Técnicas de Planificación Temporal

May 7, 2024

1 Representaciones Gráficas

Las representaciones gráficas son esenciales para analizar y visualizar el flujo de datos de los algoritmos de procesamiento de señales digitales (DSP) y para explotar el paralelismo inherente entre diferentes sub-tareas. Estas representaciones son de gran ayuda para *mapear* algoritmos en hardware, permitiendo una exploración más efectiva del diseño y su arquitectura. Vamos a trabajar con cuatro tipos de representaciones gráficas de algoritmos DSP:

- Diagrama de bloques
- Grafo de flujo de señal (signal-flow graph - SFG)
- Grafo de flujo de datos (data-flow graph - DFG)
- Grafo de dependencia (dependency graph - DG)

Cada una de estas representaciones describe los algoritmos en diferentes niveles de abstracción. En general, DG muestra el paralelismo inherente y las restricciones máximas en el flujo de datos de un algoritmo, siendo útil para el diseño de un arreglo sistólico. Por otro lado SFG y DFG se utilizan para analizar propiedades estructurales y explorar arquitecturas alternativas mediante transformaciones de alto nivel.

1.1 Diagrama de bloques

Los diagramas de bloques son utilizados comúnmente para representar sistemas de procesamiento de señales digitales (DSP). Estos diagramas están compuestos por bloques funcionales conectados a través de aristas dirigidas (directed edges) que simbolizan el flujo de datos desde el bloque de entrada hasta el bloque de salida.
Ejemplo: 3-Tap FIR

$$y(n) = b_0 x(n) + b_1 x(n-1) + b_2 x(n-2) \tag{1}$$

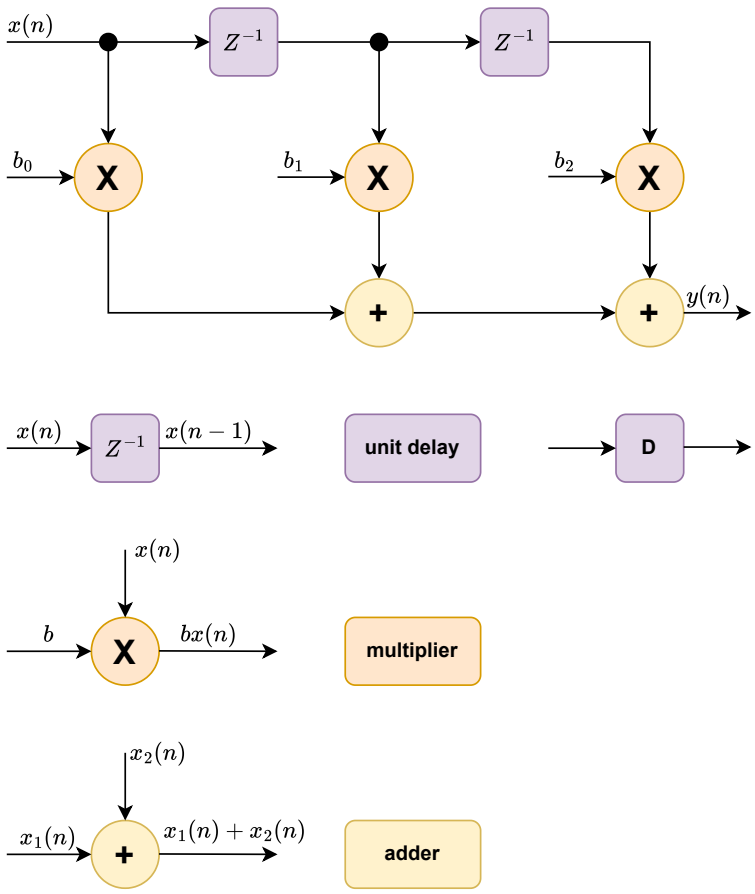


Figure 1: Diagrama Bloques 3-Tap Fir

1.2 Grafos de Flujo de Señal (SFG)

Un grafo de flujo de señal es una colección de nodos y aristas dirigidas. Los nodos representan cálculos o tareas. Las aristas dirigidas (j, k) representan transformaciones lineales de la señal en un nodo de entrada (j) hacia un nodo de salida (k). Los SFG son especialmente útiles para la representación, análisis y evaluación de redes y estructuras de filtros digitales. Estos grafos permiten describir sistemas DSP donde las aristas normalmente se restringen a:

- Multiplicadores de ganancia constante
- Delay

Los SFG pueden transformarse en direfentes formas sin cambiar la funcion de sistema representada.
Se utilizan principalmente para describir algoritmos DSP, en arquitecturas no son muy utilizados.

Ejemplo: *Flow graph reversal or transposition*: reversión del sentido de las aristas, se intercambian Inputs/outputs, lo cual es aplicada en sistemas de entrada/salida única para obtener estructuras transpuestas equivalentes

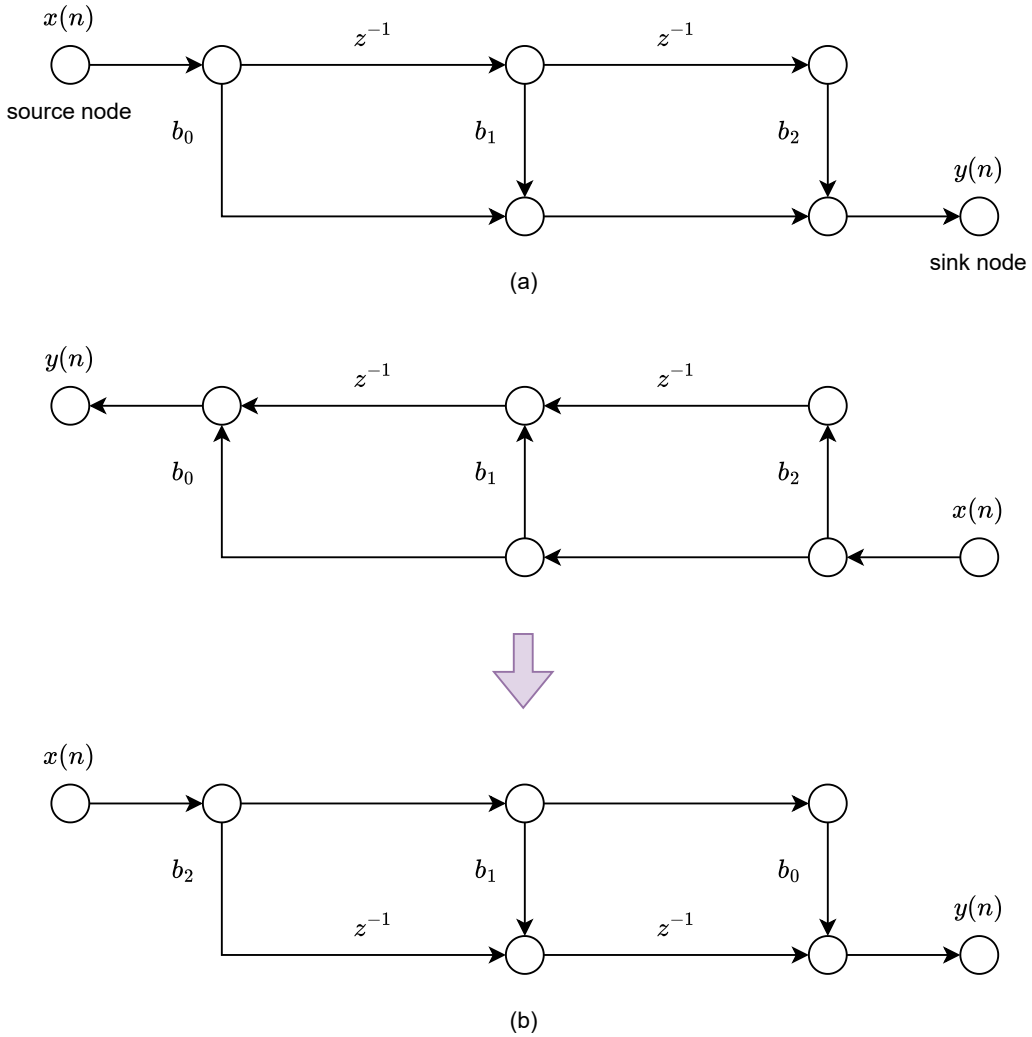


Figure 2: SFG 3-tap FIR Filter. (a) Direct-Form FIR fitler SFG; (b) Transposed FIR Filter SFG.

1.3 Grafo de Flujo de Datos (DFG)

En esta representación, los nodos representan cómputos (o funciones o sub-tareas) y las aristas dirigidas represetan el camino del dato (comunicación entre los nodos), y cada arista tiene asociado un número no negativo de delay.

Ejemplo:

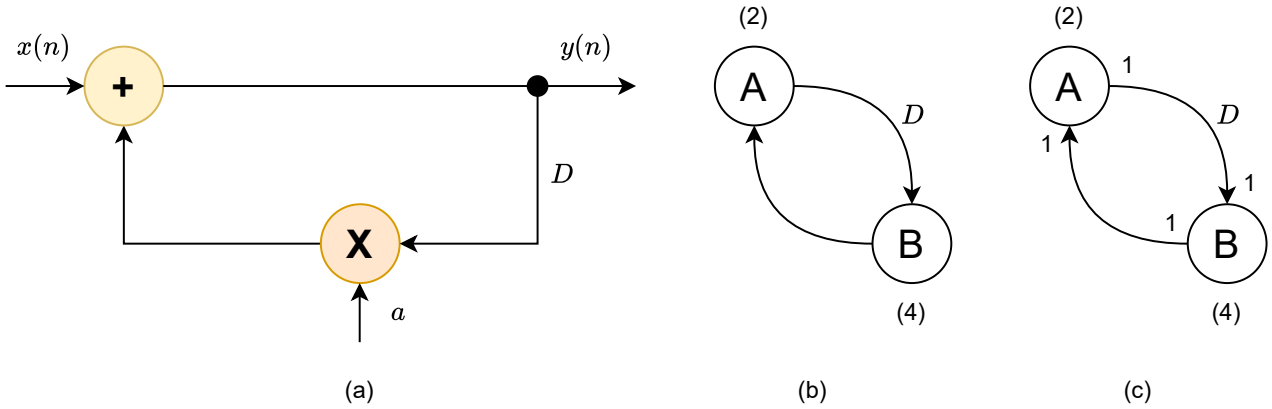


Figure 3: (a) Diagrama en bloques de $y(n) = ay(n-1) + x(n)$. (b) DFG Convencional. (c) DFG Síncrono

Donde A representa la suma y B la multiplicación, la arista de A hacia B ($A \to B$) contiene un delay, mientras que $B \to A$ no contiene ninguno. Asociado a cada nodo tenemos su tiempo de ejecución normalizado en unidades de tiempo (u.t.; unit time) y pueden dispararse cuando todas las entrada necesarias estén disponible, respetando ciertas restricciones de procedencia dentro y entre iteraciones de los datos procesados.

Los DFG, como también los diagramas en bloques se utilizan para describir tanto los sistemas lineares single-rate como tambien no-lineales multi-rate.

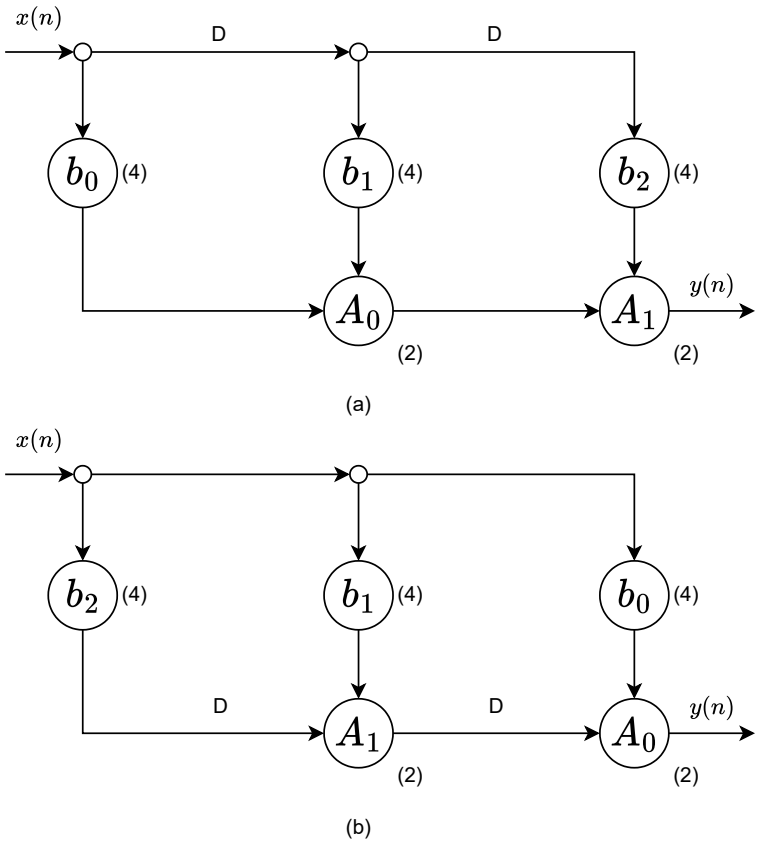


Figure 4: (a) DFG 3-tap FIR filter direct form. (b) DFG para su forma transpuesta.

El grafo de flujo de datos síncrono (SDFG) es una variante especial del grafo de flujo de datos donde la cantidad de muestras de datos producidas o consumidas en cada ejecución está especificada de antemano. Este tipo de grafos es adecuado para describir sistemas que operan en tasa única o múltiples tasas de procesamiento.

