operación es representado por un carácter en cada ventana de simulación. Estos caracteres se obtienen del análisis de los valores simulados en cada ventana de simulación (un carácter por ventana). La concatenación ordenada de los caracteres que definen el valor del bit en cada ventana muestra su valor a lo largo de toda la simulación.

Dado un operando en una ventana de simulación, los primeros caracteres representan la actividad de conmutación de los bits más significativos, los últimos caracteres representan la actividad de conmutación de los bits menos significativos y los caracteres intermedios representan las conmutaciones en los bits intermedios.

El proceso de generación de patrones de operación se lleva a cabo una única vez y tiene lugar antes del proceso de síntesis de alto nivel. Las operaciones con patrones más cercanos (según el orden alfabético) son más apropiadas para compartir la misma unidad funcional.

Número y longitud de las ventanas de simulación

Estos parámetros permiten el ajuste de precisión en la representación de los valores simulados. El número más apropiado de ventanas y la longitud más apropiada dependen del tiempo empleado para la simulación y la variabilidad en el conjunto de estímulos usados para simular el comportamiento.

Para simulaciones mayores con una alta variabilidad de los datos de entrada, es preferible un mayor número de ventanas.

Para poder afinar lo más posible en el número y longitud de ventanas, los valores simulados deberían ser analizados para identificar los intervalos donde se dan pocos cambios en los valores de las señales. Cada intervalo debería ser considerado como una ventana de simulación diferente.

Ambos parámetros, el número y la longitud de las ventanas de simulación, afectan al resultado de la síntesis de alto nivel, aunque la influencia principal la produce el número de ventanas.

Por un lado, el uso de una única ventana puede llevar a que el algoritmo tome algunas decisiones de diseño que no ayuden a reducir el número de conmutaciones debido a la falta de precisión de los patrones de las operaciones.

Pensemos por ejemplo que dos señales opuestas siendo 0 y 1 respectivamente, durante la primera mitad de la simulación, y los valores opuestos durante la segunda mitad de la simulación serían representadas con el carácter M. Si usásemos dos ventanas resolveríamos el problema ya que los patrones asignados para cada señal serían A y Z en la primera ventana; y Z y A en la segunda ventana respectivamente.

Por otro lado, un número alto de ventanas puede incrementar el tiempo de diseño y no contribuir a la reducción adicional del número de conmutaciones. Un número de ventanas apropiado puede ser seleccionado en función del número de conmutaciones que diferencian una iteración de los valores de entrada con la siguiente.

En todos nuestros experimentos se han usado varias ventanas de la misma longitud obteniendo una reducción significativa del número de conmutaciones.

Cálculo de la similitud de los patrones

Para poder identificar las operaciones o los fragmentos de operaciones con patrones de actividad de conmutación similares, la similitud de estos patrones se mide usando una redefinición de la distancia de Hamming llamada Patrón de la distancia de Hamming (PHD).

El cálculo del PHD solo se realiza para parejas de operaciones o fragmentos de operaciones de la misma longitud ya que no se permite la ejecución de operaciones en unidades funcionales de mayor anchura.

El PHD de dos patrones se define como la suma de las distancias entre los caracteres de los patrones que ocupan las mismas posiciones.

Como las ventanas de simulación pueden tener diferente duración, se asocia un peso a cada carácter del patrón (en relación directa con la longitud de la ventana de simulación correspondiente).

En esta versión del algoritmo lo hemos fijado teniendo en cuenta el número de iteraciones de los valores de entrada que contiene la ventana correspondiente.

#PHD(pa1, pa2) = $\sum_{i=0}^{width-1}(\mathrm{CHD}(\mathrm{pa(i)},\mathrm{pb(i)}) * \mathrm{weight(i)})$ siendo

Width: anchura del patrón pa y del patrón pb.

Pa(i), pb(i): iésimo carácter del patrón pa y pb respectivamente.

Weight(i): peso asociado a la ventana cuyos patrones estamos comparando.

CHD(x, y): Distancia de Hamming entre dos caracteres (CHD) del alfabeto inglés, x e y, representando los valores de simulación de dos bits de dos operandos en dicha ventana.

El CHD de dos caracteres se resuelve como el número de caracteres intermedios en el alfabeto teniendo un valor entre 0 y 25.

Por ejemplo, CHD(a, a) = 0, CHD(a, b)= 1, y CHD(a, c) = 2.

La figuras 1 a) a') a'') muestran los patrones asociados a tres operaciones diferentes extraídas de la especificación del comportamiento tras la simulación en las tres ventanas resultantes. El PHD entre los tres patrones se muestra en las figuras 1 b) b') b'').

Figura 1 a)

Operación	**Patrón**
Add1 (A + B)	AAAAAAAZ + AAAAAAZA
Add2 (C + D)	AAAAAAZZ + AAAAAZAA
Add3 (G + H)	AAAAAZAZ + AAAAAZZA
Mul1 (E x F)	AAAAAAZZ x AAAAAZZZ
Mul2 (I x K)	AAAAAZZZ x AAAAZAZZ
Mul3 (J x L)	AAAAAAAAAAAZAZAZ x AAAAAAAAAZAAZZAZ

Figura 1 a')

Operación	**Patrón**
Add1 (A + B)	AAAAAAAZ + AAAAAAZA
Add2 (C + D)	AAAAAAZZ + AAAAAZAA
Add3 (G + H)	AAAAZZAZ + AAAAAZAA
Mul1 (E x F)	AAAAAAZZ x AAAAAZZZ
Mul2 (I x K)	AAAAAAZA x AAAZAAAZ
Mul3 (J x L)	AAAAAAAAAAAZAZAZ x AAAAAAAAAAZAAAZA

Figura 1 a'')

Operación	Patrón
Add1 (A + B)	AAAAZZZA + ZZZZAAAA
Add2 (C + D)	AAZZZAAA + AAAAAZZZ
Add3 (G + H)	AZAZAZAA + AAZZZZZA
Mul1 (E x F)	ZZZZZZZA x AAZZZZZZ
Mul2 (I x K)	AAAAAAAA x ZAAZAAZA
Mul3 (J x L)	AAZZZZZAZAAAAAZA x AAAAAAAAAAAAAAAA

Figura 1 b)

Operaciones	PHD
Add1 → Add2	150
Add1 → Add3	100
Add2 → Add3	150
Mul1 → Mul2	150

Figura 1 b')

Operaciones	PHD
Add1 → Add2	150
Add1 → Add3	200
Add2 → Add3	150
Mul1 → Mul2	200

Figura 1 b'')

Operaciones	PHD
Add1 → Add2	550
Add1 → Add3	450
Add2 → Add3	400
Mul1 → Mul2	600

EJEMPLO MOTIVACIONAL

Para poder mostrar la ventaja añadida de considerar la información de conmutaciones a nivel de subpalabra, hemos sintetizado el diagrama de flujo de datos mostrado en la figura 2 a) con los valores de los operandos de entrada que se muestran a continuación:

```
force A 00000001
force B 00000010
force C 00000011
force D 00000100
force G 00000101
force H 00000110
force I 00000111
run 100ns

force A 00000001
force B 00000010
force C 00000011
force D 00000100
force G 00001101
force H 00000100
force I 00000010
run 100ns

force A 00001110
force B 11110000
force C 00111000
force D 00000111
force G 01010100
force H 00111110
force I 00000000
run 100ns

force A 00000001
force B 00000010
force C 00000011
force D 00000100
force G 00000101
force H 00000110
force I 00000111
run 100ns
```

```
force A 00000001
force B 00000010
force C 00000011
force D 00000100
force G 00001101
force H 00000100
force I 00000010
run 100ns

force A 00001110
force B 11110000
force C 00111000
force D 00000111
force G 01010100
force H 00111110
force I 00000000
run 100ns
```

Hemos realizado diferentes simulaciones de la representación de alto nivel del diseño, en modo normal de operación de las señales de entrada. En las figuras 2 b) b') b'') podemos ver un resumen de la información de la actividad de conmutación recogida en tres ventanas diferentes cada una de ellas correspondientes a dos iteraciones del programa con valores diferentes de los valores de entrada.

Esta tabla muestra un promedio estimado del número de transiciones de cero a uno y de uno a cero dados en una unidad funcional debido a la ejecución de dos operaciones en ciclos consecutivos.

Además de las transiciones de cero a uno, que tienen una influencia directa en el consumo dinámico, también hemos medido las transiciones de uno a cero, debido a su efecto en el consumo de potencia interno en cortocircuito.

