Processing Elements Design

Shao-Yi Chien

Introduction

- Implementation of basic arithmetic operations
- Number systems
  - Conventional number systems
  - Redundant number systems
  - Residue number systems
- Arithmetic
  - Bit-parallel arithmetic
  - Bit-serial arithmetic
  - Serial-parallel arithmetic
  - Division
  - Distributed arithmetic
  - CORDIC

Conventional Number Systems

- Conventional number systems are nonredundant, weighted, positional number systems

$$x = \sum_{i=0}^{W_d - 1} w_i x_i$$

Nonredundant: one number has only one representation
W_d: word length
w_i: weights→**weighted**
w_i depends only on the position of the digit→**positional**
For fix-radix systems, $w_i = r^i$

- Fix-point: the position of binary point is fixed
- Floating point: signed mantissa and signed exponent

Signed-Magnitude Representation

- Range
  - [-1+Q, 1-Q]
  - Q=(0.00..01)
- Complex for addition and subtraction
- Easy for multiplication and division

$$x = (1-2x_0)\sum_{i=1}^{W_d-1} x_i 2^{-i}$$

$$(+0.828125)_{10} = (0.110101)_{SM}$$

$$(-0.828125)_{10} = (1.110101)_{SM}$$

$$(0)_{10} = (0.000000)_{SM} \text{ or } (1.000000)_{SM}$$

One's Complement

- Range
  - [-1+Q, 1-Q]
- Change sign is easy
- Addition, subtraction, and multiplication are complex

$$x = -x_0(1-Q) + \sum_{i=1}^{W_d-1} x_i 2^{-i}$$

$$(+0.828125)_{10} = (0.110101)_{1C}$$

$$(-0.828125)_{10} = (1.001010)_{1C}$$

$$(0)_{10} = (0.000000)_{1C} \text{ or } (1.111111)_{1C}$$

Two's Complement

$$x = -x_0 + \sum_{i=1}^{W_d - 1} x_i 2^{-i}$$

$$(+0.828125)_{10} = (0.110101)_{2C}$$

$$(-0.828125)_{10} = (1.001010)_{2C} + (0.000001)_{2C} = (1.001011)_{2C}$$

$$(0)_{10} = (0.000000)_{2C}$$

- Range
  - [-1, 1-Q]
- The most widely used representation

Binary Offset Representation

$$x = (x_0 - 1) + \sum_{i=1}^{W_d - 1} x_i 2^{-i}$$

$$(+0.828125)_{10} = (1.110101)_{BO}$$
$$(-0.828125)_{10} = (0.001011)_{BO}$$
$$(0)_{10} = (1.000000)_{BO}$$

- Range
  - [-1,1-Q]
- The sequence of digits is equal to the two's complement representation, except for the sign bit

Redundant Number Systems (1/2)

- Redundant: one number has more than one representation
- Advantages
  - Simply and speed up certain arithmetic operation
  - Addition and subtraction can be performed without carry (barrow) paths
- Disadvantages
  - Increase the complexity for other operations, such as zero detection, sign detection, and sign conversion

Redundant Number Systems (2/2)

- Signed-digit code
- Canonic signed digit code
- On-line arithmetic

Signed-Digit Code (1/4)

$$\boldsymbol{x} = \sum_{i=0}^{W_d - 1} x_i 2^{-i} \text{ where } x_i = -1, 0, \text{or} +1$$

- Range: [-2+Q, 2-Q]
- Redundant
  - $(15/32)_{10}=(0.01111)_{2C}=(0.1000\text{-}1)_{SDC}=(0.01111)_{SDC}$
  - $(-15/32)_{10}=(1.10001)_{2C}=(0.\text{-}10001)_{SDC}$
    $=(0.0\text{-}1\text{-}1\text{-}1\text{-}1)_{SDC}$

Signed-Digit Code (2/4)

- SDC number is not unique
- Has problems to
  - Quantize
  - Compare
  - Overflow check
  - Change to conventional number systems for these operations

Signed-Digit Code (3/4)

- Example of addition
  - $(1\text{-}11\text{-}1)_{SDC}=(5)_{10}$
  - $(0\text{-}111)_{SDC}=(-1)_{10}$
- Rules for adding SDC numbers

$x_i y_i$ or $y_i x_i$	0 0	0 1	0 1	0 -1	0 -1	1 -1	1 1	-1 -1
x_{i+1} y_{i+1}	--	Neither is -1	At least one is -1	Neither is -1	At least one is -1	--	--	--
c_i	0	1	0	0	-1	0	1	-1
z_i	0	-1	1	-1	1	0	0	0

- $s_i=z_i+c_{i+1}$

Signed-Digit Code (4/4)

i		0	1	2	3
x_i		1	–1	1	–1
y_i		0	–1	1	1
c_{i+1}	0	–1	1	0	—
z_i		1	0	0	0
s_i		0	1	0	0

- $(0100)_{SDC}=(4)_{10}$

Canonic Signed Digit Code (1/3)

$$x = \sum_{i=0}^{W_d - 1} x_i 2^{-i} \text{ where } x_i = -1, 0, \text{ or } +1$$

$$x_i \cdot x_{i+1} = 0, \quad 0 \le i \le W_d - 2$$

- Range: [-4/3+Q, 4/3-Q]
- CSDC is a special case of SDC having a minimum number of nonzero digits

Canonic Signed Digit Code (2/3)

- Conversion of two's-complement to CSDC numbers
  - $2^{k+n+1} - 2^k = 2^{k+n} + 2^{k+n-1} + 2^{k+n-2} + \ldots + 2^k$
  - $(0.011111)_{2C} = (0.10000\text{-}1)_{CSDC}$
  - Convert in iterative manner
  - Step1: 011…1→100…-1
  - Step2: (-1,1)→(0,-1), (0,1,1)→(1,0,-1)
  - Ex: $(0.110101101101)_{2C}$
    $=(1.00\text{-}10\text{-}100\text{-}10\text{-}101)_{CSDC}$