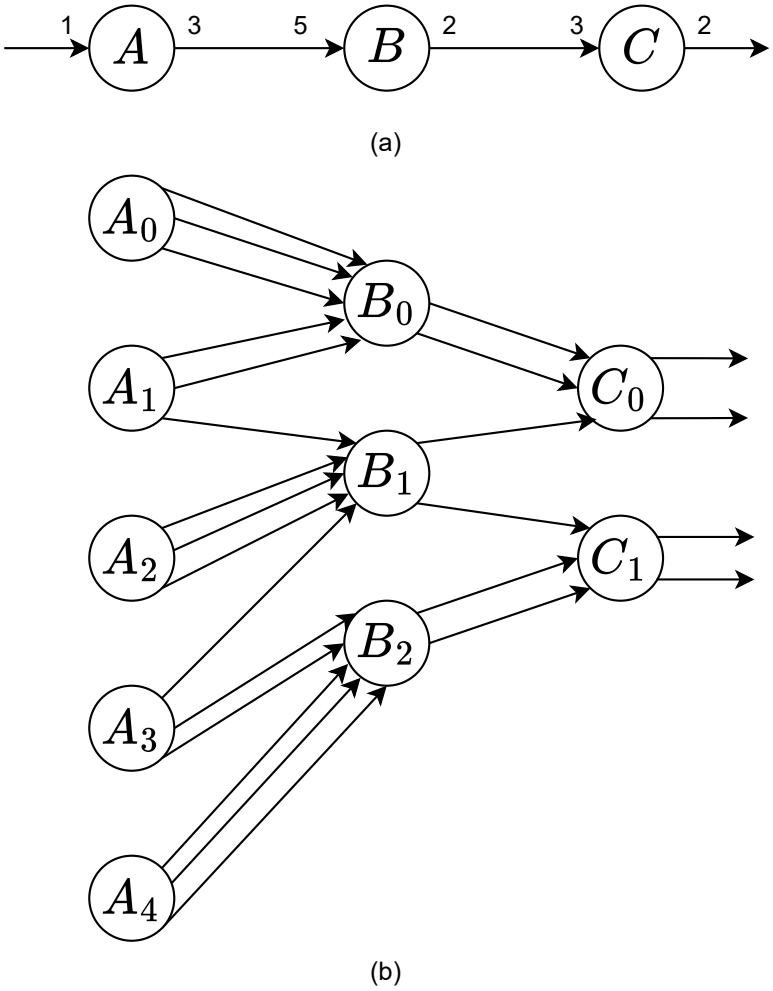


Figure 5: Multirate DFG (a), puede ser convertido a single-rate DFG (b).

1.4 Grafo de dependencia (DG)

Tipo de grafo dirigido que muestra las dependencias entre las operaciones o cómputos en un algoritmo. En este grafo, cada nodo representa un cómputo específico y las aristas indican restricciones de precedencia entre estos cálculos.

- **Creación de Nodos**: Cada vez que se introduce un nuevo cómputo en el algoritmo, se crea un nodo correspondiente en el grafo. Los nodos en un DG no se reutilizan para operaciones repetitivas.
- **Diferencia con DFG**: En los DFG, los nodos representan cómputos de una iteración del algoritmo que pueden ser ejecutadas repetitivamente y tienen elementos de delay. Mientras que DG cubre todas las iteraciones de un algoritmo sin contener elementos de delay.

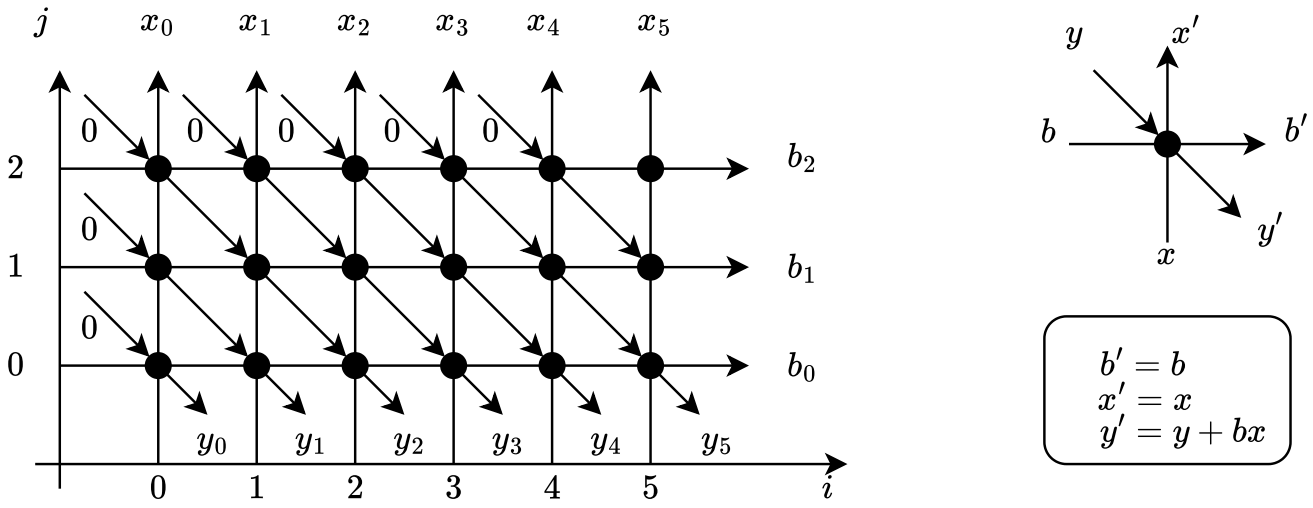
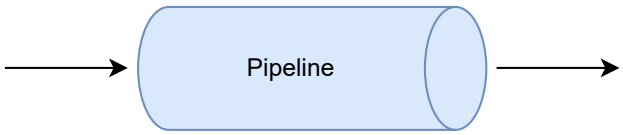


Figure 6: Grafo de dependencias para un 3-tap FIR filter $y(n) = b_0x(n) + b_1x(n-1) + b_2x(n-2)$

2 Técnicas de Planificación temporal

2.1 Introducción

- Pipelining viene de la idea de un caño de agua, donde se continua enviando agua sin esperar que termine de salir por el final del caño.



- La técnica de *pipelining* permite incrementar la capacidad de procesamiento (throughout/data rates) y mejorar el rendimiento de las aplicaciones.
  - Definición **feed-forward**: La salida depende de la entrada actual y las entradas previas. No depende de salidas anteriores.
  - Un **camino crítico** combinacional que se encuentra en un sistema *feed-forward*, puede ser reducido agregando registros de pipeline.
  - Los registros de pipeline agregan **latencia**. Si se agregan L registros, la función transferencia del sistema se multiplica con z^{-L}
  - En una nube combinacional del tipo *feedback*, los registros de pipeline no se pueden agregar de la misma manera ya que estos modificarían la función de transferencia del sistema y este caso resulta en una modificación del orden de las ecuaciones de diferencia.
- Pipelining y Retiming son 2 aspectos diferentes del diseño digital. En el caso de Retiming, se reubican los registros dentro de un circuito para mejorar diversos objectivos de diseño como:
  - Reducción de la longitud del *critical path*.
  - Minimización del número de registros
  - Otros tales como: optimización del consumo de energía y aumentar la testabilidad de la implementación.

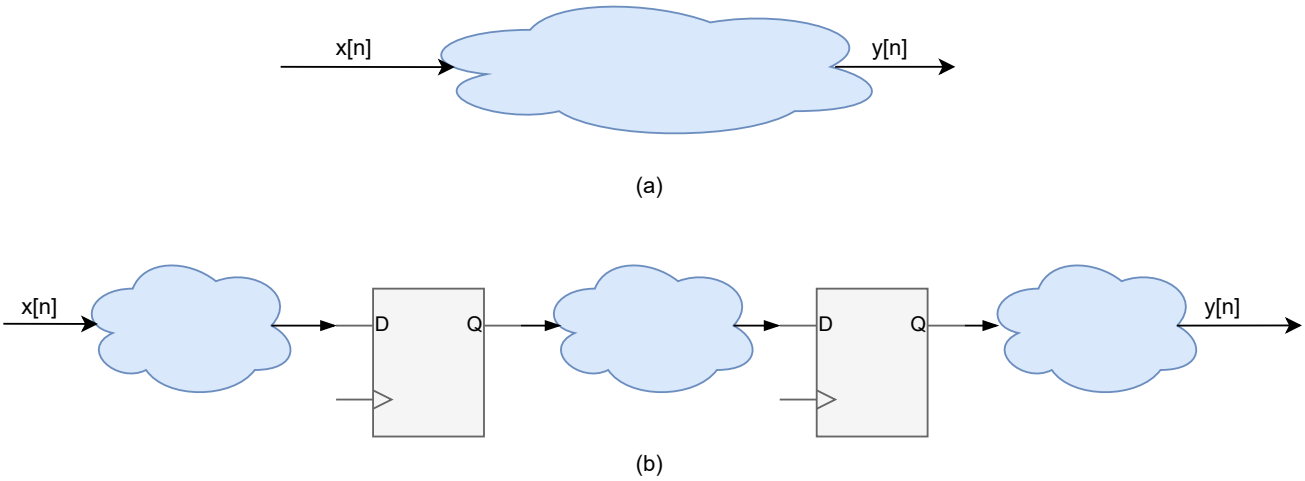


Figure 7: Los registros de pipeline se agregan para reducir el critical path de la nube combinacional. (a) Lógica combinatoria inicial. (b) Diseño con 3 etapas de pipeline utilizando 2 conjuntos de registros de pipeline.

2.2 Resumen

Retiming	Pipelining
No introduce Latencia	Aumenta la Latencia
Reordena los delays usando *feed-back cutsets*	Agrega delays en *feed-forward cutsets*
Puede disminuir el *critical path*	Puede disminuir el *critical path*
Puede aumentar o reducir el número de delays	Siempre aumenta el número de delays

3 Pipelining

3.1 Pipelining usando Feedforward Cut-set

La figura 8 muestra un ejemplo de un *feed-forward cut-set*. Si las 2 aristas $1 \rightarrow 2$ y $1 \rightarrow 3$ son removidas, el grafo se vuelve disjunto, quedando el nodo 1 en un grafo y los nodos 2 y 3 en otro grafo. Agregando N registros al circuito se mantiene la coherencia de los datos pero al *output* se le agrega un delay de N ciclos.

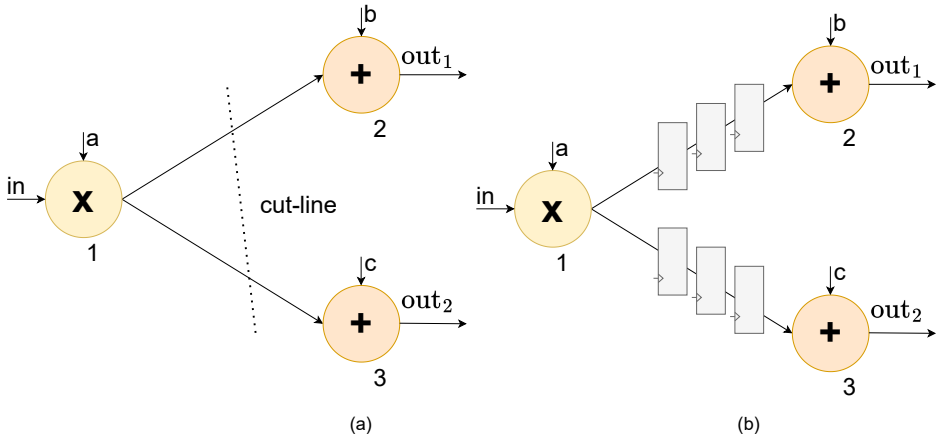


Figure 8: (a) Feed-forward cut set. (b) Registros pipeline agregados.

3.1.1 Ejemplo 1

Consideremos el siguiente 3-Tap FIR Filter (que veremos la próxima clase):

$$y(n) = ax(n) + bx(n-1) + cx(n-2) \tag{2}$$

El diagrama en bloques de este filtro puede verse a continuación:

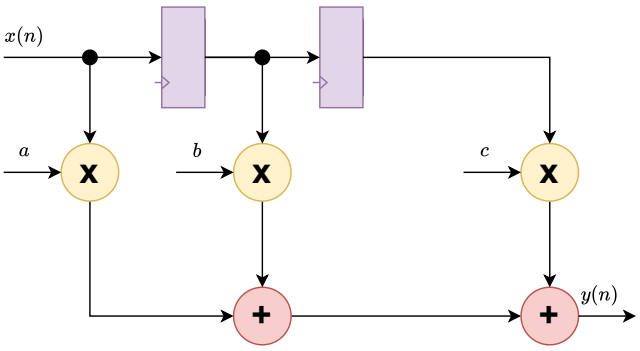


Figure 9: 3-Tap FIR Filter

Podemos ver que el critical path, o el tiempo mínimo para procesar una nueva muestra está limitado por 1 multiplicador y 2 sumadores. Asumimos que los tiempos de ejecución del multiplicador y del sumador son T_m y T_a

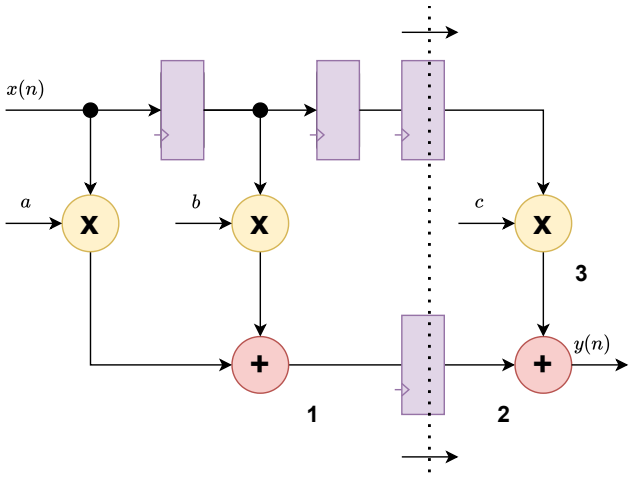


Figure 10: 3-Tap FIR Filter Pipeline

respectivamente, entonces el período de iteración es de $T_m + 2T_a$ y debe satisfacer el período de sampling T_s:

$$T_m + 2T_a \leq T_s \tag{3}$$

Usando una etapa de pipeline, se reduce el *critical path* a $T_m + T_a$. Podemos observar que en esta configuración, mientras el sumador de la izquierda inicia el cómputo de la iteración actual, el sumador derecho está completando el cómputo de la iteración anterior. En este sistema, en todo momento, 2 outputs consecutivas se computan de forma intercalada.

