

Síntese do *datapath*

unidades funcionais

- Unidades funcionais do *datapath*
 - operadores aritméticos
 - operandos inteiros, vírgula fixa ou vírgula flutuante, dimensões de dados específicas
 - operadores com constantes são mais eficientes do que operadores genéricos
 - operadores lógicos, manipulação de bits
 - funções lógicas, rotações, deslocamentos, *bitreverse*,...
 - operadores combinados específicos para uma aplicação
 - $A*B - (C+D)$, $x[k] * x[k+t1] * x[k+t2]$
- Desenhando com modelos RTL + síntese
 - operadores inferidos e construídos no processo de síntese
 - implementados como circuitos combinacionais
 - otimizações conduzidas pelos parâmetros do processo de síntese
 - dimensões dos operadores inferida da dimensão dos operandos e resultado
 - só é suportada aritmética inteira/vírgula fixa
 - diferença entre **reg/wire** e **reg/wire signed** ?
 - *floating point* não é (para já...) suportado em ferramentas de síntese RTL (porquê?)

Operadores aritméticos

- Adição/subtração
 - *Ripple carry adder*
 - propagação do *carry* limita a rapidez de cálculo
 - resultado garantido após o maior tempo de propagação do *carry*
 - *Carry lookahead*
 - Carry em cada andar gerado com funções combinatórias das entradas
 - Reduz o tempo de propagação do carry
 - *Carry select*
 - Duplicar secções do somador para $C_{in}=1$ e $C_{in}=0$
 - Saída selecionada com o real valor do carryin.
 - *Carry save*
 - Reduz $A+B+C$ para $Z+Y$ em tempo constante, i.e. não depende do nº de bits.
 - Como cresce o tempo de propagação em árvores de somadores?
 - Se um somador de N bits tem $t_p=X$ ns, qual o t_p de M somadores em cascata?

Operadores aritméticos

adição e subtracção

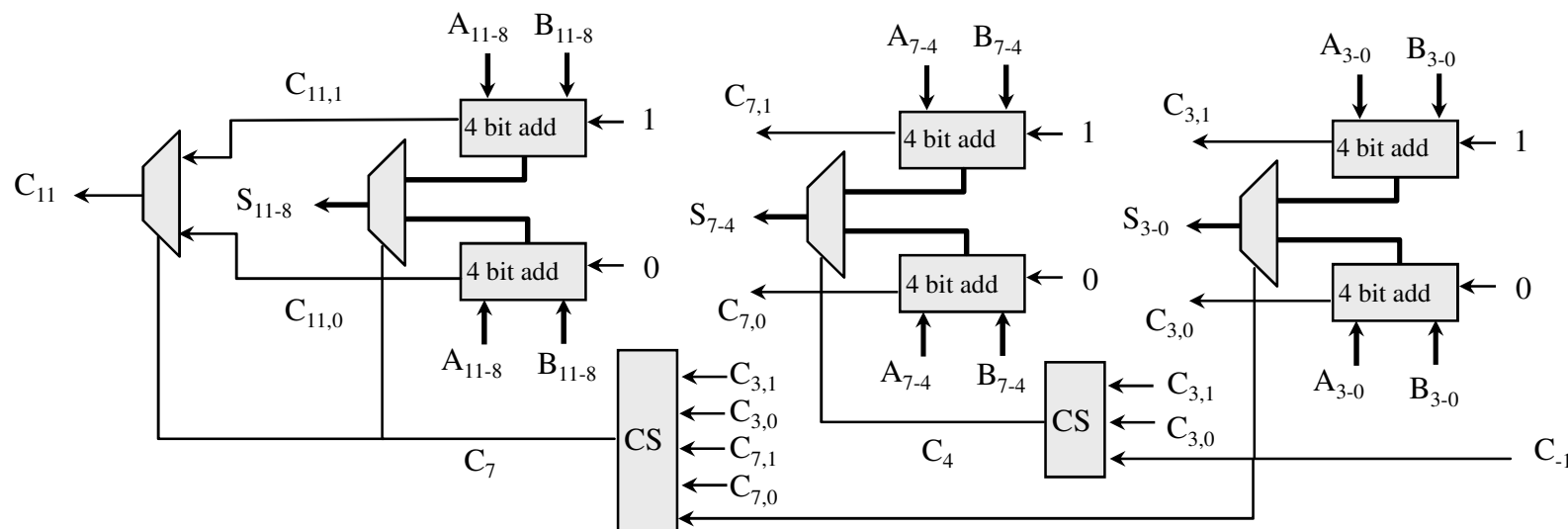
- somadores assíncronos
 - tempo de propagação depende dos operandos
 - circuitos para detetar que a propagação do *carry* está concluída
 - somador de 32 bits: média de 5 bits de *carry*
 - 6 para um somador de 64 bits
 - tempo pior é igual ou pior do que um somador *ripple carry*
- subtratores
 - em complemento para dois basta complementar o subtrator
 - trocar todos os bits (com XOR) e adicionar 1 fazendo $C_{-1}=1$
- melhor implementação depende da tecnologia
 - em FPGAs Xilinx e usando *fast carry logic*
 - somador *ripple carry* gerado pela síntese RTL melhor do que *carry select* ou *lookahead carry* “feito à mão” com funções lógicas
 - para ASICs há arquitecturas mais eficientes
 - usando portas complexas CMOS ou modelos ao nível do interruptor

Operadores aritméticos

adição e subtração

– carry select adder

- dividido em secções, cada secção inclui 2 somadores de k bits
- exemplo: somador de 12 bits em secções de 4 bits:



Kay Hwang, "Computer Arithmetic - Principles, Architecture and Design", John Wiley & Sons, 1990

Operadores aritméticos

adição e subtração

– *carry lookahead*

- *Generate carry at stage i* $G_i = A_i \cdot B_i$

- *Propagate carry at stage i* $P_i = A_i \oplus B_i$

- Boolean functions for a generic adder

$$S_i = A_i \oplus B_i \oplus C_{i-1} = P_i \oplus C_{i-1}$$

$$C_i = A_i \cdot B_i + A_i \cdot C_{i-1} + B_i \cdot C_{i-1} = \dots = G_i + P_i \cdot C_{i-1}$$

- All carry bits generated in parallel with a two level AND-OR logic circuit

$$C_1 = G_1 + C_0 \cdot P_1$$

$$C_2 = G_2 + C_1 \cdot P_2 = G_2 + (G_1 + C_0 \cdot P_1) \cdot P_2 = G_2 + G_1 \cdot P_2 + C_0 \cdot P_1 \cdot P_2$$

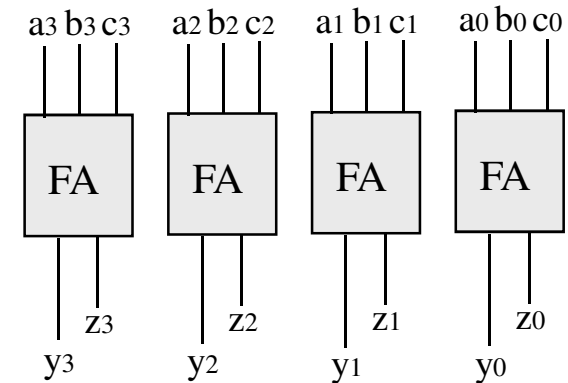
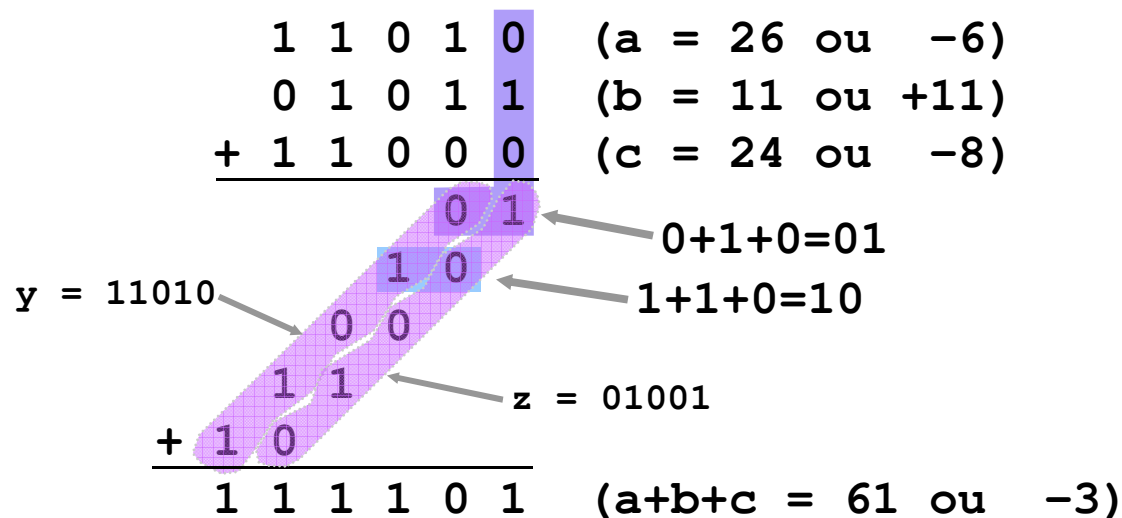
...

$$C_k = G_k + G_{k-1} \cdot P_k + G_{k-1} \cdot P_{k-1} \cdot P_k + \dots + C_{-1} \cdot P_0 \cdot P_1 \dots P_k$$

Operadores aritméticos

adição e subtração

- Adição de vários operandos
 - *Carry-save adder*
 - produz 2 resultados (**z** e **2y**) cuja soma é igual à adição dos 3 operandos
 - Os resultados **y** e **z** são gerados em tempo constante ($O(1)$)
 - Para obter o resultado **z+2y** é necessário um somador adicional

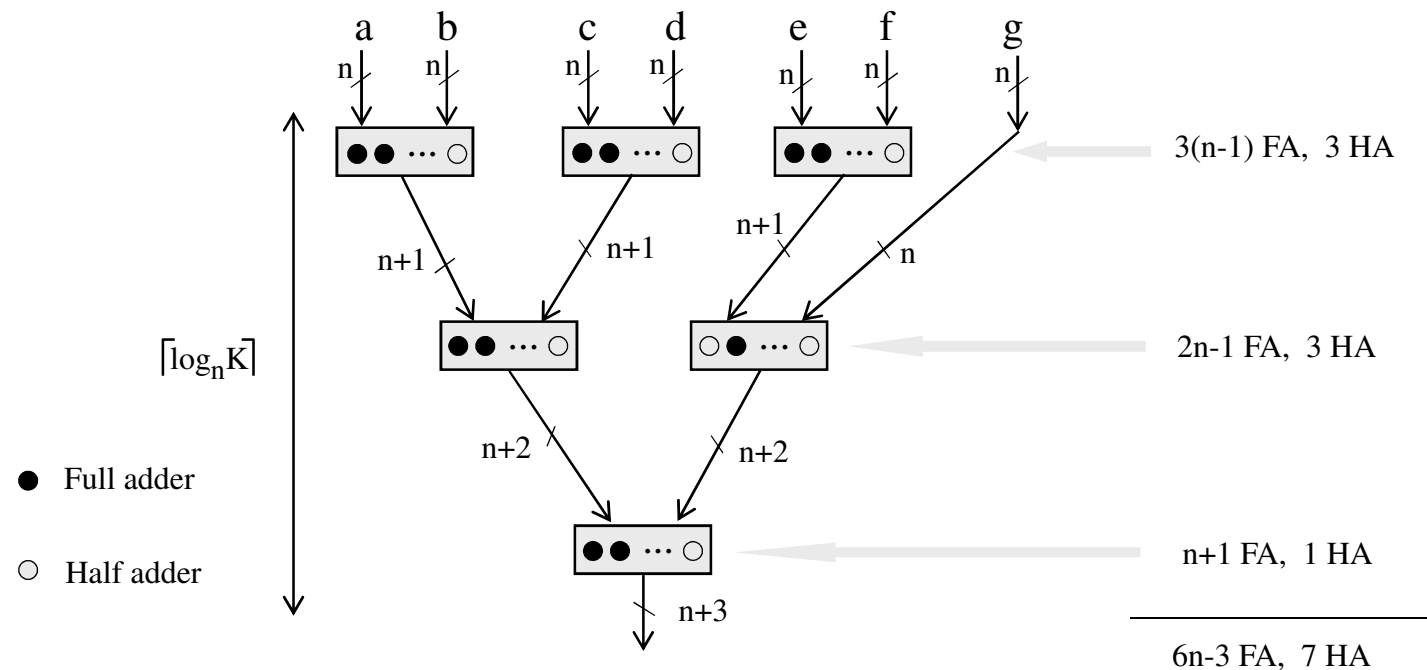


$$a+b+c = z+2y$$

Operadores aritméticos

adição e subtração

- Adição de vários operandos com RCA
 - Exemplo: adição de 7 operandos com n bits cada

