Módulo 7:

Alternativas para implementación de circuitos típicos

Contenidos del módulo 7

- Circuitos aritméticos (*carry save, carry select, carry look ahead*, aritmética serial, contadores seriales).
- Aplicación de técnicas de *pipeline*.

Análisis de circuitos aritméticos en FPGAs

- Dentro de los bloques funcionales usuales, los circuitos aritméticos relacionados a la suma conforman un grupo muy usado
  - Un sumador es básico en etapas de cálculo, y si se agrega memoria es la base de un acumulador
  - Un negador es una forma de complemento a ‘1’ y suma
  - Un restador es un sumador donde el sustraendo es negado
  - Un incrementador/decrementador es una variación de suma y resta.
  - Y un contador binario es un incrementador/decrementador registrado
- Un bloque aritmético normalmente requiere más de una macrocelda, y por ello la optimización de las conexiones y tiempos de propagación se hace crítica, más cuantos más bits tiene el campo numérico.
- Todos los fabricantes de FPGA ofrecen algunos “atajos” especiales, que aprovechan conexiones dedicadas o modos de operación especiales, para hacer circuitos más eficientes.

Análisis de circuitos aritméticos en FPGAs

- Dentro de los bloques funcionales usuales, los circuitos aritméticos relacionados a la multiplicación han tomado cada vez más vigencia debido a la incorporación de funciones DSP en aplicaciones de comunicaciones y control.
- Las soluciones habituales puede ser
  - Totalmente paralelas: caso de los multipicadores embebidos de las FPGA o de las CPU de los DSP. Es un importante problema de microelectrónica.
  - Paralelo/secuencial (algorítmicas): mediante algoritmos de shift y suma
  - Combinadas con el uso de tablas: aprovechando el uso de bloques de RAM en modo ROM para almacenar productos precalculados, muy útil en aplicaciones de DSP
  - Seriales: caso algorítmico donde la suma y shift se simplifica al resolverse de a un bit a la vez.
- En todos los casos debe considerarse la multiplicación con y sin signo

Explotando los dominios velocidad y silicio

- Al seleccionar una FPGA para comenzar un diseño muchas personas consideran como parámetro prioritario sólo la cantidad de macroceldas disponibles
- Y suelen considerar la velocidad de operación como un parámetro secundario, de importancia sólo para alcanzar la máxima frecuencia deseable

Esto es un error

- El análisis de la velocidad alcanzable en una FPGA debe ser tenido en cuenta al comenzar el diseño, porque la arquitectura a elegir depende de la frecuencia a la que debe operar cada bloque en relación a la obtenible con esa FPGA
- Por ejemplo, si un bloque de un diseño debe funcionar a 10MHz y la FPGA puede operar a 160MHz, se disponen 16 ciclos de reloj para realizar esa tarea

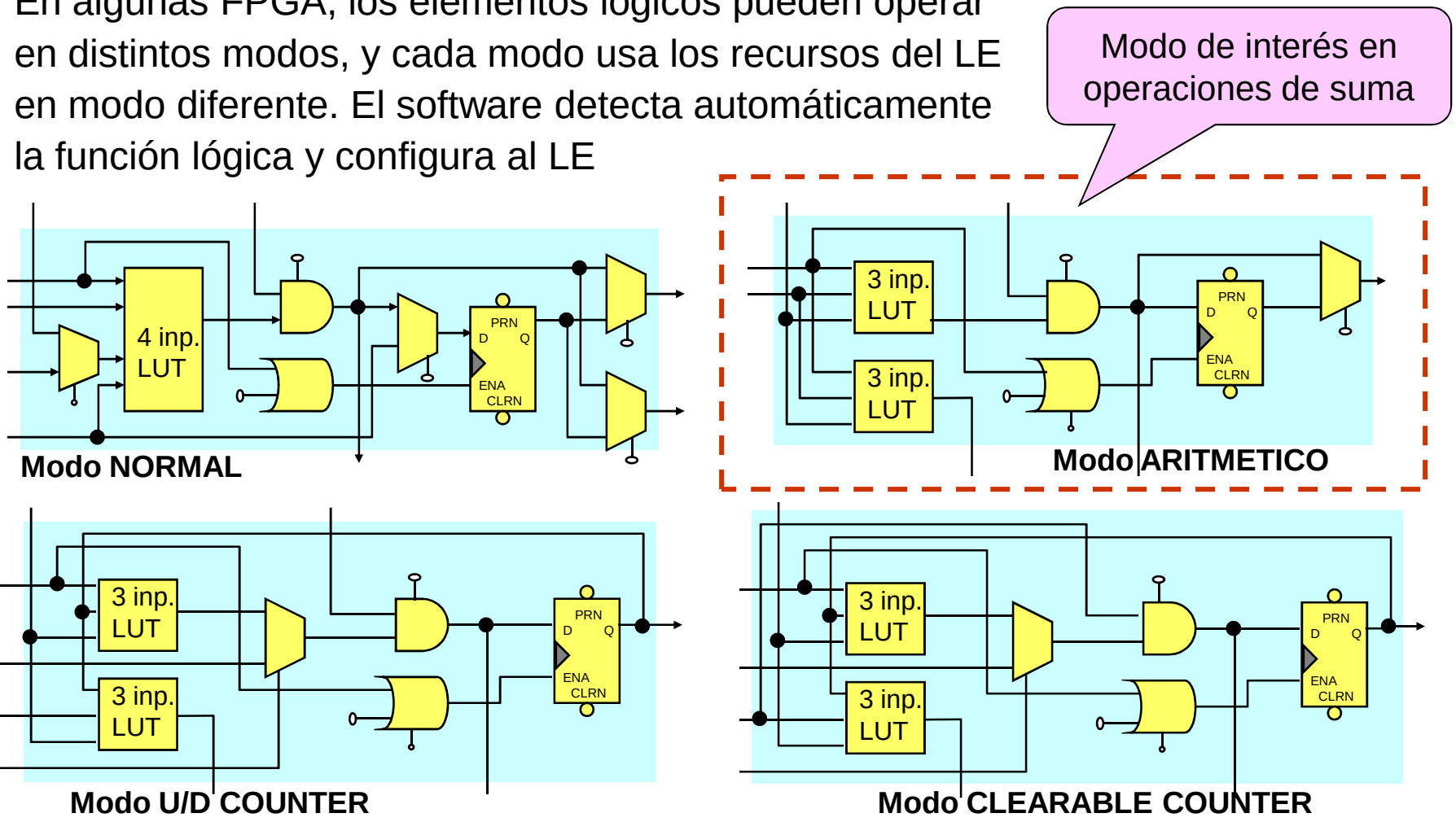
Análisis de circuitos aritméticos de suma en FPGAs

Esquemas básicos de suma:

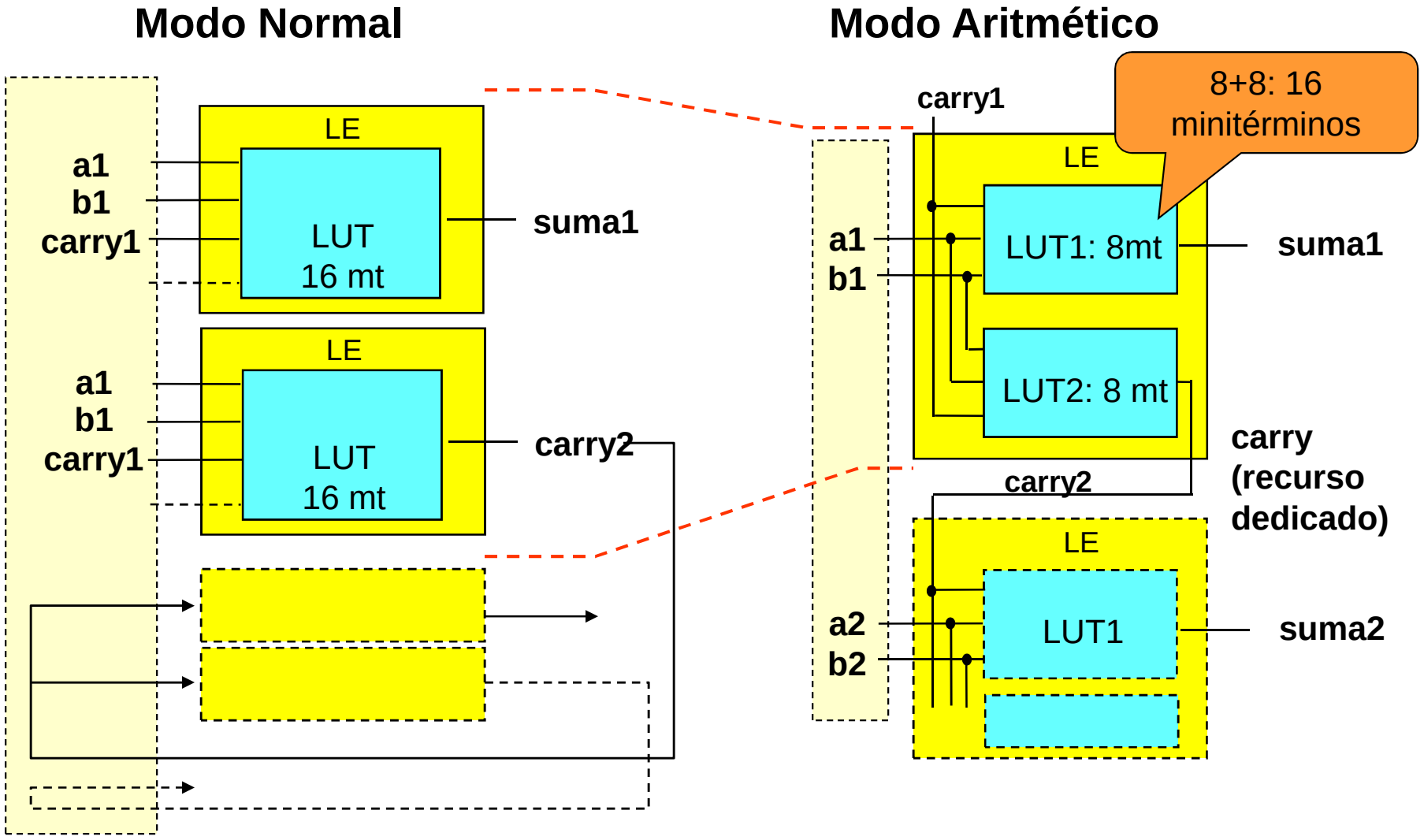
- Optimizaciones embebidas: las cadenas de Carry y el caso de los “modos de operación” de ciertas FPGA
- Suma y resta con:
  - Ripple carry
  - Carry select: a nivel de macrocelda (optimización embebida) y a nivel de bloque funcional
  - Carry look ahead
  - Carry save
- Aritmética Serial:
  - sumador
  - sumador/restador
  - complementador

Los modos de operación de algunas FPGA de ALTERA

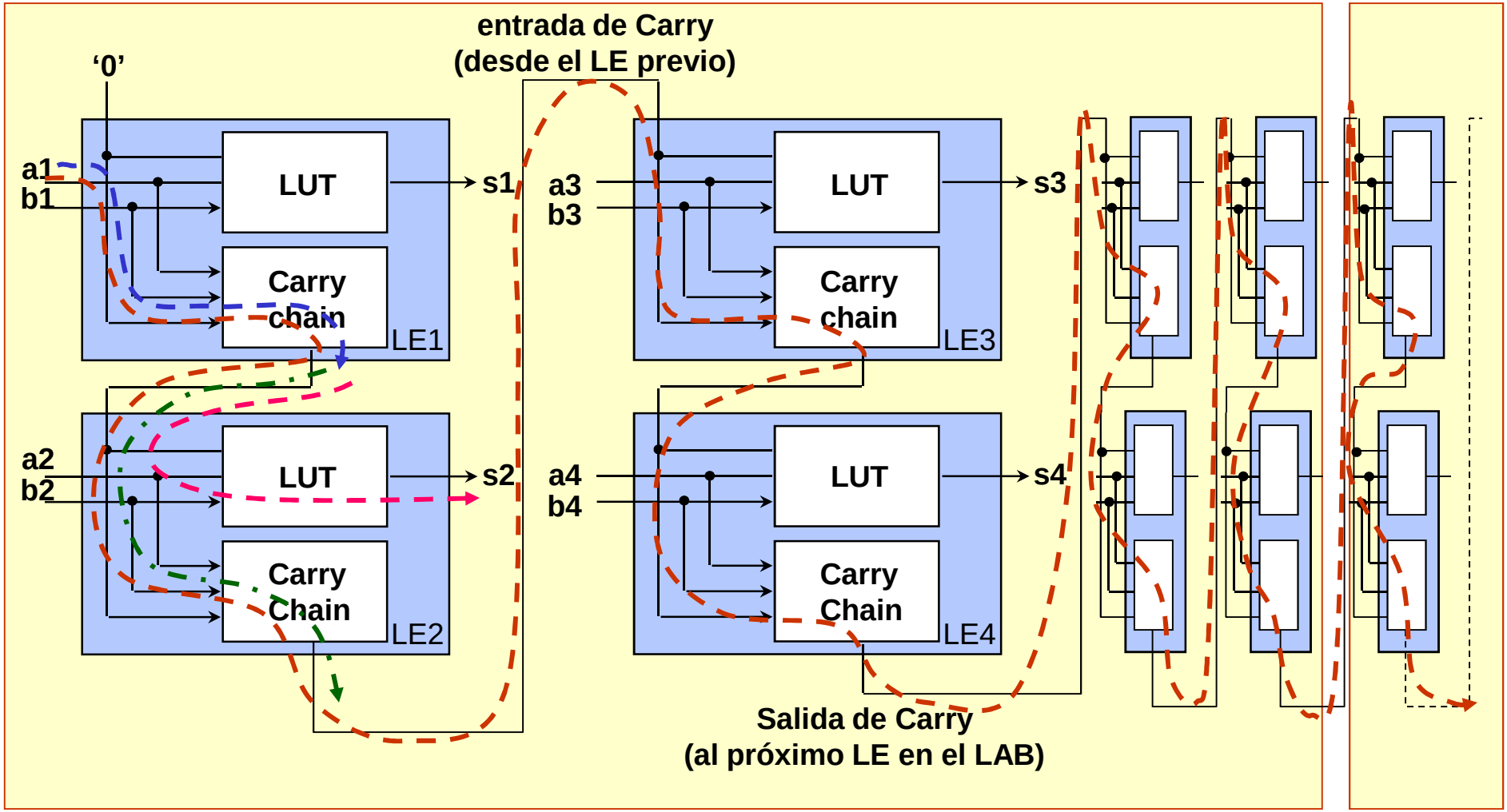
- En algunas FPGA, los elementos lógicos pueden operar en distintos modos, y cada modo usa los recursos del LE en modo diferente. El software detecta automáticamente la función lógica y configura al LE