Figura 2 a)

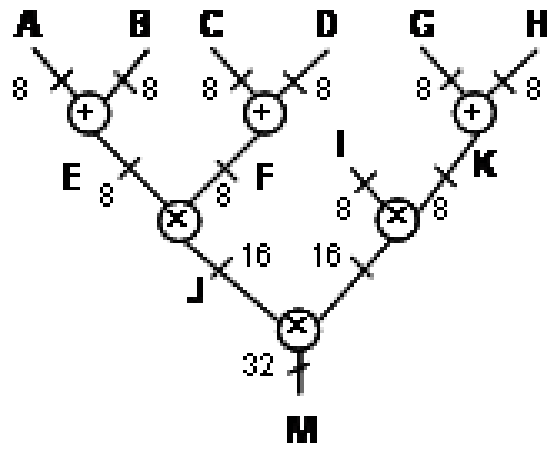


Figura 2 b)

	Transiciones 0 → 1 / 1 → 0	
A + B → C + D	1/0	1/1
A + B → G + H	1/0	1/0
C + D → A + B	0/1	1/1
C + D → G + H	1/1	1/0
G + H → A + B	0/1	0/1
G + H → C + D	1/1	0/1
Ex F → I x K	1/0	1/1
I x K → E x F	0/1	1/1

Figura 2 b')

	Transiciones 0 → 1 / 1 → 0	
A + B → C + D	1/0	1/1
A + B → G + H	2/0	1/0
C + D → A + B	0/1	1/1
C + D → G + H	2/1	0/0
G + H → A + B	0/2	0/1
G + H → C + D	1/2	0/0
Ex F → I x K	0/1	1/2
Ix K → E x F	1/0	2/1

Figura 2 b'')

	Transiciones 0 → 1 / 1 → 0	
A + B → C + D	2/2	4/3
A + B → G + H	2/2	2/3
C + D → A + B	2/2	3/4
C + D → G + H	2/2	3/1
G + H → A + B	2/2	3/2
G + H → C + D	2/2	1/3
Ex F → I x K	0/7	1/4
Ix K → E x F	7/0	4/1

Figura 2 c)

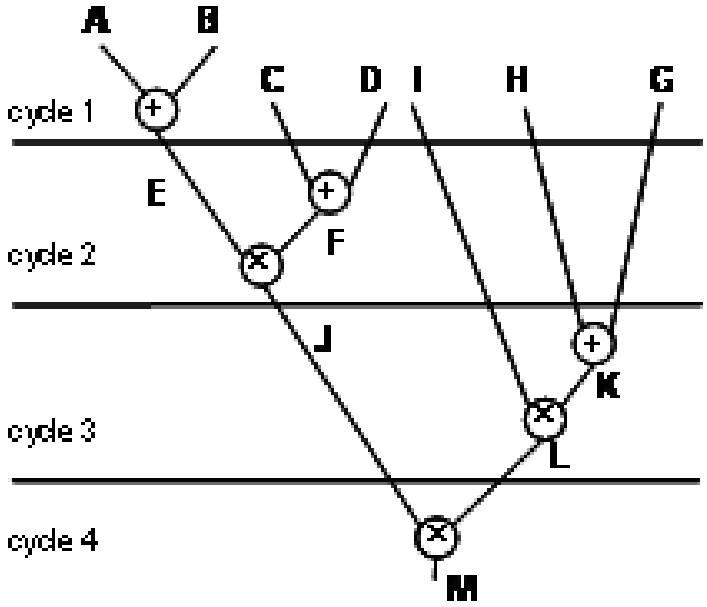
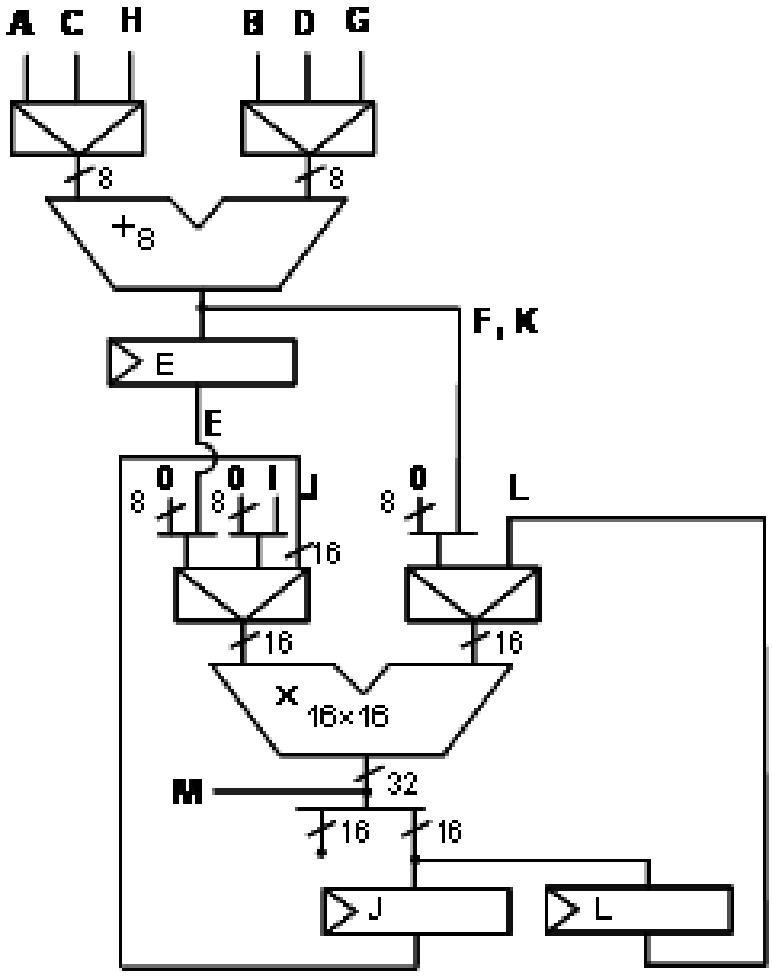


Figura 2 d)

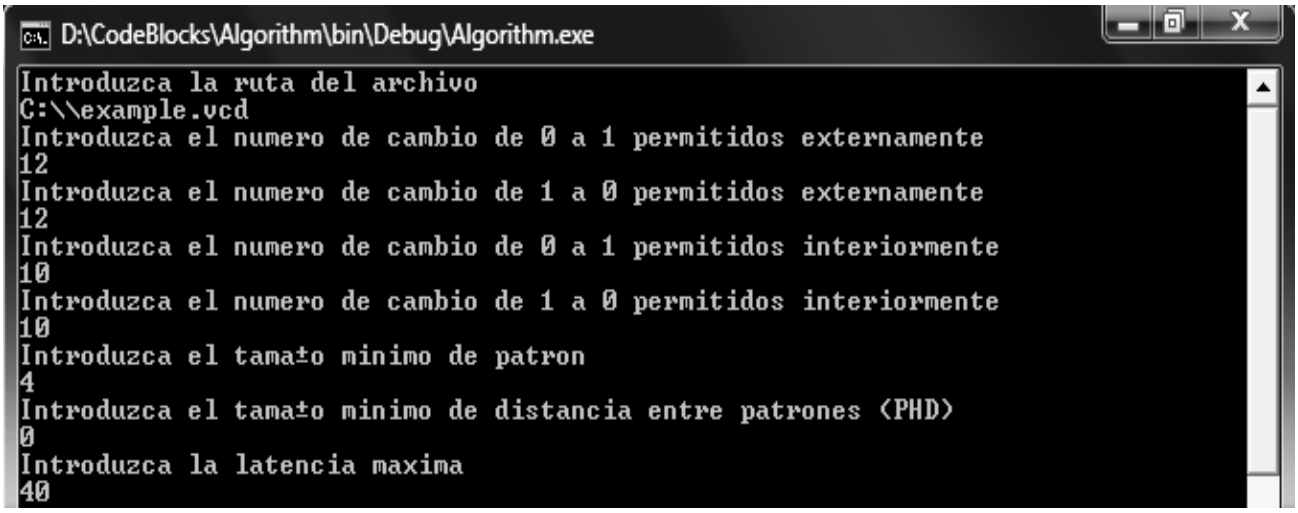


La primera columna muestra los pares de operaciones ejecutados en la misma unidad funcional en ciclos consecutivos y en la segunda columna vemos el número de transiciones.

Los algoritmos convencionales de reducción de potencia usan la información de actividad de conmutación para minimizar el número de conmutaciones en la planificación y la asignación.

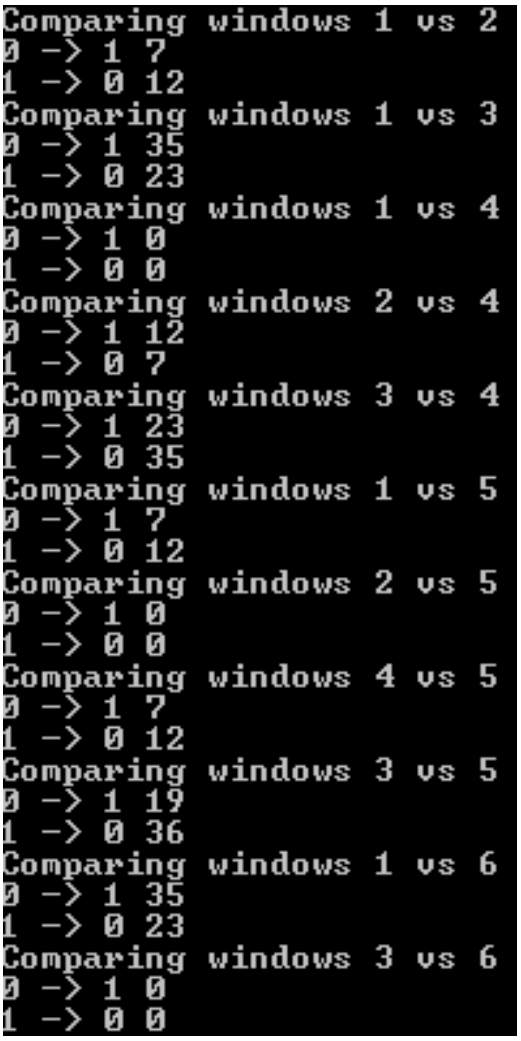
En la figura 2 c) podemos ver el diagrama final y en la figura 2 d) la ruta de datos. Esta solución ha sido obtenida mediante un algoritmo de programación lineal entera (ILP) basado en la actividad de las operaciones y considerando la propiedad conmutativa. Las conmutaciones no deseadas son evitadas latcheando los operandos de entrada de las unidades funcionales siempre y cuando no se requiera el cálculo de una nueva operación. Esta técnica se aplica también para mejorar la ruta de datos que describiremos después.

Iniciando la misma simulación pero ahora buscando una solución con nuestro algoritmo lo primero que tenemos que hacer es definir el valor de los parámetros como mostramos a continuación:



Necesitamos introducir un archivo .vcd con la simulación sobre la que queremos ejecutar el algoritmo, los cambios de cero a uno y de uno a cero máximos para incluir en una misma ruta de datos diferentes ventanas iniciales del programa, los cambios de cero a uno y de uno a cero máximos para no añadir a la ruta de datos una modificación, el tamaño mínimo del patrón a tener en cuenta, el valor de PHD para compartir una unidad funcional entre varios patrones, y la latencia máxima necesaria para la planificación de las operaciones.

