

5.4 – 5.5.7

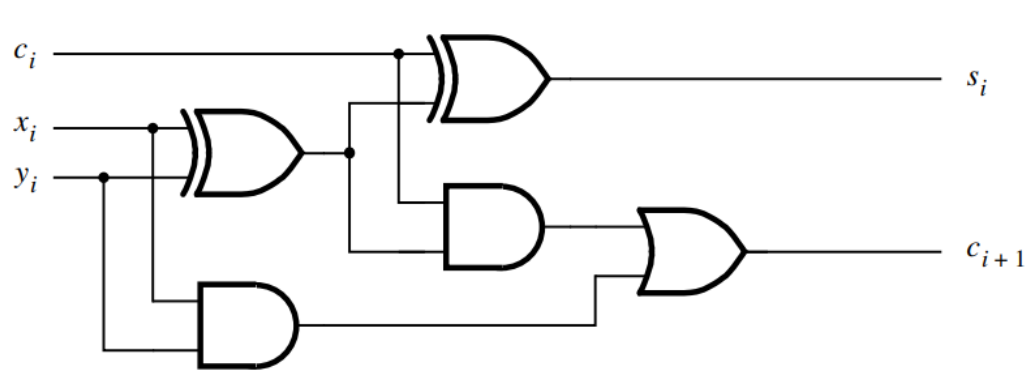
Sumadores rápidos y diseño con CAD

angel@uc.cl

IEE2713 Sistemas Digitales

5.3.6 Problemas de desempeño

- Al comprar un circuito digital (ej: un computador) es importante considerar el cuociente velocidad/precio del circuito

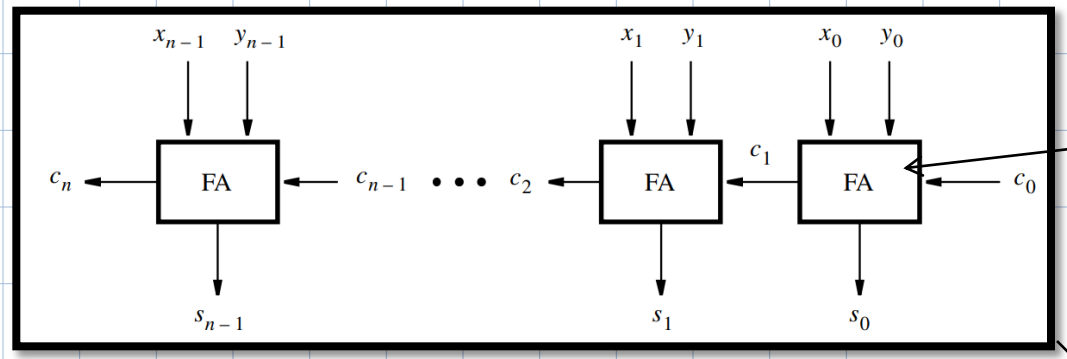


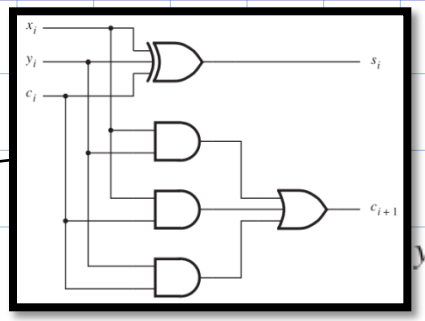
En un análisis "de primer orden", podemos asumir que **cada compuerta agrega un retardo Δt**

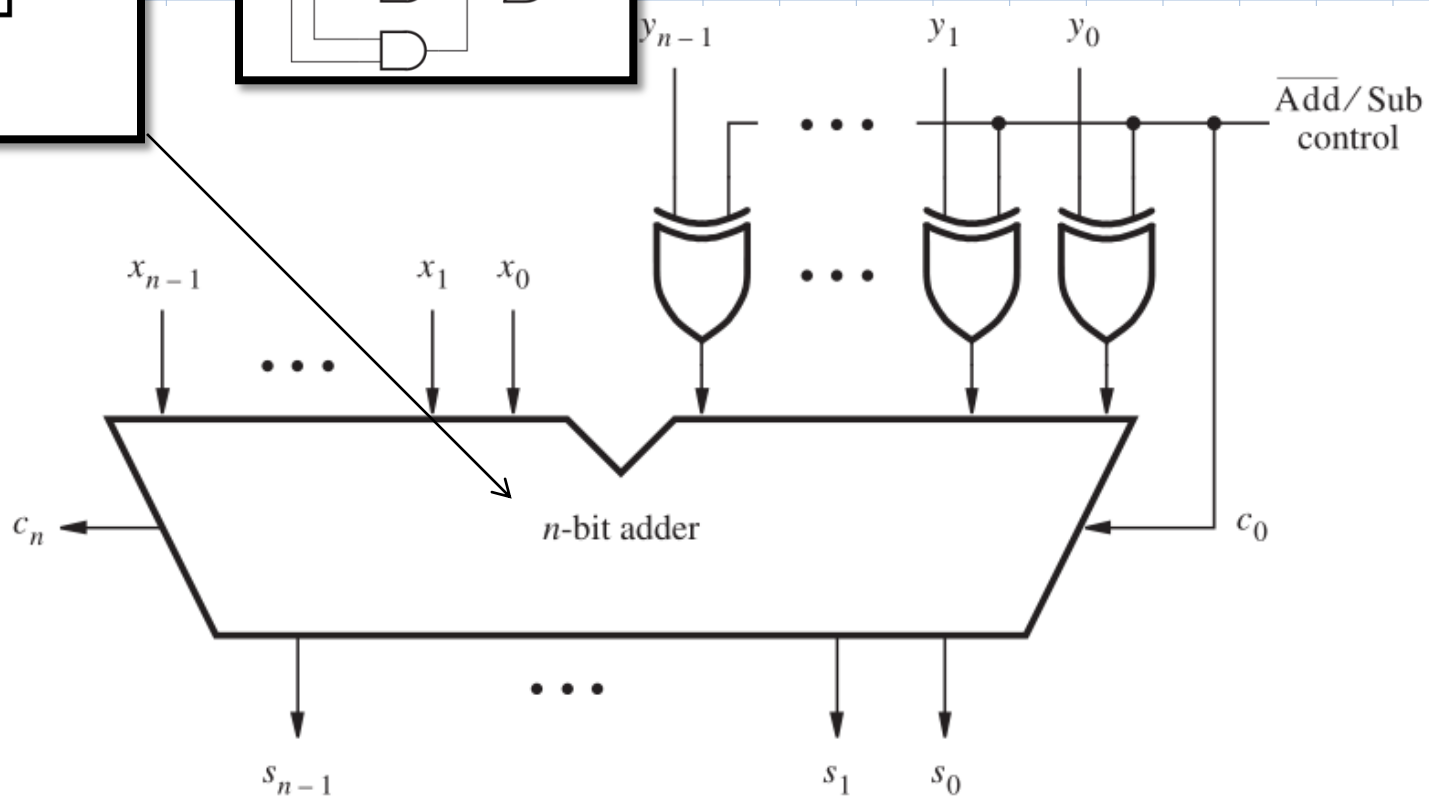
Una forma simple de analizar este parámetro es viendo que velocidad tendrá el circuito:

- Si asumimos que cada transistor tiene un delay genérico Δt entonces, cada vez que se pasa por una compuerta lógica más se demora el circuito en responder. El camino que involucra más compuertas lógicas (n) se llama *critical-delay path* y demoraría $n\Delta t$

Camino crítico en el ripple-carry adder







- Carry out requiere 2Δt
- El mayor retardo de entrada a salida es (2n+1)Δt, donde n es el número de bits del sumador
- Si n = 64, este retardo es inaceptable

5.4 Sumadores rápidos

- Podemos mejorar el desempeño del sumador de dos maneras:
  - Tecnología más nueva y más cara
  - Mejor arquitectura
- A continuación veremos cómo mejorar la arquitectura del sumador