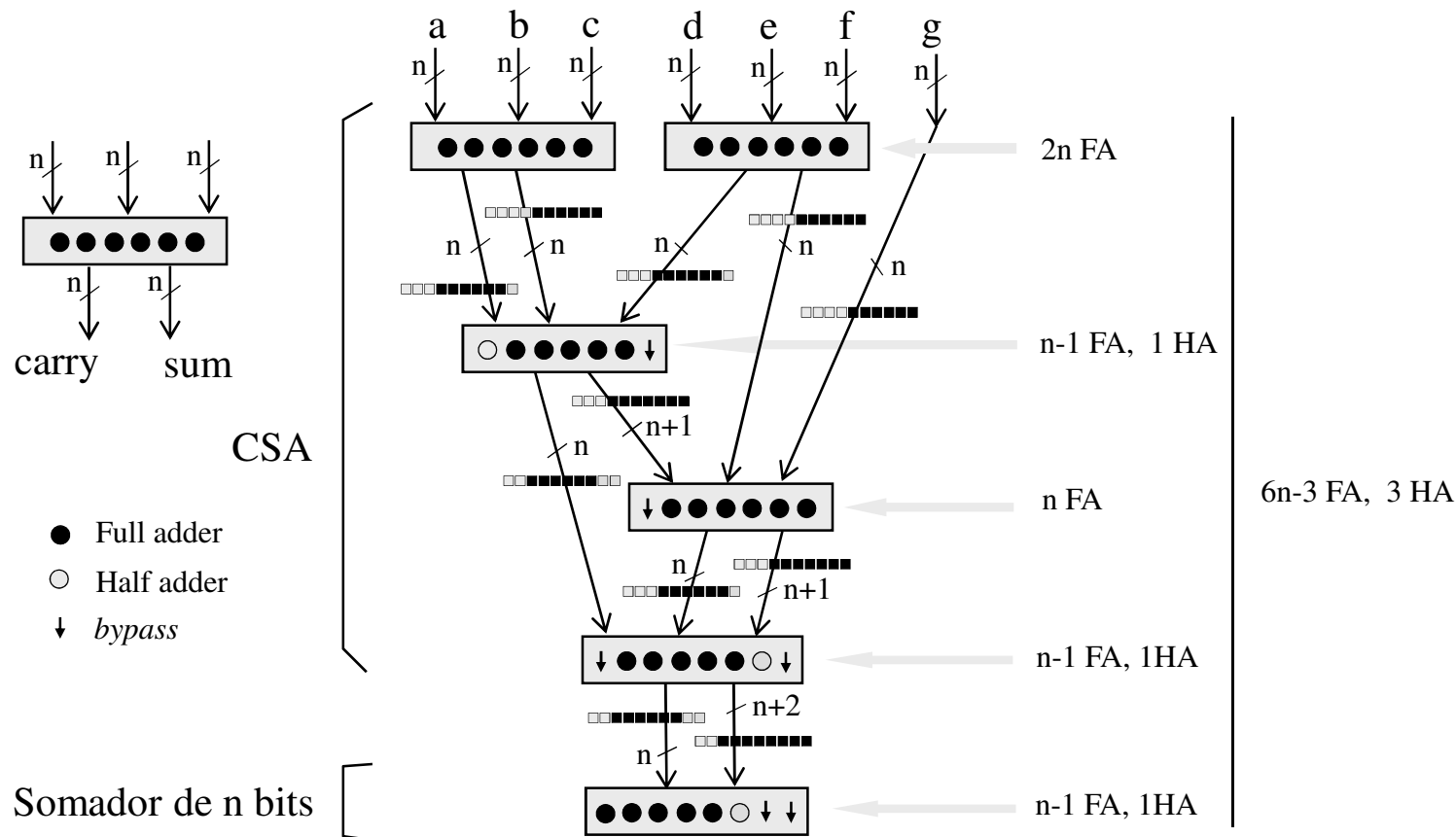


Tempo de propagação: $(n+3+1)tp_{FA}$

Operadores aritméticos

adição e subtração

- Árvore de CSA para adicionar n operandos



Adder tree

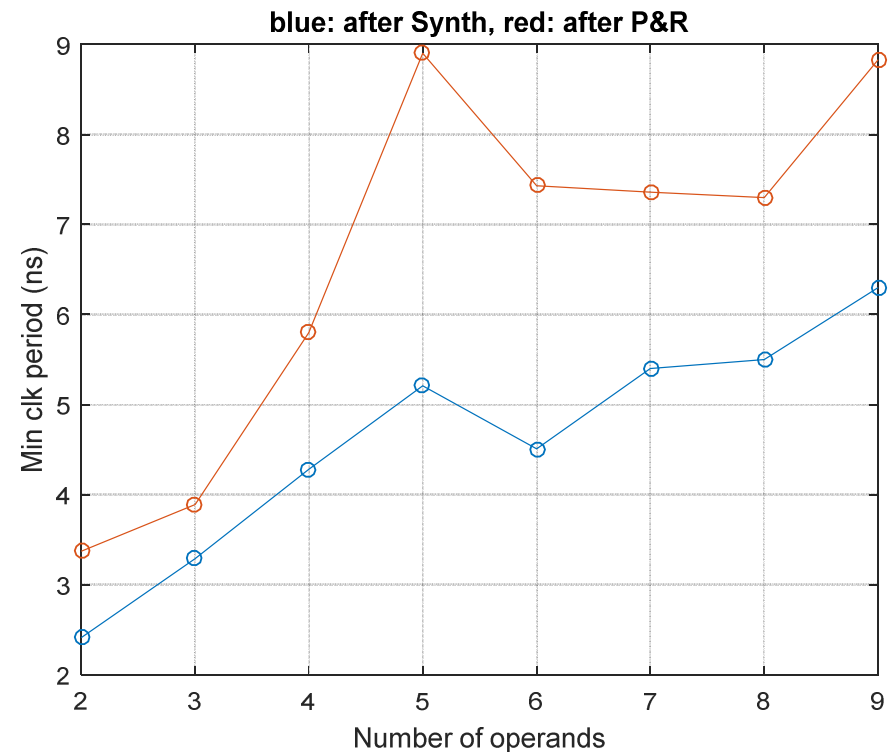
how propagation delays increase?

```
always @(posedge clock)
```

```
...
```

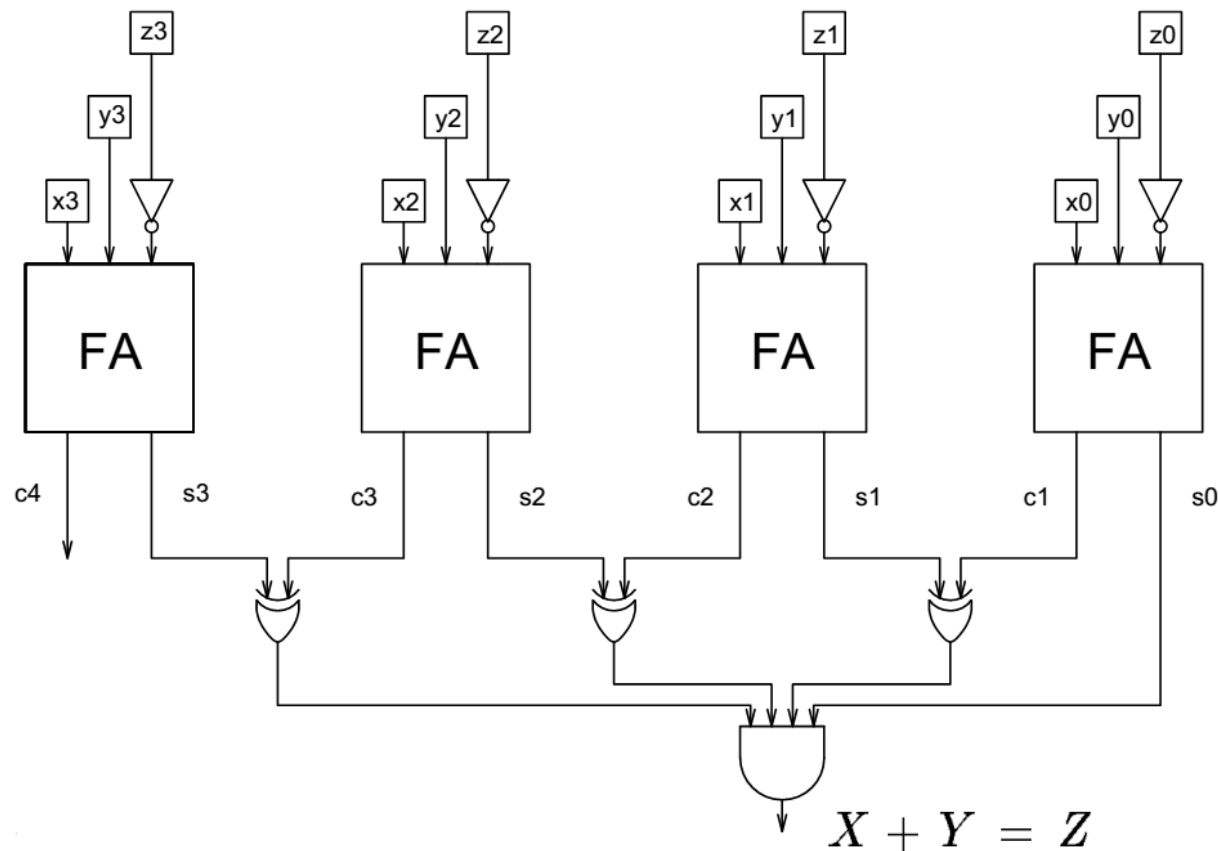
```
sum <= ra + rb + rc + rd + re + rf + rg + rh;
```

# operands	min period (after P&R)	
2	2.41	(3.37)
3	3.29	(3.89)
4	4.28	(5.80)
5	5.21	(8.90)
6	4.51	(7.43)
7	5.40	(7.36)
8	5.50	(7.30)
9	6.30	(8.83)



Carry-save adder como comparador

Theorem 1 Let X , Y , and Z be n -bit two's complement numbers, and let $(C, S) = X + Y + \overline{Z}$. Then $X + Y = Z \Leftrightarrow C$ and S differ in every bit position.