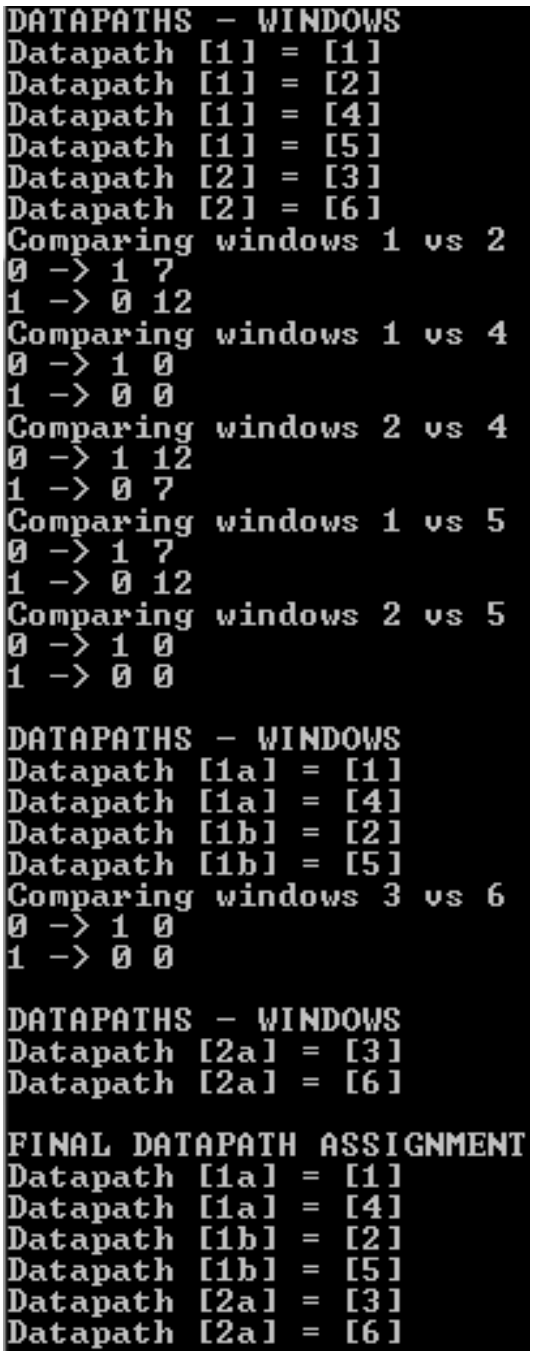
Una vez hemos dado valor a los parámetros y hemos cargado en el programa las operaciones y dependencias entre ellas, el algoritmo pasaría a fragmentar las iteraciones de los operandos de entrada (ventanas iniciales) entre las ventanas necesarias para cumplir los requisitos impuestos a través de los parámetros como se muestra a continuación.



Aquí vemos como el algoritmo va comparando las ventanas iniciales (una ventana inicial es una iteraciones de los valores de entrada, en nuestro ejemplo son seis ventanas iniciales), para dar lugar a las rutas de datos necesarias una por cada ventana final.

Finalmente como podemos comprobar se necesitan dos rutas de datos (ruta de datos 1 y ruta de datos 2) teniendo la primera una modificación que permitirá ejecutar ventanas iniciales que no se podrían ejecutar en esta ruta de datos sin dicha modificación cumpliendo los requisitos impuestos por los parámetros.

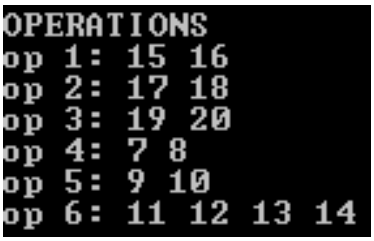
En la ruta de datos 1a (ruta de datos 1 sin modificación) se ejecutaran las ventanas iniciales 1 y 4, en la ruta de datos 1b (ruta de datos 1 con modificación) se ejecutaran las ventanas iniciales 2 y 5; y en la ruta de datos 2 (se ejecutaran las ventanas iniciales 3 y 6).



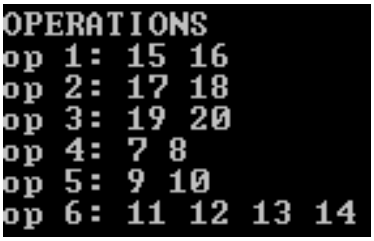
Como podemos ver en las imágenes que se muestran a continuación cada operación se divide en operaciones menores para así poder aprovechar la cercanía de los patrones menores encontrados dentro de las operaciones.

En esta ocasión coincide que en las planificaciones para las tres rutas de datos la descomposición de operaciones como se puede ver en las imagenes.

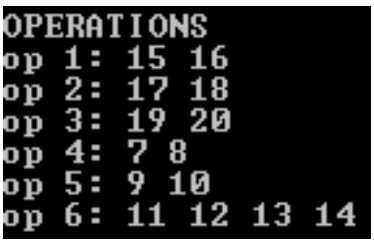
Ruta de datos 1 a



Ruta de datos 1 b



Ruta de datos 1 c



Para simplificar el ejemplo, dividimos el tiempo total de ejecución en 6 ventanas iniciales (una por cada iteración de los valores de entrada). Tras esto y debido a los parámetros el ejemplo resulta finalmente en tres ventanas finales correspondiéndose cada una con una ruta de datos. Aunque al estar la ventana final 1 y la ventana final 2 dentro de las restricciones que imponen los parámetros, la ruta de datos que ejecuta la ventana final 1 y la ruta de datos que ejecuta la ventana final 2 son la misma pero con una ligera modificación.

En la figura 2 e) mostramos los patrones extraídos en las simulaciones sobre la actividad de conmutación.

Figura 2 e)

Operación	**Patrónes**
Op1: A + B	AAAAAAAZ + AAAAAAZA
Op2: C + D	AAAAAAZZ + AAAAAZAA
Op3: G + H	AAAAAZAZ + AAAAAZZA
Op4: E x F	AAAAAAZZ x AAAAAZZZ
Op5: I x K	AAAAAZZZ x AAAAZAZZ
Op 6: J x L	AAAAAAAAAAAAZAZAZ x AAAAAAAAAZAAZZAZ
Op7: $E_{7..4}$ x $F_{7..4}$	AAAA x AAAA
Op8: $E_{3..0}$ x $F_{3..0}$	AAZZ x AZZZ
Op9: $I_{7..4}$ x $K_{7..4}$	AAAA x AAAA
Op10: $I_{3..0}$ x $K_{3..0}$	AZZZ x ZAZZ
Op11: $J_{15..12}$ x $L_{15..12}$	AAAA x AAAA
Op12: $J_{11..8}$ x $L_{11..8}$	AAAA x AAAA
Op13: $J_{7..4}$ x $L_{7..4}$	AAAZ x AZAA
Op14: $J_{3..0}$ x $L_{3..0}$	AZAZ x ZZAZ
Op15: $A_{7..4}$ + $B_{7..4}$	AAAA + AAAA
Op16: $A_{3..0}$ + $B_{3..0}$	AAAZ + AAZA
Op17: $C_{7..4}$ + $D_{7..4}$	AAAA + AAAA
Op18: $C_{3..0}$ + $D_{3..0}$	AAZZ + AZAA
Op19: $G_{7..4}$ + $H_{7..4}$	AAAA + AAAA
Op20: $G_{3..0}$ + $H_{3..0}$	AZAZ + AZZA

Esta nueva representación de la actividad de conmutación facilita la identificación de los patrones similares dentro de las operaciones de la especificación inicial.

Por ejemplo, las partes coloreadas en los patrones iníciales corresponden a los fragmentos de las operaciones que, ejecutándose sobre la misma unidad funcional, producen un menor número de conmutaciones (en este caso particular ninguna). Para poder explotar esta ventaja, estas sumas son fragmentadas en varias sumas menores y ejecutadas en unidades funcionales diferentes.

En el ejemplo podemos ver que si ejecutamos los bits más significativos (MSB) de A + B y los bits más significativos de C + D sobre la misma unidad funcional no tendríamos ninguna conmutación. Los bits menos significativos de A + B y de C + D se ejecutarían en otra unidad funcional que si tendría conmutaciones.

En la multiplicación el comportamiento es idéntico. Para poder encontrar los patrones de menor anchura se tiene en cuenta la actividad de conmutación interna.

En la figura 2 e) se muestran algunos patrones de ejecución internos de la multiplicación. La identificación de los patrones comunes dentro de las operaciones permite una mejor alineación (en términos de consumo dinámico) de los bits de los operandos que comparten las unidades funcionales.

En el ejemplo podemos ver que si ejecutamos los bits más significativos (MSB) de E x F y los bits más significativos de I x K sobre la misma unidad funcional no tendríamos ninguna conmutación. Los bits menos significativos de E x F y de I x K se ejecutarían en otra unidad funcional que si tendría conmutaciones.

Como se ve estas alineaciones solo pueden ser alcanzadas a través de la descomposición de algunas de las operaciones de la especificación.

La figuras 3 a) a') y a'') muestran la ruta de datos de bajo consumo resultantes.