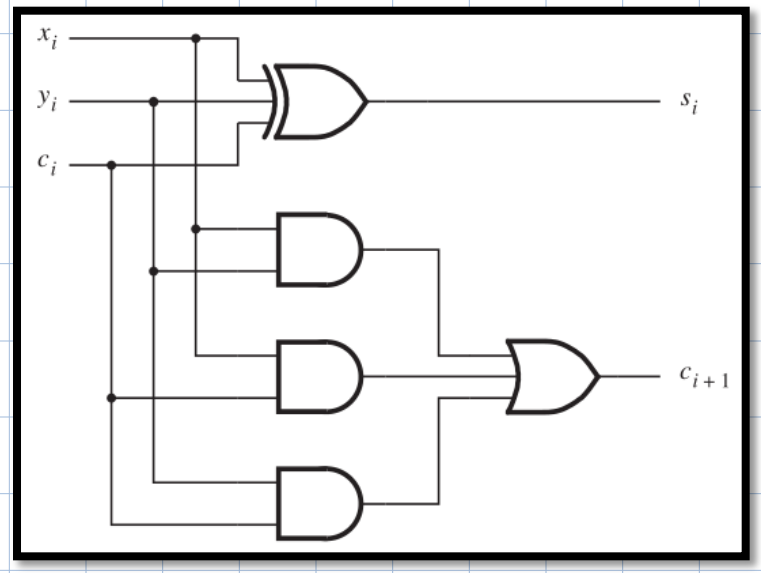
  Factoricemos la expresión para el carry:

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

$$c_{i+1} = x_i y_i + c_i(x_i + y_i)$$

$$c_{i+1} = g_i + c_i p_i \qquad g_i = x_i y_i, p_i = x_i + y_i$$

¿Alguna conclusión de esto?



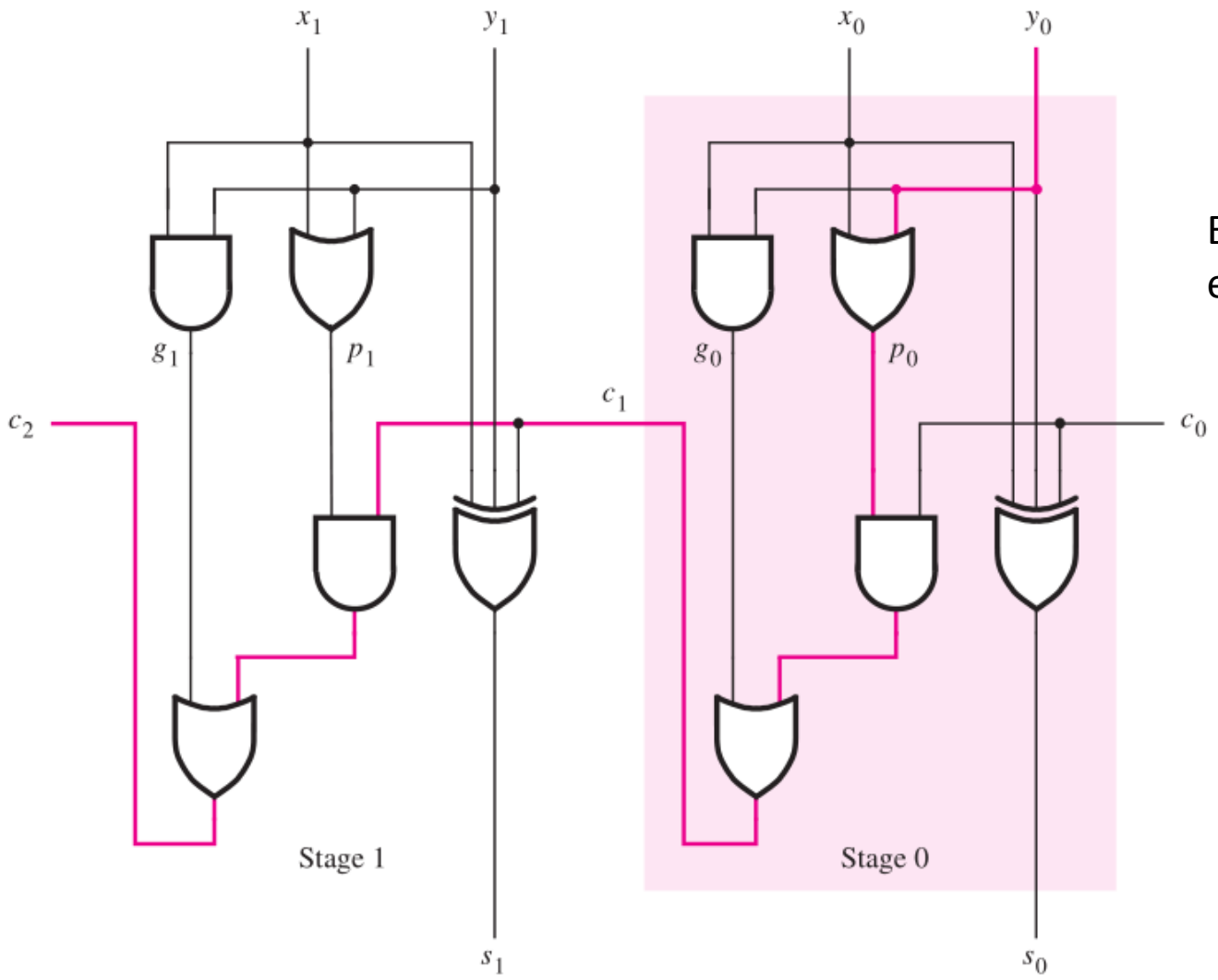
5.4.1 Carry-lookahead adder

- La función g *genera* un carry, independientemente del carry anterior
- La función p *propaga* el carry anterior
- Veamos cómo queda la expresión en términos de etapa anterior:

$$c_{i+1} = g_i + p_i(g_{i-1} + p_{i-1}c_{i-1})$$
$$c_{i+1} = g_i + p_i g_{i-1} + p_i p_{i-1} c_{i-1}$$
$$c_{i+1} = g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + \cdots + p_i p_{i-1} \cdots p_2 p_1 g_0 + p_i p_{i-1} \cdots p_1 p_0 c_0$$