Modo Normal versus Aritmético



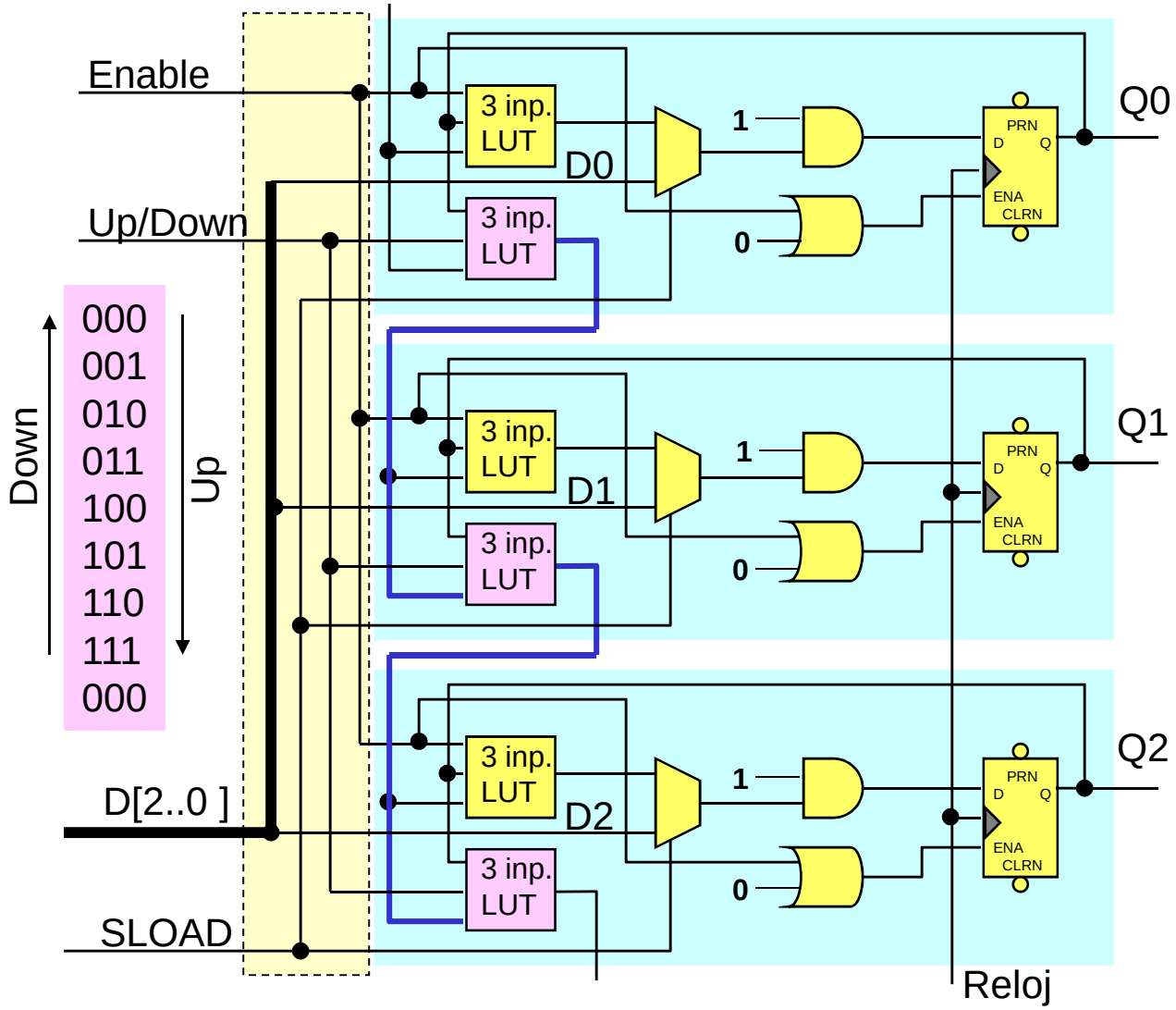
Cadena de CARRY: topología de conexionado



Contador U/D programable usando Carry Chain

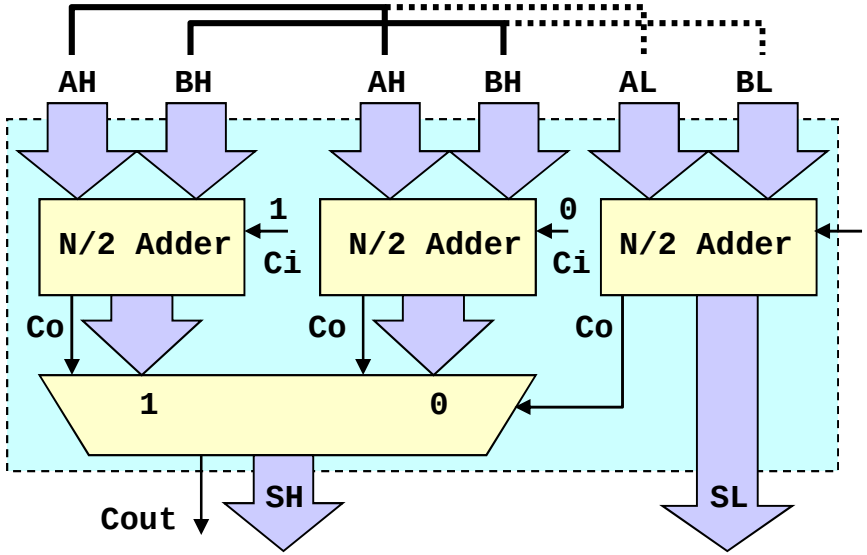
- Para implementar contadores, se emplea feedback desde el registro del LE, la cadena de Carry, un multiplexor disponible a la salida de la LUT, y señales de control independientes (Enable e Up/Down)

Analizar la función lógica necesaria para cada sub-LUT



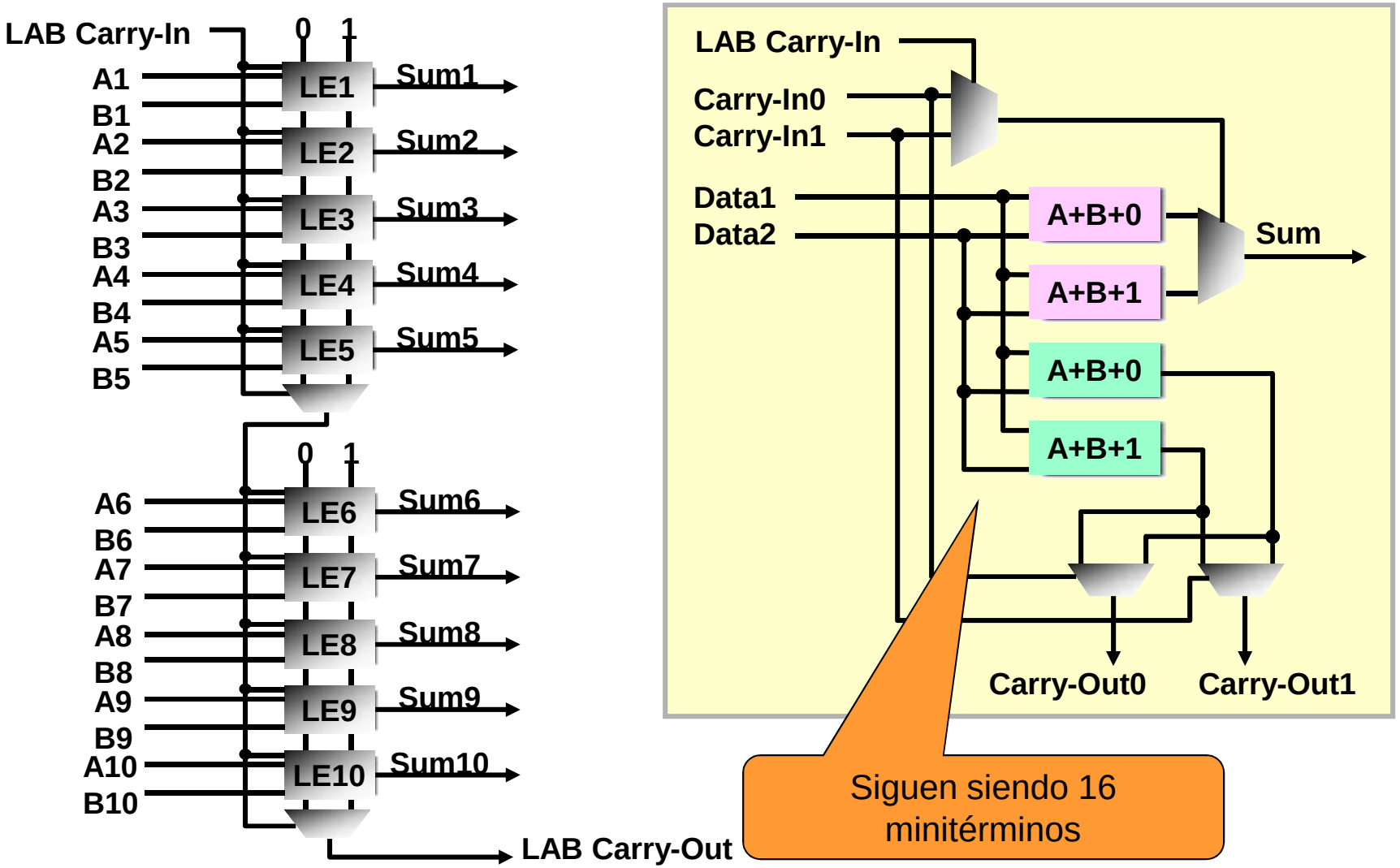
Optimización de la suma. Un ejemplo de “Carry-select” por diseño

```
LIBRARY ieee;USE ieee.std_logic_1164.ALL;
LIBRARY ieee;USE ieee.std_logic_unsigned.ALL;
ENTITY cysel64 IS
PORT (a,b:IN std_logic_vector(64 DOWNTO 1);
      s: OUT std_logic_vector(64 DOWNTO 1);
      cin:IN std_logic; cout:OUT std_logic);
END cysel64;

ARCHITECTURE a OF cysel64 IS
SIGNAL slo,shi0,shi1:
  std_logic_vector(33 DOWNTO 1);
BEGIN
 slo  <= '0'& a(32 DOWNTO 1)
           + b(32 DOWNTO 1) + cin;
 shi0 <= '0'& a(64 DOWNTO 33)
           + b(64 DOWNTO 33);
 shi1 <= '0'& a(64 DOWNTO 33)
           + b(64 DOWNTO 33) + '1';

 s(32 DOWNTO 1) <= slo(32 DOWNTO 1);
 s(64 DOWNTO 33) <=
    shi0 (32 DOWNTO 1) WHEN slo(33)='0'
    ELSE shi1 (32 DOWNTO 1);
 cout <= shi0 (33) WHEN slo(33)='0'
    ELSE shi1 (33);
END a;
```

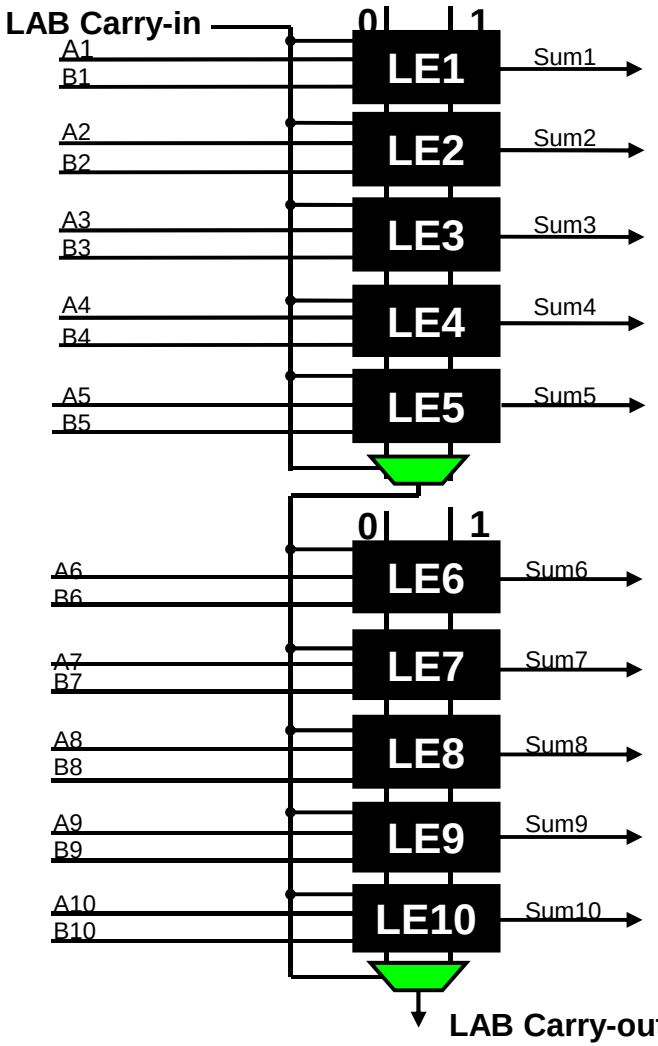


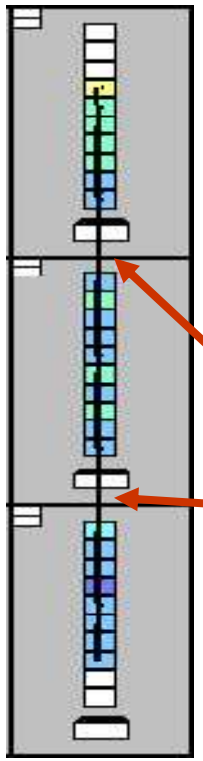
- La idea de Carry-select es simple:
- Usa 3 sumadores (***slo***, ***shi0***, y ***shi1***) de N/2 bits:
  - uno para la mitad inferior de los sumandos
  - dos para la parte superior, donde en uno de éstos se supone que el carry de entrada será ‘0’ (***shi0***) y en el otro que será ‘1’(***shi1***).
- En función de la salida de carry de ***slo*** se opta entre los posibles resultados ***shi0*** y ***shi1***
- Consume el doble de recursos (132 vs 66 LEs), pero también casi duplica la velocidad