Clock	Input	Nodo 1	Nodo 2	Nodo 3	Output
0	$x(0)$	$ax(0) + bx(-1)$	-	-	-
1	$x(1)$	$ax(1) + bx(0)$	$ax(0) + bx(-1)$	$cx(-2)$	$y(0)$
2	$x(2)$	$ax(2) + bx(1)$	$ax(1) + bx(0)$	$cx(-1)$	$y(1)$
3	$x(3)$	$ax(3) + bx(2)$	$ax(2) + bx(1)$	$cx(0)$	$y(2)$

3.1.2 Ejemplo 2

Consideremos el siguiente sistema dado por:

$$y_n = h_0 x_n + h_1 x_{n-1} + h_2 x_{n-2} + h_3 x_{n-3} + h_4 x_{n-4} \tag{4}$$

Hay múltiples formas de proponer un *cut-set*.

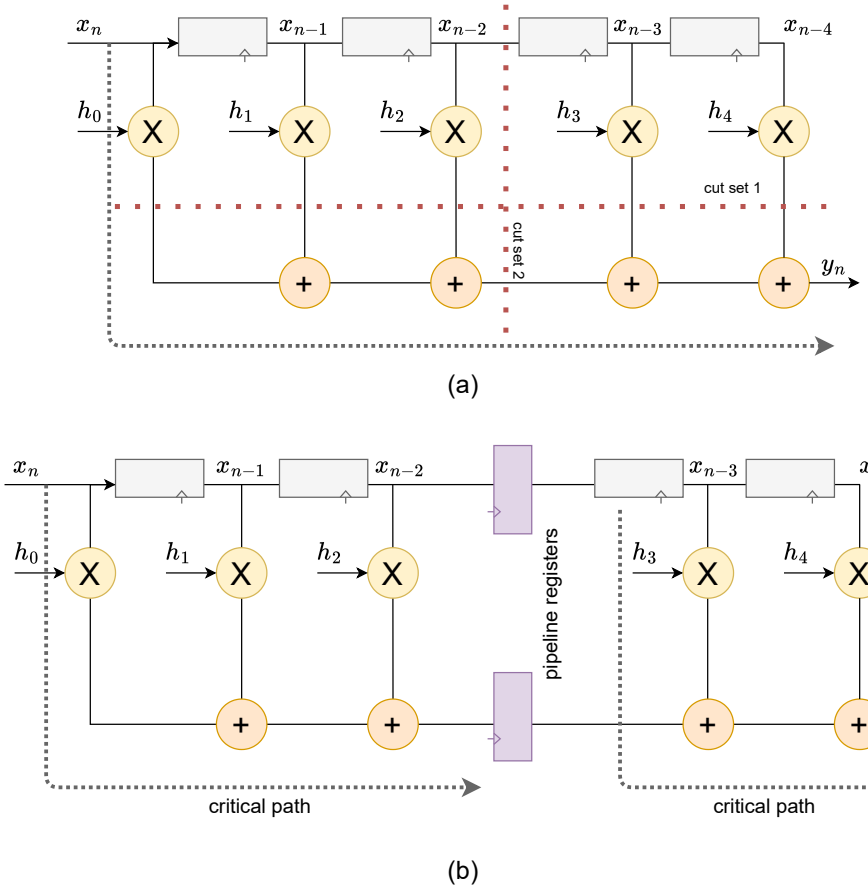


Figure 11: Pipelining mediante cut-set. (a) 2 posibles candidatos para realizar el corte. (b) Un nivel de pipeline agregado utilizando el candidato 2

El diseño pipeline implementa la siguiente función de transferencia:

$$H_r(z) = z^{-1} H(z) \tag{5}$$

3.2 Ejemplo: Fine Grain Pipelining

Que es fine-grain pipelining? Cuando los registros se ubican dentro de una unidad de cómputo, en este ejemplo un sumador RCA.

- El RCA 2-etapas de pipeline tiene 2 FullAdders en el *critical path*.
- El resultado ahora esta disponible 1 clock después que la versión original, es decir tiene 1 clock de latencia.

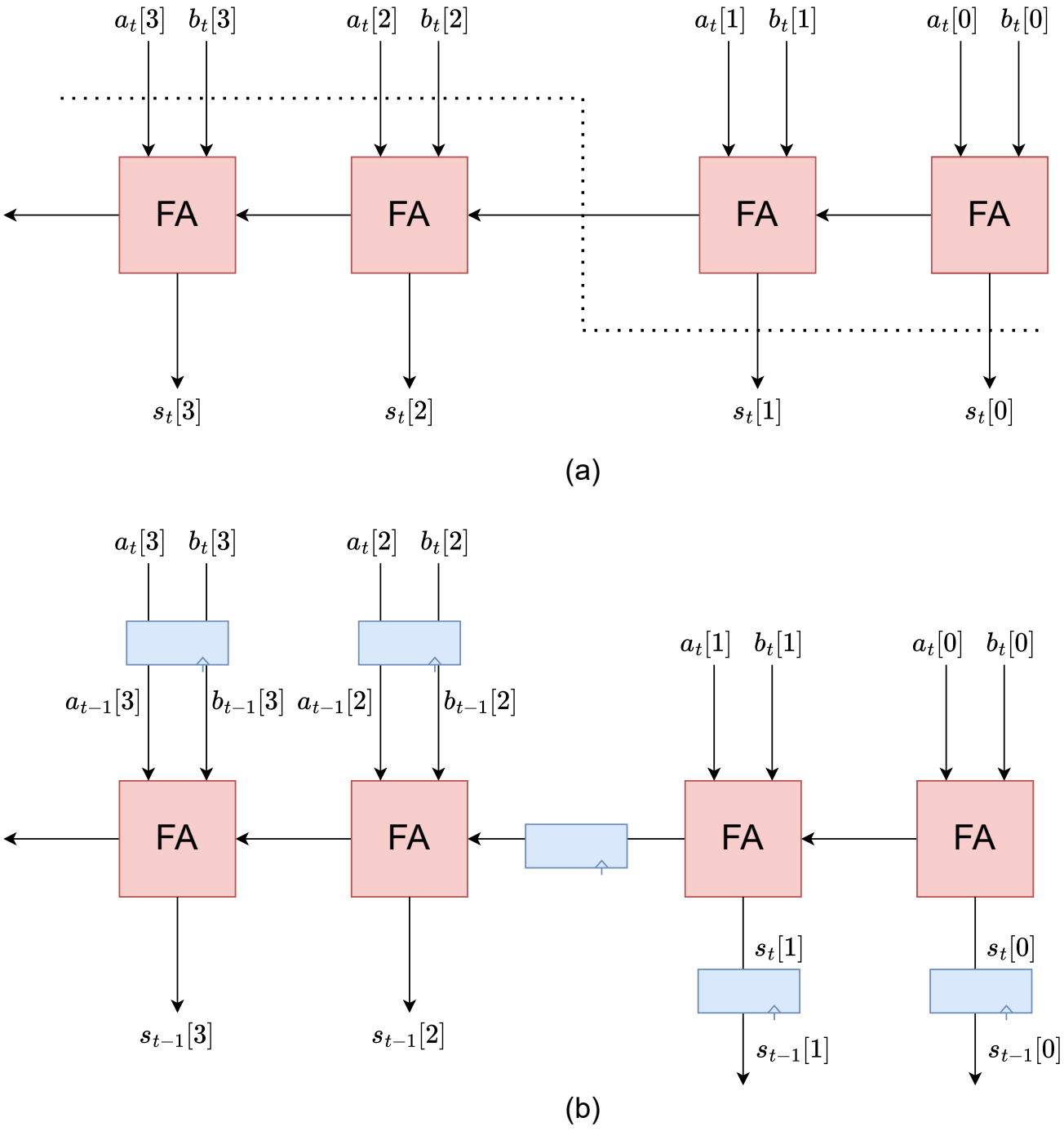


Figure 12: Pipelining mediante cut-set. (a) Cut-set propuesto para 4-bits RCA. (b) Registros ubicados

```
// Modulo que implementa un sumador 4bits pipeline de 2 etapas
module pipeline_adder(
    input  wire       clk,
    input  wire       rst_n,
    input  wire [3:0] a,
    input  wire [3:0] b,
    input  wire       cin,
    output reg  [3:0] sum_p,
    output reg        cout_p
);

// Pipeline Registers
reg [3:2] a_preg, b_preg;
reg [1:0] s_preg;
reg       c2_preg;
// Internal wires
reg [3:0] s;
reg       c2;

// Nube Combinacional
always @* begin
    // Combinacional 1
    {c2, s[1:0]} = a[1:0] + b[1:0] + cin;
    // Combinacional 2
    {cout_p, s[3:2]} = a_preg + b_preg + c2_preg;
    // Output
    sum_p = {s[3:2], s_preg};
end

// Sequencial
always @ (posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        s_preg  <= 0;
        a_preg  <= 0;
        b_preg  <= 0;
        c2_preg <= 0;
    end else  begin
        s_preg  <= s[1:0];
        a_preg  <= a[3:2];
        b_preg  <= b[3:2];
        c2_preg <= c2;
    end
end
endmodule
```

3.3 Ejemplo 2: RCA 4-etapas pipeline

- El RCA 4-etapas de pipeline tiene un único FullAdder en el *critical path*.
- Para mantener la coherencia de los datos, el cut-set propuesto asegura que todos los paths desde input hacia un output, tengan 3 registros.

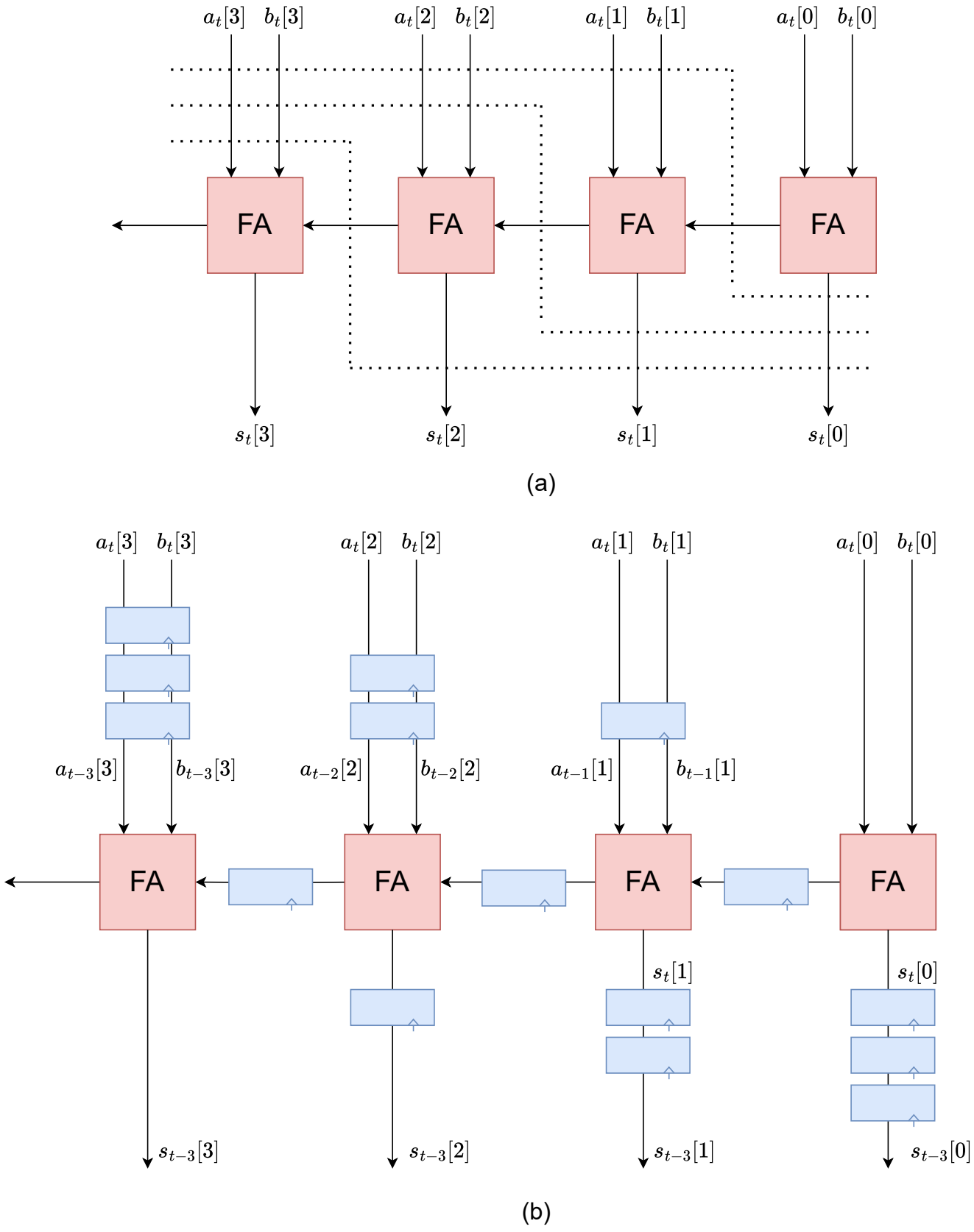


Figure 13: (a) Tres cut-sets para agregar cuatro etapas de pipeline en el RCA 4-bits. (b) Pipeline Final

Es una buena práctica reacomodar las etapas de pipeline de la siguiente manera, para poder claramente visualizar las distintas etapas.