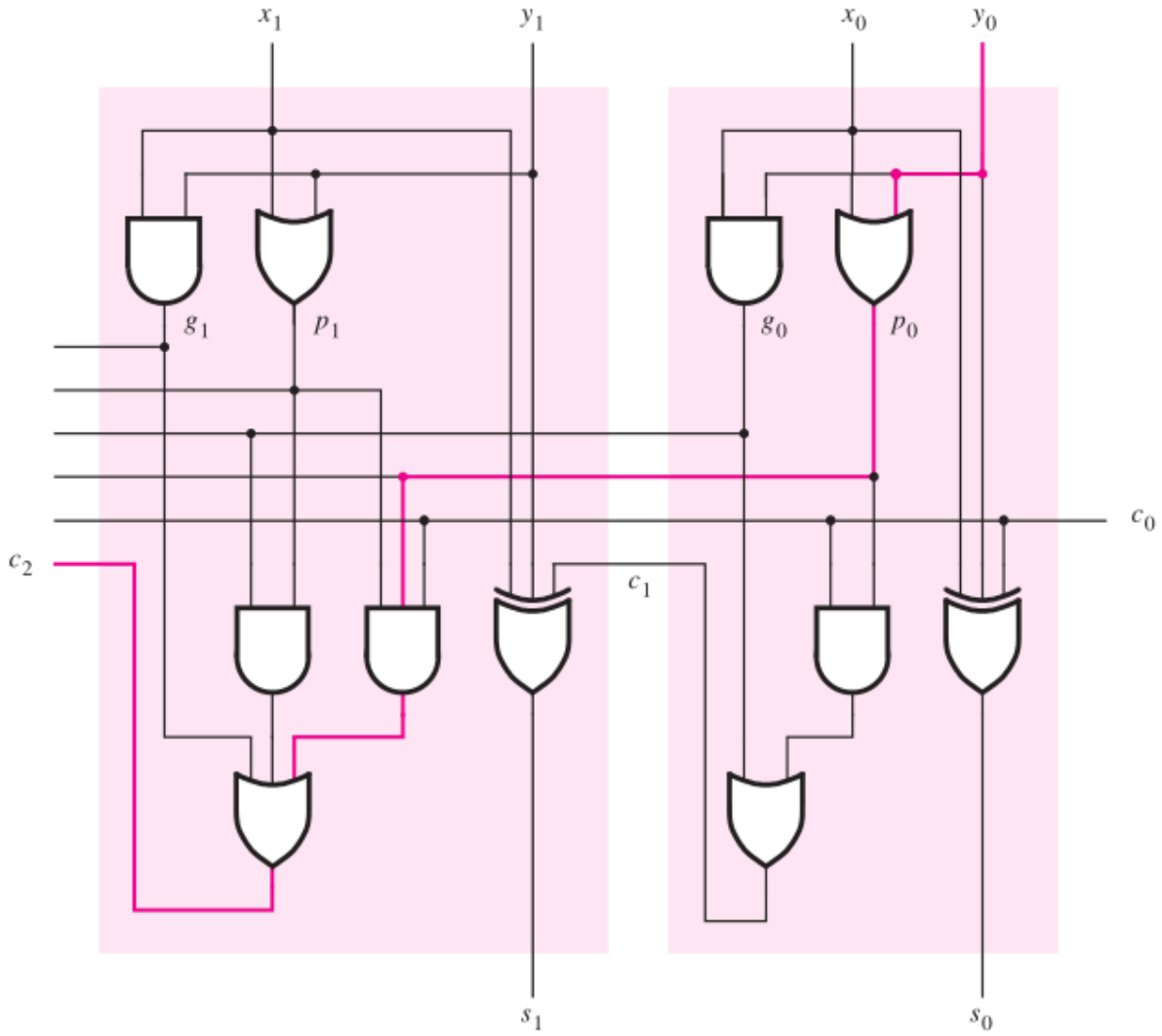
- Esta expresión es de 2 niveles AND-OR y puede ser evaluada muy rápidamente
- Los sumadores que la utilizan se denominan carry-lookahead adder

Comparación: ripple-carry adder



El camino crítico es 2n+1 retardos, donde n es el número de bits del sumador

Comparación: carry-lookahead adder

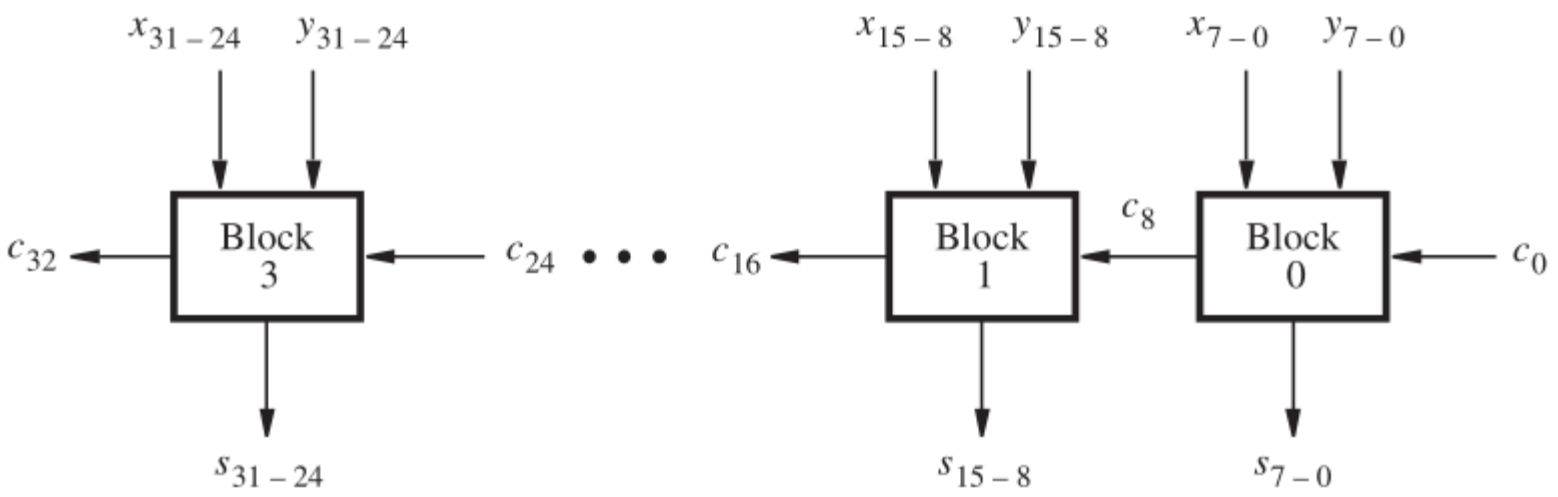


El camino crítico es 3 retardos de compuerta, independientemente del número de bits del sumador (wow!!!)
El sumador completo es sólo 4 retardos de compuerta gracias a la evaluación temprana de los carry

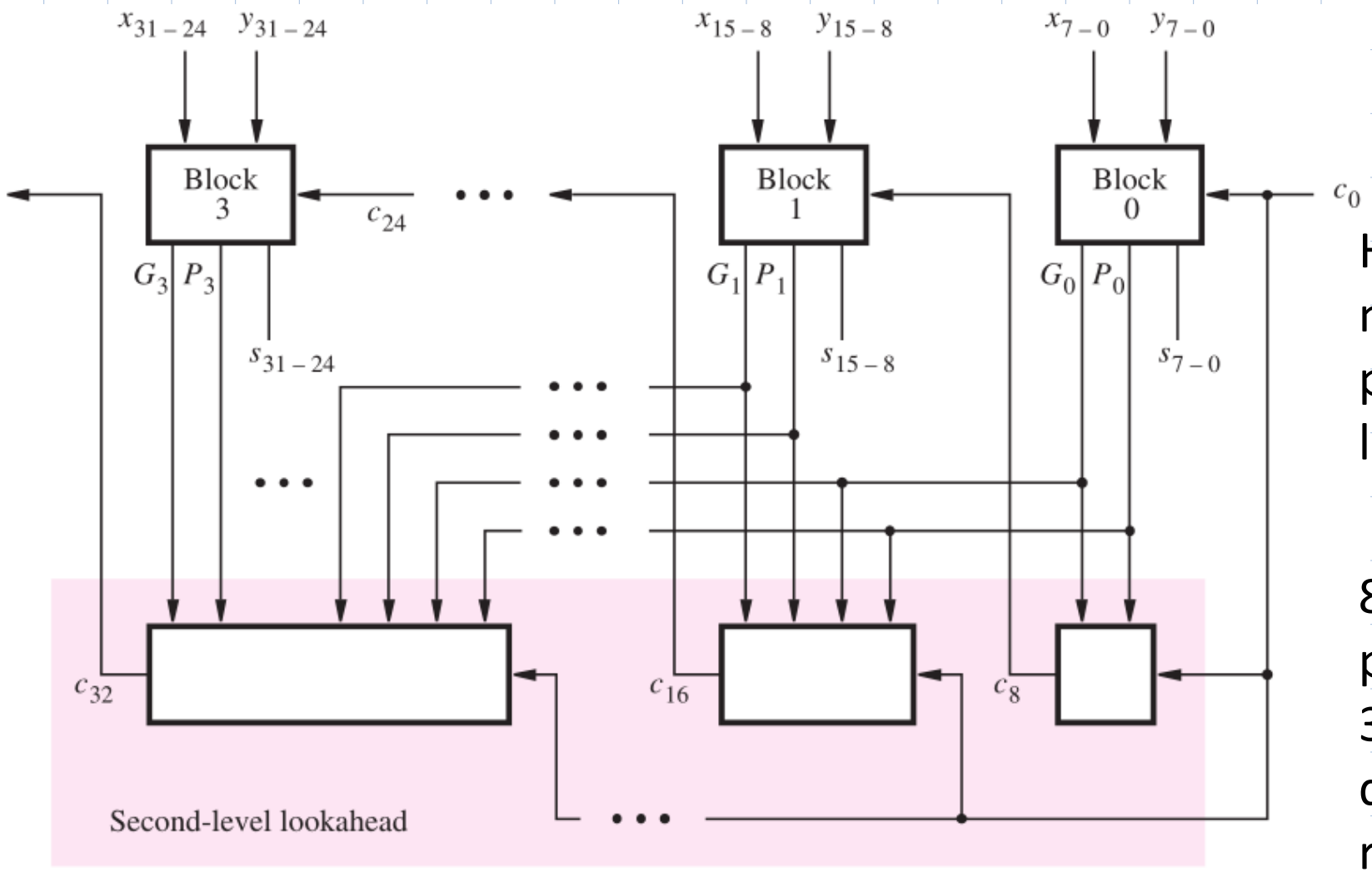
Pero la complejidad crece exponencialmente con el número de entradas… ¿Qué hacer?