Canonic Signed Digit Code (3/3)

- Conversion of SDC to two's complement numbers
  - Separate the SDC number into two parts
    - One parts holds the digit that are either 0 or 1
    - The other part has –1 digits
  - Subtract these two numbers

On-Line Arithmetic

- The number systems with the property that it is possible **to compute the i-th digit of the results using only the first (i+d)-th digit**, where d is a small positive constant
- Favorable in recursive algorithm using numbers with very long word lengths
- SDC can be used for on-line addition and subtraction, d=1

Residue Number Systems (1/2)

- For a given number x and moduli set $\{m_i\}$, i=1, 2, …, p
  - $x=q_i m_i+r_i$
  - RNS representation: $x=(r_1, r_2, \ldots, r_p)$
- Advantages
  - The arithmetic operations (+, -, *) can be performed for each residue independently
- Disadvantages
  - Hard for comparison, overflow detection, and quantization
  - Not easy to convert to other number systems

Residue Number Systems (2/2)

- Example
  - Moduli set={5,3,2}
  - Number range=5*3*2=30
  - $9+19=(4,0,1)_{RNS}+(4,1,1)_{RNS}$
    $=((4+4)_5,(0+1)_3,(1+1)_2)_{RNS}=(3,1,0)_{RNS}=28$
  - $8*3=(3,2,0)_{RNS}+(3,0,1)_{RNS}$
    $=((3*3)_5,(2*0)_3,(0*1)_2)_{RNS}=(4,0,0)_{RNS}=24$

Bit-Parallel Arithmetic (1/2)

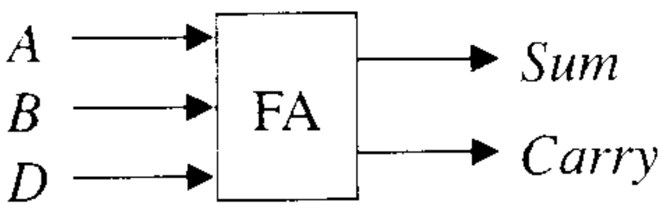
- Addition and subtraction
  - Ripple carry adder (RCA) (carry propagation adder, CPA)
  - Carry-look-ahead adder (CLA)
  - Carry-save adder
  - Carry-select adder (CSA)
  - Carry-skip adder
  - Conditional-sum adder

Bit-Parallel Arithmetic (2/2)

- Multiplication
  - Shift-and-add multiplication
  - Booth's algorithm
  - Tree-based multipliers
  - Array multipliers
  - Look-up table techniques

Ripple Carry Adder (RCA) (1/2)

- Also called carry propagation adder (CPA)
  - Full adder

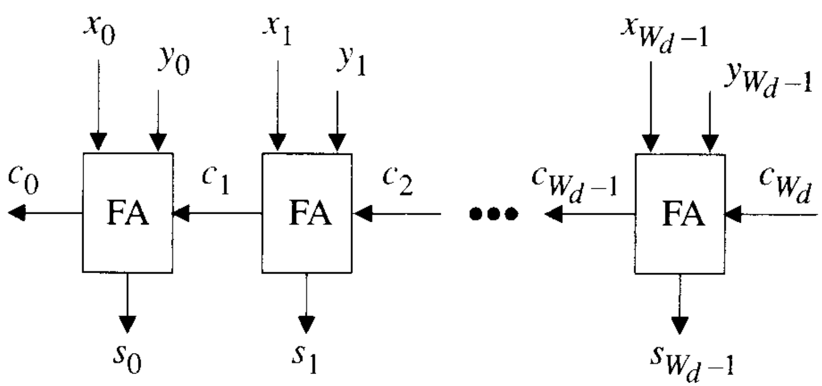


$$S = A \oplus B \oplus D = \{\text{Parity}\}$$
$$= A \cdot B \cdot D + A \cdot \overline{B} \cdot \overline{D} + \overline{A} \cdot \overline{B} \cdot D + \overline{A} \cdot B \cdot \overline{D}$$

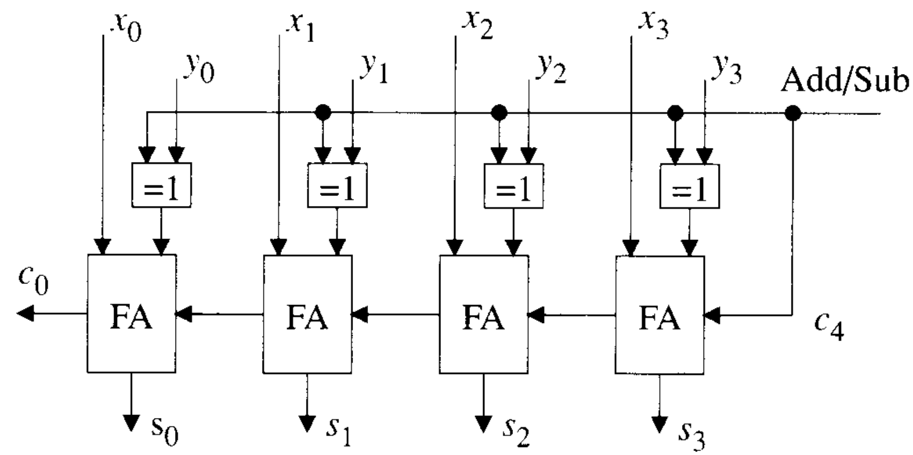
$$C = A \cdot B + A \cdot D + B \cdot D = A \cdot B + D \cdot (A + B)$$

Ripple Carry Adder (RCA) (2/2)