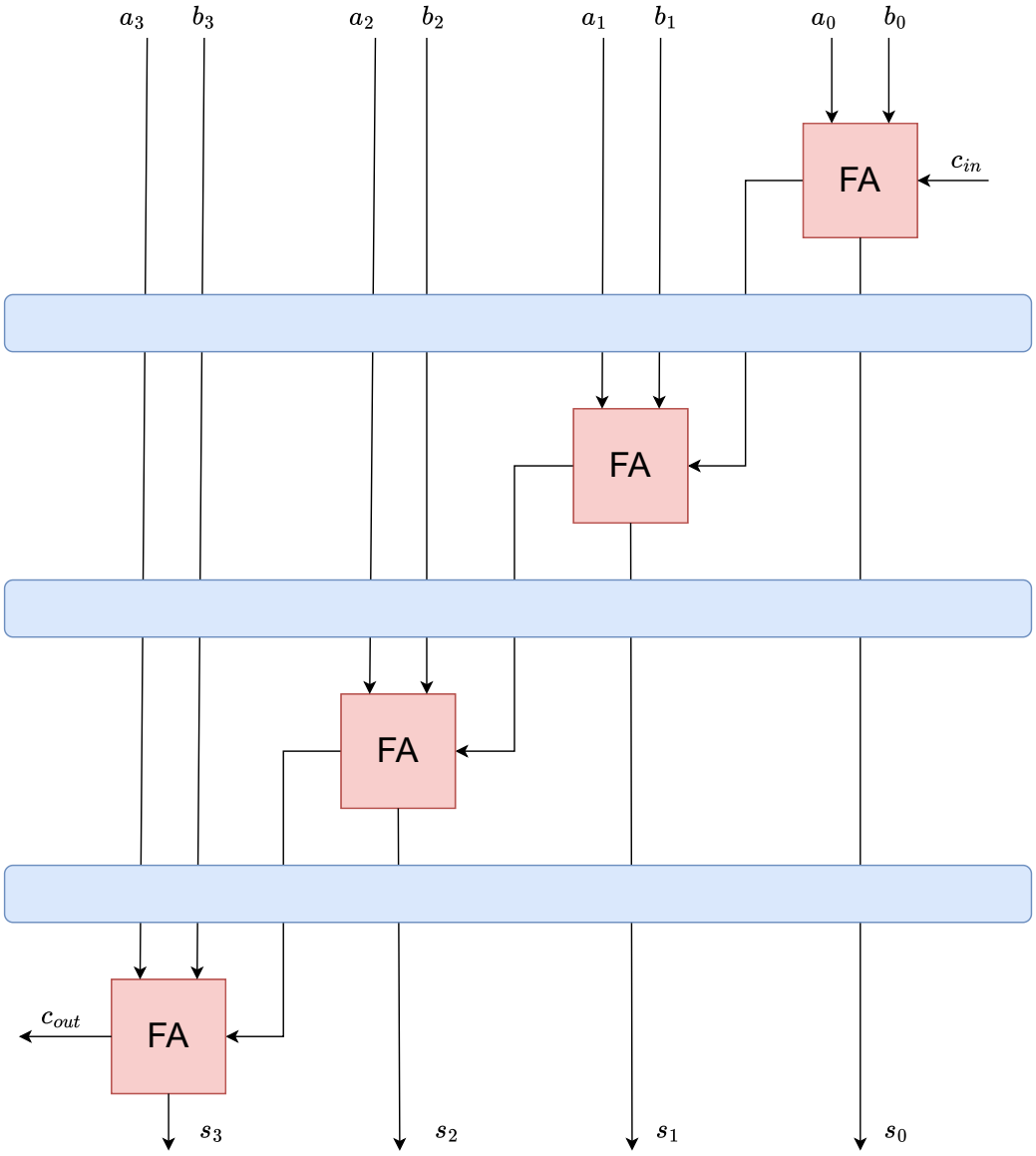


Figure 14: Siguiendo buenas práctica de diseño mediante la alineación de diferentes etapas de pipeline

3.4 Teorema de transferencia de Delay

- Establece que, sin afectar la función de transferencia del sistema, se pueden transferir N registros desde cada arista entrante de un nodo del DFG, hacia todas las aristas salientes del mismo nodo, o viceversa.
- Es un caso particular de *cut-set retiming* donde la línea de corte se utiliza para separar un nodo de un DFG.

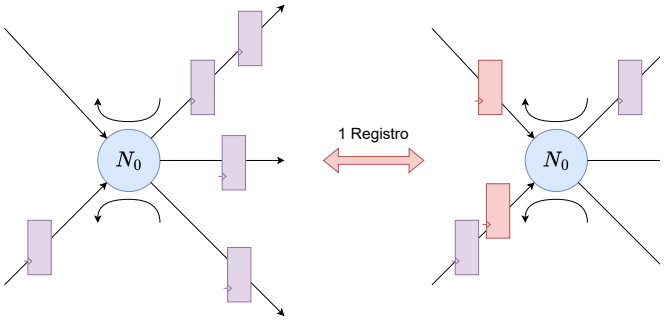
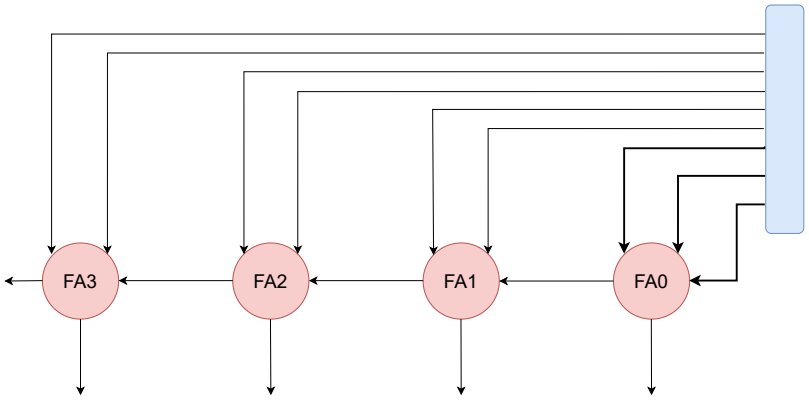


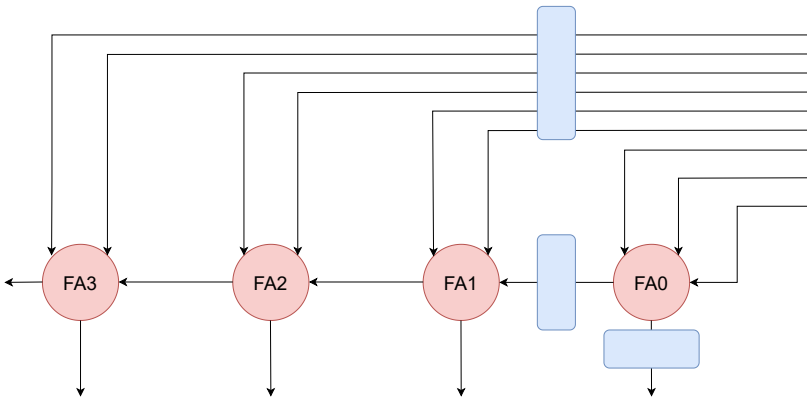
Figure 15: Aplicando el teorema de transferencia de delay movemos un registro a través del nodo N_0

3.5 Pipelining usando el teorema de transferencia de delay

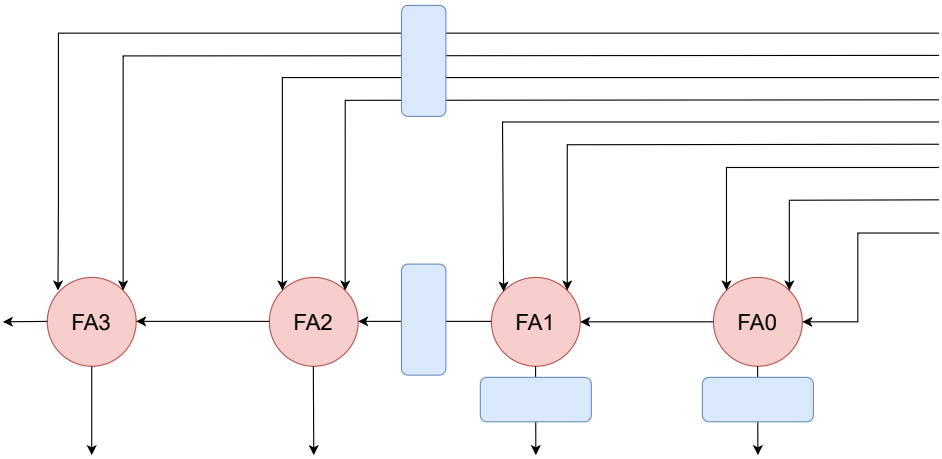
Una forma conveniente de implementar pipelining es agregar el número deseado de registros en todas las entradas y luego aplicar sistemáticamente el teorema de transferencia de delay, hasta llegar al *critical path*.



(a)

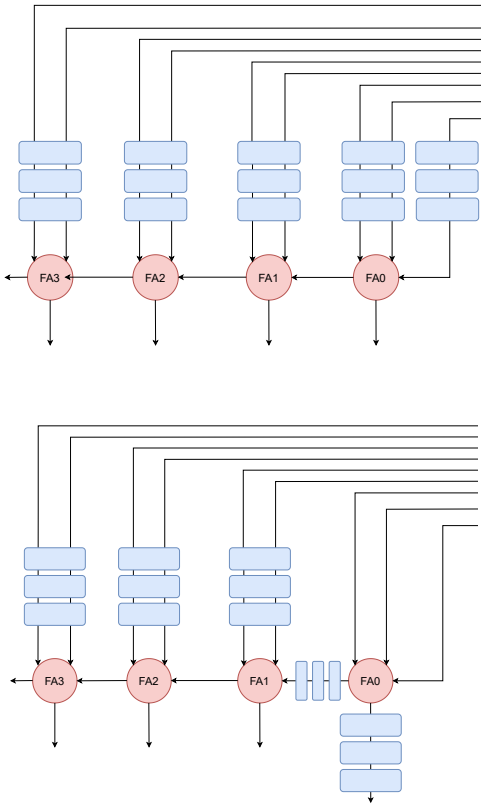


(b)



(c)

Figure 16: Teorema transferencia de delay para agregar 1 nivel de pipeline. (a) DFG Original. (b) Teorema aplicado sobre el nodo FA0. (c) Teorema aplicado sobre el nodo FA1.



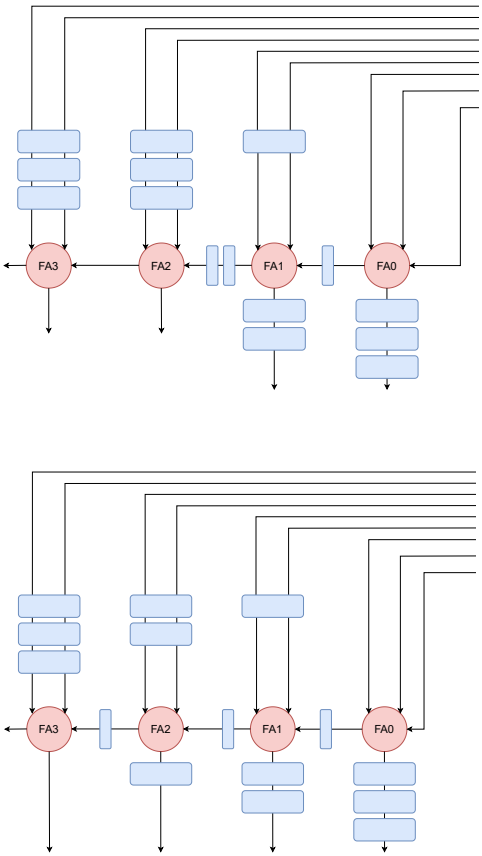


Figure 17: Usando el teorema de transfencia de delay se agregan 3 etapas de pipeline. (a) DFG Original. (b) FA0. (C) FA1. (d) FA2

3.6 Ejemplo: RISC-v Pipeline

A modo de ejemplo cualitativo, observamos en la siguiente imagen un diagrama en bloques de un RISC-V básico con 5 etapas de pipeline.