Diseño jerárquico de un carry-lookahead adder

- Una alternativa es dividirlo en adders más cortos
  - Por ejemplo, si queremos diseñar un adder de 32 bits, partámoslo en 4 carry-lookahead adders de 8 bits cada uno
- Luego podemos conectar los bloques jerárquicamente y hacer que el carry salte (haga ripple) entre bloques, aumentando el retardo



Diseño jerárquico de un carry-lookahead adder



Hacemos un segundo nivel de carry-lookahead para producir un carry-lookahead jerárquico

8 retardos de compuerta para sumar números de 32 bits, vs. 65 retardos de compuerta para el ripple-carry adder

Consideraciones de la tecnología

- Una implementación práctica debe tomar en cuenta las limitaciones de fan-in de la tecnología
- Es posible factorizar y reutilizar algunos términos para reducir el fan-in necesario
- Un diseño optimizado debe tomar en cuenta todos estos aspectos

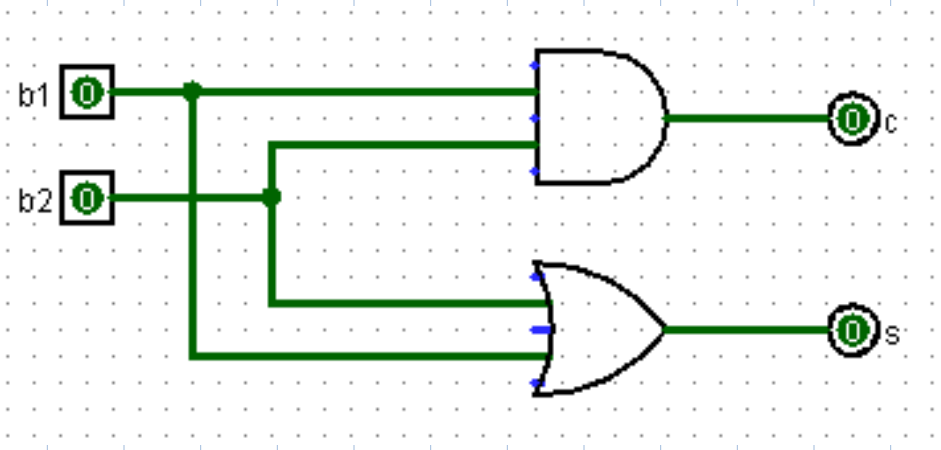
5.5 Diseño de circuitos aritméticos usando herramientas CAD

Existen algunas estrategias útiles al usar Verilog y otras herramientas de diseño CAD:

1. Emplear jerarquía para diseñar circuitos complejos (ej: si construyo un sumador en serie, hacer un modulo para un full adder e instanciarlo con los bits de entrada y salida correspondiente)
2. Emplear buses de bits de datos en vez de datos individuales (definir [3:0]X en vez de variables distintas para cada entrada del full adder)
3. Emplear compuertas adicionales al AND y OR cuando reduzca la cantidad de compuertas (ej: XOR)
4. Emplear la declaración *generate,* que permite generar múltiples instancias de un mismo bloque.

5.5.1 Diseño de circuitos aritméticos usando captura de esquemático

- Una opción es ocupar herramientas de captura de esquemático, que simulan el circuito en función de las compuertas dibujadas (Ej: Logisim).



Algunos CADs incluyen circuitos de mediana complejidad, como sumadores, empleando compuertas lógicas. A estos circuitos medianamente complejos se les denomina megafunciones o macrofunciones. Algunas macrofunciones tienen una implementación dependiente o amarrada a una tecnología particular. Simplemente instanciamos, indicando número de bits del sumador como parámetro.

Si bien son cómodas para circuitos sencillos, para circuitos amplios este proceso puede ser más engorroso que otras herramientas como Verilog y VHDL

5.5.2 Diseño de circuitos aritméticos usando Verilog

Ejemplo: ripple-carry adder

```
module adder4 (carryin, x3, x2, x1, x0, y3, y2, y1, y0, s3, s2, s1, s0, carryout);
    input carryin, x3, x2, x1, x0, y3, y2, y1, y0;
    output s3, s2, s1, s0, carryout;

    fulladd stage0 (carryin, x0, y0, s0, c1);
    fulladd stage1 (c1, x1, y1, s1, c2);
    fulladd stage2 (c2, x2, y2, s2, c3);
    fulladd stage3 (c3, x3, y3, s3, carryout);

endmodule

module fulladd (Cin, x, y, s, Cout);
    input Cin, x, y;
    output s, Cout;

    assign s = x ^ y ^ Cin;
    assign Cout = (x & y) | (x & Cin) | (y & Cin);

endmodule
```

1. Emplear jerarquia e instanciar módulos según se necesite (ej: En un sumador en serie, implementar un full adder e instanciarlo según sea necesario)
2. Emplear compuertas alternativas a AND y OR, como el XOR

5.5.3 Señales vectorizadas

- Otra forma conveniente de definir señales en Verilog es como **vectores** que representan **buses de datos**. Por ejemplo, podemos definr C como:

```
input [3:0]C;
```

- Esto define a C como una variable de 4 bits, donde el bit más significativo es C[3].
- Es posible referirnos a los tres bits del medio como:

```
wire [2:1]C;
```

- En el código que aparece a la derecha usamos C[1] para conectar el carry de salida del primer bloque al carry de entrada del segundo
- Es posible **concatenar** dos vectores {C1, C2[3:2]}
- También es posible definir [0:3]C, pero no se usa mucho

```
module adder4 (carryin, X, Y, S, carryout);
   input carryin;
   input [3:0] X, Y;
   output [3:0] S;
   output carryout;
   wire [3:1] C;

   fulladd stage0 (carryin, X[0], Y[0], S[0], C[1]);
   fulladd stage1 (C[1], X[1], Y[1], S[1], C[2]);
   fulladd stage2 (C[2], X[2], Y[2], S[2], C[3]);
   fulladd stage3 (C[3], X[3], Y[3], S[3], carryout);

endmodule
```

5.5.4 Especificación genérica

```
module addern (carryin, X, Y, S, carryout);
   parameter n = 32;
   input carryin;
   input [n–1:0] X, Y;
   output [n–1:0] S;
   output carryout;
   reg [n–1:0] S;
   reg carryout;
   reg [n:0] C;
   integer k;

   always @(X or Y or carryin)
   begin
      C[0] = carryin;
      for (k = 0; k < n; k = k+1)
      begin
         S[k] = X[k] ^ Y[k] ^ C[k];
         C[k+1] = (X[k] & Y[k]) | (X[k] & C[k]) | (Y[k] & C[k]);
      end
      carryout = C[n];
   end

endmodule
```

¿Qué pasa si queremos definir en Verilog un sumador de tamaño arbitrario?