- The speed of the RCA is determined by the carry propagation time



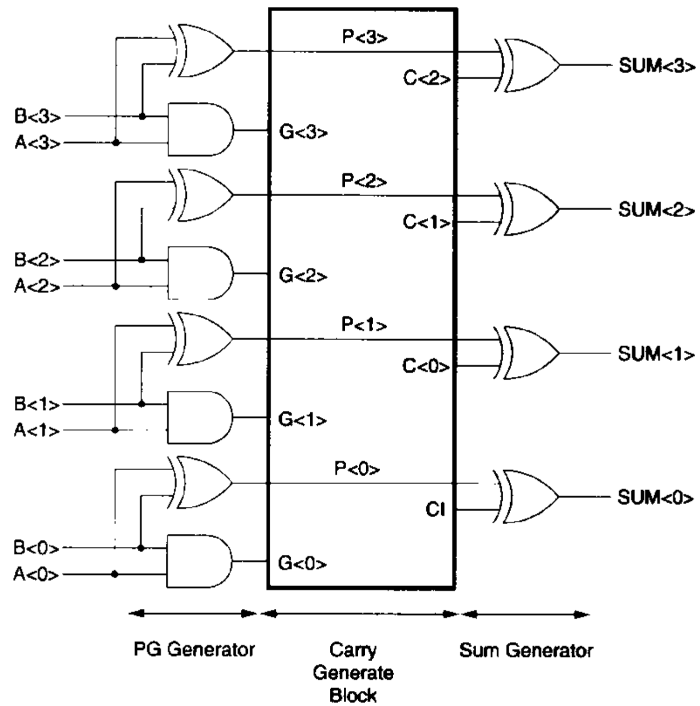
Ripple-carry adder



Ripple-carry adder/subtractor

Carry-Look-Ahead Adder (CLA)

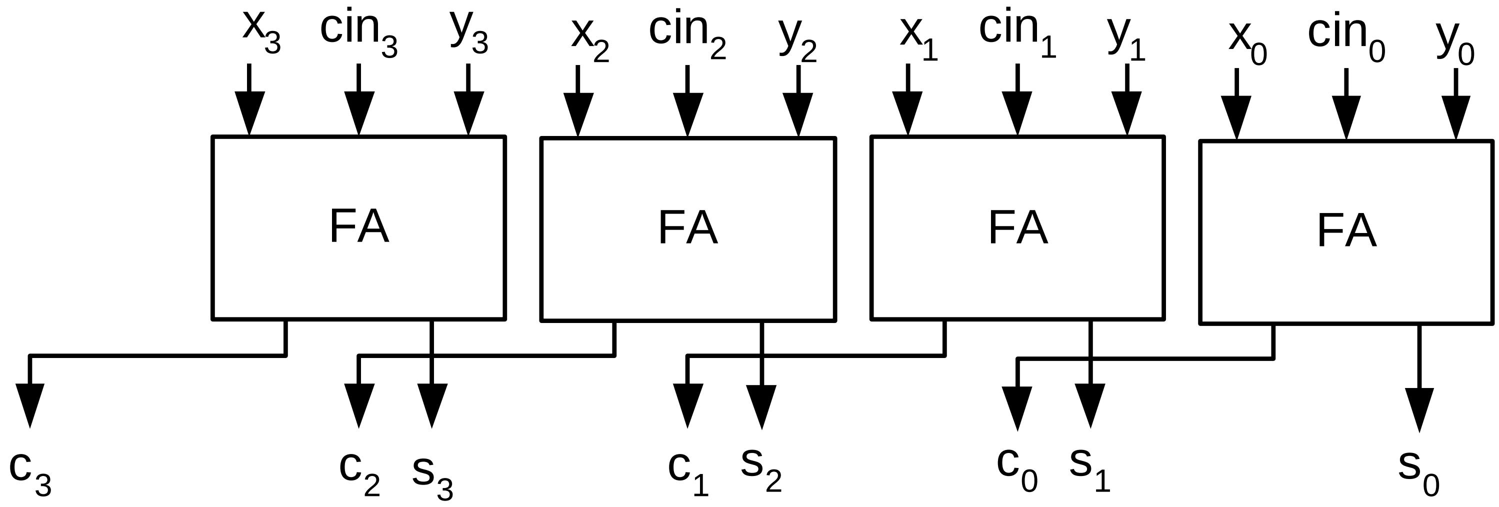
- Generate the carry with separate circuits
- $C_i = G_i + P_i . C_{i-1}$
- $G_i = A_i . B_i$
- $P_i = A_i + B_i$