Operadores aritméticos

multiplicação e divisão de N por M bits

- Circuitos combinacionais
 - Arrays 2D de $N \times M$ full-adders com alguma lógica adicional
 - O multiplicador usa uma rede de ANDs para gerar os produtos parciais
 - O divisor usa XORs para criar uma rede de somadores/subtratores em casacata
 - Área em número de blocos (full-adder+...)
 - É proporcional ao produto do número de bits N e M dos dois operandos
 - Tempo de propagação
 - No multiplicador é proporcional à soma do número de bits dos operandos
 - No divisor é proporcional ao produto do número de bits dos operados
- Pipelined
 - Onde quebrar e meter os registos de pipeline?
- Circuitos sequenciais
 - circuito mais simples: uma iteração por bit do multiplicador / dividendo
 - redução do número de ciclos de relógio analisando 2 ou mais bits/iteração

Operadores aritméticos multiplicação


- Multiplicação binária

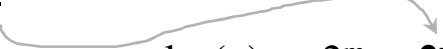
	sem sinal	complemento para dois
1101	(13)	(-3)
x 0101	(5)	(5)
1101		
0000		
1101		
0000		
01000001	(65	≠ -15)

- multiplicação de números com sinal
 - facilmente tratados com representação em sinal e magnitude
 - custo: complementar os operandos e o resultado
 - Custo equivalente a um somador/subtrator


Operadores aritméticos multiplicação

- Multiplicação binária em complemento para dois
 - se o multiplicador for negativo
 - basta subtrair o último produto parcial (é igual a zero se for positivo)
 - Recordando a representação complemento para dois
 - para um número positivo com m bits
 - se for negativo
 - para os dois casos


$$\text{valor}(x) = \mathbf{0} + X_{m-2} \cdot 2^{m-2} + \dots + X_1 2^1 + X_0 \cdot 2^0$$


$$\text{valor}(x) = -2^m + \mathbf{2^{m-1}} + X_{m-2} \cdot 2^{m-2} + \dots + X_1 2^1 + X_0 \cdot 2^0$$

$$\text{valor}(x) = - X_{m-1} \cdot 2^{m-1} + X_{m-2} \cdot 2^{m-2} + \dots + X_1 2^1 + X_0 \cdot 2^0$$



bit de sinal X_{m-1} tem peso -2^{m-1}

Operadores aritméticos multiplicação

- Multiplicação binária, complemento para dois
 - se só for negativo o multiplicando
 - basta estender o sinal dos produtos parciais

extensão de sinal →

$$\begin{array}{r}
 1101 \quad (-3) \\
 \times 0101 \quad (5) \\
 \hline
 1111101 \\
 000000 \\
 11101 \\
 \hline
 0000 \\
 \hline
 11110001 \quad (-15)
 \end{array}$$

Operadores aritméticos

multiplicação

- Multiplicação em complemento para dois
 - se o multiplicador for negativo

se o multiplicador
for positivo, o último
produto parcial é zero!

0101	(5)	
x 1101	(-3)	
0101		
0000		} ← somar os 3 produtos parciais
+ 0101		
0011001		
- 0101		} ← subtraír o último produto parcial
11110001	(-15)	

- ... e se ambos negativos

extensão de sinal

→

→

1101	(-3)	
x 1101	(-3)	
111 1101		
00 0000		} ← somar os 3 produtos parciais
+ 1 1101		
11110001		
- 1 1101		} ← subtraír o último produto parcial
00001001	(+9)	

- implementação: fácil incluir no multiplicador *shift-add*!

Operadores aritméticos multiplicação

- Multiplicador iterativo (*shift-add*)
 - multiplicando=Md[m-1:0], multiplicador=Mr[n-1:0]
 - $Acc[m+n:0] = 0$
 - para cada bit k do multiplicador desde 0 até n-1
 - se $Mr_k = 1$

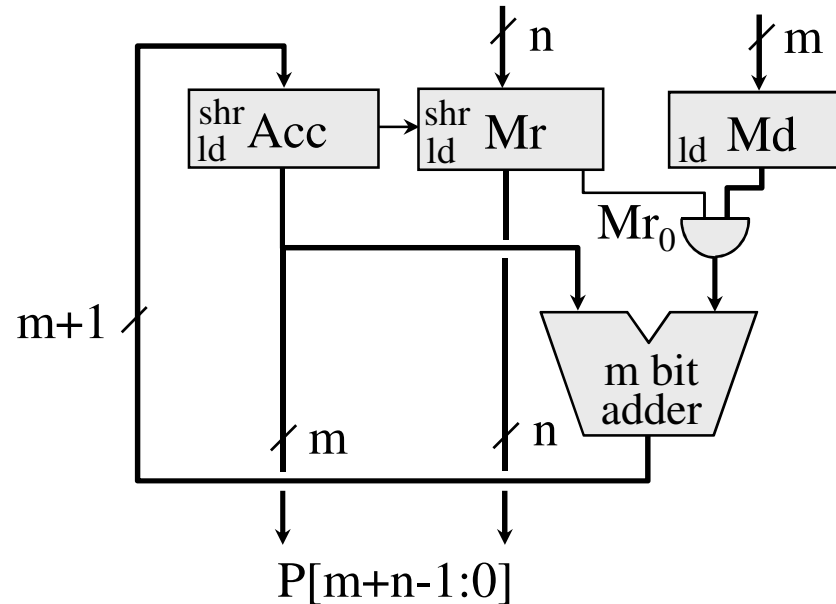
$$Acc[m+n:n] = Acc[m+n-1:n] + Md[m-1:0]$$
 - $Acc = Acc \gg 1$
 - produto = $Acc[m+n-1:0]$
 - exemplo:

Md	Mr	k	Mr_k	Acc	oper.
1101	0101	–	–	00000:0000	
1101	010 1	0	1	01101 :0000	add
				00 110 :1000	shift
1101	01 0 1	1	0	00110 :1000	add
				00 011 :0100	shift
1101	0 1 01	2	1	10000 :0100	add
				0 1000 :0010	shift
1101	0 101	3	0	01000 :0010	add
				00 100 :0001	shift

Operadores aritméticos multiplicação

- Multiplicador *shift-add*

- **Mr** é o multiplicador, **Md** é o multiplicando
 - em cada ciclo, os **m+1** bits do produto parcial são carregados em **Acc**
 - o resultado parcial é deslocado para o registo que contém inicialmente **Mr**
 - calcula o produto em **n** iterações (**n** = número de bits do multiplicador)



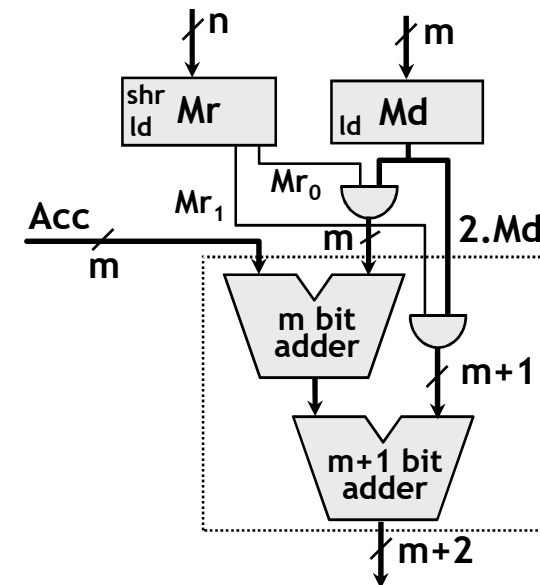
Operadores aritméticos

multiplicação sequencial

- Multiplicador *shift-add*
 - avaliação de k bits do multiplicador de cada vez
 - número de ciclos reduzido k vezes (*shift* de k bits de cada vez)
 - exemplo para k=2

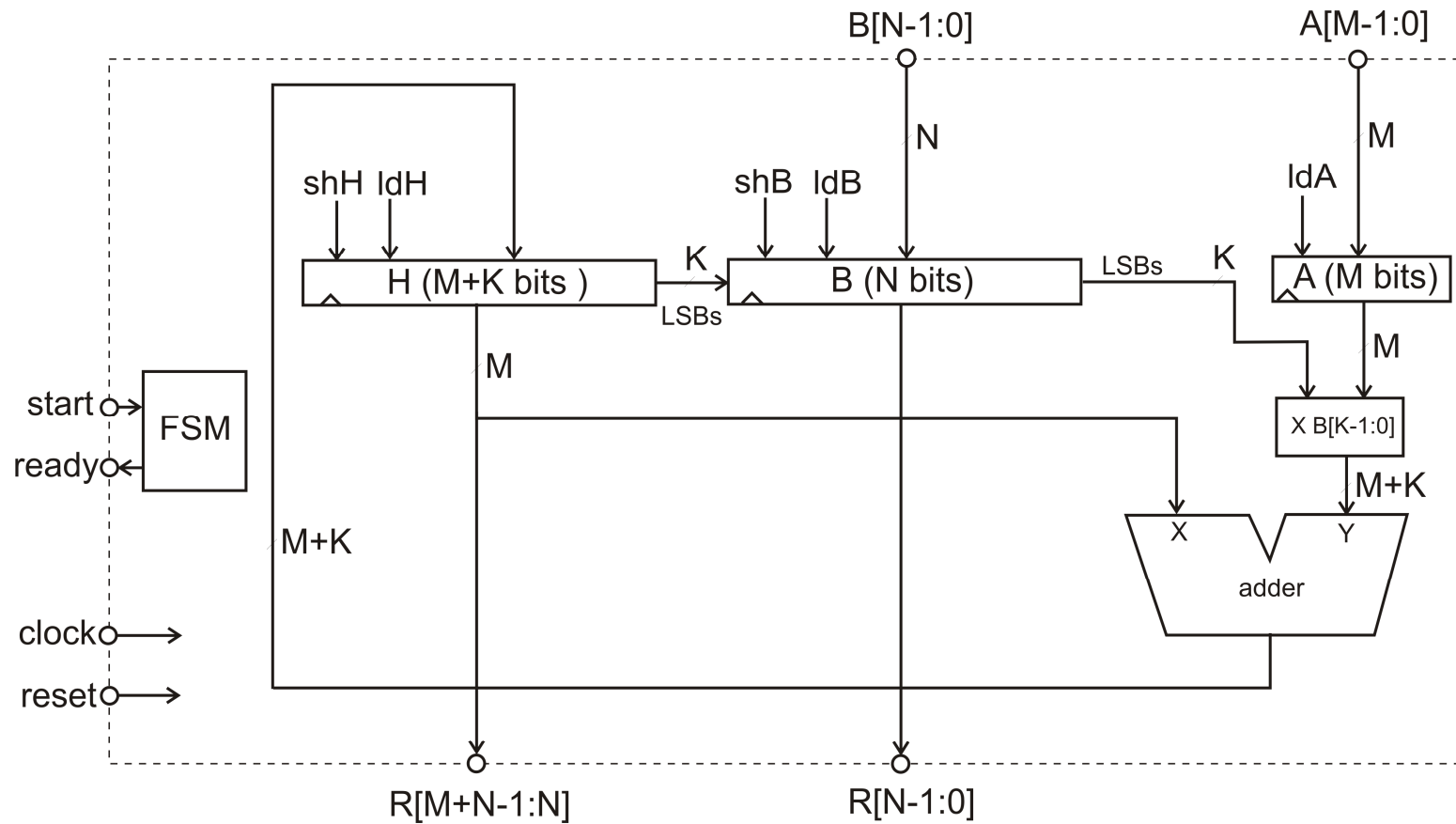
Mr1	Mr0	somar a Acc
0	0	0
0	1	Md
1	0	2.Md
1	1	2.Md+Md