Figura 3 a) Ruta de datos 1a

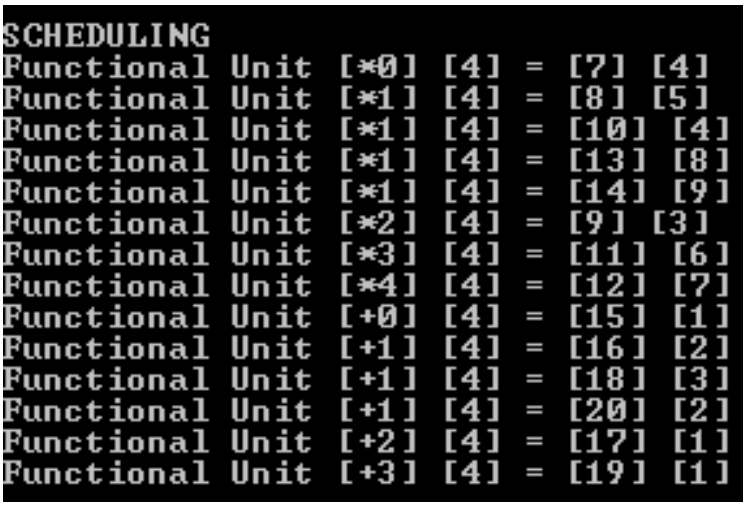


Figura 3 a') Ruta de datos 1 b)

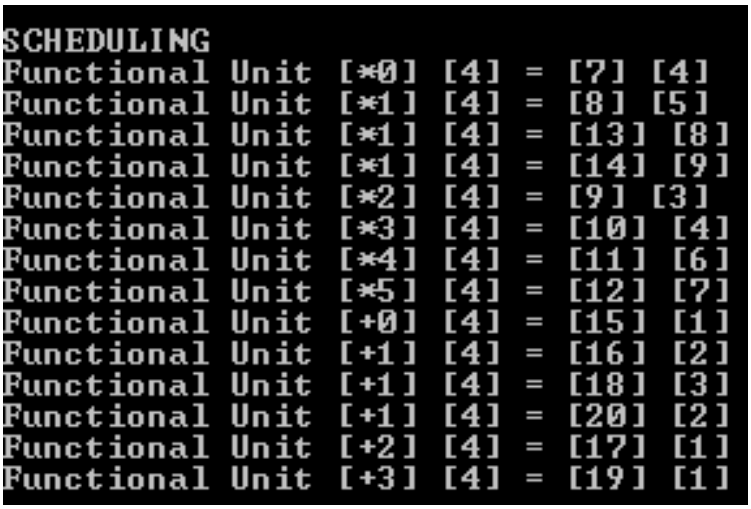
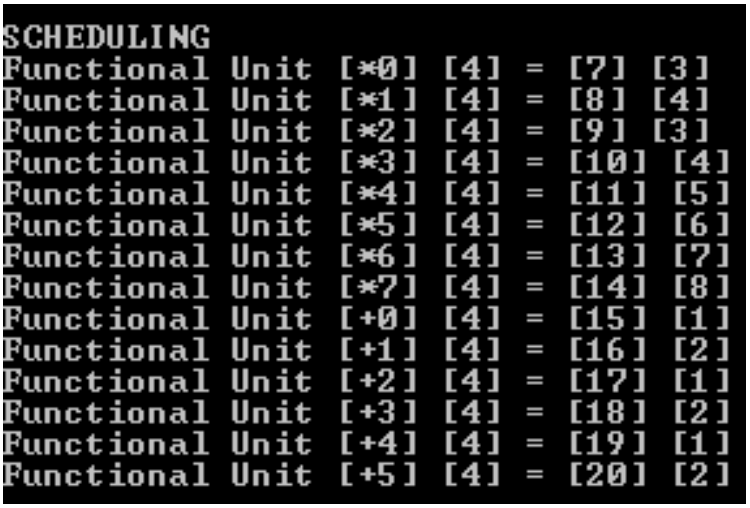


Figura 3 a'') Ruta de datos 2 a)



Hay que tener en cuenta que la descomposición, por ejemplo, de una multiplicación de 16 x 16 bits en 3 multiplicaciones y 2 sumas menores no incrementa el área ni el consumo estático.

También hay que tener en cuenta que los cables en la ruta de datos resultante tampoco complican la interconexión, ya que existían previamente, incluso de mayor medida, dentro de las unidades funcionales que componían la ruta de datos inicial.

El área de multiplexores es similar en ambos diseños, ya que el mayor número de unidades en la ruta de datos optimizada se compensa con su menor área.

Queda demostrado que la técnica propuesta ofrece una mayor flexibilidad. Con esta técnica se permite la ejecución de varios fragmentos de la misma operación en diferentes ciclos, e incluso no consecutivos, para así reducir las transiciones ocurridas en los recursos de la ruta de datos.

Las nuevas características de la implementación optimizada (operaciones parcialmente conmutadas, operaciones descompuestas y alineamientos de bits desordenados) no son posibles con los enfoques anteriores que consideran la actividad de conmutación a nivel de operación.

El estudio de la actividad de conmutación a nivel de subpalabra ha expandido el espacio de diseño explorado en el ejemplo, dando lugar a una reducción de aproximadamente el 80% en el número de transiciones de cero a uno, no solo en las unidades funcionales sino también en el enrutado y los recursos de almacenamiento.

El ahorro total de consumo alcanzado es del 40% tras una síntesis lógica con la herramienta “Synopsys Design Compiler” y la librería de diseño “TSMC 65 nm”. El área del circuito y el rendimiento permanecen prácticamente constantes.

ALGORITMO PROPUESTO

El algoritmo de síntesis de alto nivel que se propone para el ahorro de la potencia consumida lleva cabo la planificación y la asignación de las operaciones de la especificación tratando de minimizar el consumo dinámico de potencia.

Se explota el paralelismo a nivel de datos, es decir, las cadenas de bits en los operandos de entrada que muestran un menor número de conmutaciones a lo largo de la ejecución normal del circuito.

Se desarrolla una fragmentación guiada por patrones para identificar los patrones más comunes (MCP) entre las operaciones, y así asignarlos al mismo operador en ciclos consecutivos. Como las cadenas de bits más similares producen un menor número de conmutaciones, la actividad de conmutaciones en el circuito se ve reducida.

El algoritmo toma como entradas un archivo .vcd con la simulación sobre la que queremos ejecutar el algoritmo, los cambios de cero a uno y de uno a cero máximos para incluir en una misma ruta de datos diferentes ventanas iniciales del programa, los cambios de cero a uno y de uno a cero máximos para no añadir a la ruta de datos una modificación, el tamaño mínimo del patrón a tener en cuenta, el valor de PHD para compartir una unidad funcional entre varios patrones, y la latencia máxima necesaria para la planificación de las operaciones.

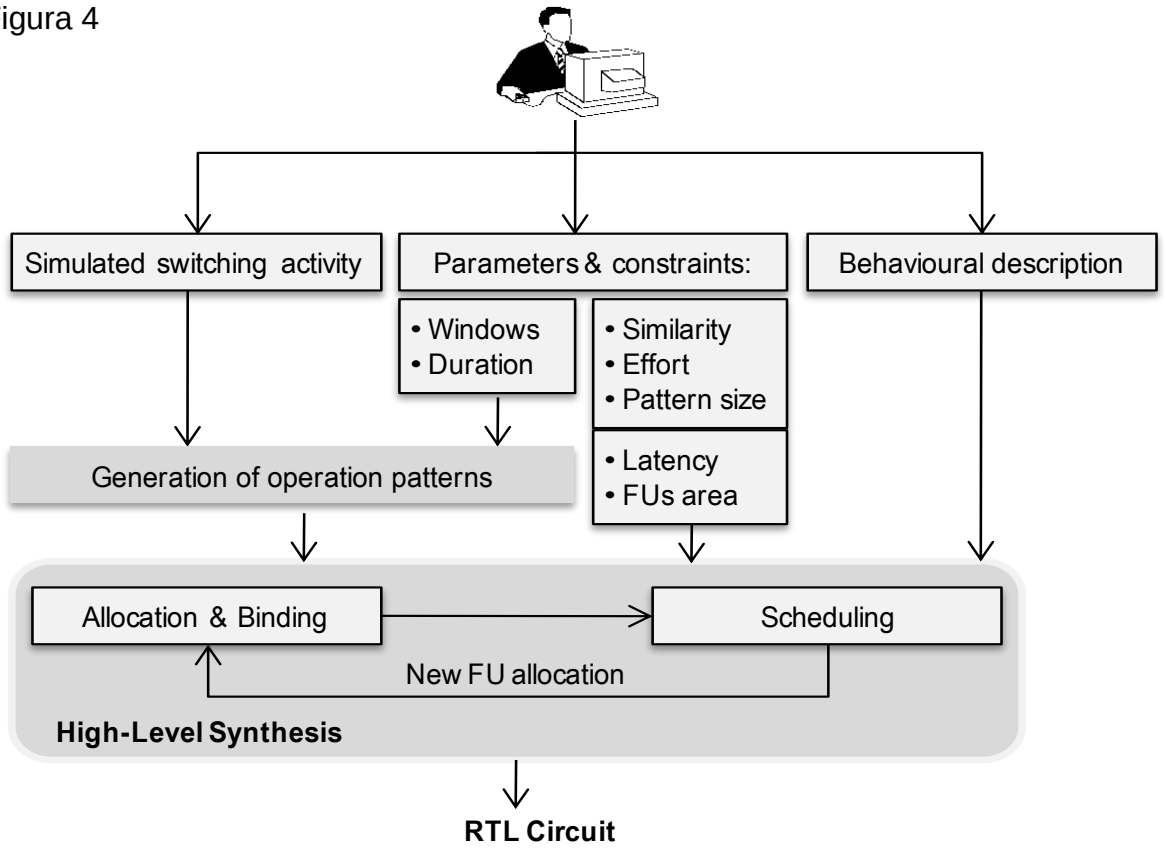
Además necesitan ser declaradas las dependencias entre las operaciones (ya que esta información no se encuentra en el archivo .vcd de la simulación).

La información recogida en los patrones de actividad de conmutación se usa para realizar la asignación de operaciones a unidades funcionales y después la planificación basándonos en la similitud de los patrones.