**Different digit notation in this slide*

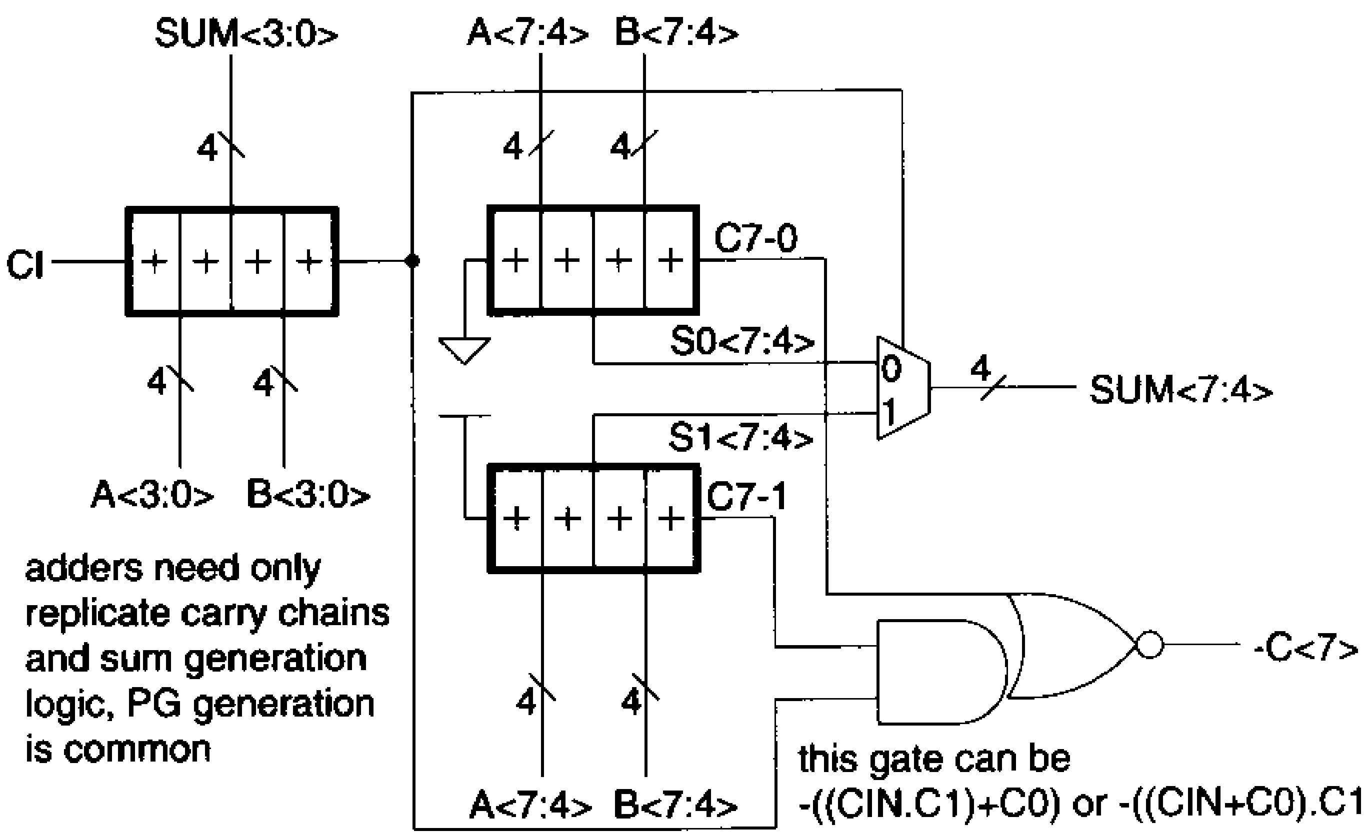
Carry-Save Adder

- Used when adding three or more operands
- Reduce the number of operands by one for each stage



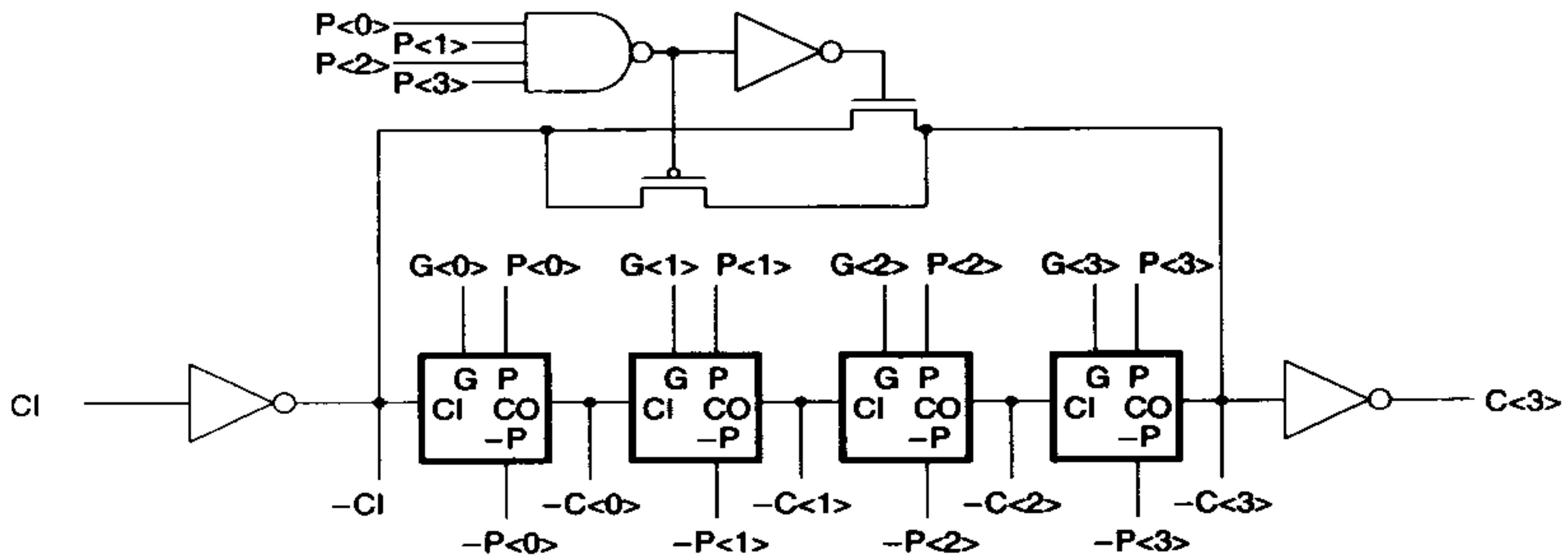
**Different digit notation in this slide*

Carry-Select Adder (CSA)