- aumento da complexidade do somador
 - é necessário somar 3.Md
 - necessário 2 somadores em cascata



Multiplicador sequencial

(K bits do multiplicador por iteração)



Operadores aritméticos multiplicação

- Multiplicação - recodificação de Booth

- pela propriedade:

$$2^{i+k} - 2^i = 2^{i+k-1} + 2^{i+k-2} + 2^{i+1} + 2^i$$

- pode substituir-se

1 0 0 **1 1 1** 0 0 1

- por

1 0 **1** 0 0 **$\bar{1}$** 0 0 1

2^{i+k} $- 2^i$

- objectivo: eliminar sequências de uns que conduzem a 3 x Md

00 10 11 10 11 01 10	fatores: 0 2 3 2 3 1 2
00 10 11 10 11 10 $\bar{1}$0	
00 10 11 1 1 00 $\bar{1}$0 $\bar{1}$0	
00 1 1 00 0 $\bar{1}$00 $\bar{1}$0 $\bar{1}$0	
01 0 $\bar{1}$00 0 $\bar{1}$00 $\bar{1}$0 $\bar{1}$0	fatores: 1 -1 0 -1 0 -2 -2

- permite evitar a multiplicação pelo fator 3 (3.Md = 2.Md+Md)

Operadores aritméticos multiplicação

- Multiplicação - recodificação de Booth

- algoritmo

- percorrer todos os *bits* desde o lsb até encontrar um 1
 - trocar esse 1 por 1 e percorrer uns até encontrar um zero
 - trocar esse 0 por 1 e continuar

- exemplo

$$0011\ 1101\ 1001 = 512 + 256 + 128 + 64 + 16 + 8 + 1 = 985$$

$$0100\ 0\bar{1}10\ \bar{1}01\bar{1} = 1024 - 64 + 32 - 8 + 2 - 1 = 985$$

- tabela de recodificação, analisando 2 bits de cada vez

- é necessário acrescentar um bit zero à direita do lsb: $b_{-1}=0$

b_i	b_{i-1}	z_i	valor	caso
0	0	0	0	cadeia de zeros
0	1	1	+1	fim dos uns
1	0	$\bar{1}$	-1	inicio dos uns
1	1	0	0	cadeia de uns

Operadores aritméticos multiplicação

- Multiplicação - recodificação de Booth
 - para cada bit do multiplicador
 - se é 0 não soma nada
 - se é 1 soma o multiplicando
 - se é $\bar{1}$ soma o simétrico do multiplicando
 - não é necessário recodificar o multiplicador explicitamente
 - exemplo

$$(-7) \times (-5) = +35$$

$$-7 = 1001$$

$$-5 = 1011$$

recodificar -5
(ver tabela)

1011 0
1011 0
1011 0
1011 0

$\bar{1}10\bar{1}$

$$\begin{array}{r} 1001 \\ \times \bar{1}10\bar{1} \\ \hline 00000111 \\ 00000000 \\ 111001 \\ + 00111 \\ \hline 00100011 \end{array}$$

$$\begin{array}{r} -7 \\ \times -5 \\ \hline -(-7) \times 2^0 \\ + (0) \times 2^1 \\ + (-7) \times 2^2 \\ + -(-7) \times 2^3 \\ \hline 7 - 28 + 56 = 35 \end{array}$$

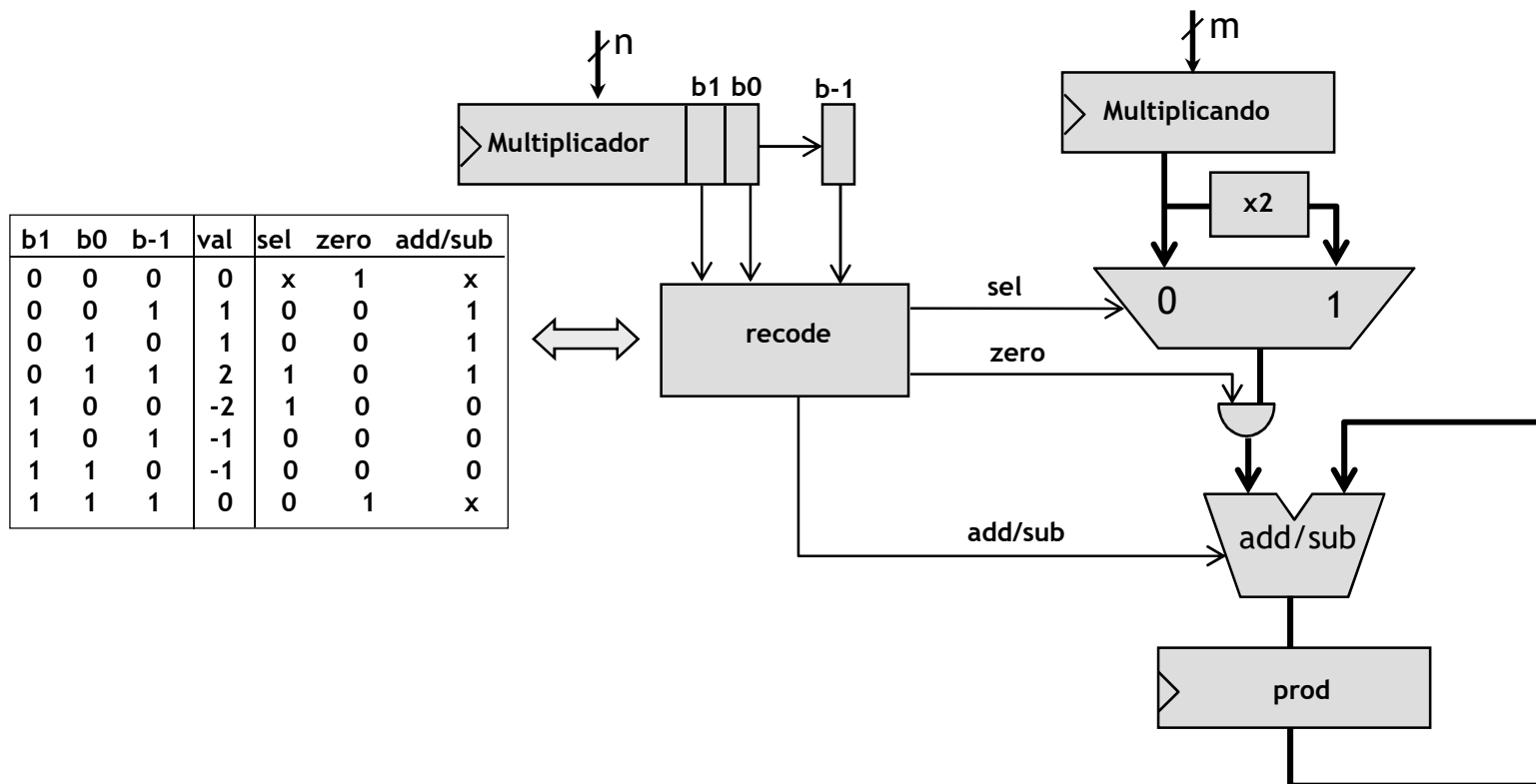
Operadores aritméticos multiplicação

- Multiplicador de Booth
 - recodificando pares de *bits* em dígitos com sinal
 - são analisados 3 *bits* de cada vez
 - cada par de *bits* produz uma multiplicação parcial por 0, +1, +2, -1 ou -2
 - reduz para metade o número de iterações (*shift* de 2 bits de cada vez)
 - tabela de recodificação

b_i	b_{i-1}	b_{i-2}	z_i	z_{i-1}	fator	caso
0	0	0		0	0	cadeia de zeros
0	0	1		1	+1	fim de uns
0	1	0		1	+1	1 isolado
0	1	1	1		+2	fim de uns
1	0	0	$\bar{1}$		-2	início de uns
1	0	1		$\bar{1}$	-1	zero isolado
1	1	0		$\bar{1}$	-1	início de uns
1	1	1		0	0	cadeia de uns

Operadores aritméticos multiplicação

- Multiplicador de Booth (n/2 iterações)



Operadores aritméticos multiplicação

- Multiplicação paralela (*unsigned*)

					a4	a3	a2	a1	a0
					b4	b3	b2	b1	b0
					a4.b0	a3.b0	a2.b0	a1.b0	a0.b0
				a4.b1	a3.b1	a2.b1	a1.b1	a0.b1	
			a4.b2	a3.b2	a2.b2	a1.b2	a0.b2		
		a4.b3	a3.b3	a2.b3	a1.b3	a0.b3			
	a4.b4	a3.b4	a2.b4	a1.b4	a0.b4				
p9	p8	p7	p6	p5	p4	p3	p2	p1	p0

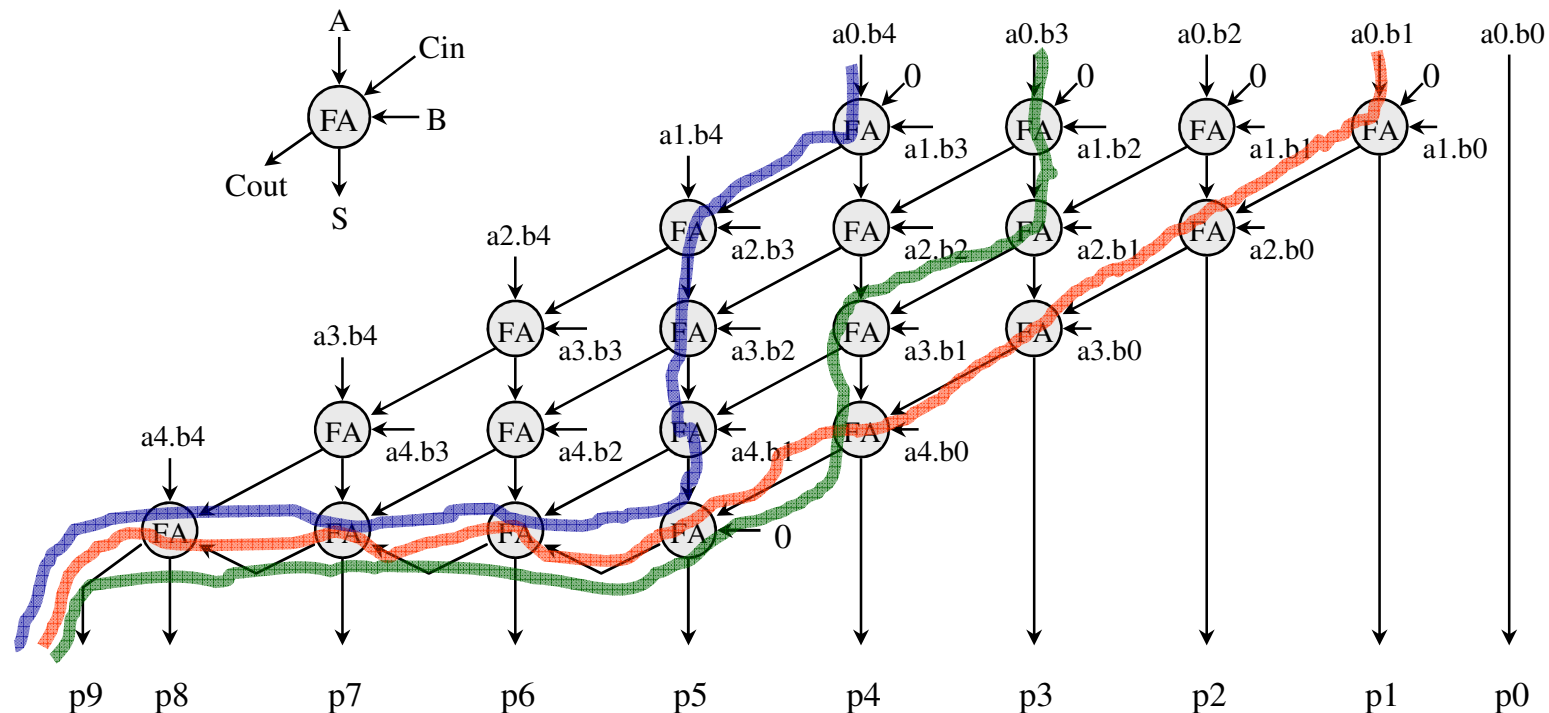
multiplicação de 2 números de $n \times m$ bits:

m produtos parciais (n ANDs cada): $n \times m$ ANDs

$m-1$ somadores de n bits para somar os produtos parciais

Operadores aritméticos multiplicação

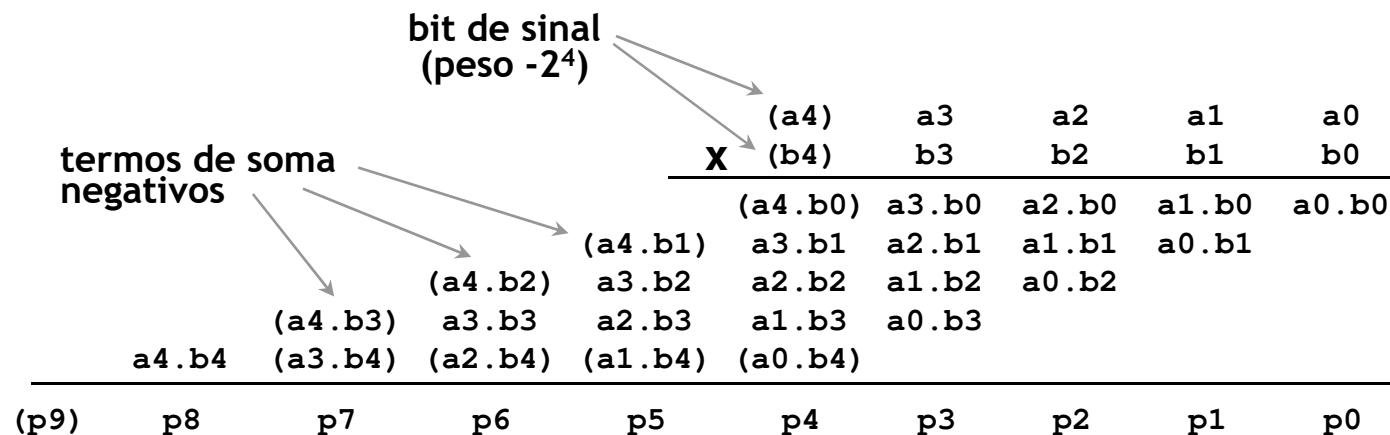
- array multiplicador (*unsigned*)
 - cada nó é um *full adder* (somador de 3 bits)



caminho crítico: $a0.b_i \rightarrow p9: ((n-1) + (n-1)) \times tp_{FA}$

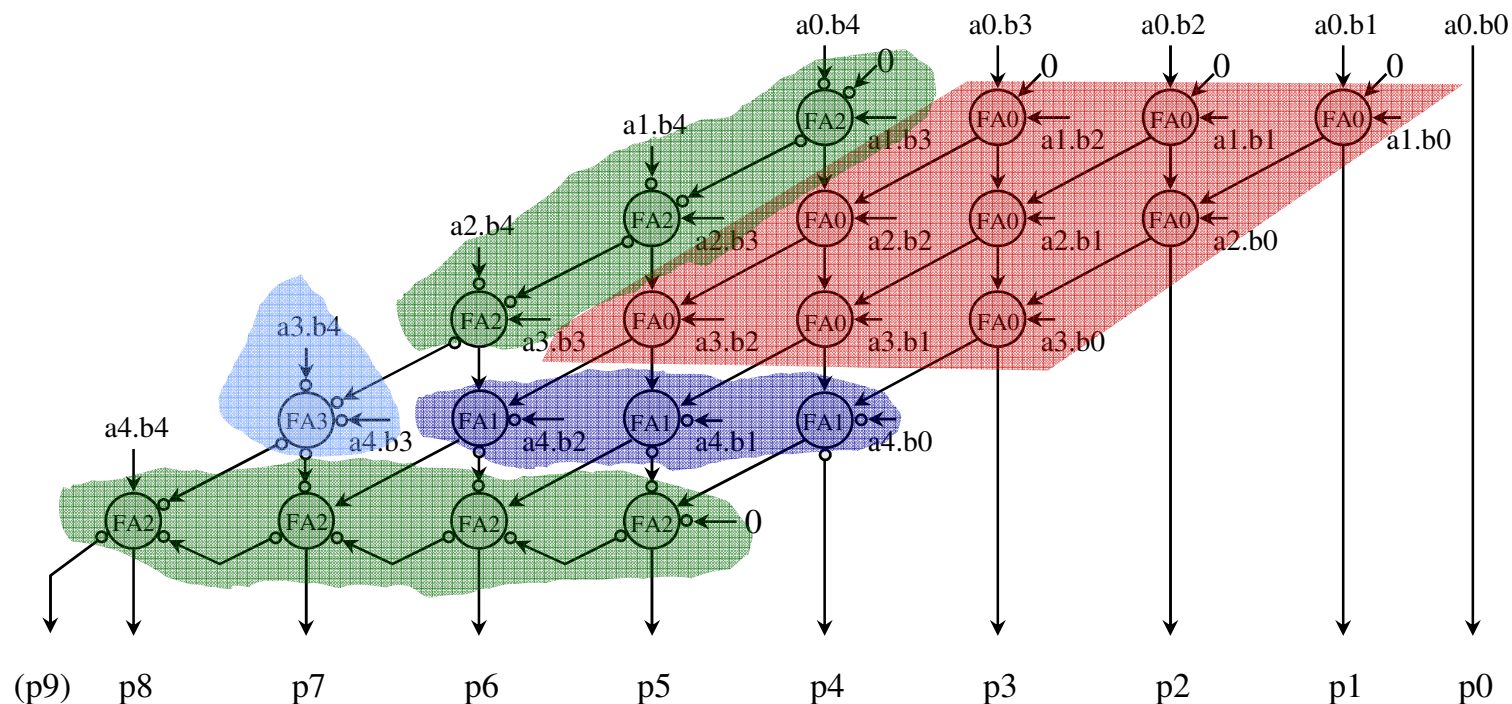
Operadores aritméticos multiplicação

- multiplicação paralela (*signed*)



Operadores aritméticos multiplicação

- array multiplicador de Pezaris ($n \times n$, *signed*)
 - 4 tipos de *full adders* diferentes



Operadores aritméticos multiplicação

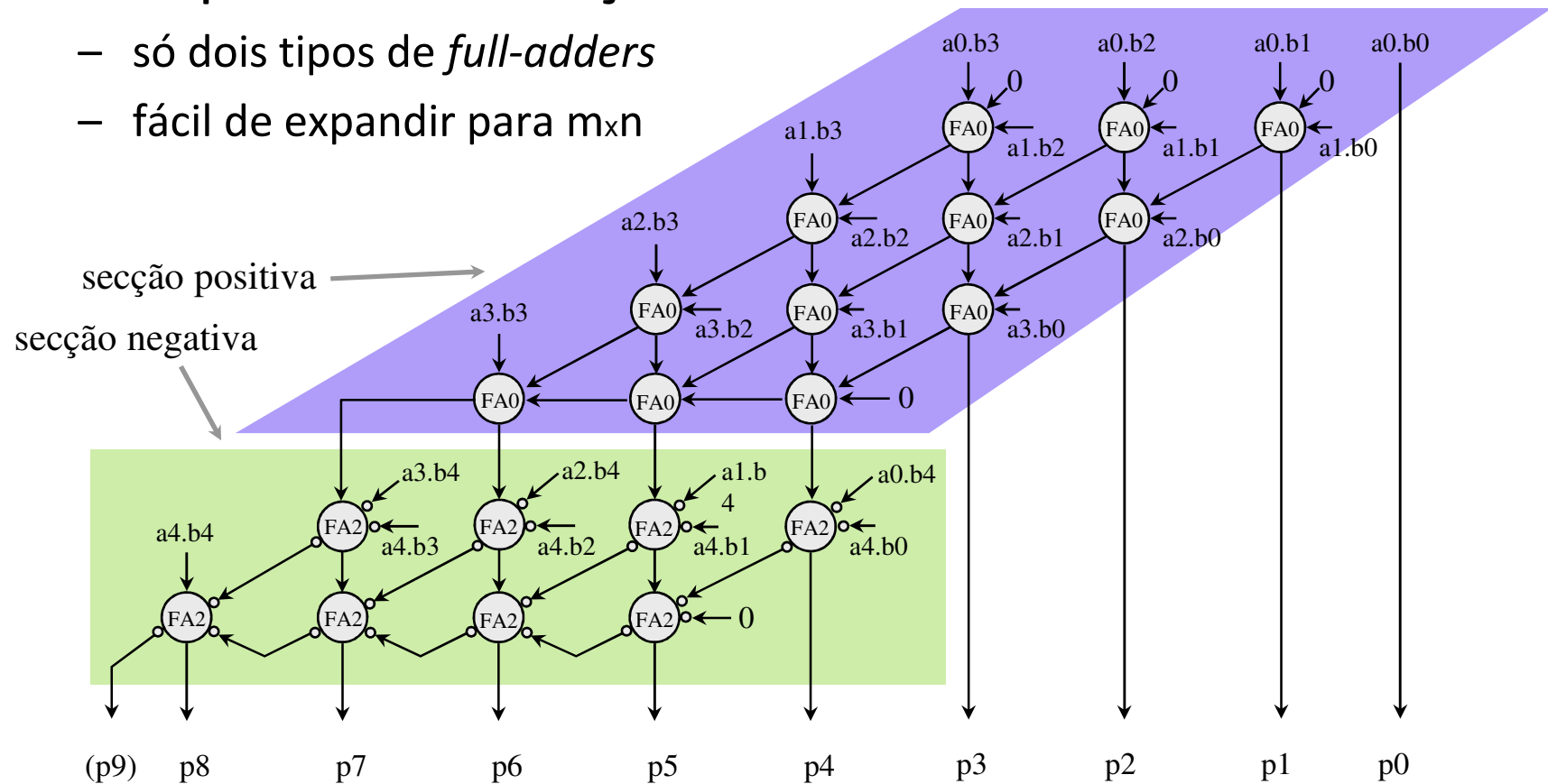
- multiplicador bi-secção (*signed*)
 - separando os termos positivos dos negativos