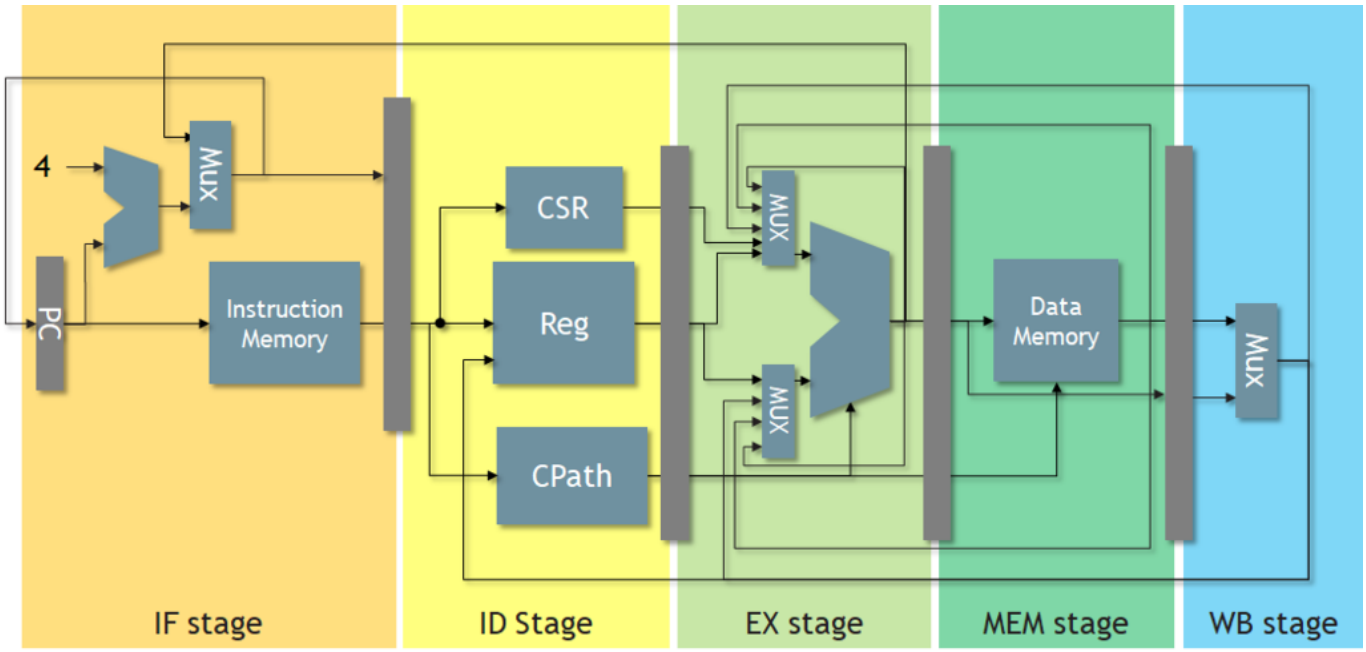


Figure 18: RISC-V con 5 etapas de pipeline

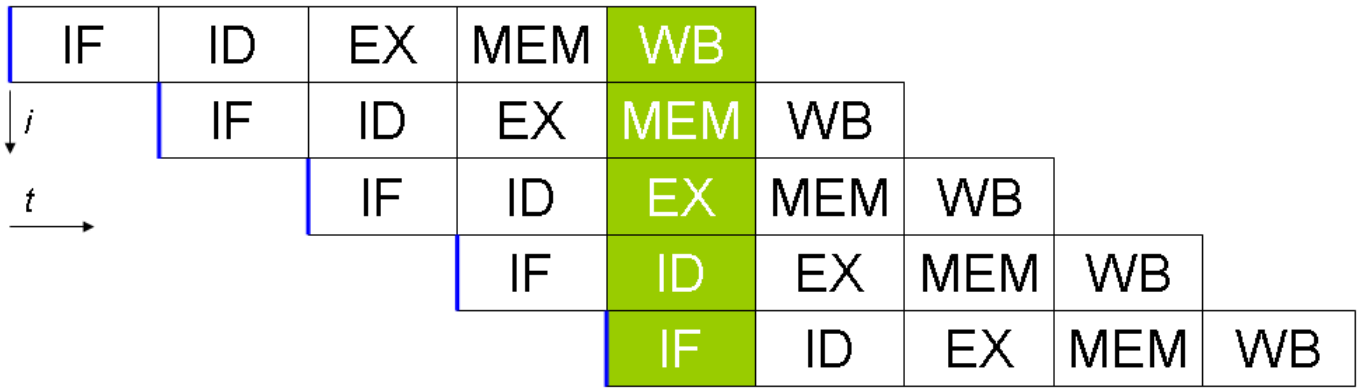


Figure 19: RISC-V con 5 etapas de pipeline: Ejecución

4 Retiming

- Esta técnica permite mejorar la performance del diseño durante síntesis sin tener que rediseñar el RTL
- Agrega una dimensión de optimización al permitir que los registros se muevan entre la lógica combinacional.
- Qué métricas mejora? Performance y Area
- Las herramientas de síntesis como Genus, suelen soportar retiming automático y manual.

4.1 Retiming aplicado a Timing

En la figura 20 podeos observar como mediante la aplicación de retiming reducimos el *critical path* de 6ns a 5ns. Retiming trata de balancear los delays en los *worst critical path*, por lo tanto reduciendo WNS (worst negative slack).

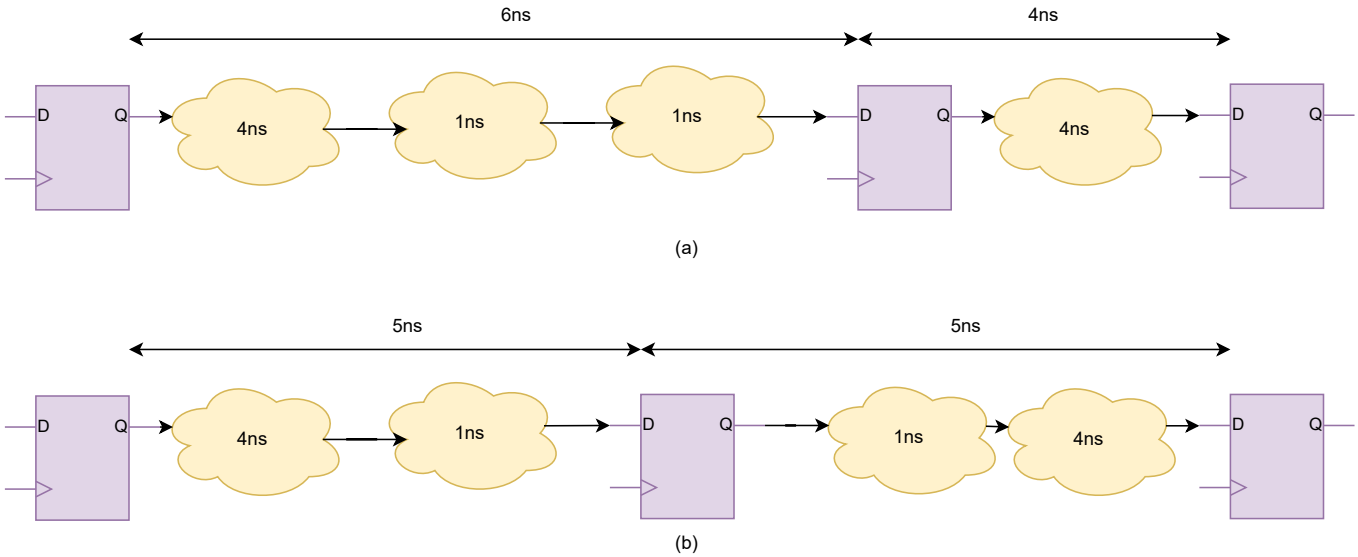


Figure 20: Retiming para reducir timing

4.2 Minimización del Número de Registros y delay del critical path

Retiming también puede ser aplicado para reducir el número de registros en el diseño.

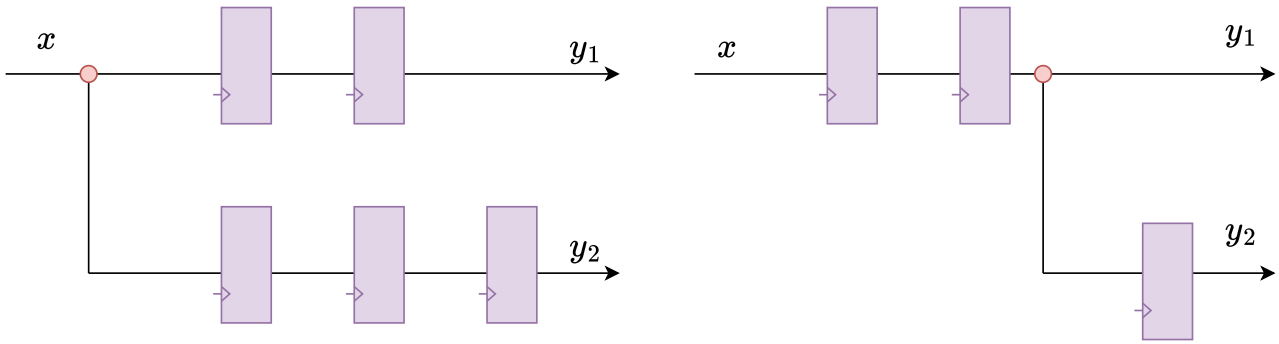


Figure 21: Retiming para minimización de registros

4.3 Cut-set Retiming

Un *cut-set* válido es un conjunto de aristas forward o backward en un DFG que intersecta una línea de corte de tal manera que, si estas aristas se eliminan del grafo, el grafo se vuelve disjunto.
El *retiming* implica transferir una serie de delays desde aristas de la misma dirección a través de una línea de corte de un DFG hacia todas las aristas de dirección opuesta a lo largo de la misma línea. Estras transferencias de delays no alteran la función de transferencia del DFG. La figura 22 muestra un DFG con una línea de corte válida. La línea divide el DFG en 2 grafos disjuntos, uno formado con los nodos N_0 y N_1 y el otro por N_2. La arista $N_1 \rightarrow N_2$ es una arista de corte hacia adelante (forward), mientras que $N_2 \rightarrow N_0$ y $N_2 \rightarrow N_1$ son aristas de corte hacia atrás (backward). Un retraso de cada una de estas aristas backward se mueve hacia la arista forward.

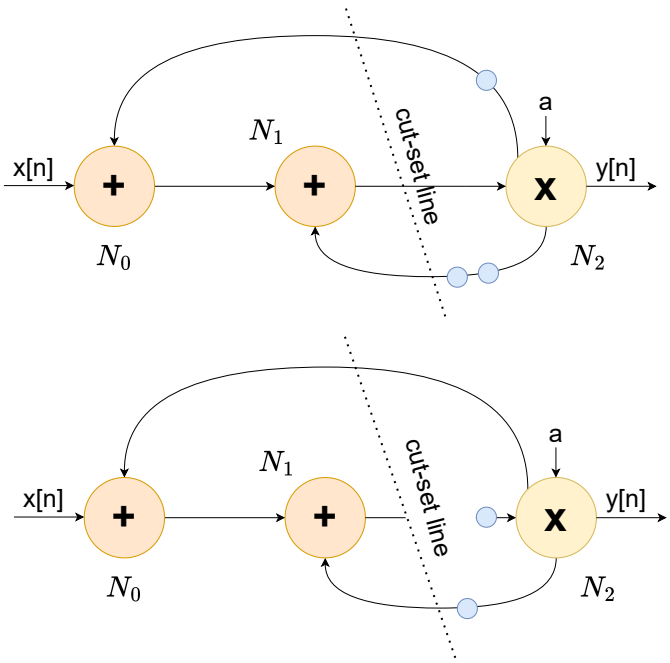


Figure 22: Cut-set retiming. (a) DFG inicial con linea de cut-set propuesta. (b) DFG luego de cut-set retiming.

4.4 Pipelining y Retiming en un sistema Feed-forward

Son metodologías complementarias.
Pipeline añade el número apropiado de registros de manera que transforma un grafo de flujo de datos G, en un grafo pipelined G_N de tal manera que las funciones de transferencia $H(z)$ de G y $H_N(z)$ de G_N, solo difieren por un delay puro z^{-L}, donde L es el número de etapas de pipeline añadidas en el DFG.
Añadir registros seguido por un retiming facilita el movimiento de registros en un DFG para reducir el *critical path*.

4.5 Retiming FIR Filter Direct Form

Retiming puede ser aplicado a un Filtro FIR en forma directa para obtener su forma transpuesta (transposed direct-form TDF). Esta forma rompe el *critical path* ubicando un registro antes de cada sumador.

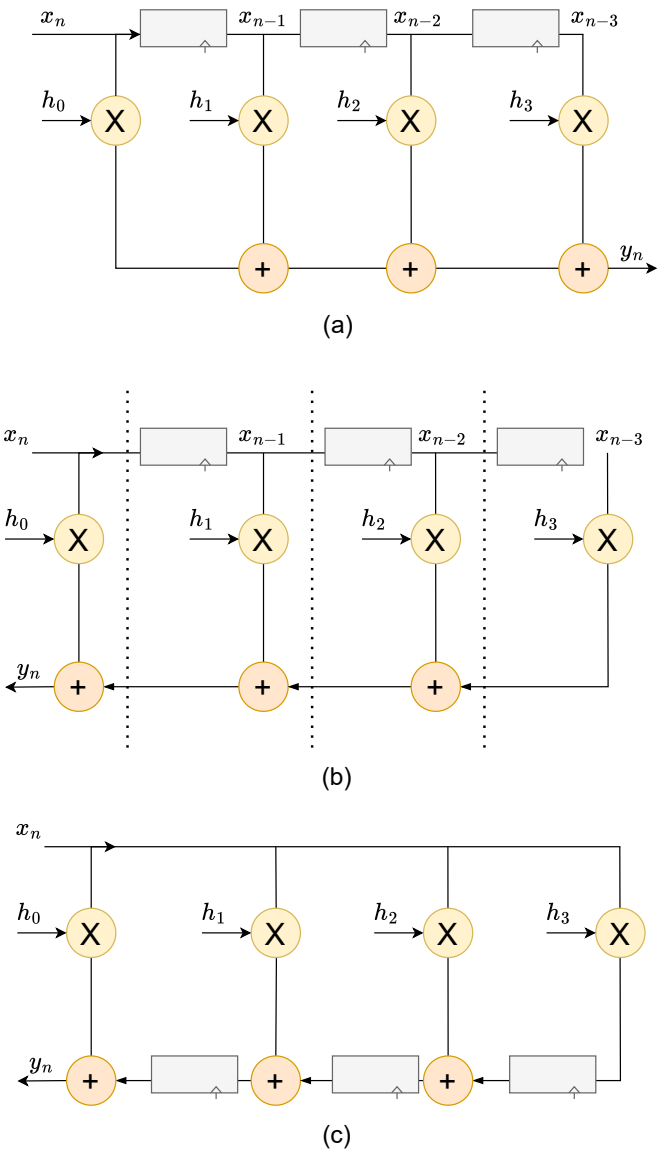


Figure 23: Filtro FIR en forma directa transformado a una estructura TDF usando cut-set retiming. (a) Filtro FIR 4-coeffs en DF. (b) Inversión de la dirección de las sumas en DF y se aplica cut-set retiming. (c) Retimed Filter en TDF.