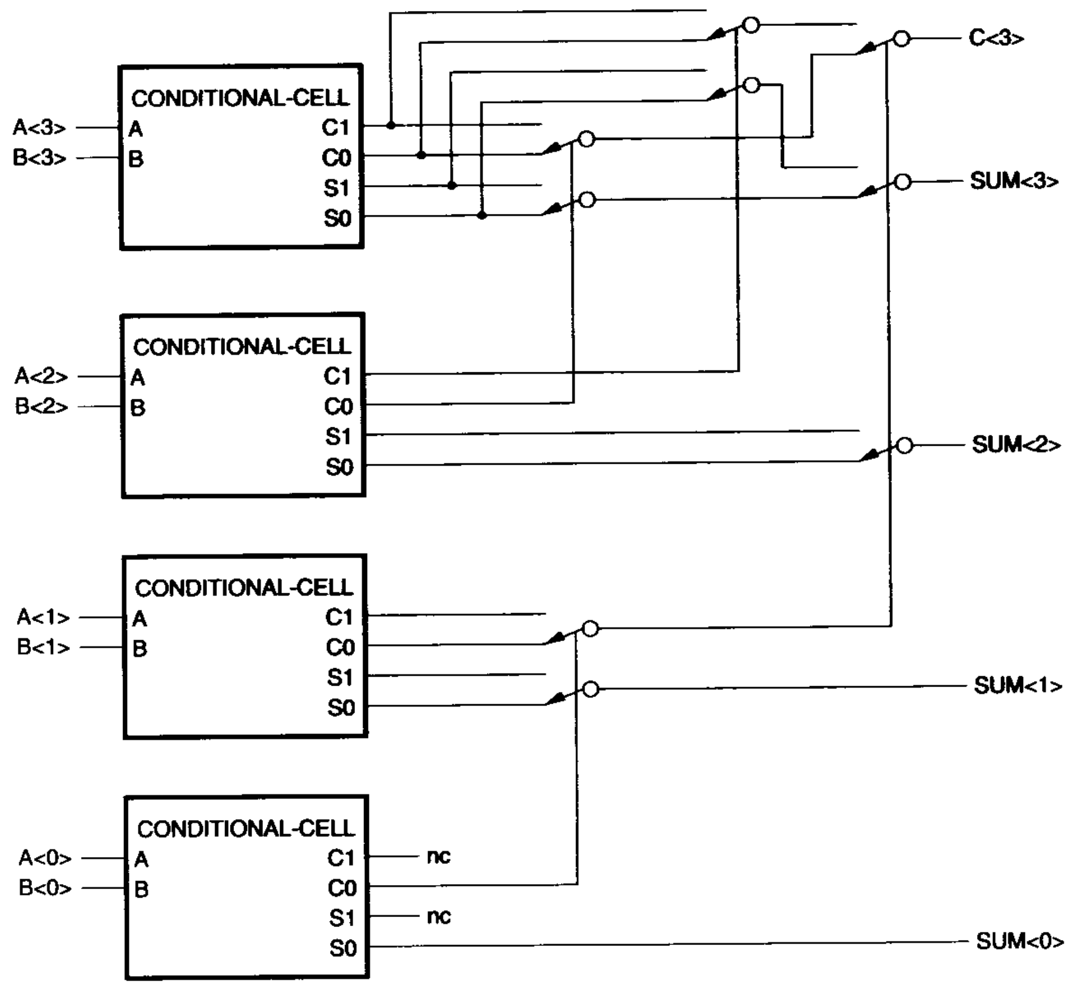
**Different digit notation in this slide*

Carry-Skip Adder



Different digit notation in this slide

Conditional-Sum Adder



$$S_0 = A \oplus B$$

$$S_1 = -(A \oplus B)$$

$$C_0 = A \cdot B$$

$$C_1 = A + B$$

**Different digit notation in this slide*

Multiplication

- Bit-parallel multiplication

$$
\begin{array}{ccccccccc}
 & & & & -a_0\cdot x_{W_d-1} & \cdots & a_{W_c-2}\cdot x_{W_d-1} & a_{W_c-1}\cdot x_{W_d-1} \\
 & & & -a_0\cdot x_{W_d-2} & a_1\cdot x_{W_d-2} & & a_{W_c-1}\cdot x_{W_d-2} & \\
 & & & & \vdots & & & \\
 & & -a_0\cdot x_2 & & & & & \\
 & -a_0\cdot x_1 & a_1\cdot x_1 & & & & & \\
 +a_0\cdot x_0 & -a_1\cdot x_0 & -a_2\cdot x_0 & \cdots & -a_{W_c-1}\cdot x_0 & & & \\
\hline
 y_{-1} & y_0 \,\bullet & y_1 & \cdots & & & y_{W_d+W_c-3} & y_{W_d+W_c-2}
\end{array}
$$

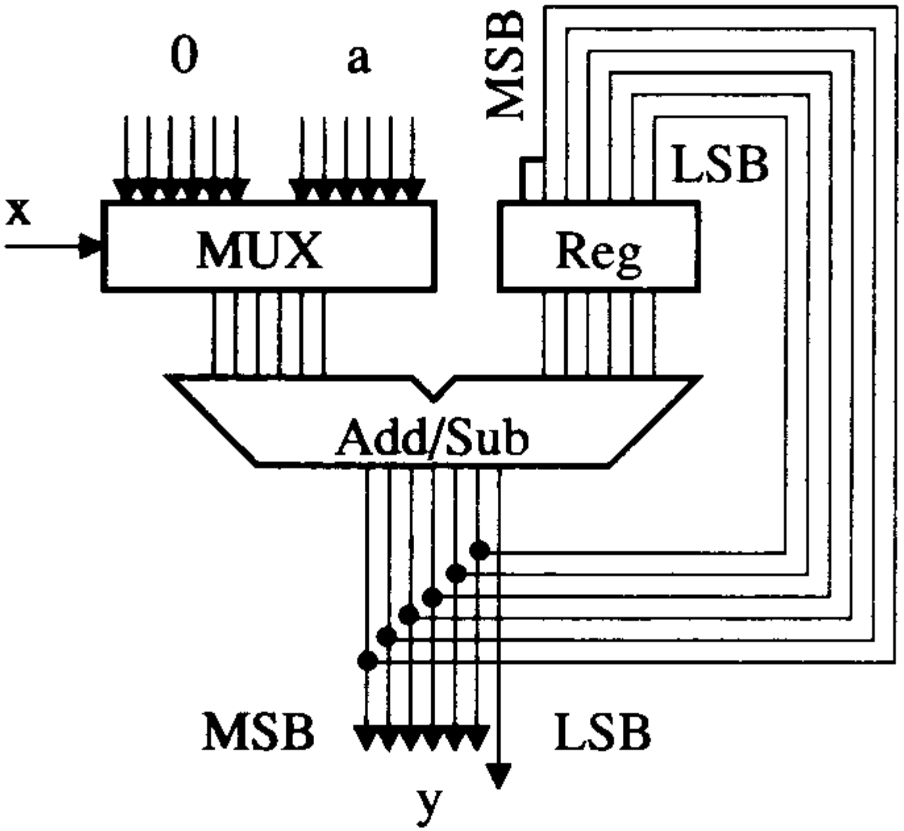
Shift-and-Add Multiplication (1/2)