Alternativa para la cadena de Carry: Carry Select dentro de la FPGA



Detalles de la cadena de carry usando Carry Select

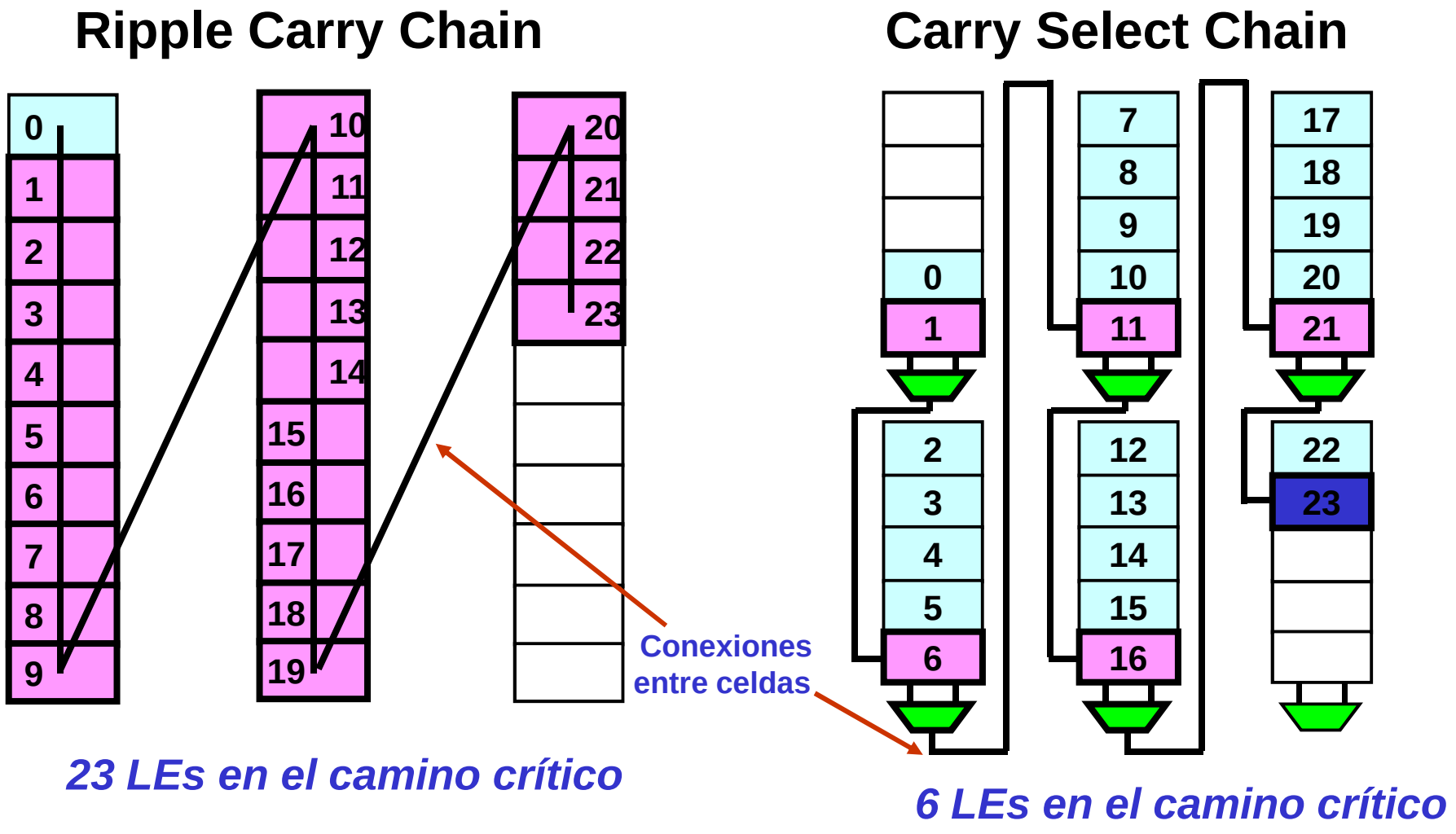




- las cadenas de Carry comienzan y terminan en cada LE
- existen 2 cadenas de Carry en cada LAB
- la señal Carry-Select se genera en los LE 5 & 10
  - Cada LE no aparece en el camino crítico de retardos
- La cadena de Carry se propaga entre LAB sucesivos verticalmente
  - la salida CarryOut del LE10 excita la entrada CarryIn de los LEs 1-5 del LAB siguiente

Cadena vertical de Carry

Diferencia de longitud de la cadena de Carry: contador de 24-Bits



Sumador CarryLookAhead

El método evita el proceso de propagación de Carry. Para ello cada sumador (bloques celestes) genera 3 funciones de 3 variables:

`s=a XOR b XOR cin;` suma de las entradas a y b y el carry de entrada

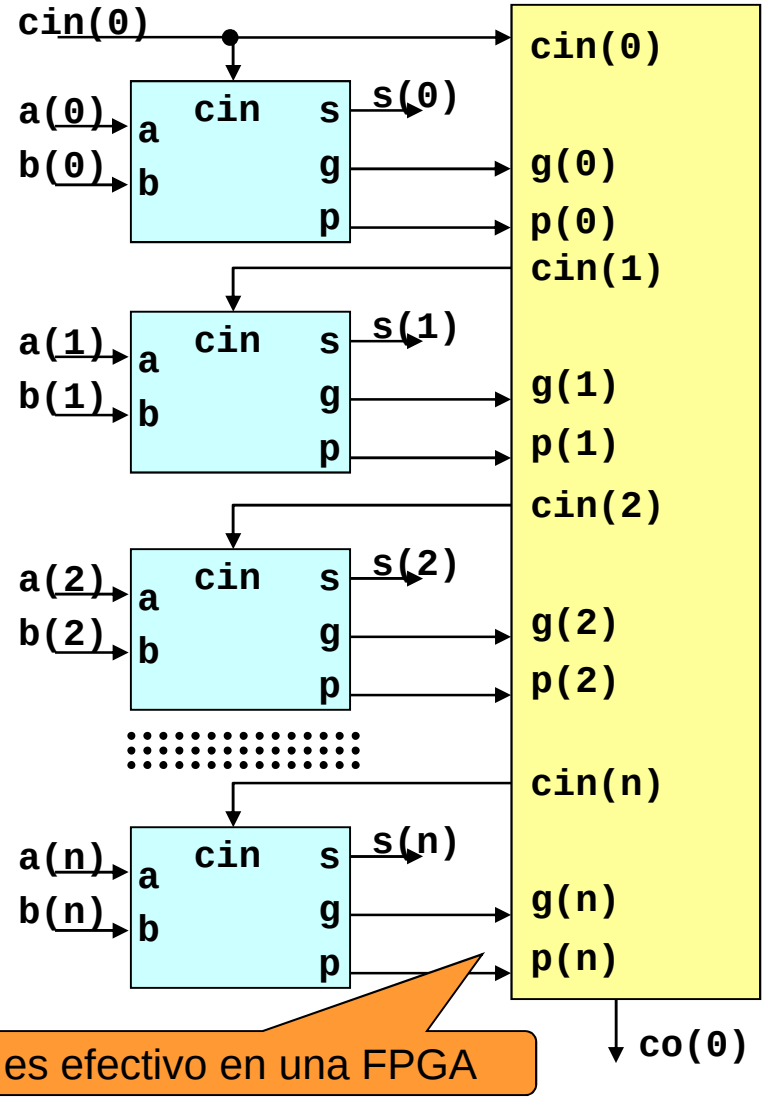
`g = a AND b;` Función *GENERATE*, vale 1 si ambas entradas valen 1

`p = a OR b;` Función *PROPAGATE*, vale 1 si alguna entrada vale 1

El Carry de entrada a una etapa puede ser calculado en base a las señales G y P de las etapas previas (bloque amarillo):

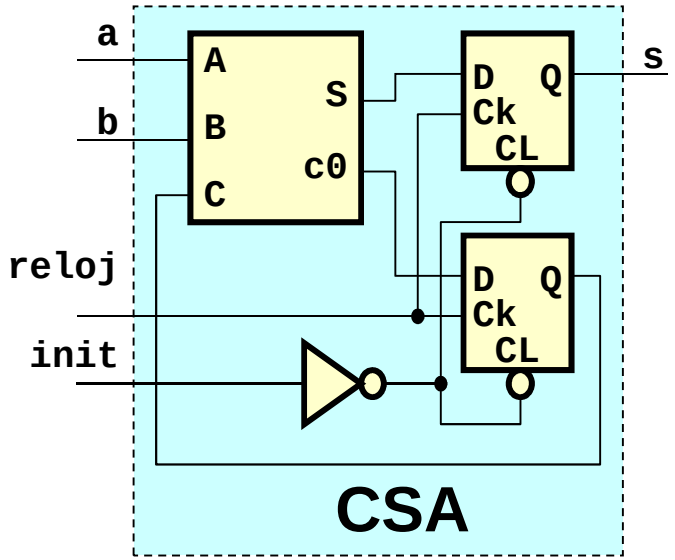
```
cin(1)=g(0)OR(p(0) AND cin(0))
cin(2)=g(1)OR
 (p(1) AND (g(0) OR (p(0)AND cin(0)))
```

etcetera.....



Ejemplos: Sumador serial "Carry Save Adder"

```
ENTITY seradd IS PORT (
 a,b,reloj,init:IN BIT; s:OUT BIT);
END seradd ;

ARCHITECTURE a OF seradd IS
SIGNAL c: BIT;
BEGIN
 PROCESS (reloj,init) BEGIN
  IF (init='1') THEN s<='0'; c<='0';
  ELSIF reloj'EVENT AND reloj='1' THEN
   s  <= a XOR b XOR c;
   c <= (a AND b) OR (a AND c) OR (b AND c);
  END IF;
 END PROCESS;
END a;
```

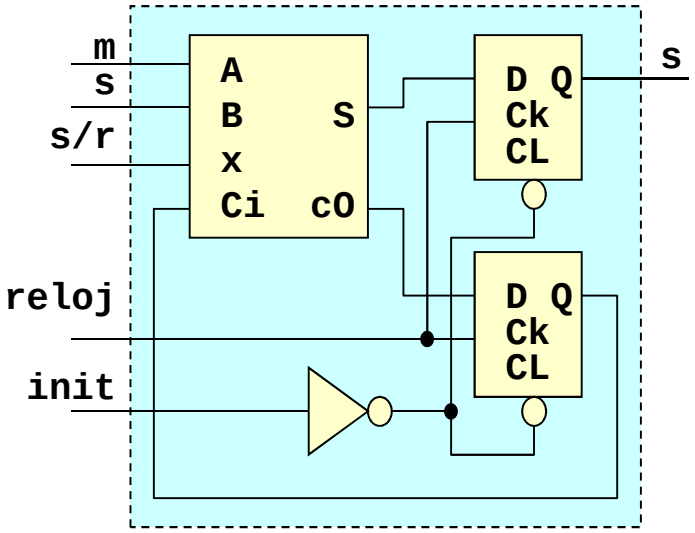


- En el caso de dos operandos a y b que ingresen simultáneamente, desde el LSB hasta el MSB, este circuito sólo requiere 2 LEs, y la latencia es de sólo 1 ciclo de reloj.
- En este circuito un LE calcula la suma de los datos de entrada más el acarreo generado en la etapa previa (salida s), y el otro LE calcula el acarreo generado en esa etapa y lo almacena ("Carry Save") para su posterior uso al calcular el bit siguiente.
- El dominio del silicio se ha intercambiado con el dominio del tiempo. Más o menos bits requieren el mismo hardware, sólo más o menos ciclos de reloj

Fuente: R.Andraka.

Ejemplo: Sumador/restador Serial

```
ENTITY seraddsub IS PORT (
 m,s,sr,reloj,init:IN BIT; s:OUT BIT);
END seraddsub;

ARCHITECTURE a OF seraddsub IS
SIGNAL c: BIT; -- carry/borrow
BEGIN
 PROCESS (reloj,init) BEGIN
  IF (init='1') THEN s<='0'; c<='0';
  ELSIF reloj'EVENT AND reloj='1' THEN
   s  <= a XOR b XOR c;
   IF sr='1'
    THEN -- sr=1 indica suma
    c <= (m AND s)
      OR (m AND c)
      OR (s AND c);
    ELSE -- sr=0 indica resta
    c <= (NOT(m) AND s)
      OR (NOT(m) AND c)
      OR (s AND c);
    END IF;
  END IF;
 END PROCESS;
END a;
```

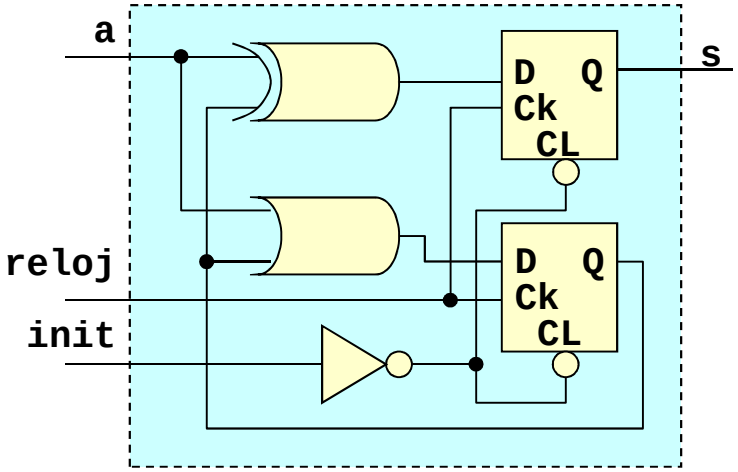


- Dado que cada LE tiene una tabla LUT de 4 entradas, ampliar el sumador serial a sumador/restador serial se hace sin consumir más recursos, pues también sólo requiere 2 LEs, y su latencia es de 1 ciclo de reloj.