4.6 Retiming Periférico

Todos los registros se mueven al perímetro del diseño, ya sea entrada o salida de la lógica. Luego la lógica combinacional es globalmente optimizada y se finaliza realizando retiming en los registros para obtener la mejor performance.

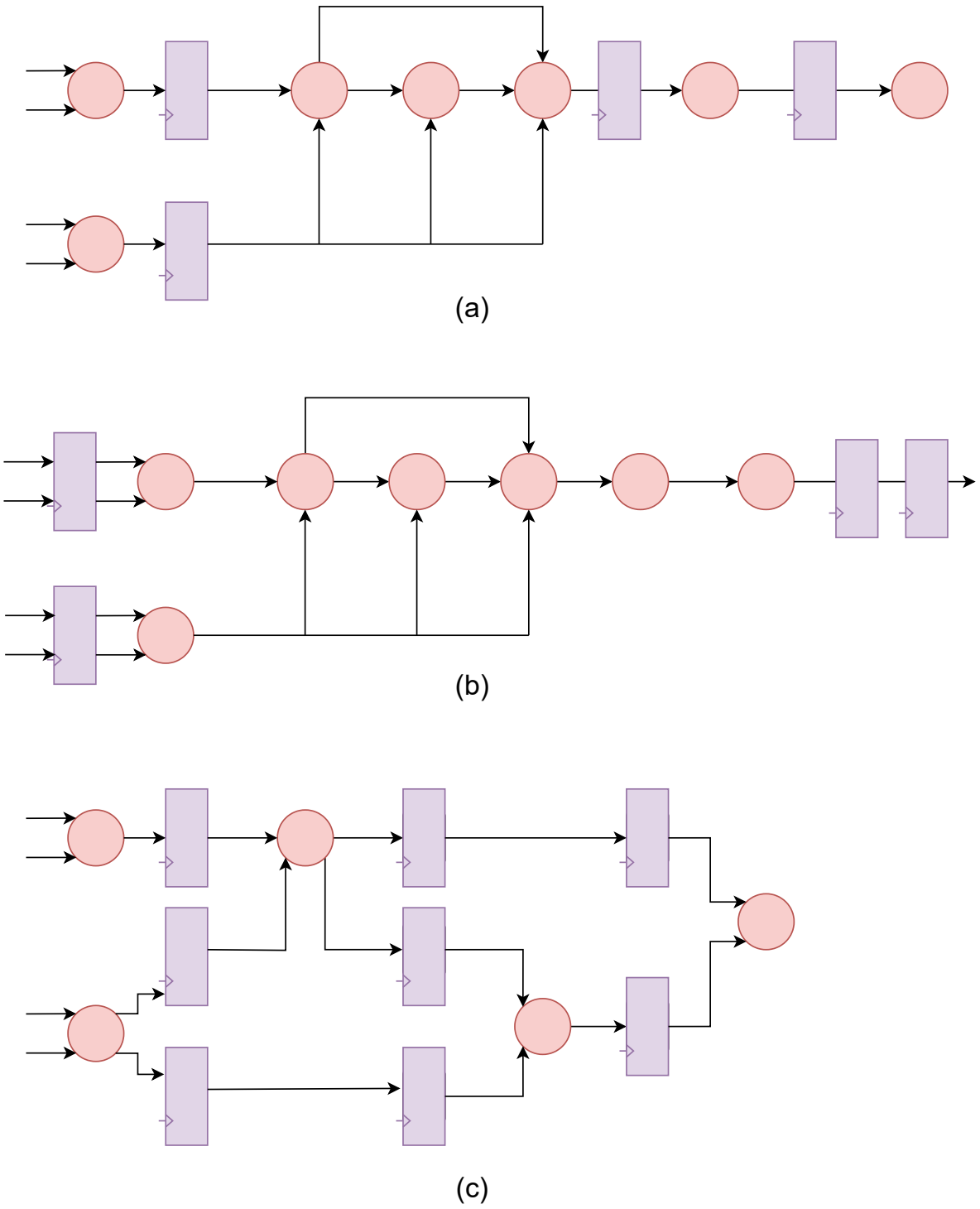


Figure 24: Retiming Periférico. (a) Diseño original. (b) Primero se mueven los registros al perímetro del diseño. (c) Luego se realiza retiming y optimización.

4.7 Retiming mediante descomposición de Shannon

La descomposición de Shannon es una transformación que puede extender el alcance de retiming. Fracciona una función booleana multivariable en una combinación de dos funciones booleanas equivalentes.

$$f(x_o, x_1, \cdots, x_{N-1}) = \overline{x_0}.f(0, x_1, \cdots, x_{N-1}) + x_0.f(1, x_1, \cdots, x_{N-1}) \tag{6}$$

Esta técnica identifica una señal x_0 que tarda en llegar al módulo (*late arrival signal*) y duplica la lógica asignándole valores a X_0 de 1 y 0. Luego un multiplexor 2:1 selecciona la salida correcta de esta lógica duplicada.

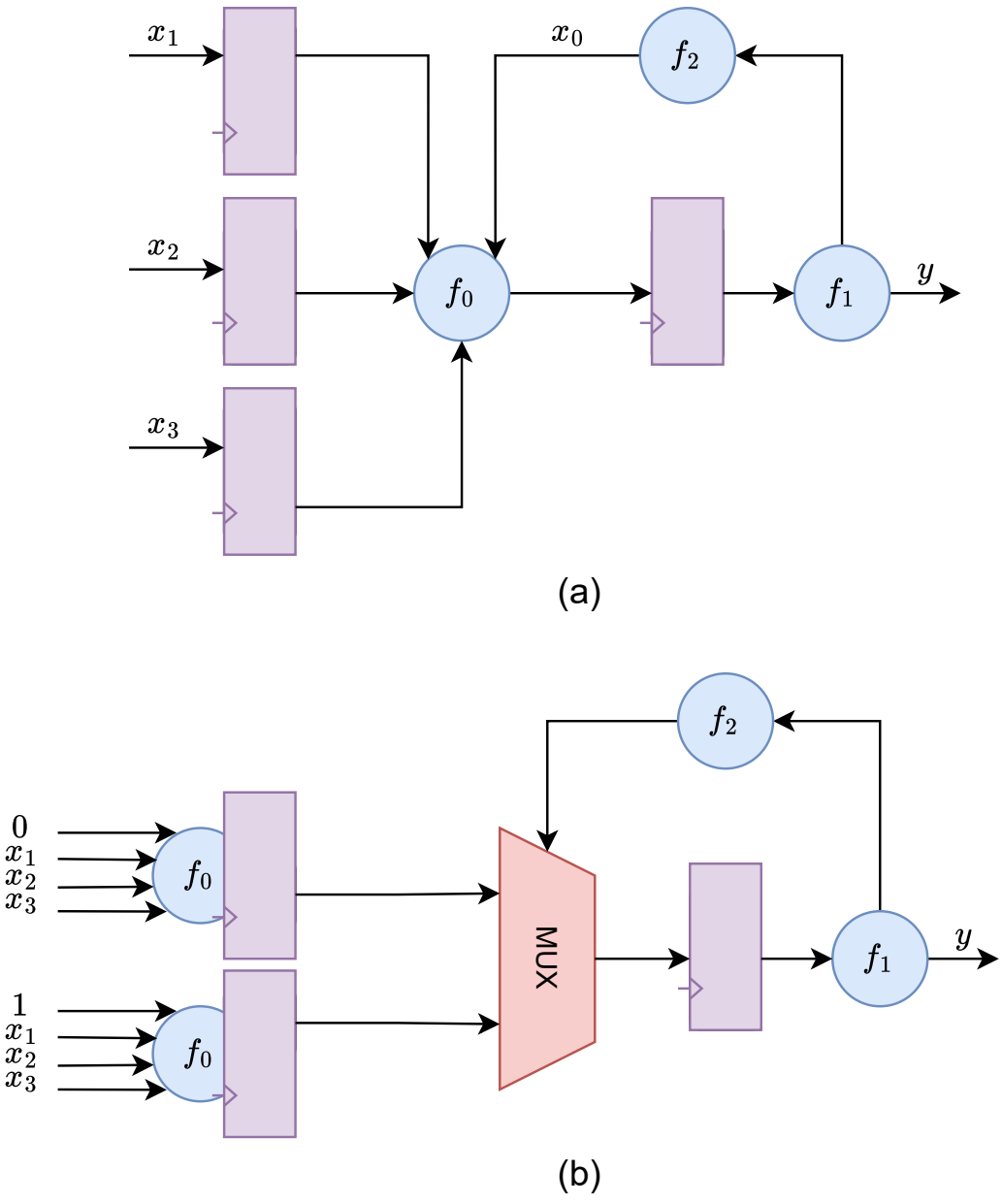


Figure 25: (a) Decomposición de Shannon para remover el la entrada mas lenta X_0 hacia f_0 y duplicar la lógica en f_0 con valores fijos para x_0. (b) Diseño luego de retiming.

5 Feedback

5.1 Definiciones

- Un sistema con feedback calcula una muestra de salida basada en salidas anteriores y entradas actuales y previas.
- La iteración se define como la ejecución de todas las operaciones en un algoritmos que son necesarias para calcular una muestra de salida
- El período de iteración es el tiempo requerido para ejecutar una *iteración* del algoritmo
- En sistemas real-time síncronos, se debe completar la ejecución de la iteración actual antes de que se adquiera la siguiente muestra de entrada. Esto impone un límite superior en el período de iteración para que sea menor o igual a la tasa de muestreo de los datos de entrada.

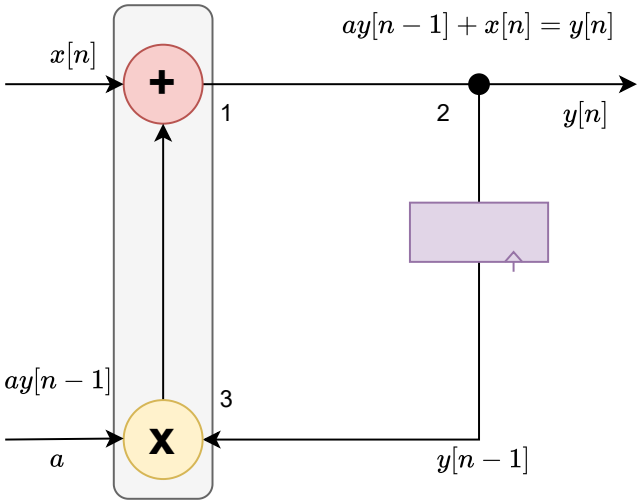


Figure 26: El período de iteración para un IIR de primer orden es igual a $T_m + T_a$

La figura 26 muestra un filtro IIR de primer orden que implementa la siguiente ecuación:

$$y[n] = ay[n-1] + x[n] \tag{7}$$

El algoritmo necesita realizar una multiplicación y una suma para computar la salida de una iteración. Asumimos que los tiempos de ejecución del multiplicador y del sumador son T_m y T_a respectivamente, entonces el período de iteración para este ejemplo es de $T_m + T_a$ y debe satisfacer respecto del período de sampling T_s:

$$T_m + T_a \leq T_s \tag{8}$$

- Un **bucle** (loop) se define como un camino dirigido que comienza y termina en el mismo nodo. En la figura 26, el camino dirigido que consisten en los nodos $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$ es un bucle.
- El **Límite del bucle** (loop bound) del i-ésimo bucle se define como $\frac{T_i}{D_i}$, donde T_i es el tiempo de cómputo del bucle y D_i es el numero de delays del bucle. Para el ejemplo de la figura 26, el límite del bucle es $\frac{T_m+T_a}{1}$
- Un **bucle crítico** (critical loop) de un DFG se define como el bucle con el límite de bucle máximo.
- El período de iteración de bucle crítico se llama límite del período de iteración (**iteration period bound - IPB**). Se expresa como:

$$IPB = max_{\text{todos } L_i} \left\{ \frac{T_i}{D_i} \right\} \tag{9}$$

  Donde T_i y D_i son el tiempo computacional acumulado de todos los nodos y el número de registros en el bucle L_i respectivamente. En diseños feedback, el IPB es la cota máxima que un diseñador puede alcanzar utilizando únicamente retiming y sin recurrir a transformaciones pipeline más complejas.