					(a4)	a3	a2	a1	a0	
					x	(b4)	b3	b2	b1	b0
secção positiva	{						a3.b0	a2.b0	a1.b0	a0.b0
						a3.b1	a2.b1	a1.b1	a0.b1	
					a3.b2	a2.b2	a1.b2	a0.b2		
		a4.b4	0	a3.b3	a2.b3	a1.b3	a0.b3			
secção negativa	{		(a4.b3)	(a4.b2)	(a4.b1)	(a4.b0)				
			(a3.b4)	(a2.b4)	(a1.b4)	(a0.b4)				
(p9)	p8	p7	p6	p5	p4	p3	p2	p1	p0	

Operadores aritméticos multiplicação

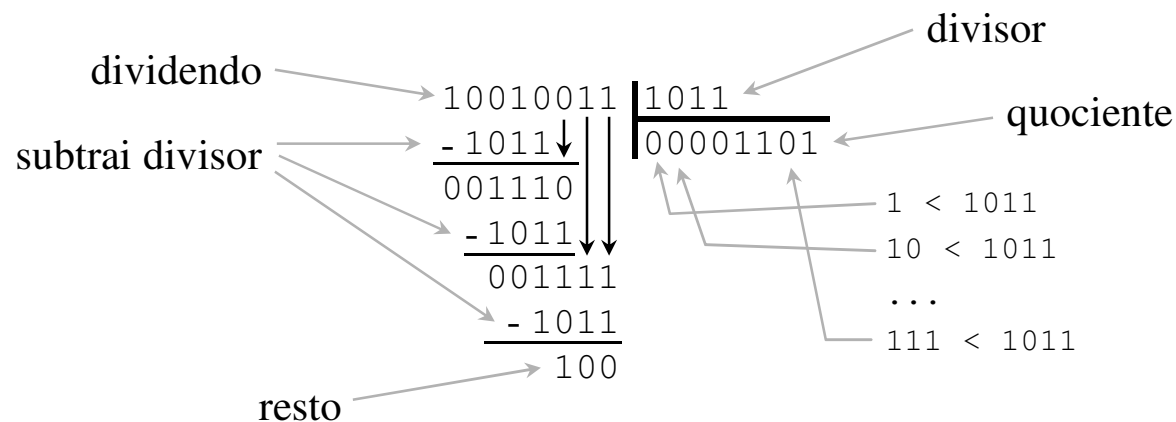
- multiplicador bi-secção

- só dois tipos de *full-adders*
- fácil de expandir para $m \times n$



Operadores aritméticos – divisão

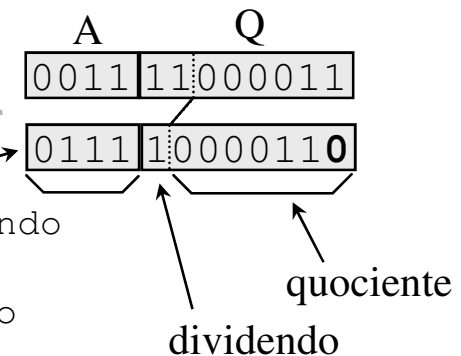
- divisão de números inteiros sem sinal
 - processo parecido com a multiplicação: *shift-subtract*
 - o resultado de uma subtração define a próxima operação
 - dependência entre as várias operações
 - mais complexo do que a multiplicação
 - exemplo (sem sinal): $147/11=13$, $147\%11=4$



Operadores aritméticos – divisão

- divisão *unsigned* – algoritmo “restoring”

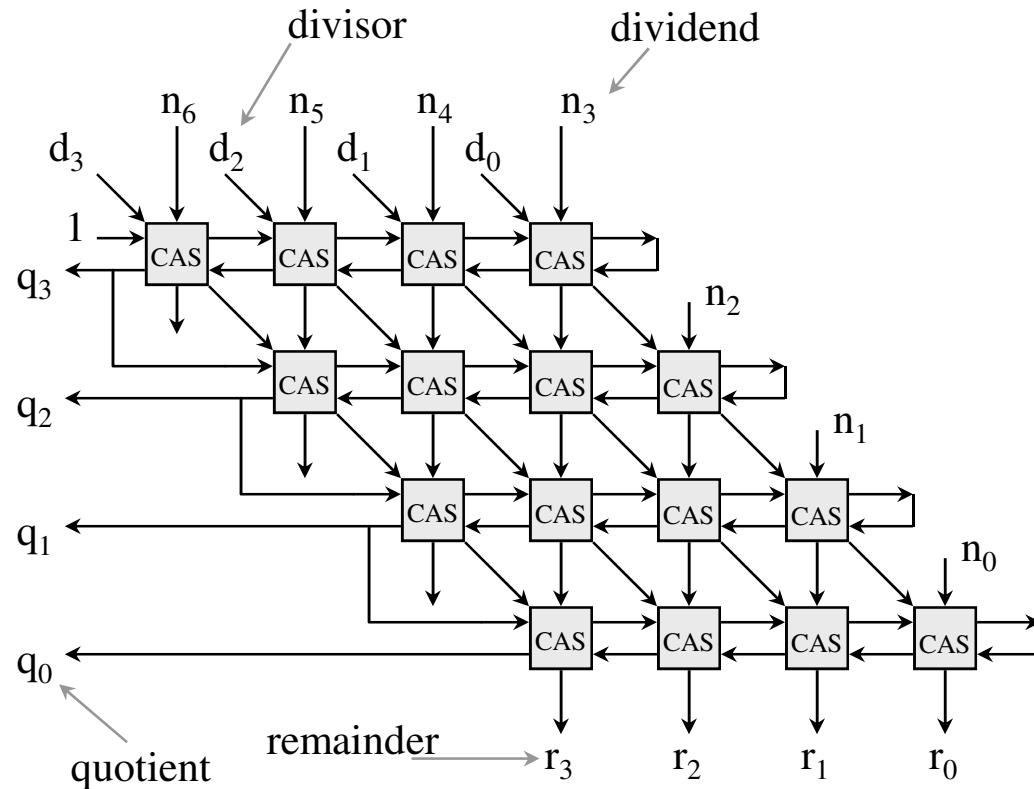
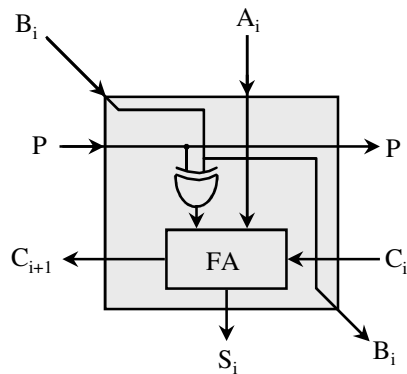
```
A=0;           // fica com o resto, n bits
M=Divisor;     // n bits
Q=Dividendo;   // “apanha” quociente, m bits
cnt=m;
repeat
  {A,Q} = {A,Q} << 1;
  A = A - M; // subtrai o divisor ao dividendo
  if (A>=0)
    Q[0] = 1; // divisor “coube” no dividendo
  else
    begin
      Q[0] = 0; // divisor não “coube” no dividendo
      A = A + M; // repõe o valor do dividendo
    end
  cnt = cnt - 1;
until (cnt==0);
```



Operadores aritméticos

divisor combinacional

CAS
(Controlled Add/Subtract)



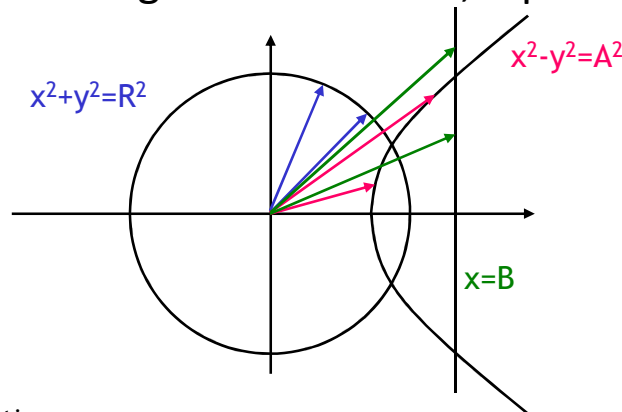
Síntese do *datapath* outros operadores

- Implementações específicas de uma aplicação
 - quadrado
 - $A \times A$ é mais eficiente do que a multiplicação
 - operações com constantes
 - propagação de constantes simplifica o *hardware* (geralmente...)
 - multiplicação traduzida em *shifts*, adições e subtracções
 - raíz quadrada
 - aproximação algorítmica
 - funções transcendentais
 - aproximações por tabelas, polinómios ou segmentos de recta
 - domínio da função e precisão pretendida condiciona o método

CORDIC

- **CO**ordinate **R**otation **D**igital **C**omputer

- "canivete suíço" para cálculo de funções transcendentas
 - \sin , \cos , \cosh , \sinh , atan , \ln , \exp , $\sqrt{}$, produto, divisão
- processo iterativo, calcula um bit do resultado por iteração (aprox.)
- Reduzida complexidade: somadores, comparadores, LUT e *shifters*
- proposto em 1959 (Volder) para o cálculo de funções trigonométricas para navegação de aviões em tempo real, generalizado por Walther em '71
 - usado na calculadora HP35 (1972) (Google "*HP35 voidware*")
- "rotação" de um vector ao longo de um círculo, hipérbole ou recta



Scott Hauck, André DeHon, "Reconfigurable Computing - the theory and practice of FPGA-based computing " (capítulo 25)

CORDIC

- Rodar um vector no plano (x,y) de θ graus
 - vector inicial (xs,ys), vector final (xf, yf)

$$\begin{bmatrix} x_f \\ y_f \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x_s \\ y_s \end{bmatrix} = \text{ROT}(\theta) \begin{bmatrix} x_s \\ y_s \end{bmatrix}$$

- Definir θ como um conjunto de "micro-rotações" de ângulos α_i

$$\theta = \sum \alpha_i$$

$$\text{ROT}(\theta) = \prod \text{ROT}(\alpha_i)$$

$$\begin{aligned} x_{i+1} &= x_i \cos \alpha_i - y_i \sin \alpha_i & x_{i+1} &= \cos \alpha_i (x_i - y_i \tan \alpha_i) \\ y_{i+1} &= x_i \sin \alpha_i + y_i \cos \alpha_i & y_{i+1} &= \cos \alpha_i (y_i + x_i \tan \alpha_i) \end{aligned}$$

CORDIC

- Escolhendo os micro-ângulos α_i adequados

$$\alpha_i = \text{atan}(\delta_i 2^{-i}), \text{ com } \delta_i = \{-1, +1\}$$

$$x_{i+1} = \cos \alpha_i (x_i - y_i 2^{-i} \delta_i)$$

$$y_{i+1} = \cos \alpha_i (y_i + x_i 2^{-i} \delta_i)$$

- à parte $\cos \alpha_i$, são só necessárias somas, subtracções e deslocamentos
- os valores de δ_i são +1 ou -1 para que o processo convirja
- define-se uma nova variável z_i que representa o ângulo de rotação:

$$z_{i+1} = z_i - \text{atan}(\delta_i 2^{-i}) = z_i - \delta_i \text{atan}(2^{-i})$$

- iniciando z_i com θ , ($z_0 = \theta$), Z é conduzido a zero somando ou subtraindo parcelas iguais a $\text{atan}(2^{-i})$, fazendo $\delta_i = +1$ ou -1

$$\delta_i = +1 \text{ se } z_i \geq 0 \qquad \delta_i = -1 \text{ se } z_i < 0$$

CORDIC

rotation mode

- Rotação de um vetor x_s, y_s de um ângulo θ :
 - Fazer $x_0, y_0 = x_s, y_s$ e $z_0 = \theta$
$$x_{i+1} = x_i - y_i 2^{-i} \delta_i$$
$$y_{i+1} = y_i + x_i 2^{-i} \delta_i$$
$$z_{i+1} = z_i - \delta_i \operatorname{atan}(2^{-i}), \text{ com } \delta_i = \operatorname{sign}(z_i)$$
- O módulo do vetor final (i.e. x_f e y_f) é escalado de:
 - Em cada rotação de um valor igual a $\frac{1}{\cos \alpha_i}$
 - No final do processo: $K = \prod_i \frac{1}{\cos \alpha_i}$

$$K = \prod_i \sqrt{1 + \tan^2 \alpha_i} = \prod_i \sqrt{1 + 2^{-2i}} \approx 1.64676026$$

Este valor não depende do número de iterações !