Fuente: R.Andraka.

Ejemplo: Complementador serial

```
ENTITY com2dos IS PORT (
 a,reloj,init:IN BIT; s:OUT BIT);
END com2dos;

ARCHITECTURE a OF com2dos IS
SIGNAL flg: BIT;
BEGIN
 PROCESS (reloj,init) BEGIN
  IF (init='1') THEN
    s<='0'; flg <='0';
  ELSIF reloj'EVENT AND reloj='1' THEN
   s   <= a XOR flg;
   flg <= a OR flg;
  END IF;
 END PROCESS;
END a;
```



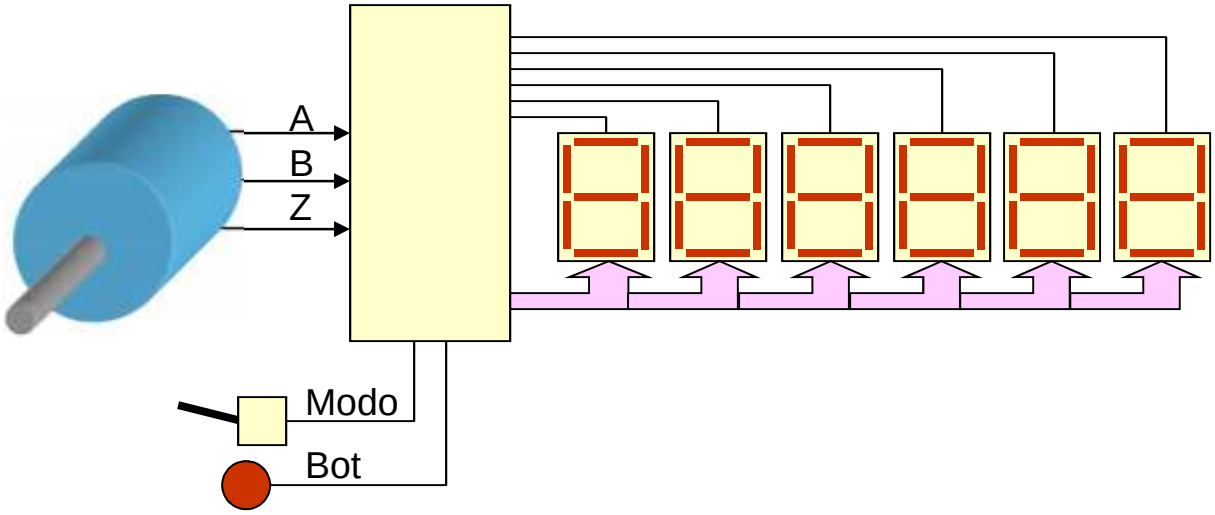
- El complementador serial implementa el algoritmo tradicional para calcular el complementar a dos de un número:
  - Desde el LSB hacia adelante, hasta encontrar el primer '1' (inclusive), la entrada es copiada en forma textual en la salida
  - A partir de allí los bits restantes se invierten

Fuente: R.Andraka.

Un ejemplo de contadores multidígito

El problema:

- Se dispone de un *Shaft Encoder* (A, B y Z) y se desea contar y mostrar en un display de hasta 6 dígitos la cantidad de pulsos por vuelta, no importa en qué sentido gire
- El sistema debe permitir que durante el giro el eje cambie de dirección
- Usa una EMP3128ATC100

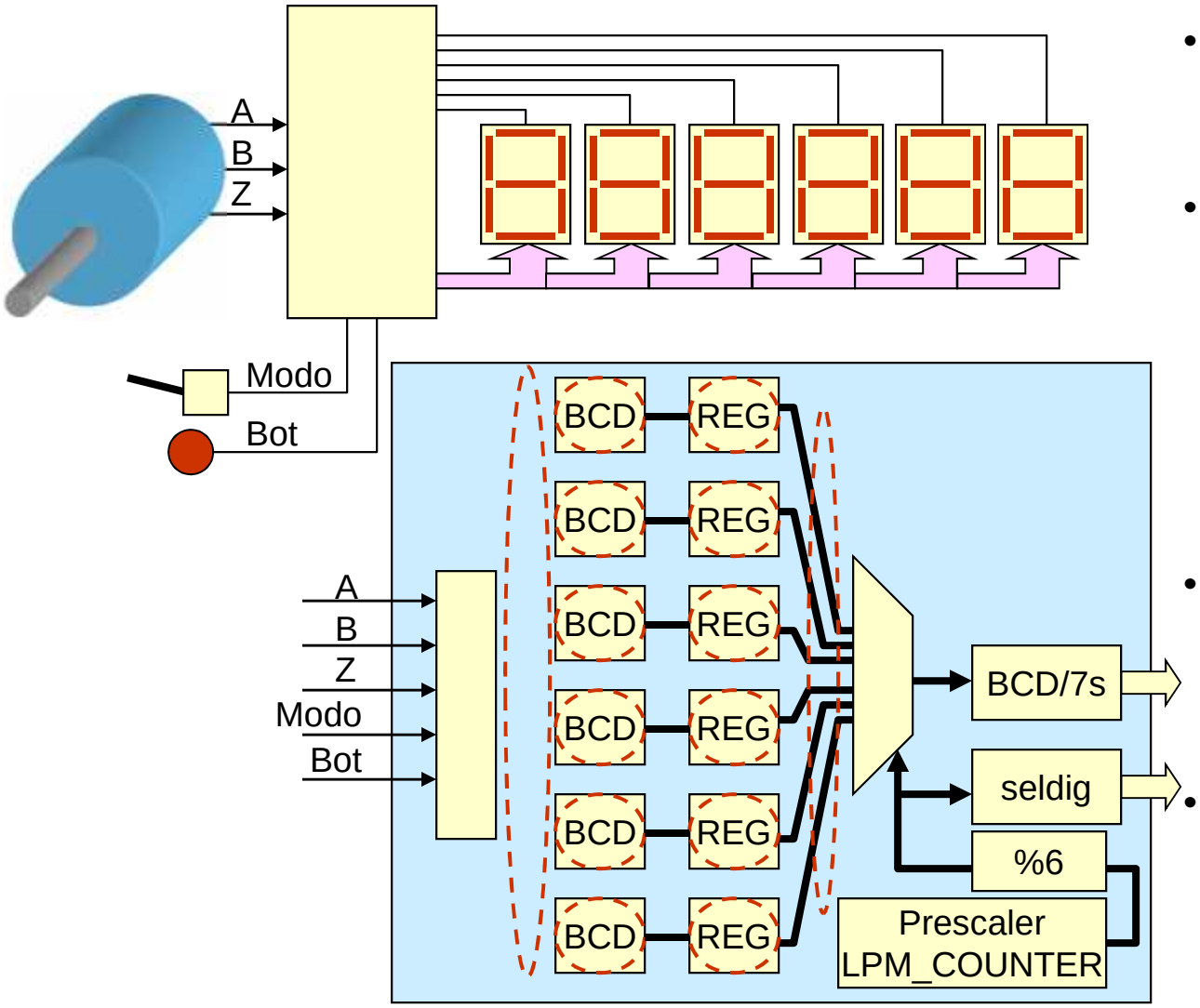


Si MODO = ’0‘, Bot no es usado y el display muestra la cantidad de pulsos de A entre pulsos de Z

Si MODO = ’1’ el pulso Z es ignorado y el display muestra los pulsos acumulados entre apretadas de Bot

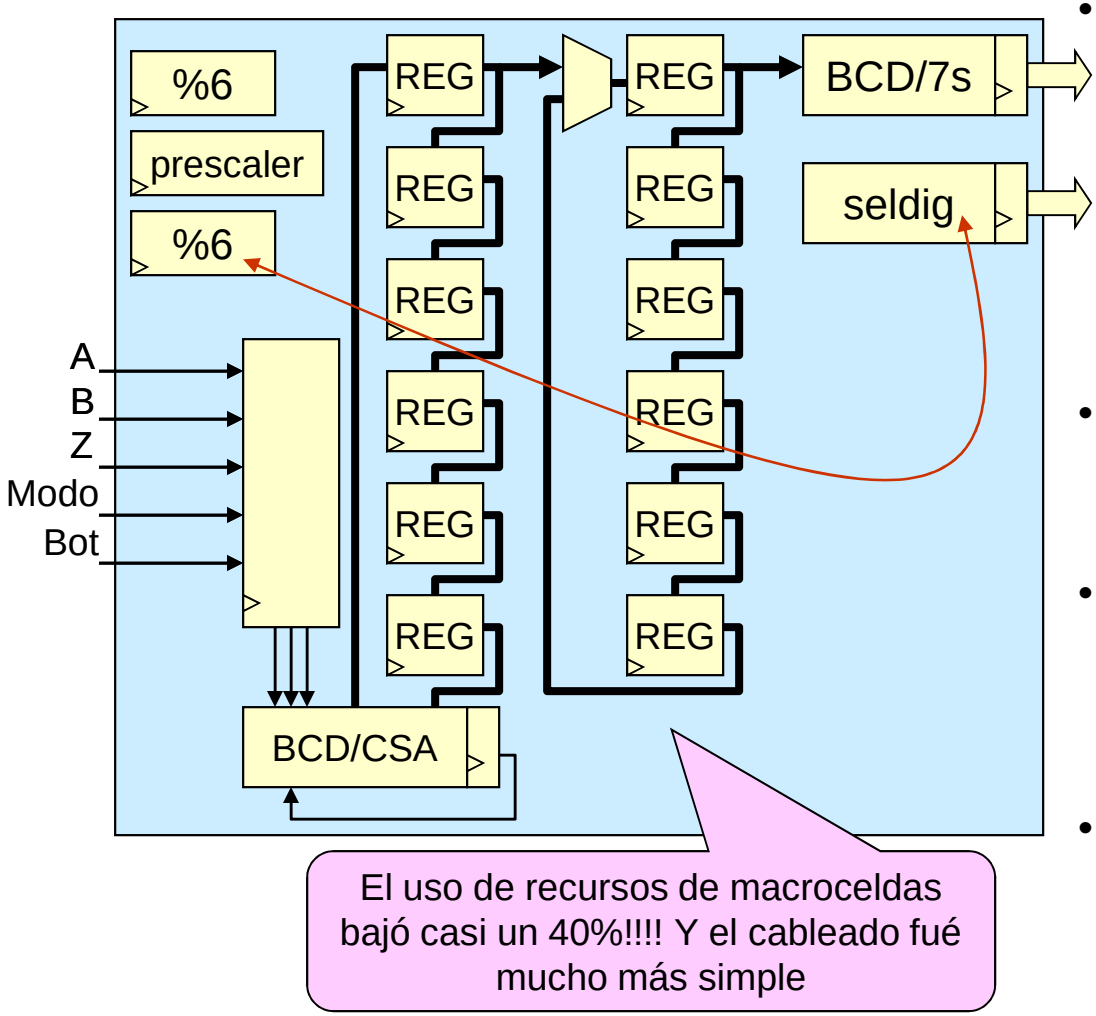
- una apretada de BOT congela el display
- otra apretada de BOT pone a cero la cuenta y muestra el display moviéndose a medida que entran pulsos

Un ejemplo de contadores multidígito: La solución obvia



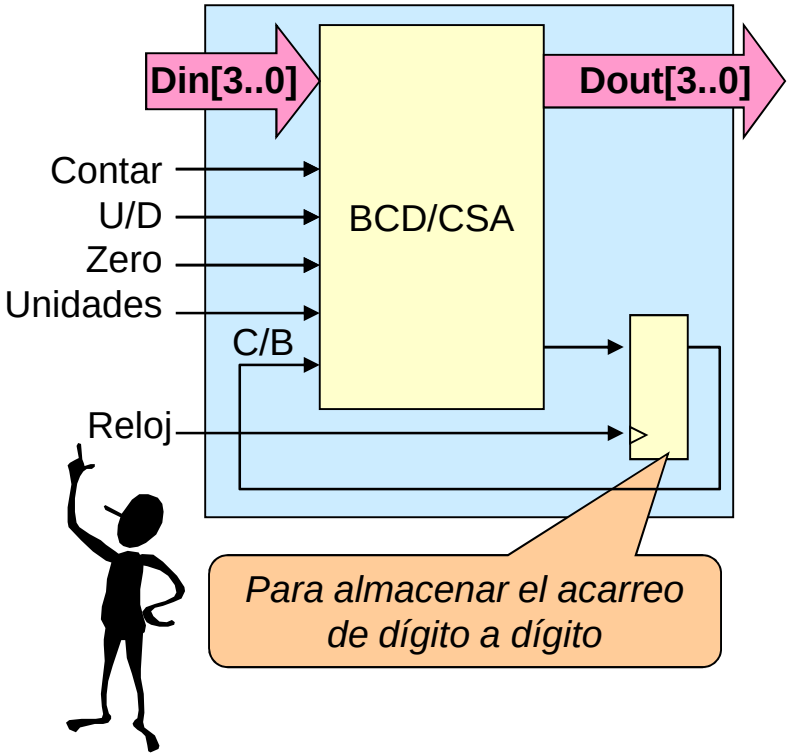
- La lógica de los contadores BCD UP/DOWN no es simple
- Y en los contadores de mayor orden requiere enables más complejos (la decena detectar el 9/0, las centenas el 99/00, los miles el 999/000, etc...)
- El multiplexor de 24 entradas y 4 salidas utiliza muchos recursos de cableado
- En una MAX los recursos de cableado no son tan abundantes

Caso de contadores multidígito
Una solución más eficiente



- Se usó una única tabla de verdad que en función del valor BCD actual, el arrastre del digito previo (o la detección de un pulso en el caso de las unidades), y el sentido de UP/DOWN define el valor BCD siguiente
- Las 6 etapas del contador son sólo 6 bloques de 4 registros, sin nada de lógica entre ellos
- El registro de salida es también un registro circular de 6 nibbles, que puede conservar su valor o copiar el de los registros de cuenta
- Seldig y el decoder BCD a 7Segmentos están registrados y sólo se actualizan cuando se pasa a mostrar otro dígito
- El prescaler es un LFSR