$$\boldsymbol{y} = \boldsymbol{a}\left(-x_0 + \sum_{i=1}^{W_d-1} x_i 2^{-i}\right) = -\boldsymbol{a}x_0 + \sum_{i=1}^{W_d-1} \boldsymbol{a}x_i 2^{-i}$$

$$\begin{array}{r}
(-a_0 \quad a_1 \quad a_2 \quad \cdots \quad a_{W_c-1})\cdot x_{W_d-1} \\
(-a_0 \quad a_1 \quad a_2 \quad \cdots \quad a_{W_c-1})\cdot x_{W_d-2} \qquad\quad \\
\vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \\
(-a_0 \quad a_1 \quad a_2 \quad\quad a_{W_c-1})\cdot x_2 \qquad\qquad\qquad\qquad\qquad\qquad \\
(-a_0 \quad a_1 \quad a_2 \quad \cdots \quad a_{W_c-1})\cdot x_1 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \\
-(-a_0 \quad a_1 \quad a_2 \quad \cdots \quad a_{W_c-1})\cdot x_0 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \\
\hline
y_{-1} \quad y_0 \;.\; y_1 \quad \cdots \quad y_{W_d+W_c-3} \quad y_{W_d+W_c-2}
\end{array}$$

Shift-and-Add Multiplication (2/2)

- The operation can be reduced with CSDC
- Can be used to design fix-operand multiplier



Booth's Algorithm (1/3)

- Used in modern general-purpose processors, such as MIPS R4000

$$x = \sum_{i=1}^{15} x_i 2^{-i} - x_0 2^0 = \sum_{i=1}^{8} x_{2i-1} 2^{-2i+1} + \sum_{i=1}^{7} x_{2i} 2^{-2i} - x_0 2^0$$

$$= \sum_{i=1}^{8} x_{2i-1} 2^{-2i+1} + \sum_{i=1}^{7} x_{2i} 2^{-2i+1} - 2\sum_{i=1}^{7} x_{2i} 2^{-2i-1} - x_0 2^0$$

$$= \sum_{i=1}^{8} x_{2i-1} 2^{-2i+1} + \sum_{i=1}^{8} x_{2i} 2^{-2i+1} - 2\sum_{i=2}^{8} x_{2(i-1)} 2^{-2i+1} - x_0 2^0$$

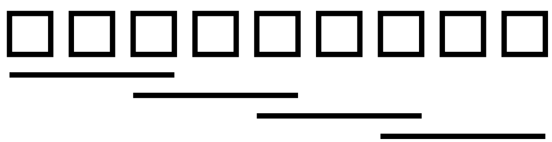
$$= \sum_{i=1}^{8} \left[x_{2i-1} + x_{2i} - 2x_{2(i-1)} \right] 2^{-2i+1}$$

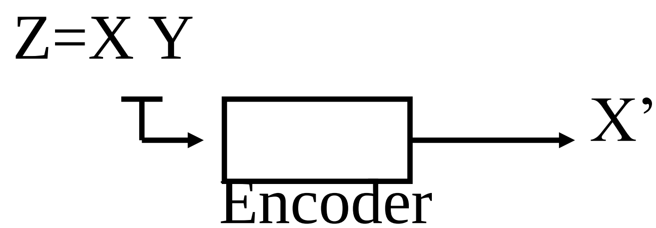
$$\boxed{x \cdot y = \sum_{i=1}^{8} \left[x_{2i-1} + x_{2i} - 2x_{2(i-1)} \right] y 2^{-2i+1}}$$

Booth's Algorithm (2/3)

x_{2i-2}	x_{2i-1}	x_{2i}	x_{2i-1}'	Operation	Comments
0	0	0	0	+0	String of zeros
0	0	1	1	+y	Beginning of 1s
0	1	0	1	+y	A single 1
0	1	1	2	+2y	Beginning of 1s
1	0	0	-2	-2y	End of 1's
1	0	1	-1	-y	A single 0 (beginning/end of 1's)
1	1	0	-1	-y	End of 1's
1	1	1	0	-0	String of 1's