Durante este proceso nuestro algoritmo aplica algunas propiedades de las operaciones aritméticas a nivel de subpalabra, y explora diferentes alineaciones de las operaciones fragmentadas para así reducir el número de conmutaciones.

En la figura que se muestra a continuación (figura 4) se muestra un esquema del algoritmo detallando las fases principales.

Figura 4



Selección y asignación de recursos

Nuestro algoritmo intenta asignar el mínimo número de unidades funcionales para ejecutar todas las operaciones de la especificación, teniendo en cuenta la información de simulación recogida con los patrones de operación.

También, se asignan un conjunto de operaciones y fragmentos de operaciones a cada recurso funcional establecido.

Esta fase consiste en un bucle que es ejecutado hasta que no es posible identificar un patrón común entre las operaciones sin asignar.

En cada iteración se identifica el MCP (patrón más común) entre las operaciones no asignadas, y se ubica un conjunto de unidades funcionales para ejecutar las operaciones que lo contienen.

Finalmente, las operaciones no asignadas sin patrones comunes son asignadas a las unidades funcionales ubicadas siempre y cuando sea posible.

En caso contrario, se seleccionarían nuevas unidades funcionales para ejecutarlas.

Los principales pasos de esta fase pueden verse resumidos en el algoritmo 1.

1) Selección del patrón más común (MCP)

El primer paso en la fase de selección y asignación es la identificación del patrón más común (MCP) entre las operaciones no asignadas.

Esto se realiza considerando como similares aquellos patrones cuya PHD (distancia de Hamming entre patrones) es menor que el parámetro de similitud definido por el diseñador.

Este parámetro se usa para poder llegar a un compromiso entre la potencia y el área, permitiendo que las operaciones con patrones más cercanos compartan las mismas unidades funcionales.

Para poder minimizar el número total de conmutaciones, diferentes alineaciones de operaciones con el mismo tipo son también evaluadas.

Hay que tener en cuenta que el número de posibles alineaciones crece de forma exponencial con el número de operaciones.

Además el diseñador dispone de otro parámetro, el tamaño mínimo de patrón, para así evitar la fragmentación excesiva de operaciones y unidades funcionales.

2) Selección de unidades funcionales y asignación de operaciones

Una vez se ha seleccionado el mejor patrón, un nuevo conjunto de unidades funcionales del mismo tipo y anchura del patrón son seleccionados de la biblioteca de diseño.

El número de unidades funcionales depende del número de operaciones y fragmentos que contienen el patrón seleccionado y la latencia del circuito (λ).

#FU(pq) = $FU(pa) = ||OP\ pa||/\lambda$

Siendo FU(pa) el número de unidades funcionales correspondientes al patrón pa y OP el conjunto de operaciones que contienen el patrón pa.

El conjunto de operaciones y fragmentos de operaciones que contienen el patrón seleccionado son asignados a las nuevas unidades funcionales.

Hay que tener en cuenta que se permite asignar operaciones con patrones similares, basándonos en el valor del parámetro de similitud, a la misma unidad funcional.

La distribución de operaciones entre las unidades funcionales tiene por objeto minimizar la actividad de conmutaciones. El PHD entre las operaciones seleccionadas se tiene en cuenta para fijar el orden de precedencia entre ellas, respetando sus posibles dependencias de datos.

En este paso algunas operaciones se fragmentan para extraer el MCP calculado antes. Los fragmentos no asignados de estas operaciones se añaden como nuevas operaciones al conjunto de operaciones sin asignar. Sus dependencias de datos, incluidas las propagaciones de acarreos, son tenidas en cuenta durante la asignación y la planificación.

A veces resulta imposible continuar extrayendo patrones comunes entre las operaciones sin asignar, entonces, un último paso intenta asignar dichas operaciones a las unidades funcionales existentes.

Estas últimas asignaciones solo se llevan a cabo si no se pierden los beneficios de consumo alcanzados anteriormente, es decir, si las nuevas operaciones pueden ser planificadas antes o después del conjunto de operaciones con patrones similares. En caso contrario se seleccionan nuevas unidades funcionales para ejecutar las operaciones no. Se debe de tener en cuenta que el valor del parámetro de similitud puede ser ajustado para evitar un alto número de operaciones en este paso.

Algoritmo 1

```
AllocationAndBinding(std::map<std::string, std::map<int, std::map<std::string, std::vector<operation>
> > >& FU,
std::map<int, operation>& operations, int dependences[100][100],
int min_pattern, int latency, int effort, int min_PHD,
std::map<std::string, std::pair<std::string, std::string> > id_var_pos, int weight)
{
    int num_fu = 0;
    pattern mcp;
    int mcp_occurrences = 0;
    std::vector<pattern> patterns;
    std::map<int, std::vector<int> > new_ops;
    std::map<int, operation> unbound_operations;
    std::map<int, operation>::iterator it_op;
    unbound_operations = operations;
    do
    {
        patterns.clear();
        mcp_occurrences = 0;
        for(it_op = unbound_operations.begin();
            it_op != unbound_operations.end(); it_op++)
        {
            Patterns(it_op->second, min_pattern, patterns);
            for(unsigned j = 0; j < patterns.size(); j++)
            {
                if(!patterns[j].second.first)
                {
                    FindPattern(patterns[j], unbound_operations, effort);
                    if(patterns[j].second.first > mcp_occurrences)
                    {
                        mcp_occurrences = patterns[j].second.first;
                        mcp = patterns[j];
                    }
                }
            }
```

```
        }
        if(patterns.size())
        {
          Allocate(FU, mcp.second.first / latency, mcp.first.second,   strlen(mcp.first.first.first.c_str()));
          Fragment(mcp.second.second, unbound_operations, operations,  dependences, new_ops,
                    mcp, id_var_pos);
          Bind(mcp.second.second, new_ops, FU, unbound_operations, strlen(mcp.first.first.first.c_str()),
                min_PHD, weight);
        }
    }
    while(patterns.size() != 0);
    if(unbound_operations.size() != 0)
      Arrange(FU, unbound_operations, weight);
}
```

---

map<string, map<int, map<string, vector<operation> > > > FU → estructura para almacenar la información relativa a las unidades funcionales resultantes, que consta de tipo de operación que realiza la unidad funcional, longitud de los operandos que acepta la unidad funcional, identificador univoco de la unidad funcional y vector de operaciones que se ejecutan en dicha unidad funcional.

map<int, operation> operations → mapa de operaciones

int dependences[100][100] → matriz de dependencias, tal que si [i][j] = 1 la operación i depende de la operación j.

int min_pattern → tamaño minimo del patrón

int latency → latencia máxima

int effort → esfuerzo

int min_PHD → PHD tope para ejecutar operaciones dentro de la misma unidad funcional.

map<string, pair<string, string> > id_var_pos → estructura que relaciona cada identificador dado por el fichero de simulación a una posición de un operando con dicho operando y la posición que el identificador representa.

int weight → peso de la ruta de datos (numero de ventanas inciales de que consta)

int num_fu → numero de unidades funcionales

pattern mcp → patrón que almacenara el patrón mas común en cada momento

int mcp_occurrences → numero de ocurrencias del patrón mas común.

vector<pattern> patterns → vector de patrones.

map<int, vector<int> > new_ops → mapa que almacena para cada operación las operaciones que representan los fragmentos en que se ha descompuesto.

map<int, operation> unbound_operations → operaciones sin asignar (no tienen unidad functional asignada en la que ejecutarse).

map<int, operation>iterator it_op → iterador para recorrer las distintas operaciones.

Patterns(it_op->second, min_pattern, patterns);

FindPattern(patterns[j], unbound_operations, effort);

Allocate(…) → añade las unidades funcionales necesarias para ejecutar las operaciones con el patrón mas común.

Fragment(…) → fragmenta las operaciones en operaciones mas pequeñas para buscar patrones comunes.

Bind(…) → asigna las unidades funcionales a las diferentes operaciones que contienen el patrón mas común

Arrange(…) →asigna las operaciones sin asignar a las unidades funcionales existentes o crea nuevas unidades funcionales en caso de que fuese necesario.

Planificación

Una vez llegado este punto, todas las operaciones han sido asignadas a unidades funcionales. Para planificar el conjunto de operaciones asignado a cada unidad funcional hemos diseñado un algoritmo de planificación basado en listas.

Guardamos una lista por cada unidad funcional con todas las operaciones asignadas a esta unidad funcional.

Las operaciones son planificadas en base a su movilidad una a una.

La movilidad se calcula usando las planificaciones ASAP (as soon as possible) y ALAP (as large as possible) para el valor de latencia definido por el diseñador.

En cada iteración, se planifican todas las operaciones sin planificar que tengan todos sus predecesores planificados teniendo en cuenta su movilidad.

Como las dependencias de datos entre operaciones no se consideraron en la fase de selección y asignación, al planificar las operaciones hay que verificar que tengan sus dependencias de datos resueltas.

Como consecuencia de las decisiones de diseño tomadas y las dependencias de datos entre operaciones algunas unidades funcionales pueden permanecer ociosas durante varios ciclos.

Para poder evitar conmutaciones innecesarias cuando no están calculando ninguna operación sus operandos de entrada son asignados a los valores anteriores.

El algoritmo 2 resume los pasos principales durante la planificación.