Caso de contadores multidígito
Una solución más eficiente

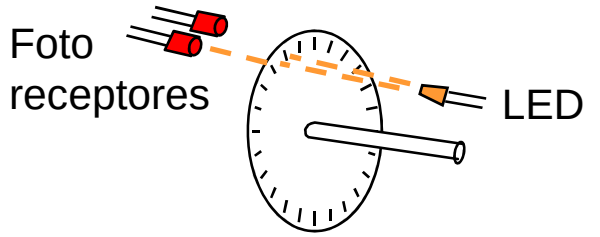


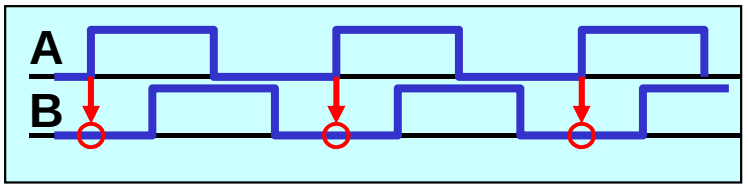
- El circuito aritmético BCD se realizó mediante el cálculo secuencial de los sucesivos dígitos, donde los 4 bits de cada dígito se evaluaron de forma paralela y combinatoria
- Todo el resto es convencional
- Como primera etapa del prescaler se usó un divisor por 6 (porque eran 6 dígitos) y la máquina de estado que procesaba las señales del encoder se procesaba una vez cada 6 ciclos de reloj. Esto es fácilmente ampliable a la cantidad de dígitos que se necesiten.
- Es un buen ejemplo de aprovechamiento de un recurso disponible en las FPGAs (su inherente elevada rapidez) para obtener soluciones más simples y más económicas

Ejemplo: Interfase a Shaft Encoder Incremental, la solución obvia

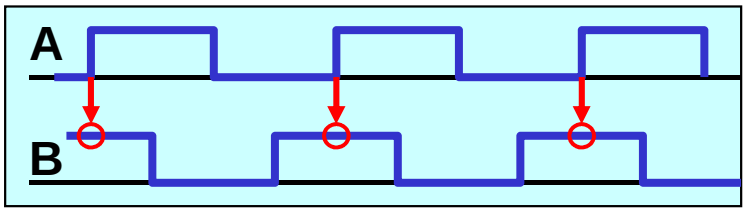
```
ENTITY shaft0 IS PORT (a,b,z:IN BIT;
 pos:OUT INTEGER RANGE 0 TO 359);
END ENTITY shaft0;
ARCHITECTURE a OF shaft0 IS BEGIN
PROCESS (a) IS
 VARIABLE cont :INTEGER RANGE -1 TO 360;
 VARIABLE sent: INTEGER RANGE -1 TO 1;
BEGIN
  IF (a ='1') THEN
   IF (b = '0') THEN sent:= 1; ELSE sent:= -1;
   END IF;
   cont := cont + sent;
   IF (z='1') OR (cont=360) THEN cont:=0; END IF;
   IF (cont=-1) THEN cont:= 359; END IF;
  END IF;
  pos <= cont;
 END PROCESS;
END ARCHITECTURE a;
```

Porqué no uso ELSIF?
Porqué no verifico que z=0 ?





Giro Horario

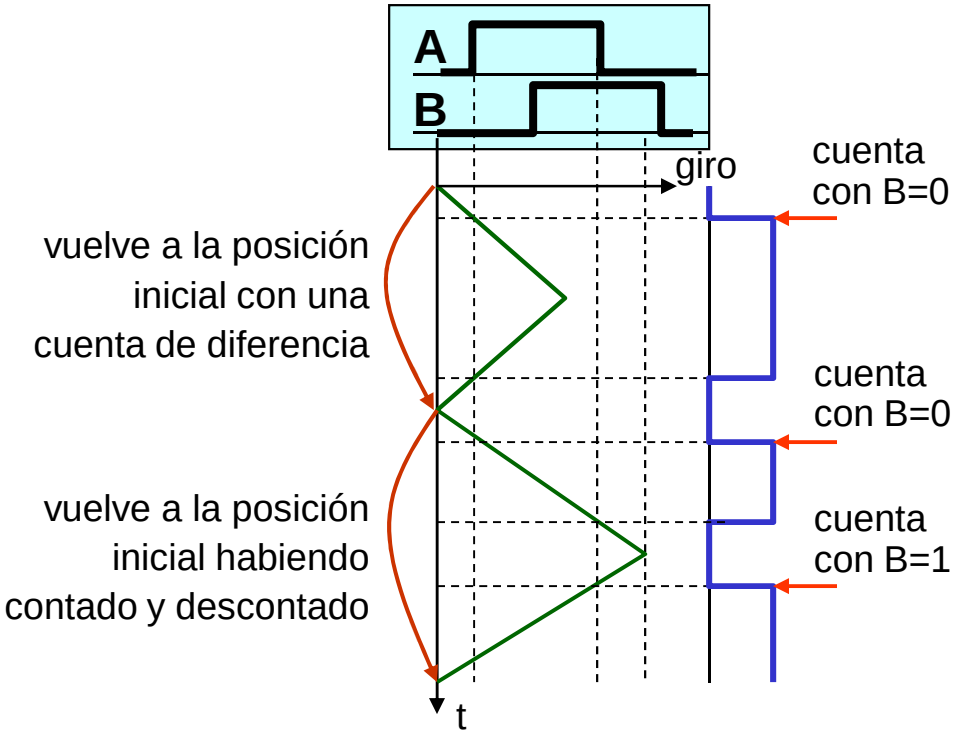


Giro Antihorario

- Es un dispositivo muy usado en aplicaciones industriales para el sensado de ángulos, y genera tres señales:
  - dos señales A y B desfasadas entre sí 1/4 de período, a razón de 360 pulsos por vuelta
  - una señal Z, de duración menor a 1/360, a razón de una por vuelta.
- La interfase más simple usa A como reloj y B como control para definir si un contador de angulo debe ser incrementado o decrementado

Ejemplo: Error de la solución obvia al cambiar el sentido de giro

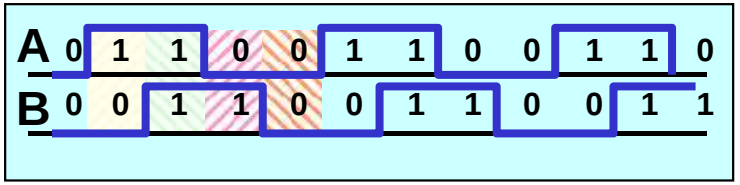
- El uso de A como reloj y B como control de UP/DOWN de un contador induce un error de pérdida de posición de un pulso en ciertos casos de cambio de giro
- En la figura se muestran dos casos en que el giro comienza y termina en idéntica posición
- En el primer caso sólo se genera un flanco creciente en A, con lo que la cuenta difiere al comienzo y al final
- En el segundo caso se generan dos flancos con valores opuestos de UP/DOWN, con lo que la cuenta inicial y final coinciden



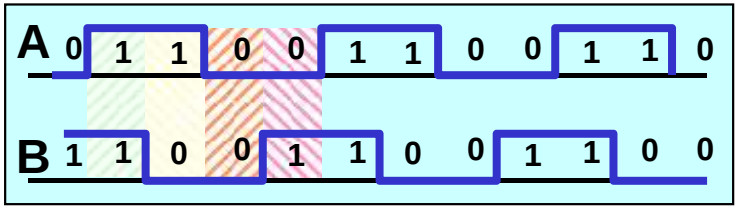
Ejemplo: Shaft Encoder, aumento 4x de la resolución (de 360 a 1440!!)

```
ENTITY shaft1 IS PORT (a,b,z,reloj:IN BIT;
 pos:OUT INTEGER RANGE 0 TO 1439);
END shaft1;
ARCHITECTURE a OF shaft1 IS BEGIN
PROCESS (reloj)
 VARIABLE cont :INTEGER RANGE -1 TO 1440;
 VARIABLE sent:INTEGER RANGE –1 to 1;
 VARIABLE stat: BIT_VECTOR (1 DOWNTO 0):= B"00";
 VARIABLE pack: BIT_VECTOR (3 DOWNTO 0);
BEGIN
 pack := stat & a & b;
 IF reloj='1' THEN
  CASE pack IS
   WHEN B"0010"|B"1011"|B"1101"|B"0100"=> sent:= 1;
   WHEN B"0111"|B"1110"|B"1000"|B"0001"=> sent:=-1;
   WHEN OTHERS=> sent := 0;
  END CASE;
  cont := cont+sent;
  stat:= a & b;
  IF (z='1') OR cont=1440 THEN cont:=0; END IF;
  IF cont=-1 THEN cont:=1439; END IF;
 END IF;
 pos <= cont;
 END PROCESS;
END a;
```

Qué cosas faltan? ...sincronización
Qué error hay todavía?



Giro Horario



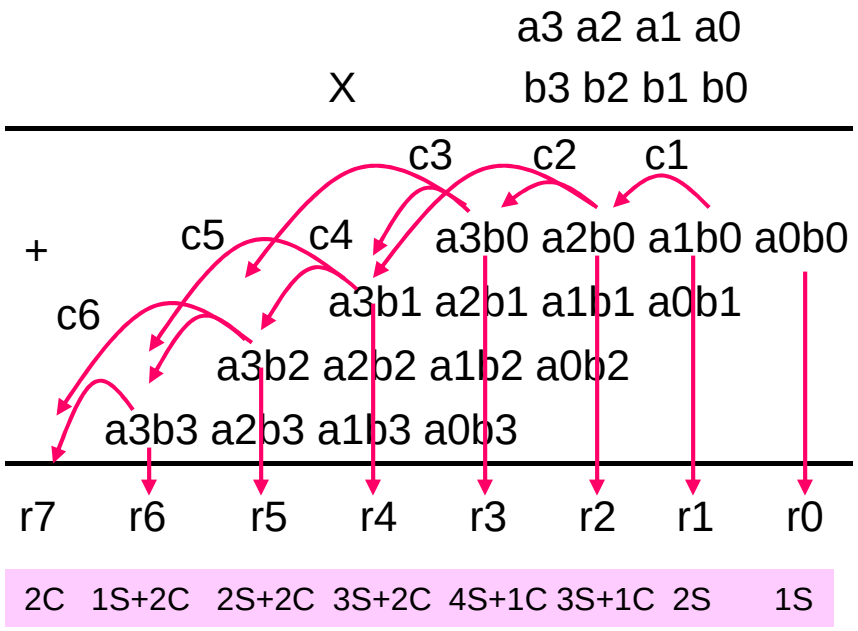
Giro Antihorario

Una alternativa que aumenta la resolución y permite un diseño sincronico con un reloj interno es observar que:

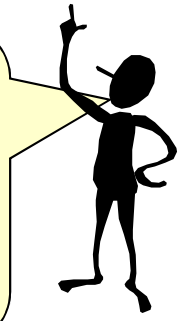
- –en sentido horario la secuencia AB es 00/10/11/01/00..
- –en sentido antihorario la secuencia AB es 00/01/11/10/00...

Multiplicadores embebidos

- Si se analiza el producto de dos palabras a[3:0] y b[3:0] y su resultado r[7:0]:
  - r0 es una simple AND
  - r1 es la suma de dos AND (4 variables)
  - c1 es también función de 4 variables
  - r2 puede verse como la suma de 4 términos (3 AND de 2 variables más C1) o como una funcíon de 6 variables
  - si r2 se calcula como sumas, c2 es el carry de salida de una suma de 4 términos (por lo que requiere 2 bits) ... o el resultado de una función de 8 variables
- Y así sucesivamente, cada resultado más complejo, en este caso para una “simple” multiplicación de 2 palabras de 4 bits

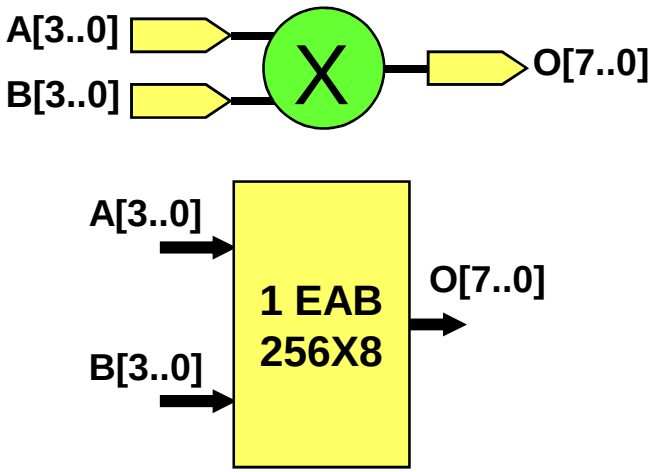


Cada sumando (S) es el resultado de una operación AND, pero la cantidad de sumandos (nS) crece en los bits centrales del resultado, así como a dependencia del arribo del acarreo complejo (nC) de bits previos