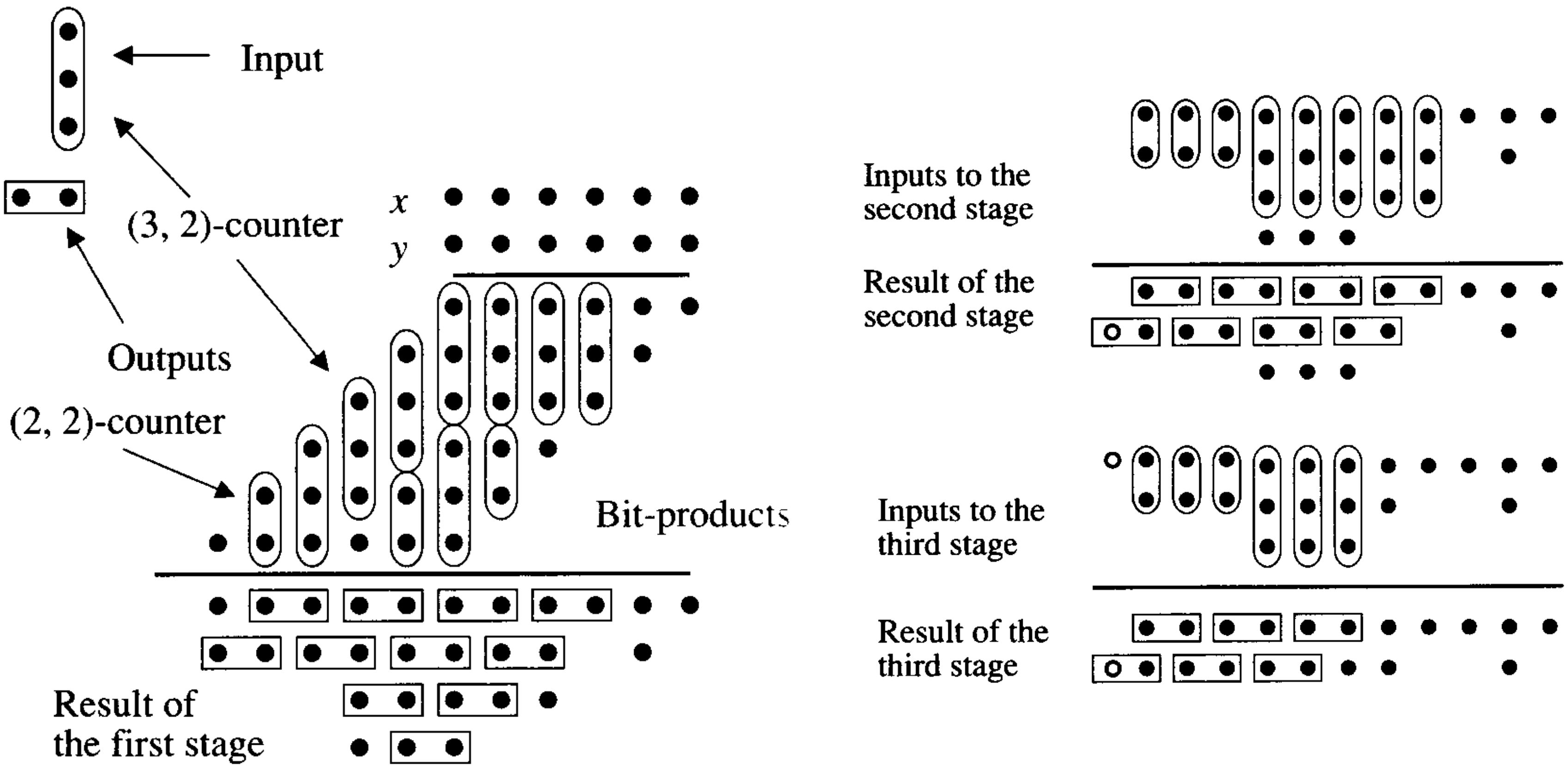
Booth’s Algorithm (3/3)





X_{i+1}	Xi	X_{i-1}		
0	0	0	0	
0	0	1	+Y	(beginning of string)
0	1	0	+Y	(isolated)
0	1	1	+2Y	(beginning of string)
1	0	0	-2Y	(end of string)
1	0	1	-Y	(beginning / end of string)
1	1	0	-Y	(end of string)
1	1	1	0	

Tree-Based Multipliers (Wallace Tree Multipliers)



Array Multipliers (1/3)

- Baugh-Wooley's multiplier

$$P = x \cdot y = \left(-x_0 + \sum_{i=1}^{W_d-1} x_i 2^{-i}\right)\left(-y_0 + \sum_{i=1}^{W_d-1} y_i 2^{-i}\right)$$

$$= x_0 \cdot y_0 + \sum_{i=1}^{W_d-1}\sum_{j=1}^{W_d-1} x_i \cdot y_j 2^{-i-j} - x_0 \sum_{i=1}^{W_d-1} y_i 2^{-i} - y_0 \sum_{i=1}^{W_d-1} x_i 2^{-i}$$

Each of the two negative terms may be rewritten

$$-\sum_{i=1}^{W_d-1} x_0 \cdot y_i 2^{-i} = -1 + 2^{-W_d+1} + \sum_{i=1}^{W_d-1} (1 - x_0 \cdot y_i) 2^{-i}$$

and by using the overflow property of two's-complement representation we get

$$-\sum_{i=1}^{W_d-1} x_0 \cdot y_i 2^{-i} = 1 + 2^{-W_d+1} + \sum_{i=1}^{W_d-1} \overline{x_0 \cdot y_i} 2^{-i}$$