CORDIC

rotation mode

- Para calcular $\sin(\theta)$ e $\cos(\theta)$ roda-se de θ o vetor $(1,0)$:
 - $x_0 = 1/K$, $y_0 = 0$ e $z_0 = \theta$
 - No final $x_f = \cos(\theta)$ e $y_f = \sin(\theta)$
- No caso geral para a rotação do vetor (a,b) :
 - $X_0 = a/K$, $y_0 = b/K$ e $z_0 = \theta$
 - No final $x_f = a \cos(\theta) - b \sin(\theta)$ e $y_f = a \sin(\theta) + b \cos(\theta)$
(o vetor (a,b) rodado do ângulo θ)
- Transformar coordenadas polares em retangulares
 - vetor com módulo M e ângulo θ : $X_0 = M/K$, $y_0 = 0$ e $z_0 = \theta$
 - No final $x_f = M \cos(\theta)$ e $y_f = M \sin(\theta)$ (coordenadas retangulares)

CORDIC

vectoring mode

- Rodar um vetor x_s, y_s até y ser igual a zero
 - Iniciar com $x_0 = x_s, y_0 = y_s$ e $z_0 = 0$
 - com $\delta_i = -\text{sign}(y_i)$
 - A variável z acumula as micro-rotações que levam y a zero
 - $z_{i+1} = z_i - \delta_i \text{atan}(2^{-i})$,
- No final obtém-se
 - $x_f = K \sqrt{(x_s^2 + y_s^2)}$, $y_f = 0$ e $z_f = \text{atan}(y_s / x_s)$
 - Transforma coordenadas retangulares em polares

CORDIC

coordenadas lineares

- Rodar um vetor ao longo da reta $x = x_s$

$$x_R = x_s \quad y_R = y_s + x_s z_s$$

- *Rotation mode:*
$$x_{i+1} = x_i$$
$$y_{i+1} = y_i + x_i 2^{-i} \delta_i$$
$$z_{i+1} = z_i - \delta_i 2^{-i}, \text{ com } \delta_i = \text{sign}(z_i)$$
$$x_f = x_s, \text{ **y}_f = \text{y}_s + \text{x}_s \text{z}_s \text{ e } z_f = 0**$$
- *Vectoring mode:*
$$x_{i+1} = x_i$$
$$y_{i+1} = y_i + x_i 2^{-i} \delta_i$$
$$z_{i+1} = z_i - \delta_i 2^{-i}, \text{ com } \delta_i = -\text{sign}(y_i)$$
$$x_f = x_s, y_f = 0 \text{ e } \text{**z}_f = \text{z}_s + \text{y}_s / \text{x}_s**$$

CORDIC

coordenadas hiperbólicas

- Rodar o vetor (xs,ys) ao longo de uma hipérbole
 - vector inicial (xs,ys), vector final (xf, yf)

$$\begin{bmatrix} x_f \\ y_f \end{bmatrix} = \begin{bmatrix} \cosh \theta & \sinh \theta \\ \sinh \theta & \cosh \theta \end{bmatrix} \begin{bmatrix} x_s \\ y_s \end{bmatrix}$$

- micro-rotações $\alpha_i = \operatorname{atanh} 2^{-i}$

$$x_{i+1} = x_i - y_i 2^{-i} \delta_i$$

$$y_{i+1} = y_i + x_i 2^{-i} \delta_i$$

$$z_{i+1} = z_i - \delta_i \operatorname{atanh}(2^{-i})$$

Rotation mode: $\delta_i = \operatorname{sign}(z_i)$

Vectoring mode: $\delta_i = -\operatorname{sign}(y_i)$

Fator de escala $K_h \approx 0.82816$

Formulação unificada

$$\begin{aligned}x[j+1] &= x[j] - m\sigma_j 2^{-j} y[j] \\y[j+1] &= y[j] + \sigma_j 2^{-j} x[j] \\z[j+1] &= \begin{cases} z[j] - \sigma_j \tan^{-1}(2^{-j}) & \text{if } m = 1 \\ z[j] - \sigma_j \tanh^{-1}(2^{-j}) & \text{if } m = -1 \\ z[j] - \sigma_j (2^{-j}) & \text{if } m = 0 \end{cases}\end{aligned}$$

$m = 1$ coordenadas circulares

$m = 0$ coordenadas lineares

$m = -1$ coordenadas hiperbólicas

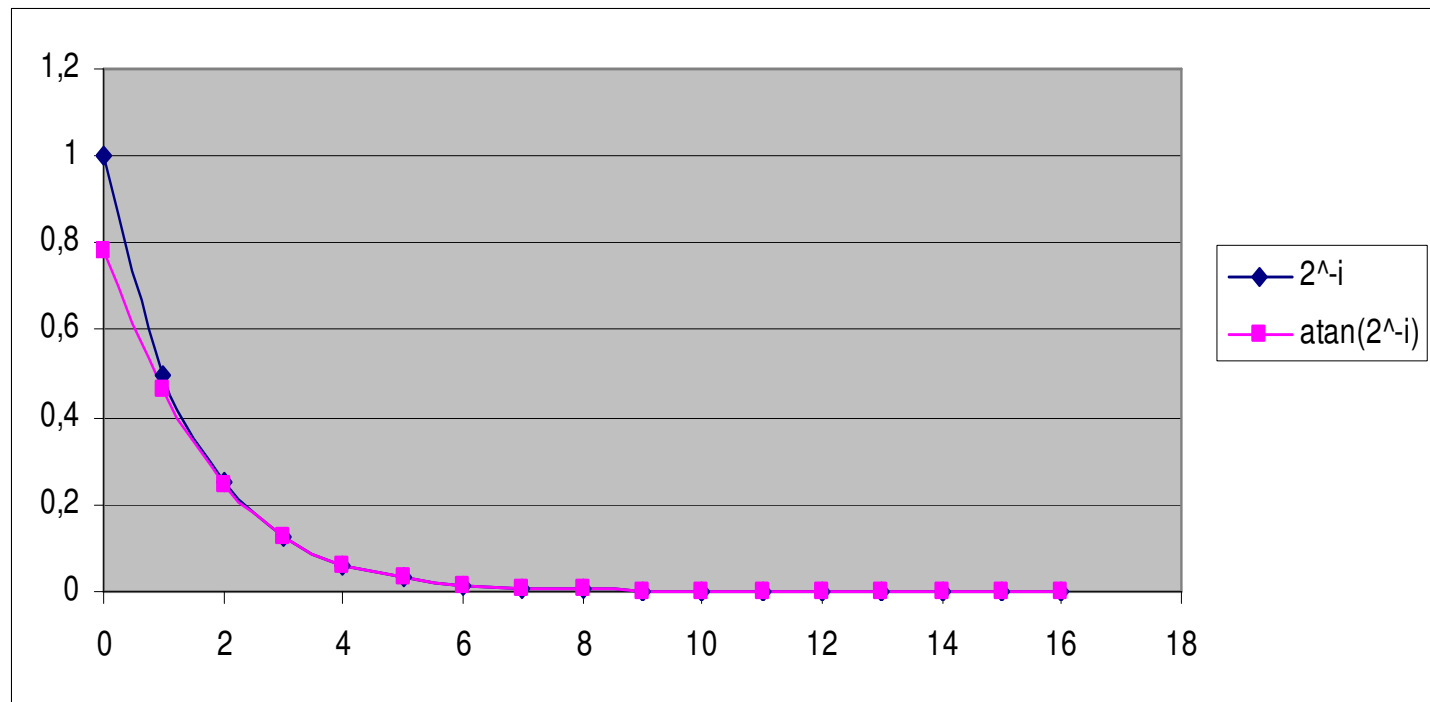
CORDIC

algumas funções

m	Mode	Initial values			Functions	
		x_{in}	y_{in}	z_{in}	x_R	y_R or z_R
1	rotation	1	0	θ	$\cos \theta$	$y_R = \sin \theta$
-1	rotation	1	0	θ	$\cosh \theta$	$y_R = \sinh \theta$
-1	rotation	a	a	θ	ae^{θ}	$y_R = ae^{\theta}$
1	vectoring	1	a	$\pi/2$	$\sqrt{a^2 + 1}$	$z_R = \cot^{-1}(a)$
-1	vectoring	a	1	0	$\sqrt{a^2 - 1}$	$z_R = \coth^{-1}(a)$
-1	vectoring	$a + 1$	$a - 1$	0	$2\sqrt{a}$	$z_R = 0.5 \ln(a)$
-1	vectoring	$a + \frac{1}{4}$	$a - \frac{1}{4}$	0	\sqrt{a}	$z_R = \ln(\frac{1}{4}a)$
-1	vectoring	$a + b$	$a - b$	0	$2\sqrt{ab}$	$z_R = 0.5 \ln(\frac{a}{b})$