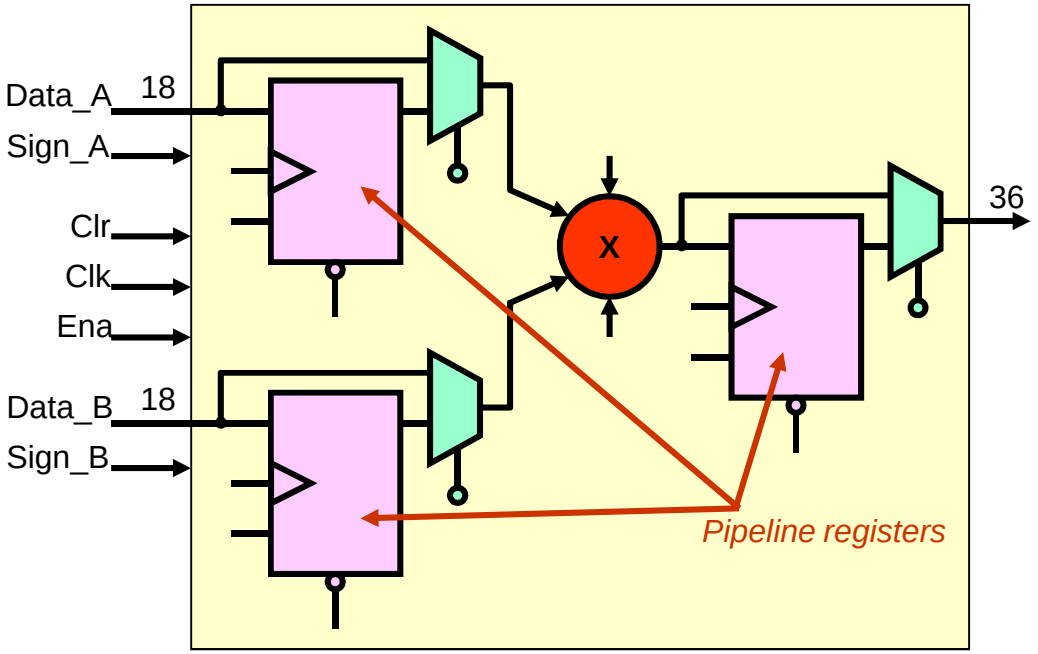
Uso de un EAB para realizar multiplicaciones

- Una multiplicación de dos palabras de 4 bits requiere una importante cantidad de elementos lógicos, consume recursos de conectividad e implica retardos importantes
- El empleo de memorias ROM donde los resultados estén almacenados en tablas permite resolver algunos casos de multiplicador en forma más eficiente
- A medida que aumenta el ancho N_A,N_B de los multiplicandos A[N_A-1..0] y B[N_B-1..0] esta solución se hace impráctica, porque requiere $2^{(NA+NB)}$ palabras de (N_A+N_B) bits
- Esta solución permite implementar otras funciones, como AGCs en comunicaciones, donde la interpretación de un multiplicando puede ser en dB



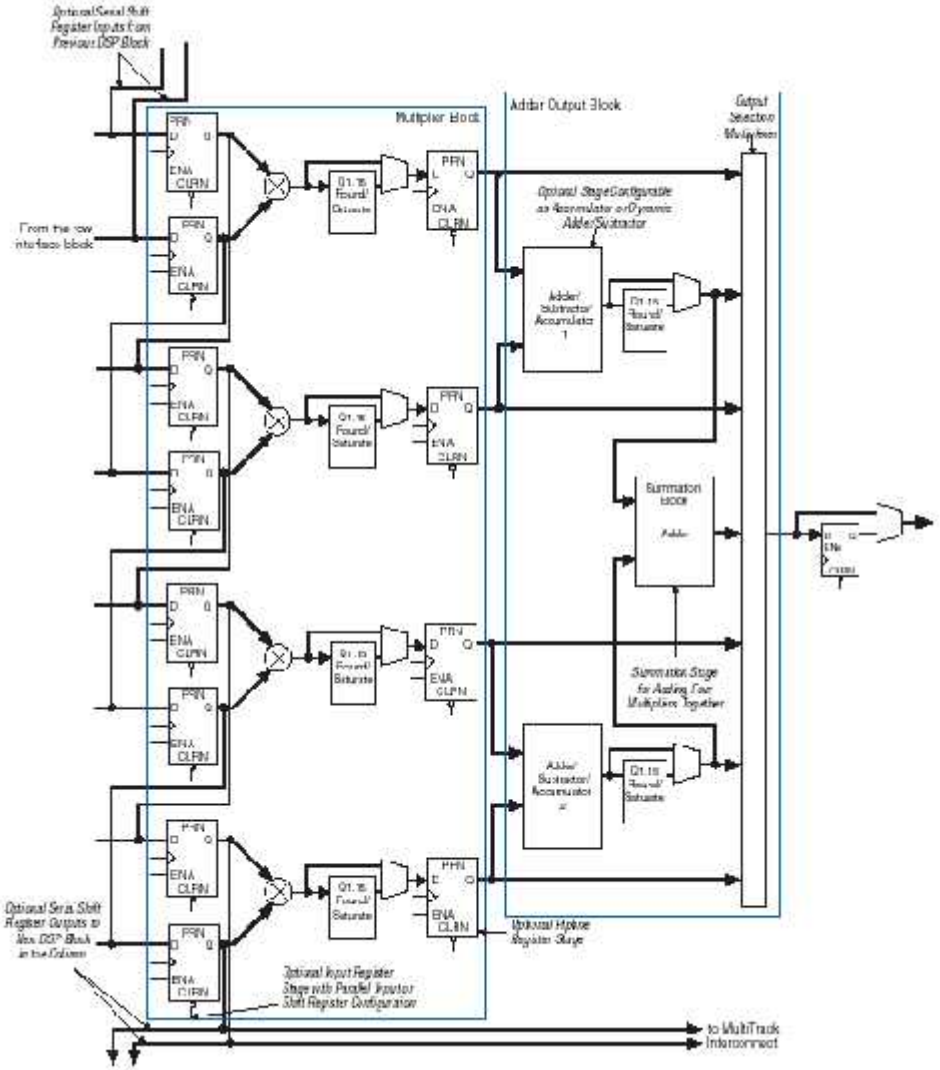
Multiplicadores embebidos

- Los multiplicadores embebidos son soluciones especializadas, no basadas en macroceldas
- Existen múltiples maneras de realizar los árboles de suma y acarreo que forman parte de un área de microelectrónica, pero para el usuario sólo importa el tiempo de ejecución
- En general los multiplicadores incluyen registros de uso opcional para tareas de pipeline, a fin de optimizar aplicaciones de procesamiento de señales
- Es normal que resuelvan operaciones de alta precisión (muchos bits)



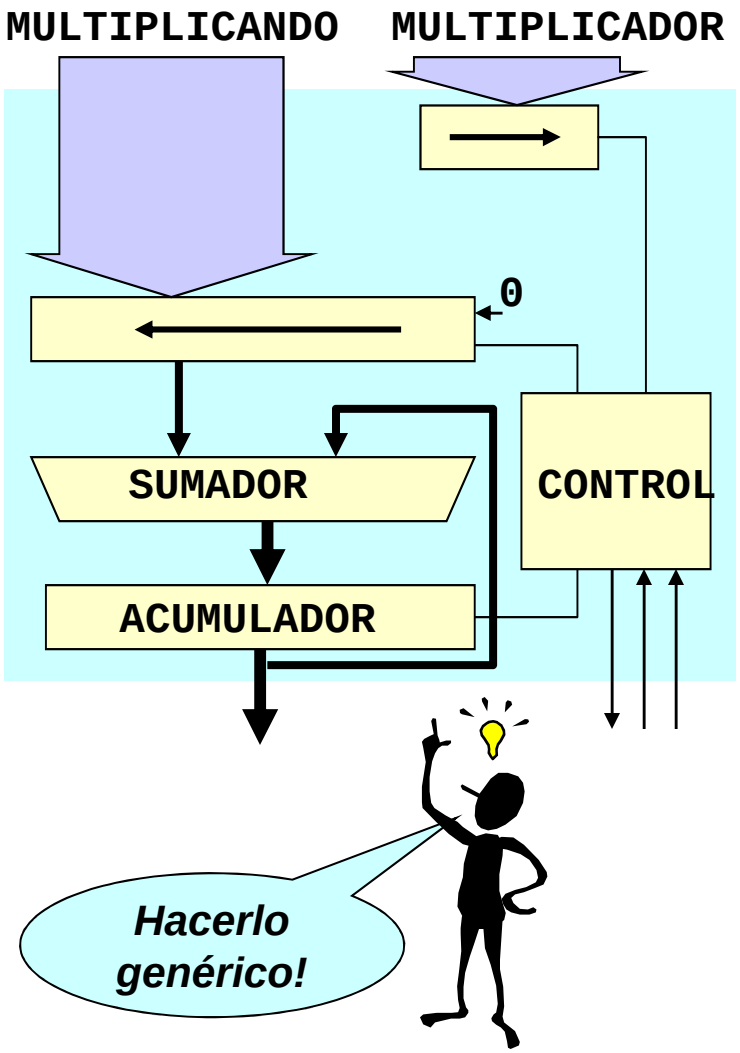
Multiplicación más suma: DSP blocks

- Los bloques DSP son un paso adelante sobre los Multiplicadores
- Son soluciones embebidas predefinidas que facilitan la realización de multiplicaciones y sumas, tal como es requerido en un filtro FIR, o en un correlador
- Y poseen caminos y registros para las cadenas de desplazamiento de datos y coeficientes
- Hay configuraciones que permiten operar con números complejos, de modo de paralelizar filtros complejos o realizar FFT en paralelo

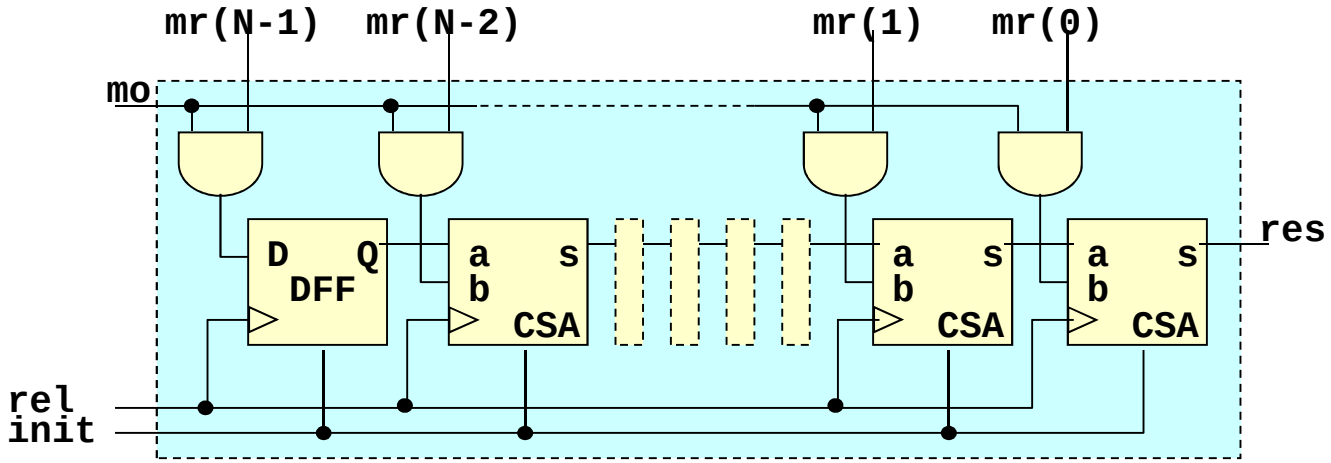


Multiplicador iterativo por shift/suma

```
LIBRARY ieee; USE ieee.std_logic_1164.ALL;
LIBRARY ieee; USE ieee.std_logic_unsigned.ALL;

ENTITY iteramult IS PORT (
 mr,mo:IN std_logic_vector (8 DOWNTO 1);
 init,reloj: IN std_logic; rdy:OUT std_logic;
 res: BUFFER std_logic_vector (16 DOWNTO 1));
END ENTITY iteramult ;

ARCHITECTURE a OF iteramult IS
BEGIN
 PROCESS (reloj) IS
  VARIABLE cnt : INTEGER RANGE 0 TO 8;
  VARIABLE moshf:std_logic_vector(16 DOWNTO 1);
  VARIABLE mrshf:std_logic_vector(8 DOWNTO 1);
  BEGIN
  IF reloj='1' THEN
   IF init = '1' THEN moshf:= X"00" & mo; cnt:=0;
    mrshf := mr; res <= X"0000";rdy <= '0';
   ELSIF cnt < 8 THEN
    IF mrshf(1) = '1' THEN res <= moshf+res; END IF;
    moshf := moshf (15 DOWNTO 1) & '0';
    mrshf := '0' & mrshf(8 DOWNTO 2);
    if cnt/=7 THEN rdy<='0'; ELSE rdy<='1'; END IF;
    cnt := cnt+1;
   END IF;
  END IF;
 END PROCESS;
END ARCHITECTURE a;
```



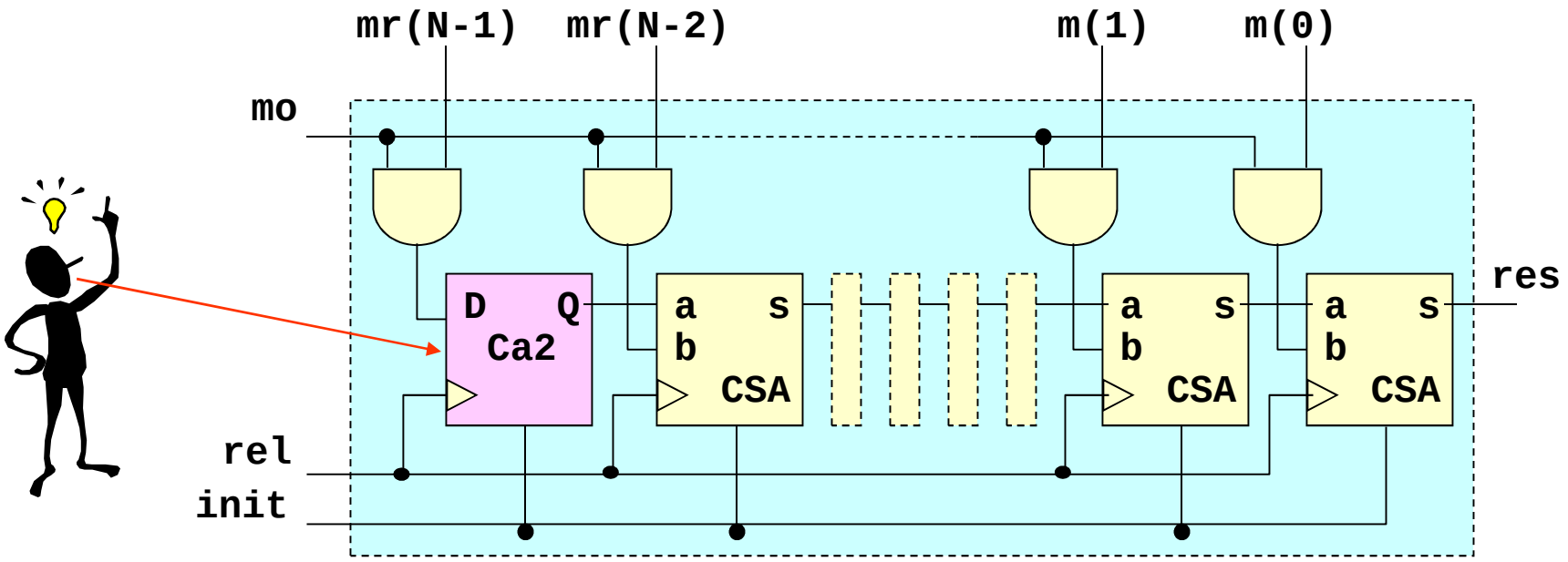
Ejemplo: Multiplicador serial sin signo



- El multiplicador serial es útil cuando, para realizar una multiplicación, el multiplicador ***mr*** es conocido en forma paralela, en tanto el multiplicando ***mo*** ingresa en forma serial
  - –El producto de cada bit de ***mr*** por ***mo*** es realizado usando N funciones AND.
  - –El resultado de cada producto parcial es sumado, mediante una cadena de sumadores seriales, al resultado acumulado hasta ese entonces y desplazado a la derecha. La primer etapa no recibe acarreo de anteriores, por lo que basta un FFD.
  - –La salida del sumador serial menos significativo (***res***) corresponde al producto.
  - –Es necesario generar 2xN-1 ciclos de reloj, durante los N primeros ingresar el multiplicando ***mo*** (LSB primero) y durante los restantes ingresar '0', para "vaciar" las etapas "carry save"

Fuente: R.Andraka.