5.2 Ejemplo

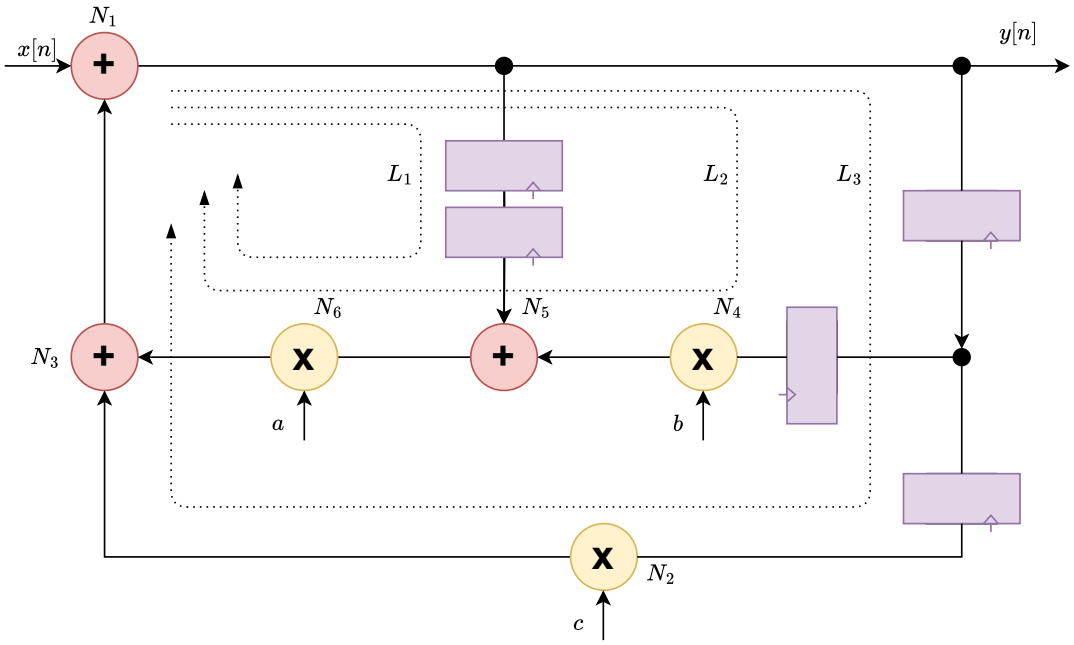


Figure 27: Grafo del flujo de la señal con 3 bucles. El bucle crítico L_2 tiene un un límite de bucle máximo de 3.5 unidades de tiempo

En el circuito de la figura 27 tenemos 3 bucles, L_1, L_2 y L_3. Asumimos que la multiplicación y la suma, respectivamente toman 2 y 1 unidades de tiempo. Luego los límites de bucles (LBs) son:

$$\begin{aligned} LB_1 &= \frac{T_1}{D_1} = \frac{1+1+2+1}{2} = 2.5 \\ LB_2 &= \frac{T_2}{D_2} = \frac{1+2+1+2+1}{2} = 3.5 \\ LB_3 &= \frac{T_3}{D_3} = \frac{1+2+1}{2} = 2 \end{aligned} \tag{10}$$

L_2 es el bucle crítico ya que tiene el máximo límite de bucle (loop bound). Esto es $IPB = max\{2.5, 3.5, 2\} = 3.5$. Entonces el camino más largo o *critical path* es $N_4 \rightarrow N_5 \rightarrow N_6 \rightarrow N_3 \rightarrow N_1$ y su delay es de 7 u.t.

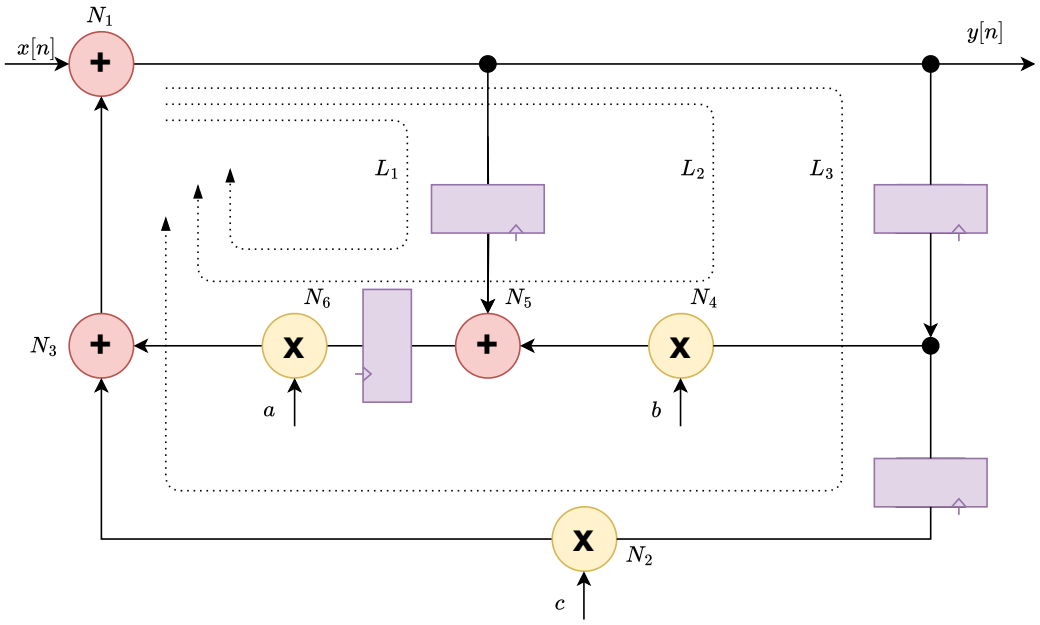


Figure 28: Retime: critical path $(N_6 + N_3 + N_1) = 4$

El valor $IPB = 3.5 < 4$ indica que hay potencial para mejorar este path utilizando retiming pero esto puede involucrar realizar *fine-grain retiming*, esto es empezar a mover los registros dentro de las unidades computacionales, lo cual es una operatoria más compleja y debe evaluarse el trade-off entre presentar una solución sub-óptima y el rediseño que involucre seguir avanzando en la optimización.

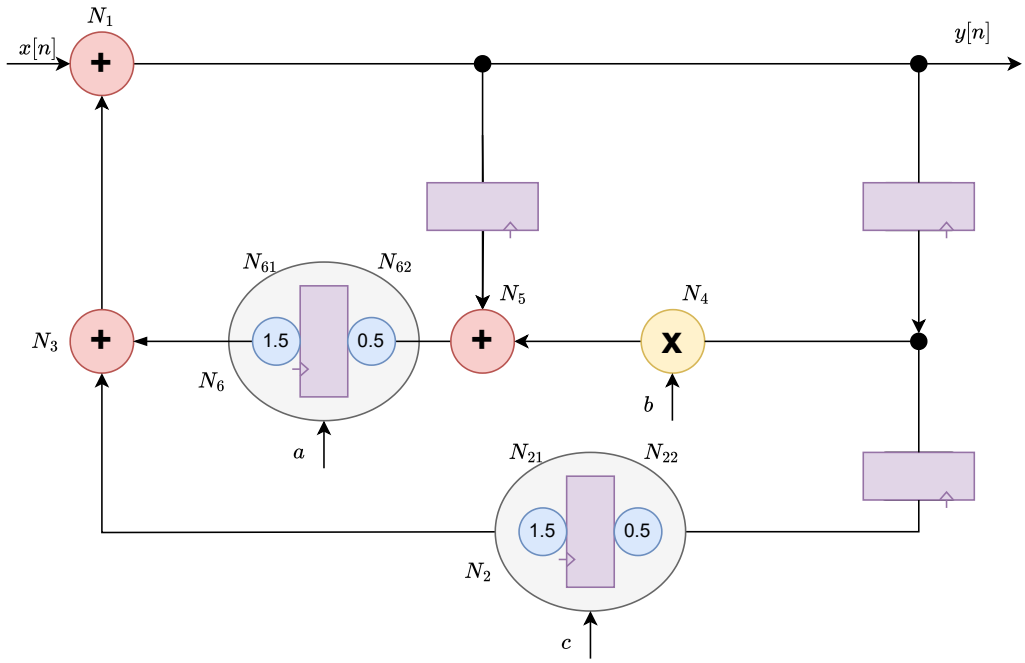


Figure 29: Fine Grain Retimed: Critical delay path = 3.5 = IPB

6 Unfolding

Técnica de transformación que permite que múltiples iteraciones de un bucle puedan ser ejecutadas en una única iteración. En el contexto de diseño de HW, unfolding es aplicar una transformación matemática a un grafo de flujo de datos para replicar su funcionalidad para calcular múltiples muestras de salida dadas múltiples muestras de entrada. Unfolding es **procesamiento en paralelo** y también suele ser llamada **loop unrolling** en SW.

```
sum = 0;
for (i = 0; i<N; i++)
    sum += a[i] * b[i];
```

Listing 1: Ejemplo: Dot Product

Veamos en que consiste esta técnica aplicandola a un simple dot-product entre arreglos de tamaño N. Para calcular cada operación MAC (multiply and accumulation) el programa necesita ejecutar una serie de ciclos para mantener el loop, esto se conoce como *loop overhead*. Este overhead incluye incrementar la variable i, comparar el valor incrementado con N y decidir si ejecutar o no la siguiente iteración.
Para mitigar esto, el loop puede ser *unrolled* en un factor de J.

```
J = 4;
sum1 = 0;
sum2 = 0;
sum3 = 0;
sum4 = 0;

for (i = 0; i<N; i=i+J) {
    sum1 += a[i] * b[i];
    sum2 += a[i+1] * b[i+1];
    sum3 += a[i+2] * b[i+2];
    sum4 += a[i+3] * b[i+3];
}
sum = sum1 + sum2 + sum3 + sum4;
```

Listing 2: Ejemplo: Dot Product

6.1 HW: Transformación Unfolding

Todo DFG puede ser unfolded por un factor J usando los siguientes 2 pasos:

1. Cada nodo U del DFG Original se replica J veces, es decir tenemos $U_0, \cdots, U_{J-1}$.
2. Para 2 nodos conectados U y V que tienen w delays, se dibujan J aristas tales que cada arista j para $j = 0, 1, \cdots, J-1$ connectan al nodo U_j con el nodo $V(j+w)\%J$ utilizando $\frac{\lfloor (j+w) \rfloor}{J}$ delays.

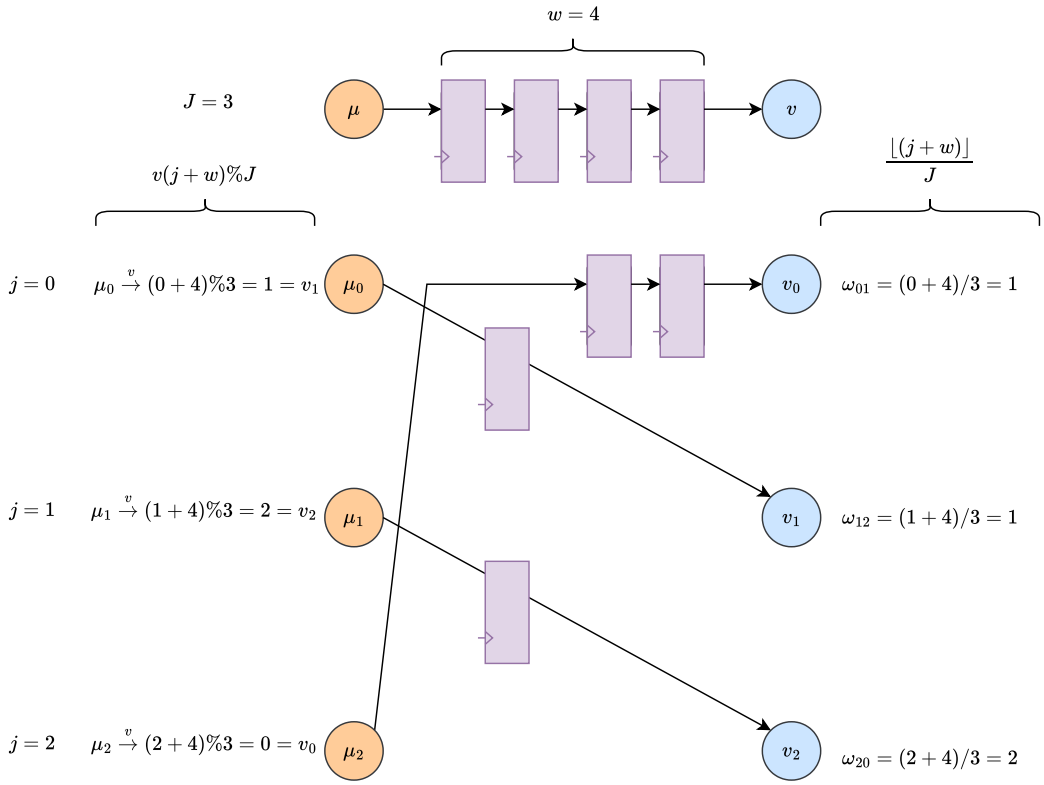


Figure 30: HW: Unfolding example

- Hacer J copias de cada nodo incrementa el área, pero el número de delays se mantiene igual que el DFG original.
- Esto aumenta el *critical path delay* y el *IPB* para un DFG recursivo, por un factor de J.