Algoritmo 2

```
void
Scheduling(std::map<std::string, std::map<int, std::map<std::string, std::vector<operation> > > >&
FU, std::map<int, operation>& operations, int dependences[100][100], int min_PHD, int max)
{
   int pos;
   std::string selected_fu;

   std::map<int, operation> asap_ops;
   std::map<int, operation> alap_ops;
   std::map<int, operation> unscheduled_operations;

   std::map<int, operation>::iterator it_op;
   std::vector<operation>::iterator it_vector;
   std::map<std::string, std::vector<operation> >::iterator it_fu;
   std::map<int, std::map<std::string, std::vector<operation> > >::iterator it_fu_size;
   std::map<std::string, std::map<int, std::map<std::string, std::vector<operation> > > >::iterator
it_fu_type;
```

```
asap_ops = operations;
alap_ops = operations;
unscheduled_operations = operations;

Asap(asap_ops, dependences);
Alap(alap_ops, dependences, max);

while(unscheduled_operations.size())
{
   for(it_op = operations.begin(); it_op != operations.end(); it_op++)
   {
      if(unscheduled_operations.find(it_op->first) != unscheduled_operations.end())
      {
         for(unsigned int i = asap_ops[it_op->first].first.second;
               i <= alap_ops[it_op->first].first.second; i++)
         {
            pos = 0;
            selected_fu = "";
            if(AllNodesScheduled(Predecessors(it_op->first, dependences), operations, i, 'p') &&
               EmptyCycle(FU[it_op->second.second.first][it_op->second.second.second.first.size()],
                           it_op->first, i, selected_fu, pos))
            {
               it_op->second.first.second = i;
               unscheduled_operations.erase(it_op->first);
               FU[it_op->second.second.first][it_op-
>second.second.second.first.size()][selected_fu].at(pos).first.second = i;
               break;
            }
         }
      }
   }
}
```

```
    for(it_fu_type = FU.begin(); it_fu_type != FU.end(); it_fu_type++)
    {
        for(it_fu_size = it_fu_type->second.begin(); it_fu_size != it_fu_type->second.end(); it_fu_size++)
        {
            for(it_fu = it_fu_size->second.begin(); it_fu != it_fu_size->second.end(); it_fu++)
            {
                for(it_vector = it_fu->second.begin(); it_vector != it_fu->second.end(); it_vector++)
                {
                    it_vector->first.second = operations[it_vector->first.first].first.second;
                }
            }
        }
    }
}
```

---

map<string, map<int, map<string, vector<operation> > > > FU → estructura para almacenar la información relativa a las unidades funcionales resultantes, que consta de tipo de operación que realiza la unidad funcional, longitud de los operandos que acepta la unidad funcional, identificador univoco de la unidad funcional y vector de operaciones que se ejecutan en dicha unidad funcional.

map<int, operation> operations → mapa de operaciones

int dependences[100][100] → matriz de dependencias, tal que si [i][j] = 1 la operación i depende de la operación j.

int min_PHD → PHD tope para ejecutar operaciones dentro de la misma unidad funcional.

int max → latencia maxima.

map<int, operation> asap_ops → mapa de operaciones planificadas segun su Asap.

map<int, operation> alap_ops → mapa de operaciones planificadas segun su Alap.

map<int, operation> unscheduled_operations → operaciones sin planificar.

map<int, operation>::iterator it_op → iterador para recorrer las operaciones.

vector<operation>::iterator it_vector → iterador para recorrer un vector de operaciones.

map<string, vector<operation> >::iterator it_fu → iterador para recorrer unidades funcionales.

map<int, std::map<std::string, std::vector<operation> > >::iterator it_fu_size → iterador para recorrer una estructura que contiene unidades funcionales ordenadas por la longitude de sus operandos.

map<string, map<int, map<string, vector<operation> > > >::iterator it_fu_type → iterador para recorrer una estructura que contiene unidades funcionales ordenadas por el tipo de operacion que ejecutan (+, *, . . .).

Asap(. . .) → planifica las operaciones segun la matriz de dependencias basandose en la planificacion Asap.

Alap(. . .) → planifica las operaciones segun la matriz de dependencias basandose en la planificacion Alap.

AllNodesScheduled(…) → comprueba que un conjunto de operaciones este planificado en un ciclo dado.

EmptyCycle(…) → comprueba la disponibilidad de una unidad functional en un ciclo dado.

En las figuras 5 a) a') a'') se muestran la planificación y asignaciones obtenidas del ejemplo anterior.

Figura 5 a)

```
SCHEDULING
Functional Unit [*0] [4] = [7] [4]
Functional Unit [*1] [4] = [8] [5]
Functional Unit [*1] [4] = [10] [4]
Functional Unit [*1] [4] = [13] [8]
Functional Unit [*1] [4] = [14] [9]
Functional Unit [*2] [4] = [9] [3]
Functional Unit [*3] [4] = [11] [6]
Functional Unit [*4] [4] = [12] [7]
Functional Unit [+0] [4] = [15] [1]
Functional Unit [+1] [4] = [16] [2]
Functional Unit [+1] [4] = [18] [3]
Functional Unit [+1] [4] = [20] [2]
Functional Unit [+2] [4] = [17] [1]
Functional Unit [+3] [4] = [19] [1]
```

Figura 5 a')

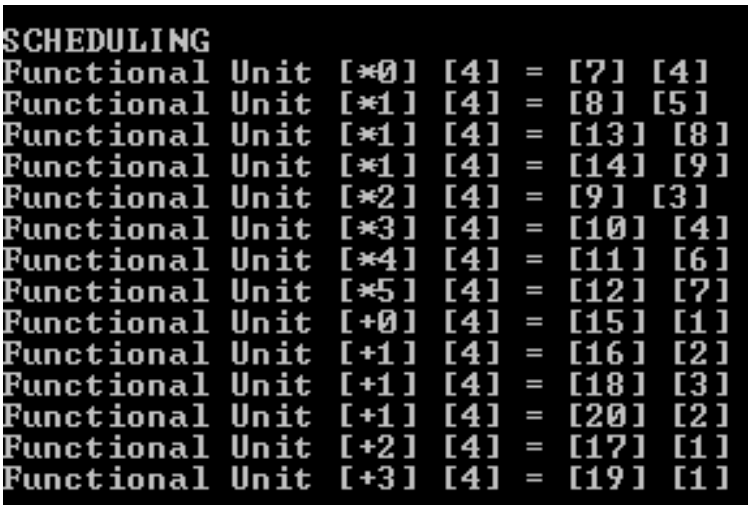
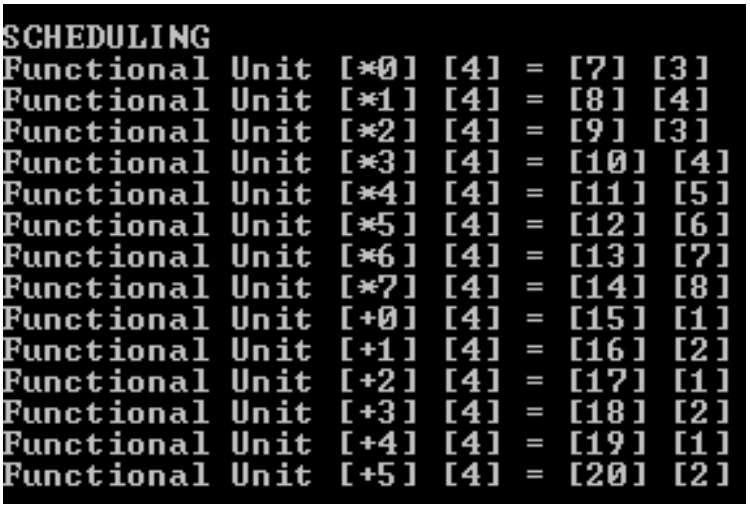


Figura 5 a'')



Como podemos ver en la planificación de la ultima ruta de datos los bits más significativos de la suma A + B se ejecutan en la unidad funcional +1 en el ciclo 2 mientras que los bits menos significativos de la suma A + B se ejecutan en la unidad funcional +0 en el ciclo 1.

Esta compartición especial de unidades funcionales entre fragmentos de diferentes operaciones no podría alcanzarse sin una planificación y asignación a nivel de subpalabra, como las presentadas en esta memoria.

La implementación sintetizada implica una reducción del 40% por ciento en la actividad de conmutación de las unidades funcionales.

Además se reduce el número de unidades funcionales en un 25% en comparación con la implementación convencional de reducción de potencia (compuesta por sumadores de 8 bits) y la longitud del ciclo permanece prácticamente constante (los retardos de dos sumadores de 4 bits encadenados y de un sumador de 8 bits son prácticamente idénticos).

RESULTADOS EXPERIMENTALES

Para poder medir la calidad del algoritmo propuesto se han desarrollado diferentes experimentos basados en la síntesis de algunos benchmarks clásicos de síntesis de alto nivel y de los decodificadores ADPCM y JPEG.

Los resultados obtenidos han sido comparados con las implementaciones sintetizadas por un algoritmo de ahorro de potencia [23] y de una formulación basada en programación lineal entera (ILP) basada en la actividad de operaciones.

La propuesta de ILP calcula el conjunto de recursos funcionales óptimos, es decir, la asignación con el mínimo número de conmutaciones en las unidades funcionales para un valor máximo de área.

El algoritmo propuesto en [23] usa una técnica de look-ahead con backtracking para reducir la actividad de conmutación.

En el algoritmo se consideran algunas propiedades de las operaciones aritméticas así como la similitud entre los valores de los operandos de entrada (obtenidas de simulaciones anteriores del comportamiento), la replicación de unidades funcionales, y la deshabilitación de unidades funcionales durante los ciclos ociosos.

Experimentos

Cada experimento de compone de los siguientes pasos:

1) Simulación de la descripción del comportamiento para obtener la información de la actividad de conmutación usando ModelSim. Las simulaciones duraron un total de 2 horas.

2) Síntesis de alto nivel de la especificación usando las técnicas de bajo consumo propuestas en [23] y la formulación ILP. Las constantes usadas para el enfoque ILP han sido fijadas a las descritas en [23]. Hay que tener en cuenta en este caso que el conjunto de unidades funcionales seleccionadas por nuestro algoritmo no podrían ser usadas como constantes, ya que las fragmentaciones realizadas dan lugar a

unidades funcionales de menor tamaño lo que hace imposible la ejecución de los comportamientos dados a nivel de operación.