Es posible definir un bloque genérico de tamaño n, definir n como **parámetro** y cambiar el tamaño de n según se necesite

La declaración `for` es un procedimiento, debe ir en bloque `always` y es ejecutado en orden

5.5.4 Especificación genérica

```
module addern (carryin, X, Y, S, carryout);
   parameter n = 32;
   input carryin;
   input [n–1:0] X, Y;
   output [n–1:0] S;
   output carryout;
   wire [n:0] C;

   genvar i;
   assign C[0] = carryin;
   assign carryout = C[n];

   generate
      for (i = 0; i <= n–1; i = i+1)
      begin:addbit
         fulladd stage (C[i], X[i], Y[i], S[i], C[i+1]);
      end
   endgenerate

endmodule

module fulladd (Cin, x, y, s, Cout);
   input Cin, x, y;
   output s, Cout;

   assign s = x ^ y ^ Cin;
   assign Cout = (x & y) | (x & Cin) | (y & Cin);

endmodule
```

Alternativamente podemos emplear el comando `generate`, que instanciará múltiples copias de un mismo bloque en un loop for o if/else

5.5.5 Nets y variables en Verilog

- Un circuito en Verilog es un conjunto de elementos lógicos interconectados y/o declaraciones de procedimientos
- A continuación definiremos nets y variables

Nets en Verilog

- Las conexiones entre elementos lógicos son **nets** y las definimos con la palabra **wire**
  - Permiten conectar entradas y salidas de bloques lógicos
  - Puede ser escalar o vector
  - No es necesario declararlas, Verilog asume que toda señal es wire
  - Existen nets tipo **tri** (3 tipos de valores), pero funcionan igual que los wires
- Son asignadas continuamente

```
module addern (carryin, X, Y, S, carryout);
    parameter n = 32;
    input carryin;
    input [n–1:0] X, Y;
    output [n–1:0] S;
    output carryout;
    wire [n:0] C;

    genvar i;
    assign C[0] = carryin;
    assign carryout = C[n];
```

Variables en Verilog

- Las señales generadas por procedimientos son **variables**
- Son usadas para describir circuitos según su comportamiento (y no según conexiones entre elementos internos)
- Retienen el valor hasta que el procedimiento vuelve a cambiarlo
- Hay dos tipos: reg e integer
  - En el ejemplo, integer no define una variable del circuito, sino el índice de un loop
  - Más adelante hablaremos sobre las variables

```
module addern (carryin, X, Y, S, carryo
  parameter n = 32;
  input carryin;
  input [n–1:0] X, Y;
  output [n–1:0] S;
  output carryout;
  reg [n–1:0] S;
  reg carryout;
  reg [n:0] C;
  integer k;

  always @(X or Y or carryin)
  begin
    C[0] = carryin;
    for (k = 0; k < n; k = k+1)
    begin
```

5.5.6 Declaraciones de asignaciones aritméticas

- En Verilog es posible realizar operaciones sobre vectores
- En el caso del ejemplo, la declaración

  S = X + Y + carryin;

  es en efecto la suma de X + Y, usando todos los bits y el carry de entrada
  - Notar que en el **always** no es necesario usar begin ni end, porque sólo contiene una declaración
- Lo malo de esta forma de implementar una suma es que no tiene carry out…
  - Pero hay formas de obtener el carry out y el overflow:

$$carryout = x_{n-1}y_{n-1} + x_{n-1}\bar{s}_{n-1} + y_{n-1}\bar{s}_{n-1}$$

$$overflow = carryout \oplus x_{n-1} \oplus y_{n-1} \oplus s_{n-1}$$

```
module addern (carryin, X, Y, S);
    parameter n = 32;
    input carryin;
    input [n–1:0] X, Y;
    output [n–1:0] S;
    reg [n–1:0] S;

    always @(X or Y or carryin)
        S = X + Y + carryin;

endmodule
```

Veamos una implementación posible para la suma con overflow y carryout

```
module addern (carryin, X, Y, S, carryout, overflow);
    parameter n = 32;
    input carryin;
    input [n–1:0] X, Y;
    output [n–1:0] S;
    output carryout, overflow;
    reg [n–1:0] S;
    reg carryout, overflow;

    always @(X or Y or carryin)
    begin
        S = X + Y + carryin;
        carryout = (X[n–1] & Y[n–1]) | (X[n–1] & ~S[n–1]) | (Y[n–1] & ~S[n–1]);
        overflow = carryout ^ X[n–1] ^ Y[n–1] ^ S[n–1];
    end

endmodule
```

Veamos otra implementación posible para la suma con overflow y carryout

```
module addern (carryin, X, Y, S, carryout, overflow);
    parameter n = 32;
    input carryin;
    input [n–1:0] X, Y;
    output [n–1:0] S;
    output carryout, overflow;
    reg [n–1:0] S;
    reg carryout, overflow;
    reg [n:0] Sum;

    always @(X or Y or carryin)
    begin
        Sum = {1'b0, X} + {1'b0, Y} + carryin;
        S = Sum[n–1:0];
        carryout = Sum[n];
        overflow = carryout ^ X[n–1] ^ Y[n–1] ^ S[n–1];
    end

endmodule
```

- El vector Sum tiene un bit más de los necesarios
  - En el MSB de Sum guardamos el carryout
- Estamos concatenando… **¿Para qué?**
- **¿Qué es eso de 1'b0?**

(En realidad aquí no es necesario concatenar, porque Verilog llena con ceros los vectores involucrados en operaciones aritméticas para hacerlos consistentes)

Otra forma mejor de hacer lo mismo

```
module addern (carryin, X, Y, S, carryout, overflow);
    parameter n = 32;
    input carryin;
    input [n–1:0] X, Y;
    output [n–1:0] S;
    output carryout, overflow;
    reg [n–1:0] S;
    reg carryout, overflow;

    always @(X or Y or carryin)
    begin
        {carryout, S} = X + Y + carryin;
        overflow = carryout ^ X[n–1] ^ Y[n–1] ^ S[n–1];
    end

endmodule
```

Aquí concatenamos la salida

Un full-adder en Verilog

La ventaja de usar esta definición del fulladder es que dejamos en el sintetizador la tarea de buscar la mejor implementación

```
module  fulladd (Cin, x, y, s, Cout);
    input  Cin, x, y;
    output  s, Cout;
    reg  s, Cout;

    always @(x or y or Cin)
        {Cout, s} = x + y + Cin;

endmodule
```

5.5.7 Representación de números en Verilog

- Los números en Verilog poseen el siguiente formato:

<tamaño en bits>'<identificador de base><dígitos significativos>

b = binario
o = octal
h = hexadecimal
o = decimal

- Es posible usar guión bajo en los números para facilitar la lectura - ej: 8b'0010_1101
- Los números negativos incorporan un guión **antes** del número - ej: -12'o4251
- En el caso de los decimales no es necesario especificar base o tamaño; Verilog le asigna tamaño según otros operandos en la expresión
  - Pero yo igual prefiero asignar tamaño para tener el control

12'b100010101001
12'o4251
12'h8A9
12'd2217

0, 1, z y x

- Hasta ahora hemos asumido que un circuito lógico tiene salida 0 o 1
- También aparecen x cuando no sabemos cuánto vale
  - Ejemplo: cortocircuitamos un 0 con un 1
- Y también existe el valor z… **¿qué representa z?**
  - Puede ser usado con base 2, 8 y hex, pero no con base 10
  - Si hacemos padding con z o x, estos valores son replicados

Extensión de signo

- La expresión C = A + B funciona OK si ambos son positivos, da lo mismo su tamaño
  - Verilog agrega ceros para igualar sus tamaños
  - Un número positivo con más ceros a la izquierda no cambia – OK
  - Un número **negativo con ceros a la izquierda, se vuelve positivo** – **FAIL!**
    - Para extender el signo a un negativo, hay que llenar de unos hacia la izquierda
    - **¿Cómo sabe Verilog que el número que queremos representar es negativo?**
- Una opción es usar el truco de extensión de signo
  - Por ejemplo, si B fuera de 4 bits y A fuera de 8 bits

  C = A + {4{B[3]}, B}

  O alternativamente, si queremos generar un carry out:

  C = {1'b0, A} + {1'b0, 4{B[3]}, B}



5.6 – 5.8

Multiplicación y otros tópicos

angel@uc.cl

IEE2713 Sistemas Digitales

5.6 Multiplicación

Antes de definir la multiplicación, podemos notar que las operaciones de desplazamiento a la izquierda y desplazamiento a la derecha (shift left y shift right) corresponden a una multiplicación y división por dos, respectivamente.