6.2 Unfolding: Ejemplo 2

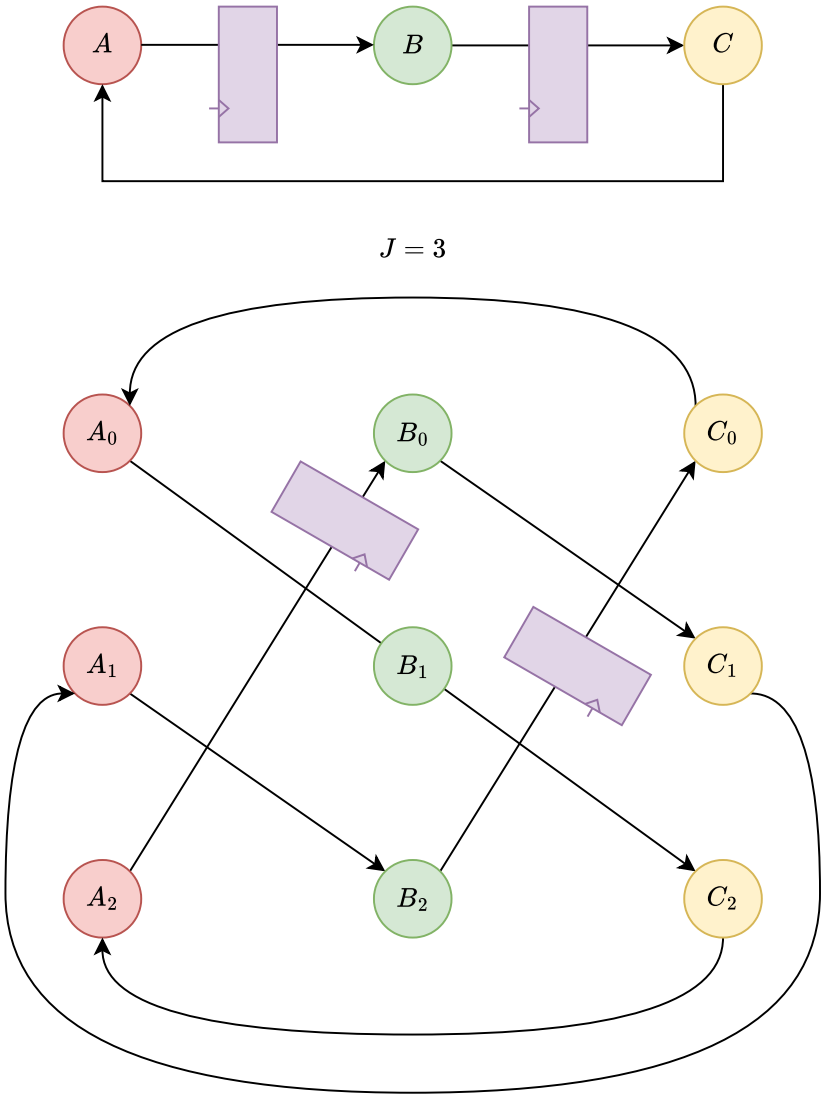


Figure 31: HW: Unfolding, ejemplo 2

6.3 Unfolding: Ejemplo 3

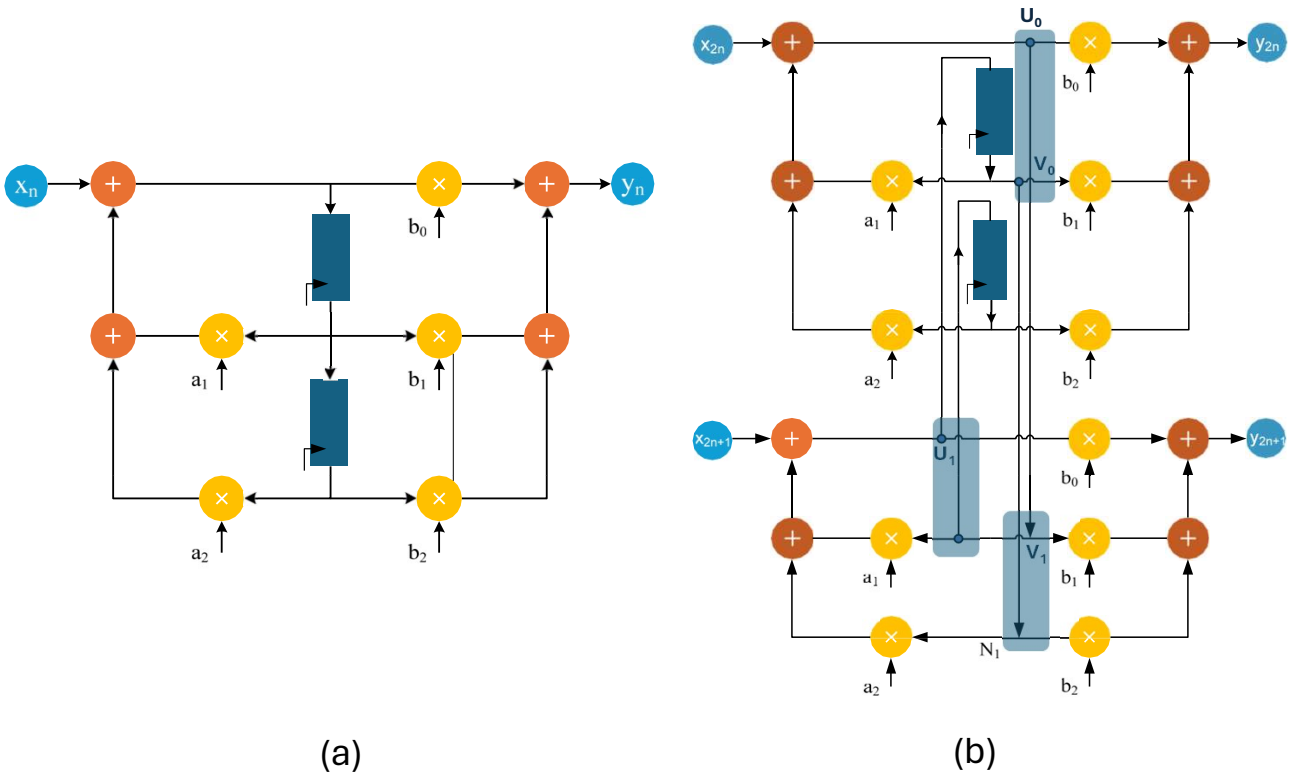


Figure 32: Transformaciones Unfolding. (a) Estructura TDF de segundo orden. (b) Unfolding usando un factor $J = 2$.

7 Folding

- Las arquitecturas *time-shared* son diseñadas si el *clock* es al menos el doble del *sampling clock*. Es decir: $f_c > 2f_s$.
- El diseño puede reusar sus recursos de hardware.
  - El dato de entrada permanece válido durante un número de múltiplos de f_c. Es decir, tengo varios clocks para procesarlo.
- Algunas definiciones básicas:
  - **Folding**: Técnica matemática que consisten en encontrar una arquitectura multiplexada en el tiempo y un schedule que dice como mapear múltiples operaciones de un DFG en las mismas unidades computacionales.
  - **Folding Factor**: Define el máximo numero de operaciones en un DFG mapeados a una unidad computacional *shareada*.
  - **Folding set o Folding Scheduler**: Es la sequencia de operaciones de un DFG mapeados a una única unidad computacional.

La transformación tiene 2 partes: La primera trata sobre encontrar un *folding factor* y el scheduling de operaciones a realizar sobre las unidades computacionales shareadas. El factor de folding óptimo es:

$$N = \left\lfloor \frac{f_c}{f_s} \right\rfloor \tag{11}$$

Donde f_c y f_s son las frecuencias de clock y de sampleo del sistema. Basado en este factor y en el scheduling, la arquitectura debe guardar resultados intermedios.
El folding de un sistema por un factor N introduce una latencia de N ciclos en el sistema. El scheduling para realizar el control de las operaciones puede ser simple como para ser implementado mediante un contador o puede basarse en una FSM.

7.1 Ejemplo: Folding

El DFG tiene 2 nodos (A_1 y A_2) realizando una suma y 2 nodos (M_1 y M_2) realizando multiplicación. Asumiendo que el dato de entrada es válido por 2 ciclos de reloj, podemos "*foldear*" el DFG por un factor de 2, utilizando unicamente un sumador y un multiplicador.
El mapeo del algoritmo en la arquitectura requiere un schedule para saber cuando realizar cada operación (en este caso particular, muy sencillo). Realizamos las operaciones en el ordenA_1 A_2 y M_1 M_2 y los multiplexores seleccionan el puerto 0 para el primer ciclo.

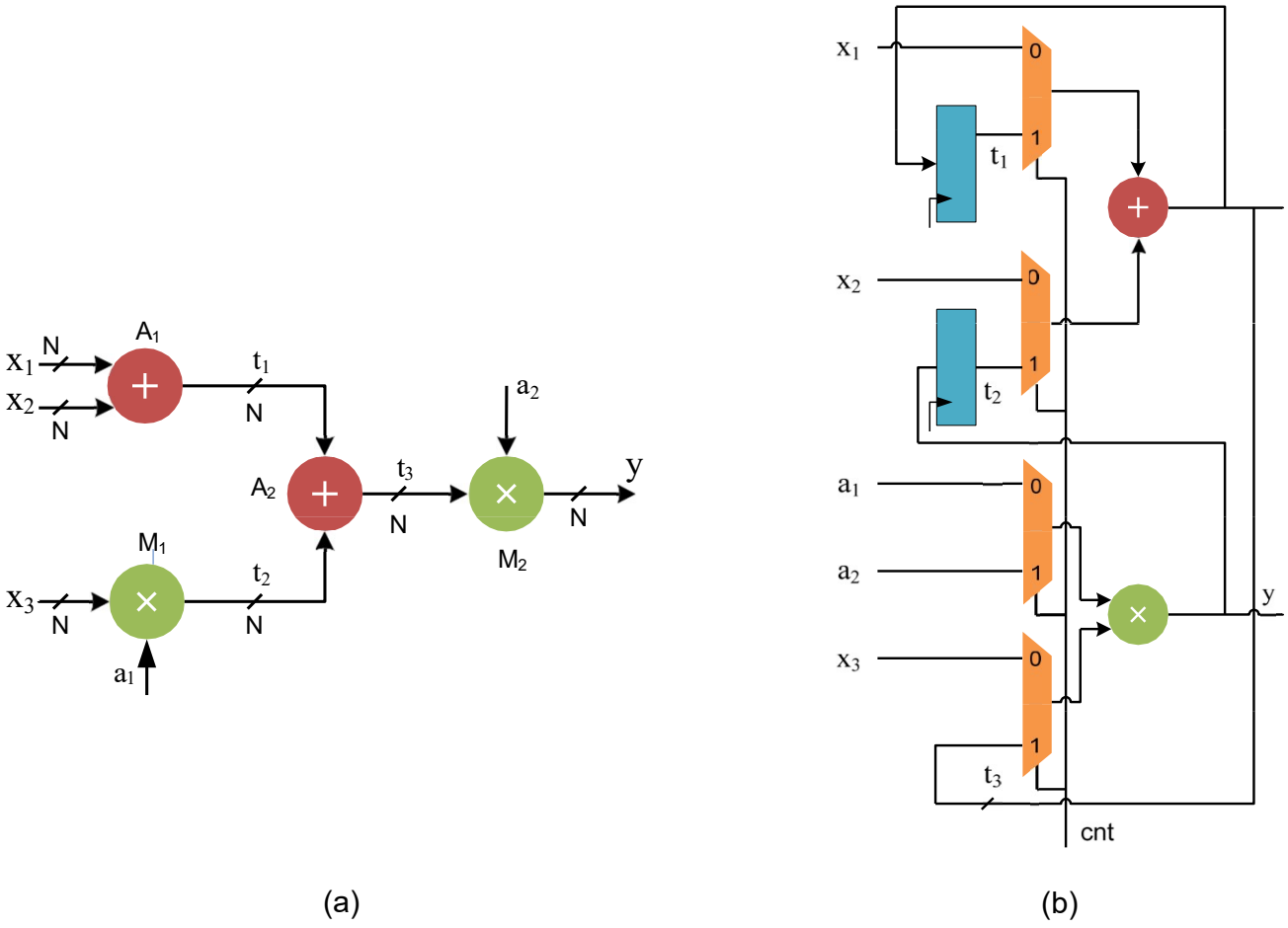


Figure 33: DFG Folding. (a) Grafo de flujo de datos ejemplo. (b) Arquitectura foldeada

8 Resource Sharing: Example

- Supongamos que tenemos un sistema que cuenta con 4 canales, A,B,C y D y llega un valor calculado x periodicamente.
- Asumimos en este ejemplo que los samples x están lo suficientemente espaciados como para poder armar un slot compartido y ubicar múltiples operaciones entre samples.
- Se busca calcular por cada canal la siguiente expresión polinómica: $a_2x^2 + a_1x^1 + a_0$, y lo hacemos en 2 steps, como muestra la siguiente tabla:

step	op	a	b	c	result
1	$out_1 = a_2x + a_1$	x	a_2	a_1	$a_2x + a_1$
2	$out_2 = out_1x + a_0$	out_1	x	a_0	$a_2x_2 + a_1x + a_0$

- Para poder compartir el *multiply and add* entre los 4 canales, se crea una tabla de *Resource Allocation* donde se especifica en cada momento temporal, que operación se está realizando.
- Esto nos permite organizar temporalmente las operaciones que se realizan sobre un recurso compartido.

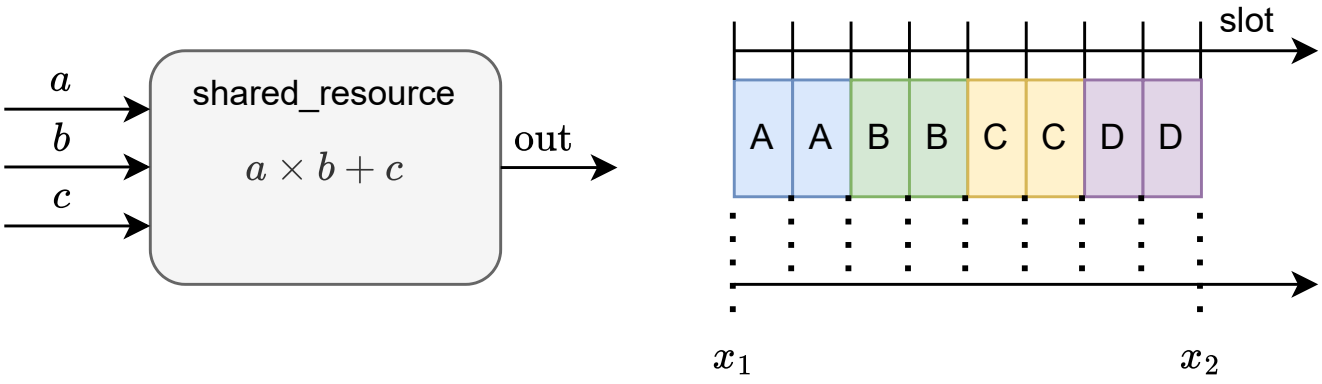


Figure 34: Resource Sharing Example