3) Procesamiento de la información de la actividad de conmutación, de acuerdo con el número de ventanas de simulación y sus duraciones correspondientes, para así obtener los patrones de actividad de conmutación. En todos los experimentos el tiempo de simulación ha sido dividido en cuatro ventanas de igual duración (30 minutos cada una). Para simulaciones mayores, seria conveniente un mayor número de ventanas.

4) Síntesis de alto nivel de la especificación del comportamiento usando nuestra propuesta.

5) Síntesis lógica de los circuitos obtenidos en 2) y 4) usando la herramienta Synopsys Design Compiler y la librería de diseño TSMC 65.

6) Los reportes obtenidos de la síntesis se han usado para obtener el consumo de potencia, el tiempo de ejecución, y el área en cada diseño. El consumo de potencia dinámica se ha medido tras la simulación de los circuitos sintetizados usando un modo normal de operación (los ejemplos seleccionados son diferentes de los usados para capturar la información de actividad de conmutación).

Rendimiento de la metodología de ahorro de potencia propuesta

El rendimiento de nuestra metodología de diseño de ahorro de potencia es sensible a los datos usados para la simulación, al igual que otros enfoques basados en la actividad de conmutación.

Además de esto, también depende de la precisión elegida para representar la actividad de conmutación, es decir, del número y longitud de las ventanas de simulación fijadas por el diseñador a través de los parámetros de entrada.

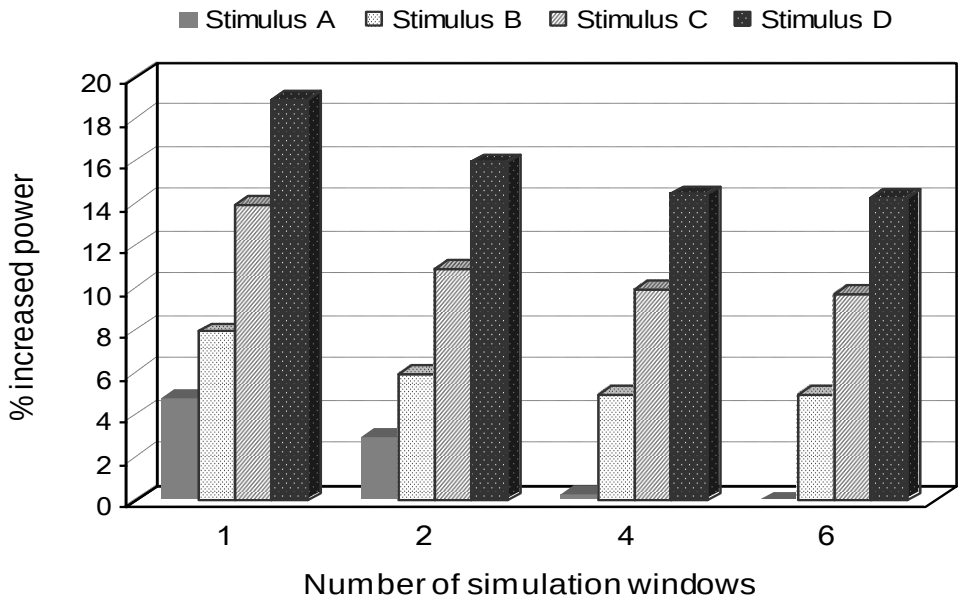
Para poder mostrar ambas dependencias hemos sintetizado un decodificador JPEG para un conjunto de imágenes como estímulo.

El proceso de síntesis se ha desarrollado en cuatro escenarios: 1, 2, 4 y 6 ventanas de simulación de igual longitud. Los valores de los parámetros del algoritmo han sido seleccionados considerando el tamaño de palabra y el número de ventanas de simulación.

Los parámetros similitud y mínimo tamaño de patrón son establecidos a tamaño de palabra * número de ventanas * 5 y tamaño de palabra * número de ventanas / 4 respectivamente. Estos valores permiten una separación máxima de 5 caracteres entre patrones considerados similares, así como la descomposición de operaciones en fragmentos cuya anchura mínima sea la longitud de palabra / 4. El consumo de potencia de los circuitos sintetizados ha sido medido después de la simulación de las implementaciones lógicas con cuatro conjuntos diferentes de imágenes de referencia (A, B, C y D). El conjunto de estímulos A es el usado para extraer la información de actividad de conmutación, siendo los restantes conjuntos B, C y D ordenados por orden decreciente de similitud con A.

La figura 6 muestra el porcentaje de incremento de consumo de potencia en función del estimulo dinámico usado. El circuito de referencia ha sido el simulado con el conjunto de estímulos A y seis ventanas de simulación.

Como se esperaba, el consumo de potencia crece con la diferencia entre los estímulos usados como perfil y aquellos usados para simular el circuito sintetizado.



Se puede ver también en la gráfica que el número de ventanas de simulación afecta a los consumos resultantes.

A medida que la precisión de la representación de la actividad de conmutación aumenta el consumo de potencia se ve reducido.

Sin embargo, esta reducción alcanza un límite superior (4 ventanas en el experimento). Por encima de este límite no hay un mayor ahorro de potencia debido al incremento de la precisión de la representación de la actividad de conmutación.

Benchmarks clásicos de síntesis de alto nivel

Los benchmarks clásicos incluyen el solucionador de ecuaciones diferenciales (DiffEq), el filtro de ondas elíptico de orden 5 (EWF), la transformación discreta del coseno (DCT), y el filtro adaptativo LMS (LMS).

Todos los benchmarks han sido sintetizados para diferentes tamaños de palabra desde 8 a 32 bits. Los valores de los parámetros han sido asignados tal y como se explicó en la sección anterior.

La tabla 1 muestra la latencia del circuito, el número de conmutaciones ocurridas en las unidades funcionales de la ruta de datos, la potencia total consumida, y el área de los circuitos sintetizados.

Tabla 1

Benchmark (word length)	Latency			Commutations in FUs			Power Consumption (uW)			Area (equivalent gates)		
	[23]	ILP	Ours	[23]	ILP	Ours	[23]	ILP	Ours	[23]	ILP	Ours
DiffEq (16)	5	5	5	1022	982	780	1205	849	672	2304	2185	2085
DiffEq (32)	5	5	5	2150	1921	1744	2854	1622	1461	4877	4669	4285
EWF (16)	17	16	16	5447	3485	3361	2210	1445	1218	3840	3831	3607
EWF (32)	17	16	16	6098	4259	4152	1645	996	870	7056	6833	6412
DCT (8)	12	12	12	3915	3384	3210	787	586	577	1214	1207	1156
Lattice (16)	10	9	9	2326	2214	2028	581	458	455	1079	1066	1021
Lattice (32)	10	9	9	4776	3993	3837	2089	1729	1554	1894	1779	1803
IDCT (8)	12	10	10	3986	3768	3494	878	680	664	1316	1301	1258
LMS (16)	11	11	12	3127	2807	2384	757	581	488	1782	1771	1670
LMS (32)	11	11	12	7515	6651	5038	2193	1831	1347	3802	3604	3482

El consumo de potencia se ve reducido en una media del 35% en comparación con [23] y cerca del 12 % comparándolo con la formulación ILP.

El ahorro total de potencia es mayor que la reducción del porcentaje de conmutaciones dada en las unidades funcionales.

La razón es que las conmutaciones en otras unidades de la ruta de datos (enrutado y almacenamiento) también se ven decrementadas, y el área y el tiempo de ejecución alcanzado en la mayoría de los casos ha contribuido también a la reducción del consumo total de potencia.

Estas reducciones adicionales son la consecuencia de haber fragmentado las operaciones, lo cual ha resultado en una menor longitud de los ciclos y unidades funcionales de menor anchura (circuitos menores en todos los casos).

Síntesis del decodificador ADPCM

También hemos implementado varios módulos del decodificador ADPCM, siguiendo el estándar G.711 de la ITU.

El conjunto de módulos sintetizados son Quantizador Adaptativo Inverso (IAQ), Factor de Quantizador de escala (QSF), Conversión de Salida en formato PCM (OPFC), Ajuste de Codificación Síncrona (SCA), y Detector de Tono y Transición (TTD). Los módulos OPFC y SCA han sido sintetizados juntos debido a su proximidad en el decodificador, y DFG, y IAQ, QSF, y TTD de forma independiente.

El rango de operaciones de los módulos sintetizados está entre 20 y 58 operaciones.

La media del número de multiplicaciones es del 24% y las operaciones restantes son principalmente sumas.

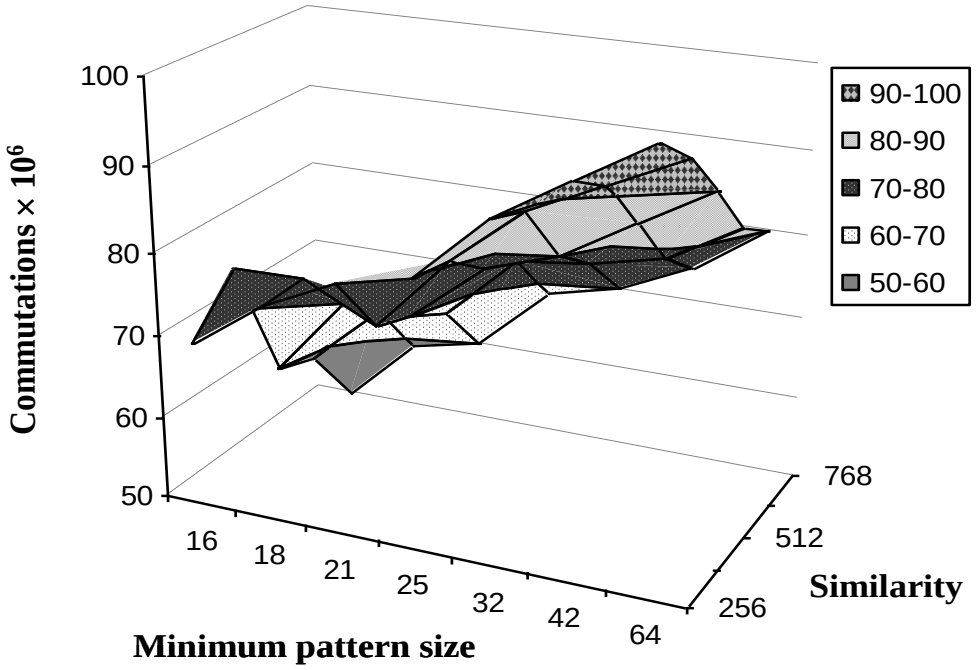
Las representaciones de datos se encuentran en complemento a dos, magnitud y signo, y binario puro para diferentes anchuras de palabra entre 4 y 16 bits.