Ejemplo: Multiplicador serial con signo



- El multiplicador serial con signo es casi idéntico al sin signo, sólo que en la primer etapa, en vez de un simple flipflop D, debe colocarse un circuito de complemento a 2

Fuente: R.Andraka.

Resolución serial de un FIR con multiplicaciones por tabla

Un filtro FIR realiza en cada ciclo la operación

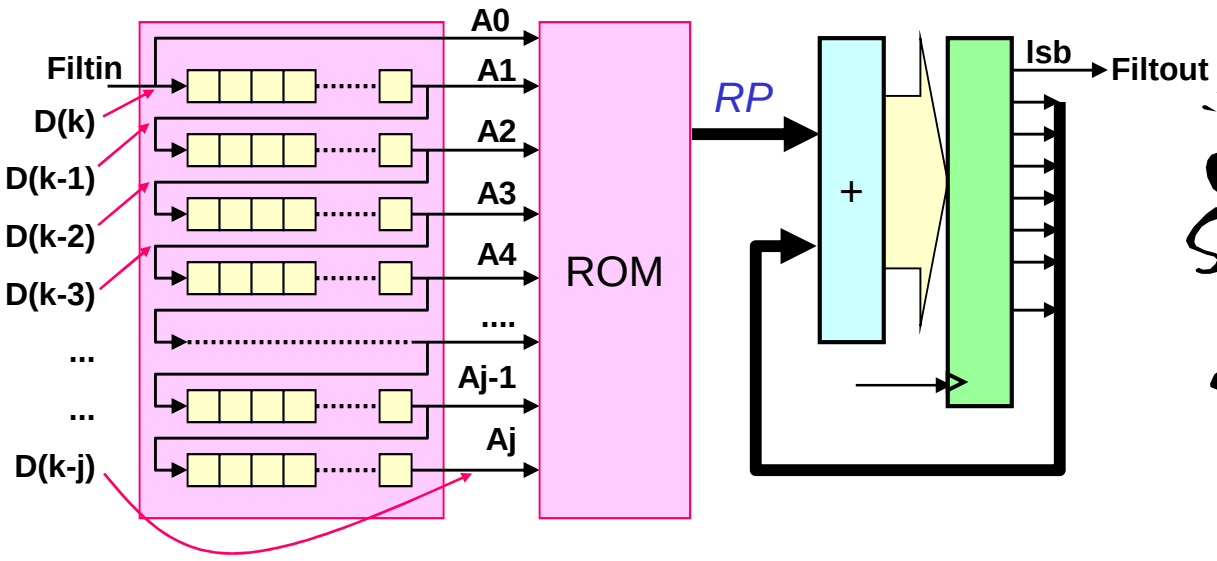
$$D(k)*C0+D(k-1)*C1+D(k-2)*C2+...+D(k-j)*Cj$$

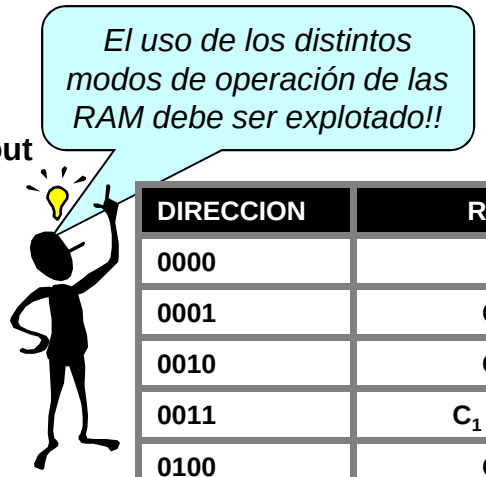
donde los $D(k-i)$ son los datos actual y previos y los Ci son los coeficientes del filtro.

Usando una RAM como shift register múltiple (disponible en algunas FPGAs) y otra RAM como ROM, es posible una solución muy eficiente de aritmética serial para resolver el FIR.

Cada palabra de ROM con dirección Aj/Aj-1/.../A1/A0 almacena el resultado parcial

$$RP = C0*A0 + C1*A1 + C2*A2 + ... Cj*Aj$$

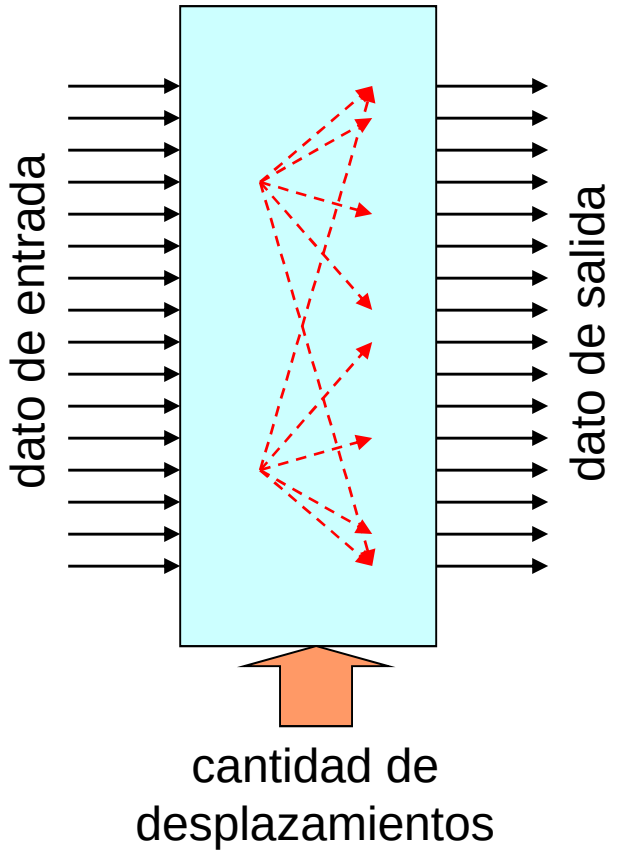




DIRECCION	RP
0000	0
0001	C_0
0010	C_1
0011	$C_1 + C_0$
0100	C_2
0101	$C_2 + C_0$
0110	$C_2 + C_1$
...	....
1111	$C_0 + C_1 + C_2 + C_3$

Barrel Shifter

- El barrel shifter es un circuito muy usado en operaciones numéricas.
- En base a una palabra de control, un dato de entrada de N bits es desplazado una cierta cantidad de bits (hasta N-1), donde el desplazamiento puede ser ***unsigned*** (siempre en un sentido) o ***signed*** (en ambos sentidos)
- A la vez, al ser desplazados los datos en un sentido existen varias alternativas respecto a qué bits ingresar por el otro extremo (cero, extensión de signo o rotación)
- A diferencia de un shift-register, donde los desplazamientos son de a un bit por ciclo de clock, en este caso la operación es combinatoria, siendo usual que el ancho de los datos de entrada (N) sea potencia de dos



Generación de números (pseudo) aleatorios, o PRNGs

A veces es necesario generar números pseudo aleatorios (son *pseudo* aleatorios porque al existir una función que define unívocamente el estado siguiente en función del estado actual, ello contradice la definición de aleatoriedad). Por ejemplo:

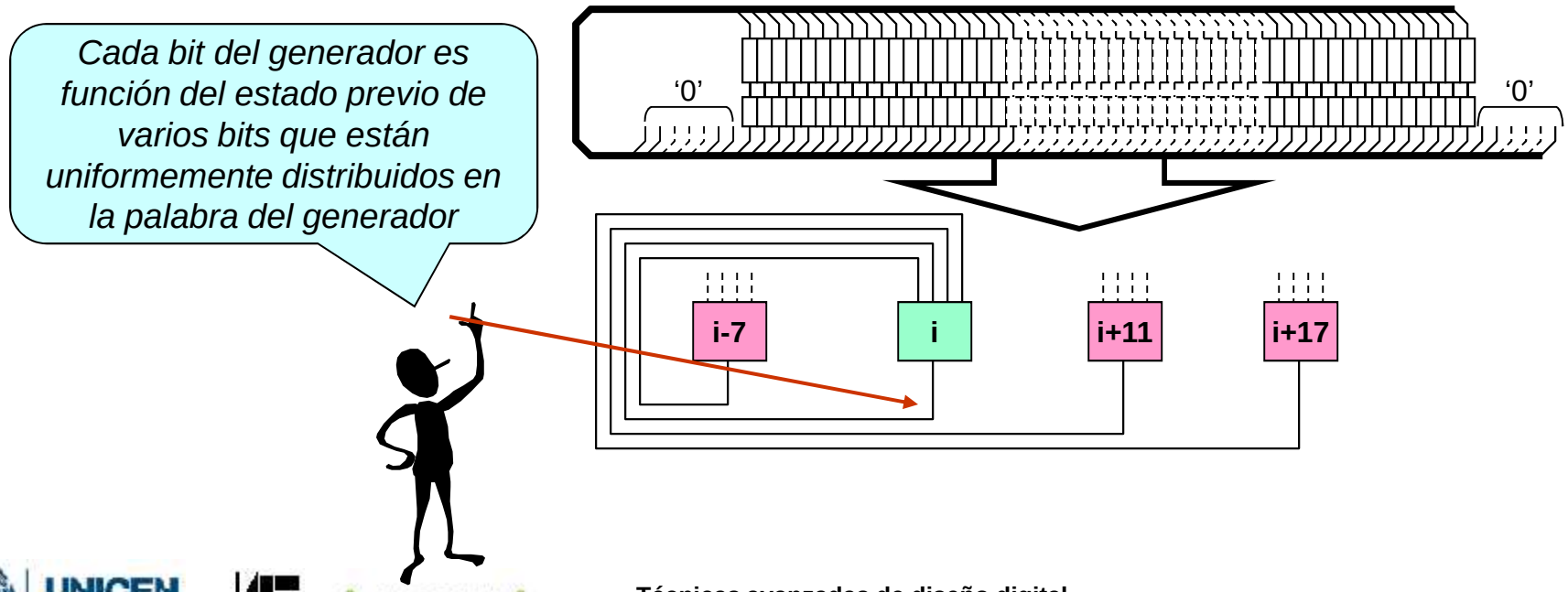
- Hardware estadístico (Monte Carlo, dinámica browniana)
- Algunos tipos de autómatas celulares
- Optimización estocástica: Algoritmos genéticos
- Aplicaciones criptográficas
- Circuitos BIST (Built In Self Test)

La bondad de un PRNG está relacionada a ciertos test de aleatoriedad, que van desde la longitud del ciclo hasta la minimización de efectos de autocorrelación. Los test DIEHARD, por ejemplo, se componen de 15 test que se aplican a 200 millones de números aleatorios sucesivos, y sólo superando los 15 test se puede considerar que la secuencia ha superado DIEHARD.

Generación de números (pseudo) aleatorios

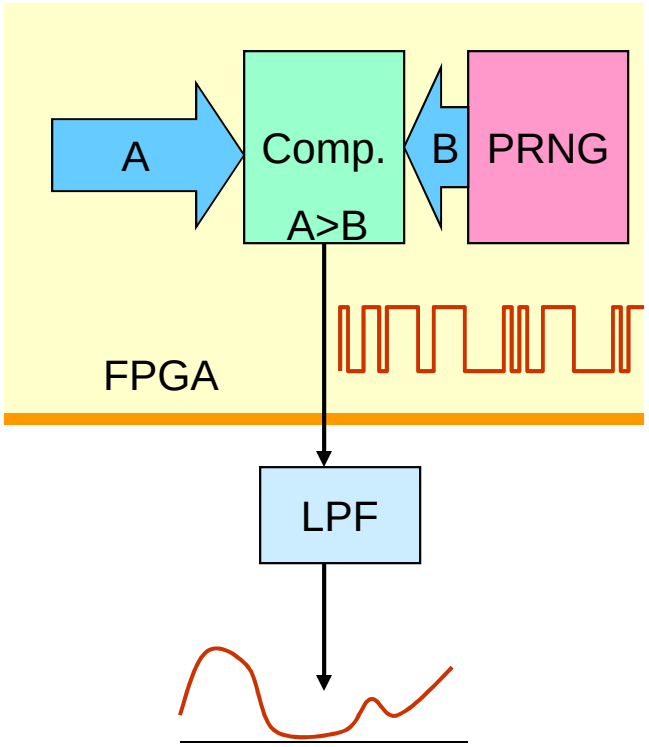
El uso de LFSRs para generadores PRNG de un bit ha demostrado excelentes propiedades estadísticas. De requerirse números de más de un bit existe una correlación temporal importante entre bits, y en ese caso el uso de varios LFSRs en paralelo puede ser una alternativa.

Pero si se aprovecha la disponibilidad de LUTs de 4 entradas como parte de cada macrocelda de FPGA puede emplearse un esquema de autómata celular con mejores prestaciones de aleatoriedad, por la menor correlación temporal entre bits.