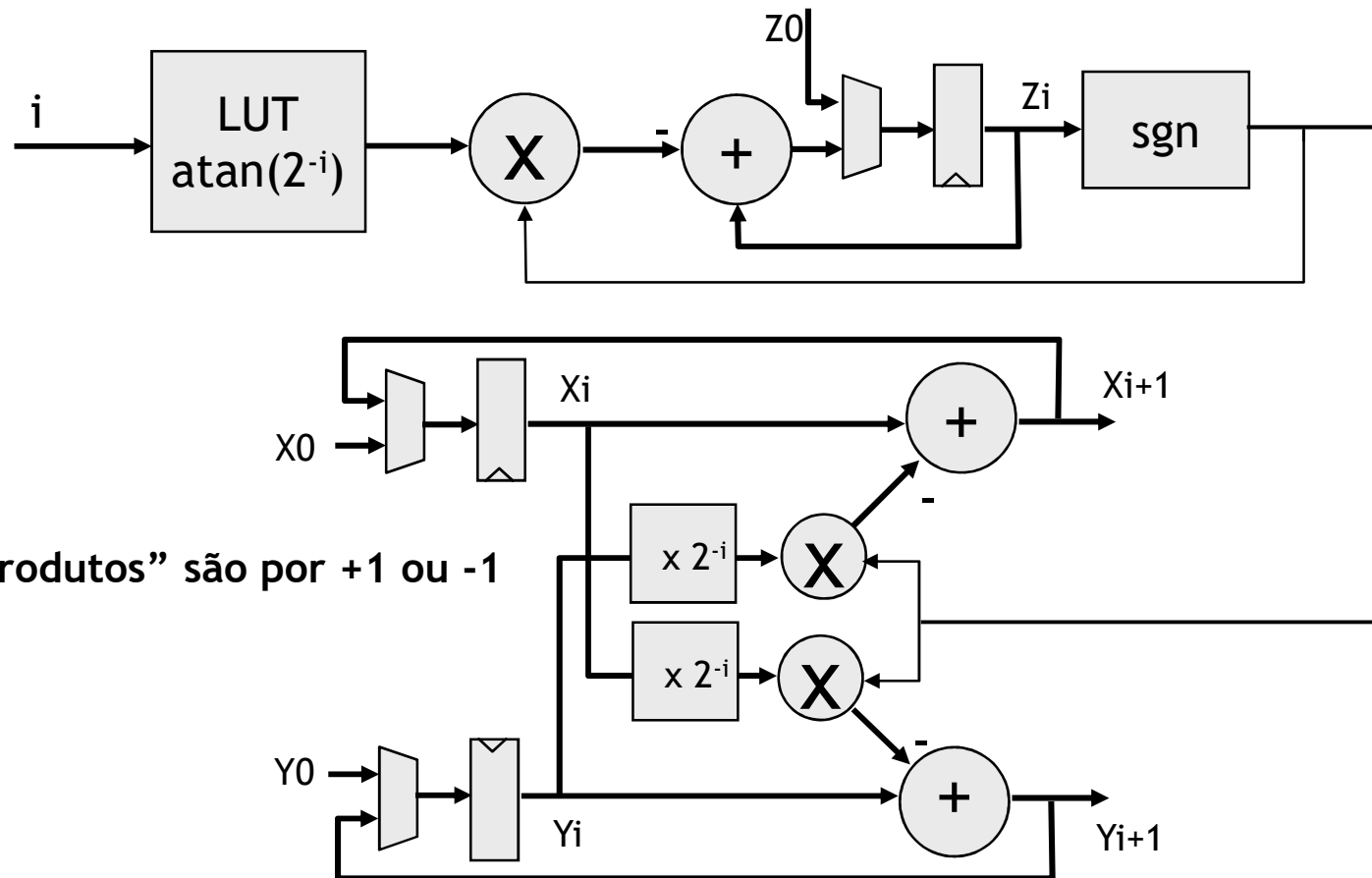
CORDIC

- quando i cresce, $\text{atan}(2^{-i})$ tende para 2^{-i}
 - a sucessão de "micro-rotações" tende para os valores 2^{-i}

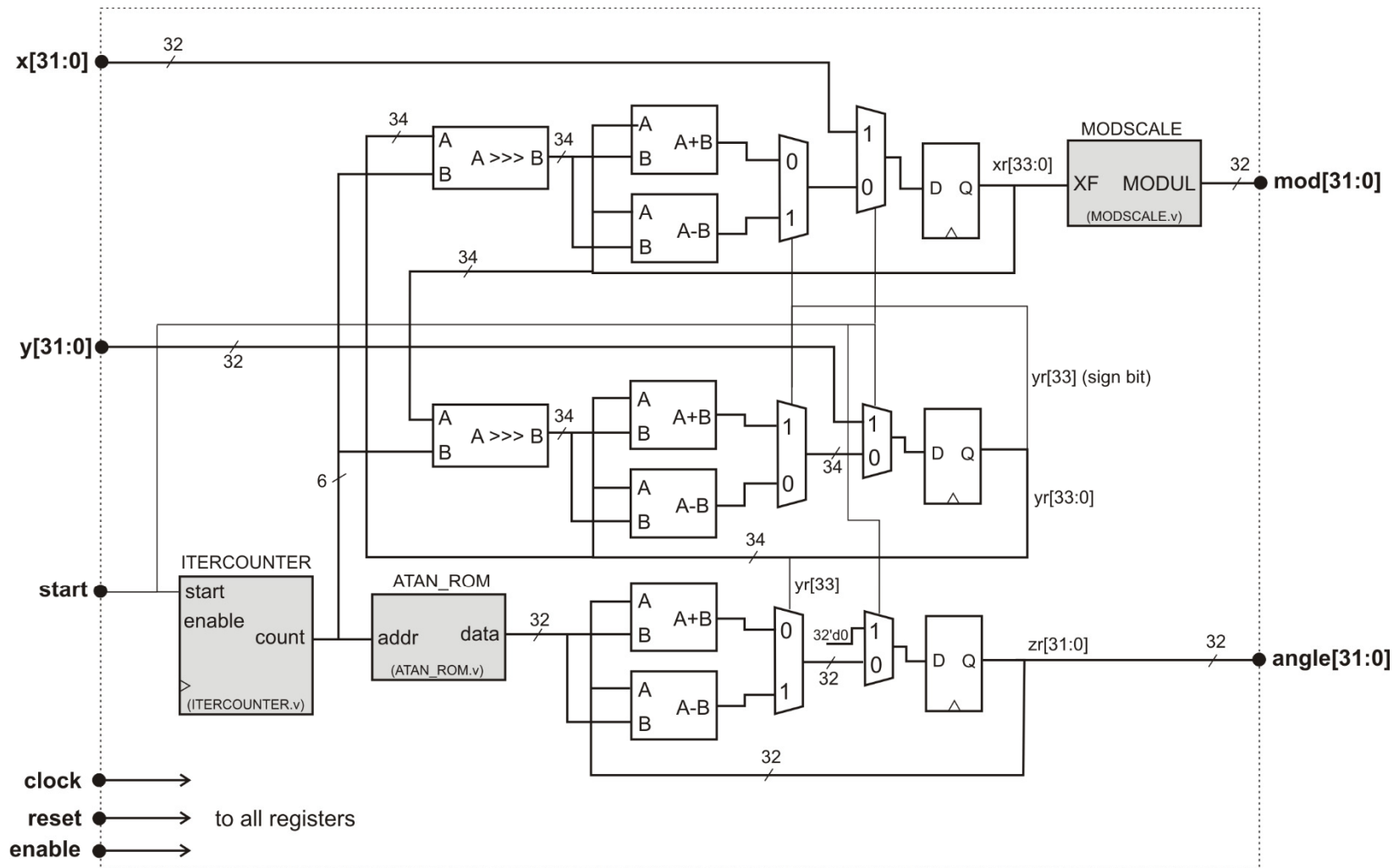


CORDIC

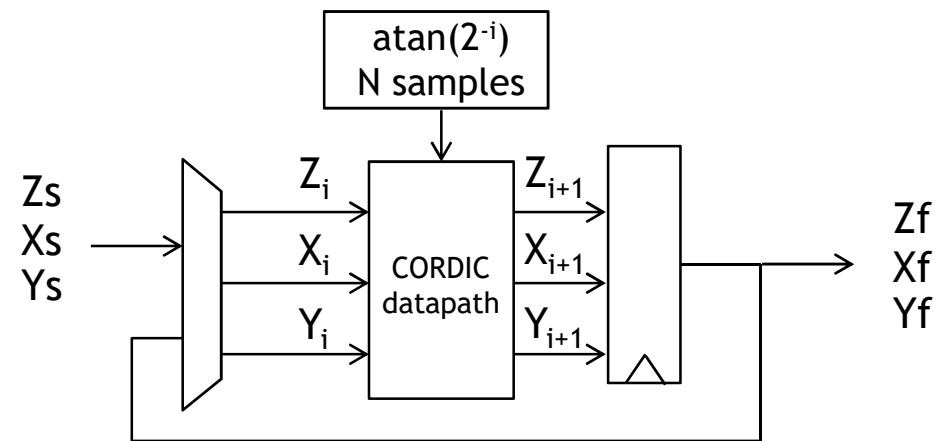
datapath para 1 iteração



Todos os “produtos” são por +1 ou -1

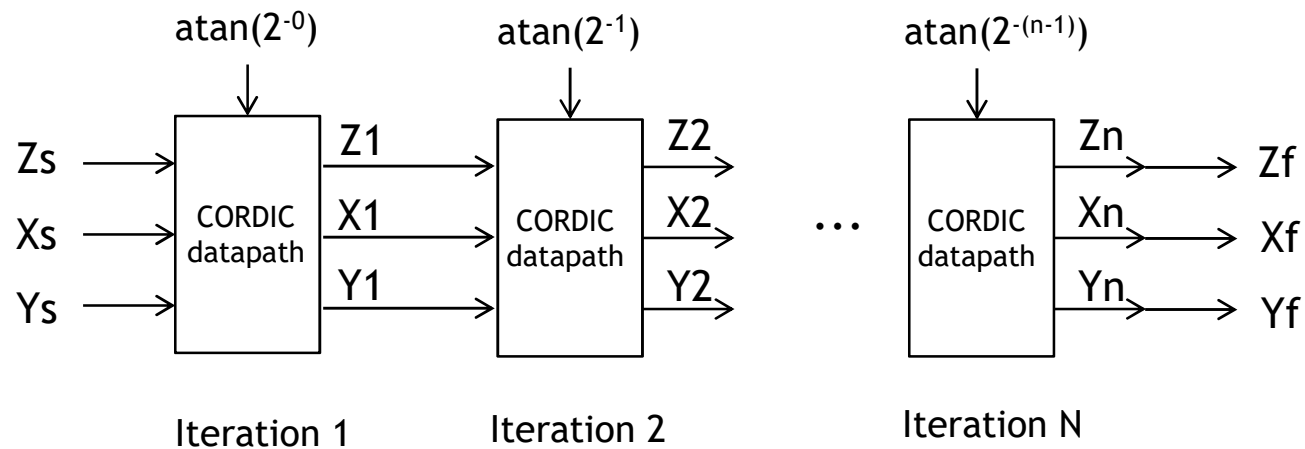


Possíveis implementações



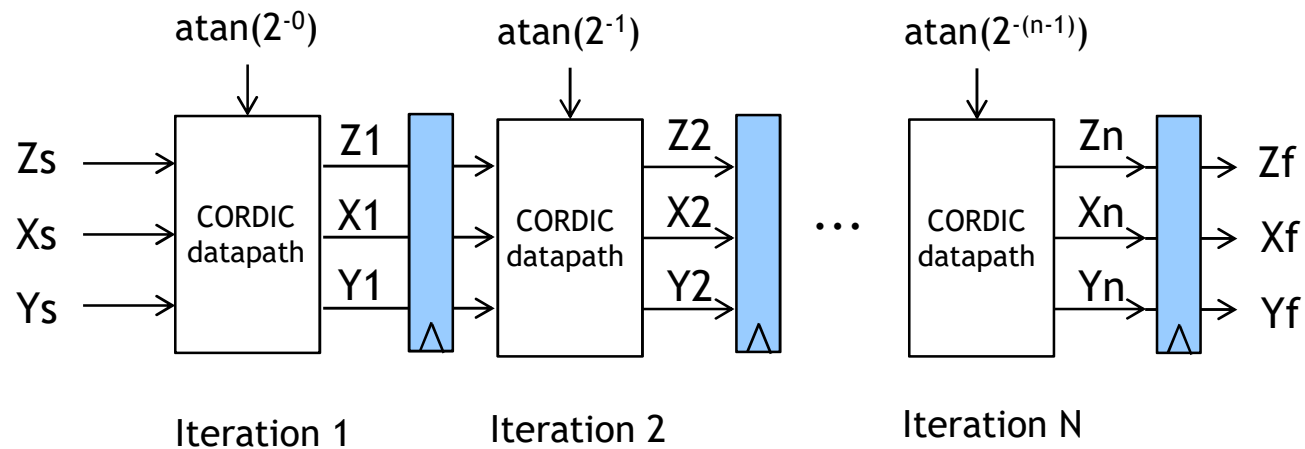
Iterative along time: minimal area, requires N clock cycles

Possíveis implementações



Iterative along space: fully combinational, N instances of the CORDIC datapath

Possíveis implementações



Iterative along space and pipelined: clock rate constrained by one iteration datapath