Los valores reales de los operandos de entrada han sido extraídos de varios ejemplos.

Estos módulos han sido sintetizados con diferentes valores para los parámetros de nuestro algoritmo. El patrón de similitud ha tomado unos valores entre longitud de palabra * 2 y longitud de palabra * 6, y el tamaño mínimo de patrón

entre longitud de palabra /2 y longitud de palabra / 8, para una longitud de operación de 128 bits.

Figura 7



La figura 7 muestra el número de conmutaciones en el modulo OPFC para diferentes valores de los parámetros.

Como se muestra en el gráfico, el número de conmutaciones puede ser reducido de forma significativa ajustando adecuadamente los valores de los parámetros.

En el ejemplo, se muestra el mínimo número de conmutaciones para un valor del parámetro de similitud igual a 768 y un tamaño mínimo de patrón de 16.

El tamaño de palabra en el circuito y el número de ventanas de simulación debe ser considerado para poder seleccionar los valores apropiados de los parámetros, ya que a través de ellos se define el tamaño de patrón.

La tabla 2 muestra la potencia consumida por los módulos sintetizados, tras la síntesis del comportamiento usando el algoritmo de baja potencia [23], la formulación ILP, y nuestro enfoque.

Los resultados de la síntesis muestran una importante reducción en el consumo de potencia, alrededor del 21% y el 11% de media comparándose con [23] e ILP respectivamente.

También se muestran decrementos menores en el área del circuito, y los tiempos de ejecución son bastante similares en los tres casos.

Los beneficios alcanzados son principalmente debidos a la implementación de un conjunto de agresivas técnicas de diseño que explotan la información de la actividad de conmutación a nivel de subpalabra.

Teniendo en cuenta que tanto la formulación ILP como [23] se centran en minimizar la actividad de conmutación a nivel de operación, los beneficios obtenidos por nuestro algoritmo deberán ser similares en el caso de compararlos con los de otros algoritmos de ahorro de potencia que trabajen a nivel de operación.

Tabla 2

Module	Consumption (uW)			Time (ns)			Area (gates)			Algorithm execution time		
	[23]	ILP	Ours	[23]	ILP	Ours	[23]	ILP	Ours	[23]	ILP	Ours
IAQ	482,76	379,15	271,56	210	208	220	1284	1264	1216	12''	12h 50'	23''
QSF	765,43	702,25	604,23	416	406	448	2686	2578	2384	14''	16h 45'	36''
TTD	645,28	598,6	589,5	311	298	306	1984	1792	1733	10''	13h 12'	21''
OPFC + SCA	807,12	748,05	739,04	576	514	547	1846	1813	1715	16''	13h 27'	34''

El tiempo de ejecución de nuestro algoritmo es mayor que el de [23], debido al coste computacional de considerar la actividad de conmutación a nivel de subpalabra y a la descomposición de las operaciones.

Sin embargo, los tiempos de ejecución presentados son bastante razonables contando con una selección apropiada de los valores de los parámetros del algoritmo que limite adecuadamente el espacio de diseño. La formulación ILP no resulta nada práctica debido a su tiempo de ejecución.

CONCLUSIONES

Esta memoria presenta un novedoso algoritmo de síntesis de alto nivel de ahorro de potencia que realiza la planificación y asignación de las especificaciones de comportamiento para reducir el consumo de potencia dinámica.

Se considera la información de actividad de conmutación a nivel de subpalabra, representada por patrones que capturan los valores de los operandos de entrada a lo largo de diferentes ventanas de simulación.

El algoritmo propuesto maneja operaciones basándose en una base de correspondencia de patrones, permitiendo la aplicación parcial de las propiedades de las operaciones aritméticas, la ejecución distribuida de operaciones sobre diferentes unidades funcionales, y la exploración de diferentes alineaciones de los subsiguientes fragmentos de operaciones.

Estas características expanden el espacio de diseño explorado y permiten un ámbito mucho mayor para la síntesis flexible que el que puede ser alcanzado con enfoques anteriores, llevando a mejoras significativas en el consumo de potencia.

La metodología propuesta de ahorro de potencia en esta memoria para los diseños ASIC también se podría aplicar al diseño con placas FPGA. En este caso, la fragmentación de operaciones debe de llevarse a cabo teniendo en cuenta, no solo la similitud de los patrones de actividad de conmutación, sino también las características de las unidades funcionales embebidas en las placas.

También, nuestro algoritmo puede ser usado en combinación con otros enfoques centrados en optimizar el consumo estático de potencia.

BIBLIOGRAFIA

[1] L. Shang, R.P. Dick N.K. Jha. “High-Level Synthesis Algorithms for Power and Temperature Minimization”. High-Level Synthesis: from Algorithm to Digital Circuit. P. Coussy and A. Morawiec Editors, 2008 Springer.

[2] M. Hartman, J. Taylor. “Solve Leakage and Dynamic Power Loss”. Mobile Handset DesignLine, Julio 2008.

[3] A. Khanna, S. McCloud. “Power Exploration in High-Level Synthesis”. FPGA and Structured ASIC Journal, Diciembre 2006.

[4] A. Stammermann, L. Kruse, W. Nebel, A. Pratsch, E. Schmidt, M.Schulte, A.Schulz. “System Level Optimization and Design Space Exploration for Low Power”. International Symposium on System Synthesis (ISSS), 2001.

[5] X. Tang, H. Zhou, P. Banerjee. “Leakage Power Optimization With Dual-Vth Library In High-Level Synthesis”. Design Automation Conference (DAC), 2005.

[6] K.S.Khouri, J.K.Niraj. “Leakage power analysis and reduction during behavioural synthesis”. IEEE Transactions on Very Large Scale Integration (VLSI) Systems. Vol. 10(6): 876-885, 2002.

[7] H.I.Chen, E.K. Loo, J.B. Kuol, M.J. Syrzycki. “Triple-Threshold Static Power Minimization in High-Level Synthesis of VLSI CMOS”. Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation (PATMOS), 2007.

[8] H. Yang, and L. Dung. “On multiple.voltage high-level synthesis using algorithmic transformations”. Asia and South Pacific Design Automation Conference (ASP-DAC), 2005.

[9] 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.

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

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

[12] M . A. Ochoa-Montiel, B. M. Al-Hashimi, P.Kollig. “Exploiting Power-Area Tradeoffs in Behavioural Synthesis through Clock and Operations Throughput Selection”. Asia and South Pacific Design Automation Conference (ASP-DAC), 2007.

[13] R. Mukherjee, S. Ogrenci, G. Memik. “Peak Temperature Control and Leakage Reduction During Binding in High Level Synthesis”. International Symposium on Low Power Electronics and Design (ISLPED), 2005.

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

[15] Y. Zhang, X. S. Hu, and D.Z. Chen. “Task Scheduling and Voltage Selection for Energy Minimization”, Design Automation Conference (DAC), 2002.

[16] L. Benini, A. Bogliolo, and C. De Micheli. “A survey of design techniques for system-level dynamic power management”. IEEE transactions on Very Large Scale Integration (VLSI) Systems, Junio 2000.

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

[18] L. Kruse, E. Schmidt, G. v. Collin, A. Stammermann, A. Schulz, E. MAcii, and W. Nebel, “Estimation of low power and upper bounds on the power comsumption from scheduled data flow graphs”. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2001.

[19] Z. Gu, J. Wang, R. P. Dick, H. Zhou. “Unified Incremental Physical-Level and High-Level Synthesis”. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 26(9), Septiembre 2007.

[20] A. Stammermann, D. Helms, M. Schulte, A. Schulz, W. Nebel. “Binding, Allocation and Floorplanning in Low Power High level Synthesis”. International conference on Computer Aided Design (ICCAD), 2003.

[21] L. Zhong and N.K. Jha. “Interconnect-aware High-Level Synthesis for Low Power”. International Conference on Computer Aided Design (ICCAD), 2002.

[22] A.A. Del Barrio, M.C. Molina, J.M. Mendias, R. Hermida. “Pattern-guided switching minimization in high level synthesis”. Design of Circuits and Integrated Systems (DCIS), 2007.

[23] X. Xing and C.C. Jonj. “A look-ahead synthesis technique with backtracking for switching activity reduction in low power high-level synthesis”. Microelectronics Journal. Vol. 38(4-5): 595-605, 2007.

AUTORIZACIÓN DE DIFUSIÓN

El abajo firmante, matriculado en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "Reducción de la actividad de conmutación a nivel de subpalabra en síntesis de alto nivel", realizado
durante el curso académico 2008-2009 bajo la dirección de Maria del Carmen Molina Prego [y con la
colaboración externa de dirección de Guillermo Botella Juan] en el Departamento de DACYA, y a la
Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en
Internet y garantizar su preservación y acceso a largo plazo.