Aplicaciones: generación PWM mediante PRNG y comparadores

- Si se supone que en un generador PRNG (*pseudo random number generator*) la distribución de combinaciones es uniforme, la simple comparación de magnitud entre la salida del PRNG con un valor binario permite generar una salida cuya probabilidad de ocurrencia es directamente proporcional a la magnitud de ese valor binario
- Este circuito permite generar una señal PWM y es útil cuando se desea que la densidad espectral de dicha señal sea lo más uniforme posible
- Este caso puede darse en fuentes de alimentación por switching que alimentan receptores de RF de muy baja señal
- Y esta salida PWM, filtrada por un filtro pasabajos, puede ser usada como salida de control analógica sin necesidad de usar un DAC



Circuitos aritméticos complejos en trigonometría, el caso de CORDIC

- El algoritmo CORDIC fue presentado por Jack Volder [IBM] en 1959, como camino para resolver problemas trigonométicos con aritmética de punto fijo.
- Ha sido usado desde entonces en todo tipo de calculadoras de mano, coprocesadores aritméticos, y en todos aquellos casos en que se desea realizar operaciones de rotación o cálculos de magnitud sobre vectores mediante la sóla aplicación de operaciones elementales de shift y suma.
- El trabajo inicial fue ampliado a otros tipos de funciones.
- En el caso de aplicaciones de comunicaciones, este algoritmo es una herramienta fenomenal al trabajar en aplicaciones DSP sobre señales complejas I+Q (*InPhase* y *Quadrature*).
  - En el modo **ROTACIÓN** un dado vector $V_0(X_0,Y_0)$ es transformado en un nuevo vector $V_1(X_1,Y_1)$ rotando a V_0 un dado ángulo A alrededor del origen
  - En el modo **VECTORIZADO**, un vector $V_0(X_0,Y_0)$ es rotado hasta transformarlo en un vector $V_1(X_1,0)$, lo que permite evaluar las coordenadas polares (magnitud y fase) del vector original

El algoritmo CORDIC: fundamentos

- CORDIC divide una rotación en una serie de rotaciones en ambos sentidos y magnitud cada vez menor hasta llegar al ángulo deseado.
- Si las rotaciones sucesivas se hacen para alcanzar un ángulo deseado se está en el modo ROTACIÓN
- Si el signo de las rotaciones se toma en cada momento para disminuir el valor Y del vector rotado, se está en modo VECTORIZADO
- La clave es que los ángulos asociados a rotaciones sucesivas son definidos por $A_i = \arctan(2^{-i})$, y por eso las multiplicaciones se transforman en operaciones de shift

Para A antihorario vale

$$X_1 = X_0.\cos(A) - Y_1.\sin(A)$$
$$Y_1 = X_0.\sin(A) + Y_1.\cos(A)$$

Saco factor común cos(A) y tengo

$$X_1 = \cos(A).(X_0 - Y_0.\tan(A))$$
$$Y_1 = \cos(A).(X_0.\tan(A) + Y_0)$$

Si supongo a $A=\text{suma}(A_i)$ con $A_i=\arctan(2^{-i})$ y i=0,1,..N se tiene

$$X_1 = \cos(\arctan(2^{-0})).(X_0 - Y_0.s.2^{-0})$$
$$Y_1 = \cos(\arctan(2^{-0})).(X_0.s.2^{-0} + Y_0)$$
$$X_2 = \cos(\arctan(2^{-1})).(X_1 - Y_1.s.2^{-1})$$
$$Y_2 = \cos(\arctan(2^{-1})).(X_1.s.2^{-1} + Y_1)$$
$$...$$
$$X_N = \cos(\arctan(2^{-N+1})).(X_{N-1} - Y_1.s.2^{-N+1})$$
$$Y_N = \cos(\arctan(2^{-N+1})).(X_{N-1}.s.2^{-N+1} + Y_1)$$

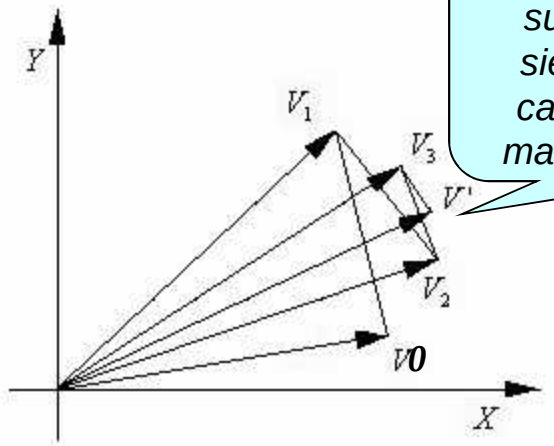
"s" vale +1 o –1 y es el signo de cada minirotación Ai

El algoritmo CORDIC

- En las sucesivas iteraciones, no importa cual sea el signo "s" de cada A_i , resulta que su coseno es positivo, por lo que los sucesivos productos por $\cos(\arctan(2^{-i}))$ son sucesivos factores de escala que convergen a un cambio de escala constante K=1.6468
- Y en cada iteración el valor de X (o Y) es el valor previo afectado de la suma o resta del respectivo Y (o X) desplazado a derecha una cantidad de bits creciente para cada iteración

Si supongo a A=suma(A_i) con A_i=arctan(2^{-i}) y i=0,1,..N se tiene

$$X_1=\cos(\arctan(2^{-0})).(X_0 - Y_0.s.2^{-0})$$
$$Y_1=\cos(\arctan(2^{-0})).(X_0.s.2^{-0} + Y_0)$$
$$X_2=\cos(\arctan(2^{-1})).(X_1 - Y_1.s.2^{-1})$$
$$Y_2=\cos(\arctan(2^{-1})).(X_1.s.2^{-1} + Y_1)$$
$$\ldots$$
$$X_N=\cos(\arctan(2^{-N+1})).(X_{N-1} - Y_1.s.2^{-N+1})$$
$$Y_N=\cos(\arctan(2^{-N+1})).(X_{N-1}.s.2^{-N+1} + Y_1)$$



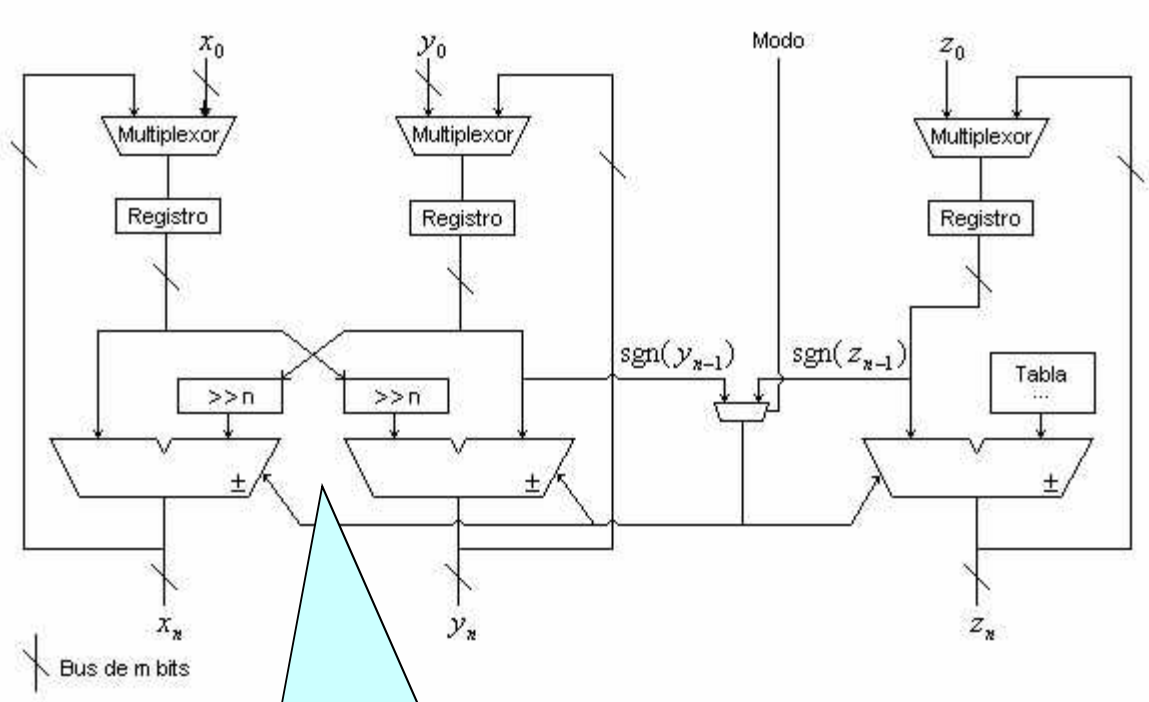
la figura muestra cómo los sucesivos V0, V1...van siendo rotados ángulos cada vez menores, y su magnitud va aumentando

El algoritmo CORDIC: pseudo código

```
x, y, z, xAnterior, yAnterior : real

x = x0;       y = y0;        z = z0
Si Rotación entonces
  Desde i = 0 hasta Numero_De_Iteraciones-1
    xAnterior = x;   yAnterior = y
    x = x + yAnterior * signo(z) * 2^-i
    y = y - xAnterior * signo(z) * 2^-i
    z = z - signo(z) * arctan(2^-i)
  Fin Desde
Sino, Vectorizado entonces
  Desde i = 0 hasta Numero_De_Iteraciones-1
    xAnterior = x;   yAnterior = y
    x = x + yAnterior * signo(yAnterior) * 2^-i
    y = y - xAnterior * signo(yAnterior) * 2^-i
    z = z + signo(yAnterior) * arctan(2^-i)
  Fin Desde
Fin Si
```

El algoritmo CORDIC implementación paralela iterativa

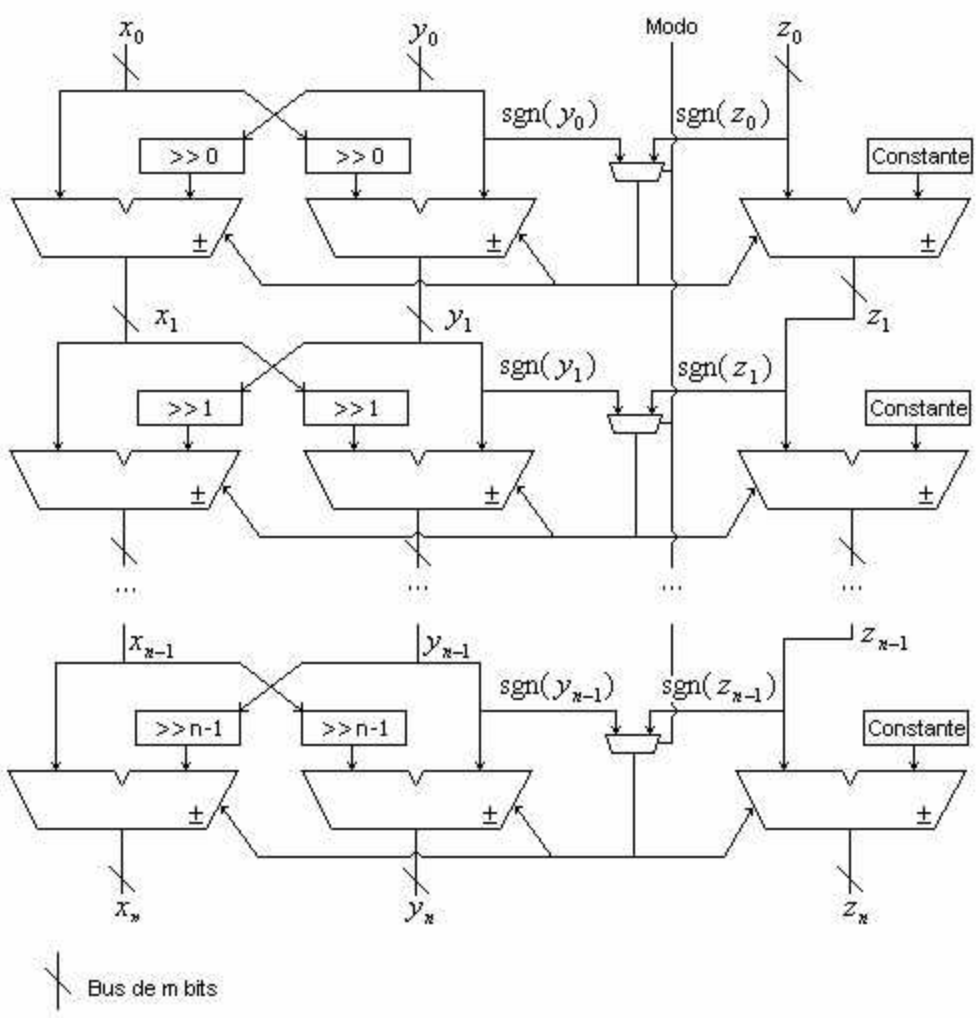
- En esta implementación, y en función del modo (VECTORIZADO o ROTACIÓN) se toman las decisiones de suma o resta.
- Una máquina de estados controla el valor “n” de los shifts a aplicar a los sucesivos valores de X e Y
- Y un acumulador de ángulo Z almacena la suma o resta de los valores Ai que corresponden a cada iteración, obtenidos de una tabla
- En “n” ciclos se obtiene el resultado



la figura muestra cómo bastan multiplexores, shifts y sumas para calcular el resultado

El algoritmo CORDIC: implementación paralela desplegada

- La idea es similar al caso previo, pero desplegando la iteración en forma de sumadores/restadores y shifters fijos en cascada
- Poniendo registros a la salida de cada sumador/restador el circuito opera en pipeline, generando una salida por pulso de reloj, con “n” ciclos de latencia
- La tabla de ángulos es reemplazada por constantes a la entrada de cada sumador Z



El algoritmo CORDIC: implementación serial

- Usando circuitos de suma/resta serial tipo Carry Save Adder se minimizan los circuitos aritméticos necesarios
- La cantidad de ciclos de reloj necesarios surge de la cantidad de iteraciones multiplicada por la precisión de cada dato
- Y la tabla de ángulos puede estar organizada como de “n” palabras más un shift, o directamente organizada de a bit