Ej: (1110) x 2 = (11100)

14 28

Ojo que al hacer shift right, hay que rellenar con MSBs

¿Por qué?

(En general la multiplicación y división es por la base)

¿Cómo multiplicamos en binario?

- Igual que en decimales – Ejemplo (4 bits)

Multiplicand M	(14)	1 1 1 0
Multiplier Q	(11)	× 1 0 1 1
		1 1 1 0
		1 1 1 0
		0 0 0 0
		1 1 1 0
Product P	(154)	1 0 0 1 1 0 1 0

Multiplicación a mano

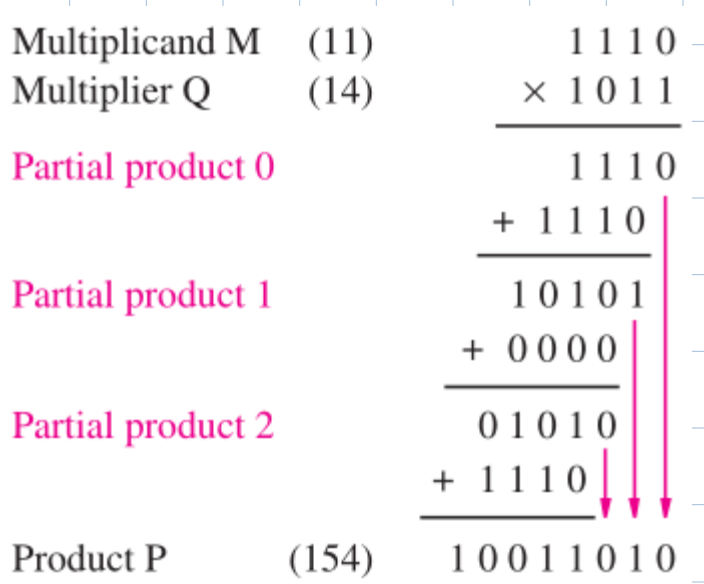
Multiplicand M	(11)	1 1 1 0
Multiplier Q	(14)	× 1 0 1 1
Partial product 0		1 1 1 0
		+ 1 1 1 0
Partial product 1		1 0 1 0 1
		+ 0 0 0 0
Partial product 2		0 1 0 1 0
		+ 1 1 1 0
Product P	(154)	1 0 0 1 1 0 1 0

Multiplicación para implementación en hardware

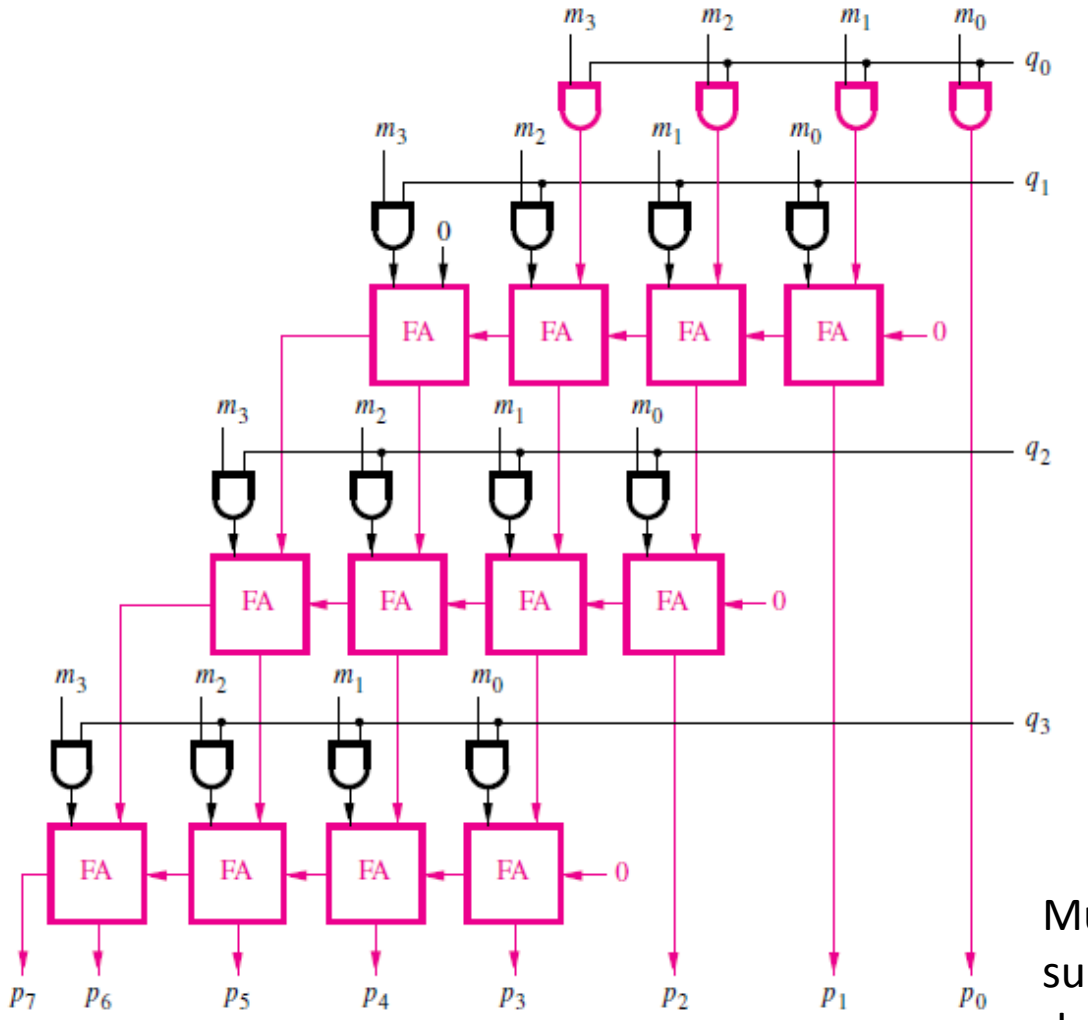
El producto de dos bits es simplemente el AND entre los dos bits

Multiplicación secuencial

- Podemos ir paso a paso
  - Primero multiplicamos
  - Luego sumamos
  - Luego multiplicamos otra vez…
  - Etc
- El problema es que son muchos pasos
  - Muy lento (pero menos hardware)
  - Más adelante veremos ejemplos
- Una alternativa es usar muchos sumadores – fuerza bruta (a continuación)



5.6.1 Multiplicador en arreglo para números sin signo



			m_3	m_2	m_1	m_0		
		×	q_3	q_2	q_1	q_0		
Partial product 0			m_3q_0	m_2q_0	m_1q_0	m_0q_0		
		+	m_3q_1	m_2q_1	m_1q_1	m_0q_1		
Partial product 1		$PP1_5$	$PP1_4$	$PP1_3$	$PP1_2$	$PP1_1$		
	+	m_3q_2	m_2q_2	m_1q_2	m_0q_2			
Partial product 2	$PP2_6$	$PP2_5$	$PP2_4$	$PP2_3$	$PP2_2$			
+	m_3q_3	m_2q_3	m_1q_3	m_0q_3				
Product P	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

(c) Hardware implementation

Multiplicar en binario es mucho más complicado que sumar binario. Hacerla con cuidado en implementaciones de tiempo real y evitarla en implementaciones livianas

5.6.2 Multiplicación de números con signo

- Aquí hay 4 casos:
  1. Ambos positivos: igual que en 5.6.1
  2. Multiplicando negativo y multiplicador positivo:
     - Añadimos bits de signo a las sumas parciales para evitar overflow
     - Es crucial mantener el signo de las operaciones intermedias
  3. Multiplicando positivo y multiplicador negativo:
     - Hacemos complemento 2 a ambos y luego realizamos la operación
     - Así logramos que el multiplicando sea negativo y el multiplicador, positivo (caso 2)
  4. Ambos negativos:
     - Hacemos complemento 2 a ambos y luego realizamos la operación
     - Quedan ambos positivos, igual que en caso 1

5.6.2 Multiplicación de números con signo

Multiplicand M	(–14)	1 0 0 1 0
Multiplier Q	(+11)	× 0 1 0 1 1
Partial product 0		1 1 1 0 0 1 0
		+ 1 1 0 0 1 0
Partial product 1		1 1 0 1 0 1 1
		+ 0 0 0 0 0 0
Partial product 2		1 1 1 0 1 0 1
		+ 1 1 0 0 1 0
Partial product 3		1 1 0 1 1 0 0
		+ 0 0 0 0 0 0
Product P	(–154)	1 1 0 1 1 0 0 1 1 0

Ejemplo: multiplicando negativo y multiplicador positivo (caso 2)

5.7 Otras representaciones numéricas

5.7.1 Representación de punto fijo

- La representación principal de números que hemos visto hasta ahora, y la que usamos comúnmente en nuestra vida cotidiana, se denomina representación de punto fijo
- Para cualquier base b:

$$B = b_{n-1}b_{b-2}\ldots b_1 b_0 . b_{-1}b_{-2}\ldots b_{-k} \xrightarrow{\text{Valor}} V(B) = \sum_{i=-k}^{n-1} b_i \times 2^i$$

- Los circuitos que usan esta representación son en esencia los mismos que hemos visto hasta ahora
- El rango está limitado por el número de bits

5.7.2 Números de punto flotante

Están representados por una mantisa y un exponente, estilo notación científica

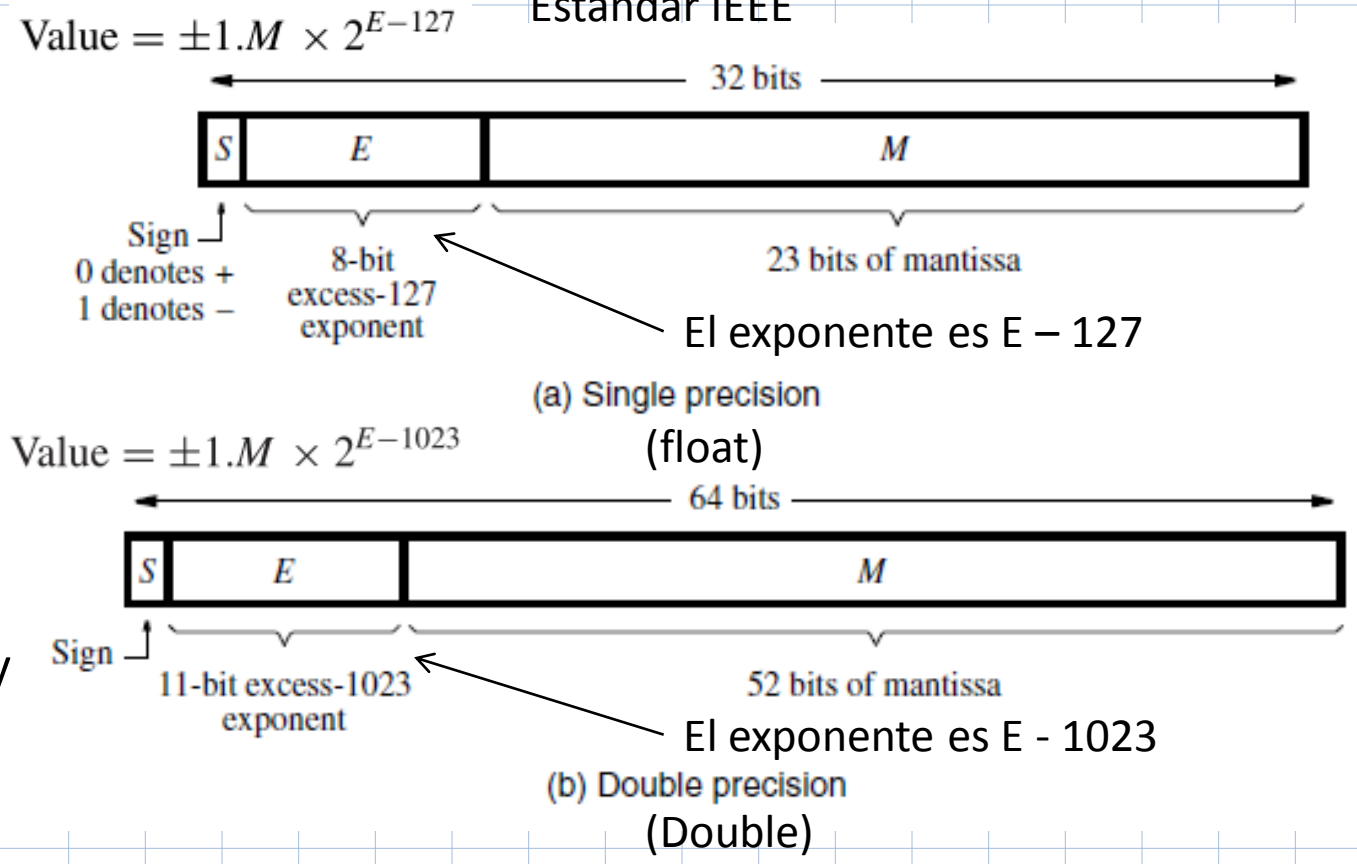
$$Mantisa \times b^{exponente}$$

Generalmente los normalizamos de manera que la coma esté después de la primera cifra significativa (siempre 1.M)

El exponente es para base 2

- El número E es siempre positivo
- Es fácil realizar operaciones entre operandos en punto flotante, primero hay que comparar los exponentes y luego desplazar la mantisa

Estándar IEEE



5.7.3 Representación decimal en código binario (BCD)

- Es la representación de dígitos decimales mediante grupos de 4 dígitos binarios
  - Útil para interfaz con humanos en display de dígitos
  - Pero desperdicia 6 de las 16 posibles representaciones
    - Generalmente tratamos esos casos como don’t cares

Decimal digit	BCD code
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Suma en BCD

X	0 1 1 1	7
+ Y	+ 0 1 0 1	+ 5
Z	1 1 0 0	12
	+ 0 1 1 0	
carry →	1 0 0 1 0	
	S = 2	

X	1 0 0 0	8
+ Y	+ 1 0 0 1	+ 9
Z	1 0 0 0 1	17
	+ 0 1 1 0	
carry →	1 0 1 1 1	
	S = 7	

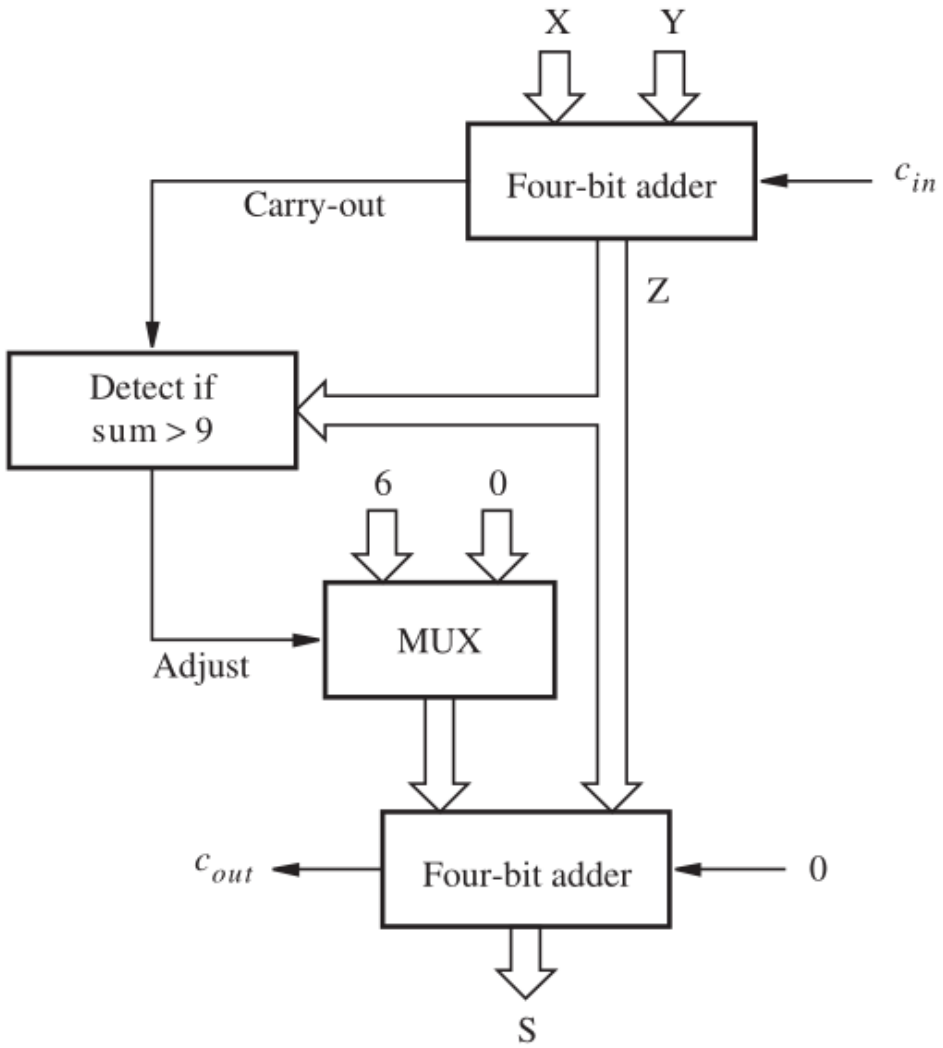
- Al sumar dígitos individuales, se puede dar el caso que la suma sea mayor que 10
- En tal caso se debe sumar 6 (0110) y el carry que quede pasa para la siguiente cifra

$$Z = X + Y$$

If $Z \leq 9$, then $S = Z$ and carry-out $= 0$

if $Z > 9$, then $X = Z + 6$ and carry-out $= 1$

Diagrama de bloques de un sumador BCD

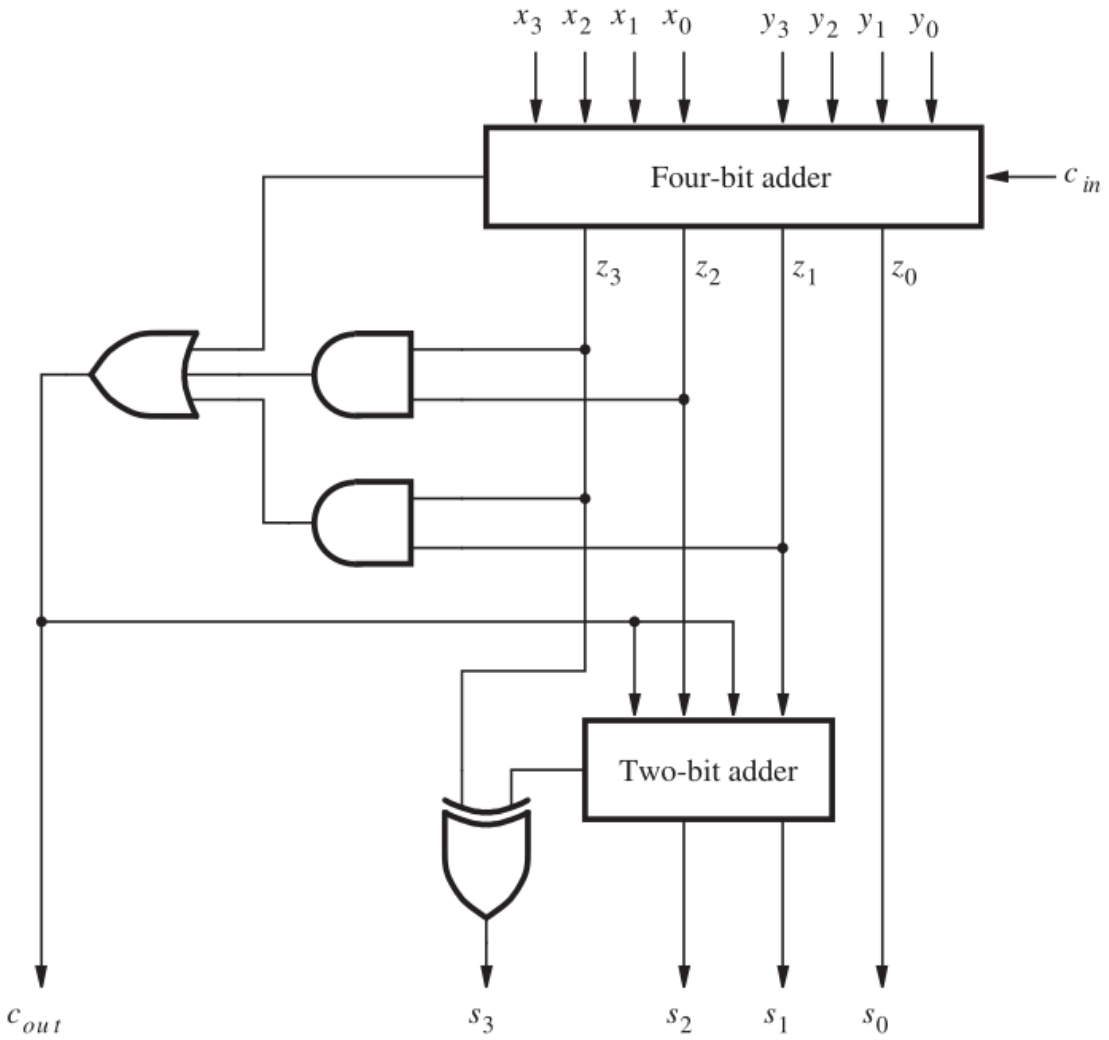



```
module bcdadd (Cin, X, Y, S, Cout);
    input Cin;
    input [3:0] X, Y;
    output [3:0] S;
    output Cout;
    reg [3:0] S;
    reg Cout;
    reg [4:0] Z;

    always @(X or Y or Cin)
    begin
        Z = X + Y + Cin;
        if (Z < 10)
            {Cout, S} = Z;
        else
            {Cout, S} = Z + 6;
    end

endmodule
```

Circuito de un sumador BCD



5.8 Código de caracteres ASCII

Bit positions	Bit positions 654							
3210	000	001	010	011	100	101	110	111
0000	NUL	DLE	SPACE	0	@	P	´	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	”	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	’	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	—	o	DEL

El Código ASCII (American standard code for information exchange) es un código para comunicaciones electrónicas donde cada variable es representada por 7 bits. Incluye caracteres imprimibles y palabras de control.

Más sobre ASCII

Bit positions	Bit positions 654							
3210	000	001	010	011	100	101	110	111
0000	NUL	DLE	SPACE	0	@	P	´	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	”	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	’	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	—	o	DEL

- Hay caracteres numéricos, alfabéticos, y símbolos
- Notar que los decimales son 011 seguido del dígito
- Existen caracteres de control
- ASCII es útil para codificar información como **texto, no como número**
- Son 7 bits que guardamos en un byte
  - El octavo bit puede ser usado como bit de **paridad (?)**

Bit de paridad

- Permite detectar errores **simples** en la transmisión de mensajes digitales
- Dos tipos: par o impar
  - Paridad par: el número de 1's en la palabra transmitida (incluyendo bit de paridad) es par
  - Paridad impar: el número de 1's en la palabra transmitida (incluyendo bit de paridad) es impar
- El bit de paridad es generado antes de enviar el dato
- Ejemplo: generación (p) y chequeo (c) de paridad en palabras de 4 bits:

$$p = x_3 \oplus x_2 \oplus x_1 \oplus x_0 \qquad c = p \oplus x_3 \oplus x_2 \oplus x_1 \oplus x_0$$