We get

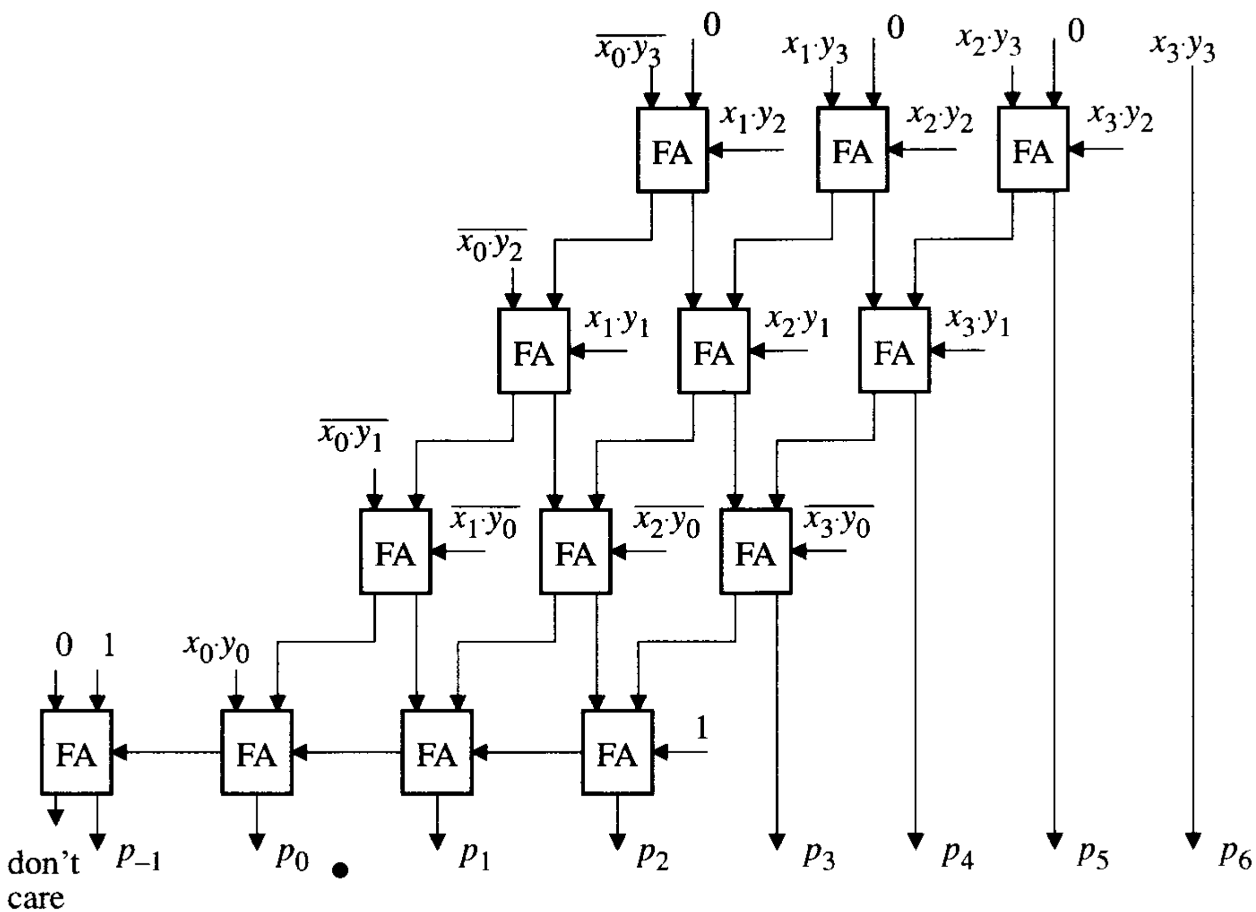
$$P = 2 + 2^{-W_d+2} + x_0 \cdot y_0 + \sum_{i=1}^{W_d-1}\sum_{j=1}^{W_d-1} x_i \cdot y_j 2^{-i-j} + \sum_{i=1}^{W_d-1} \overline{x_0 \cdot y_i} 2^{-i} + \sum_{i=1}^{W_d-1} \overline{y_0 \cdot x_i} 2^{-i}$$

Array Multipliers (2/3)

- Partial products

$$
\begin{array}{cccccccc}
 & & & & x_0 & x_1 & x_2 & x_3 \\
 & & & & y_0 & y_1 & y_2 & y_3 \\
\cline{4-8}
 & & & 1 & \overline{x_0 \cdot y_3} & x_1 \cdot y_3 & x_2 \cdot y_3 & x_3 \cdot y_3 \\
 & & & \overline{x_0 \cdot y_2} & x_1 \cdot y_2 & x_2 \cdot y_2 & x_3 \cdot y_2 & \\
 & & \overline{x_0 \cdot y_1} & x_1 \cdot y_1 & x_2 \cdot y_1 & x_3 \cdot y_1 & & \\
1 & x_0 \cdot y_0 & \overline{x_1 \cdot y_0} & \overline{x_2 \cdot y_0} & \overline{x_3 \cdot y_0} & & & \\
\hline
p_{-1} & p_0 \bullet & p_1 & p_2 & p_3 & p_4 & p_5 & p_6
\end{array}
$$

Array Multipliers (3/3)



Look-Up Table Techniques

- A multiplier AxB can be done with a large table with 2^{WA+WB} words
- Simplified method

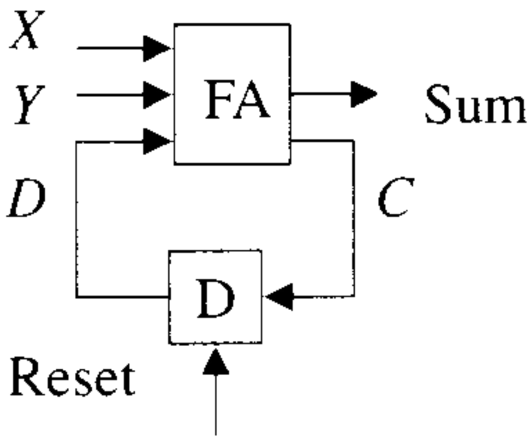
$$x \cdot y = \frac{(x+y)^2}{4} - \frac{(x-y)^2}{4}$$

  - Can be implemented with one addition, two subtraction, and two table look-up operations

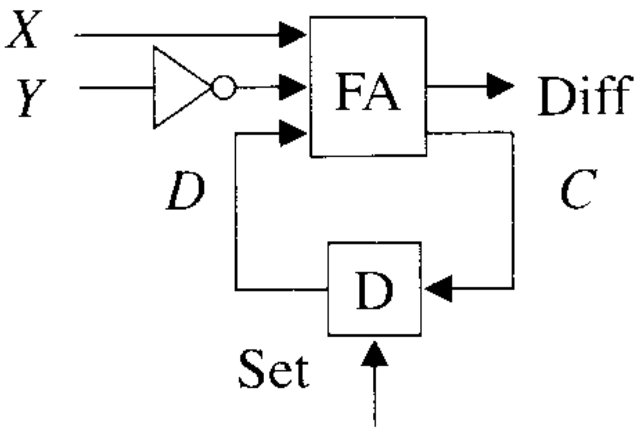
Bit-Serial Arithmetic

- Advantages
  - Significantly reduce chip area
    - Eliminate wide bus
    - Small processing elements
  - Higher clock frequency
  - Often superior than bit-parallel
- Disadvantages
  - S/P P/S interface
  - Complicated clocking scheme

Bit-Serial Addition and Subtraction



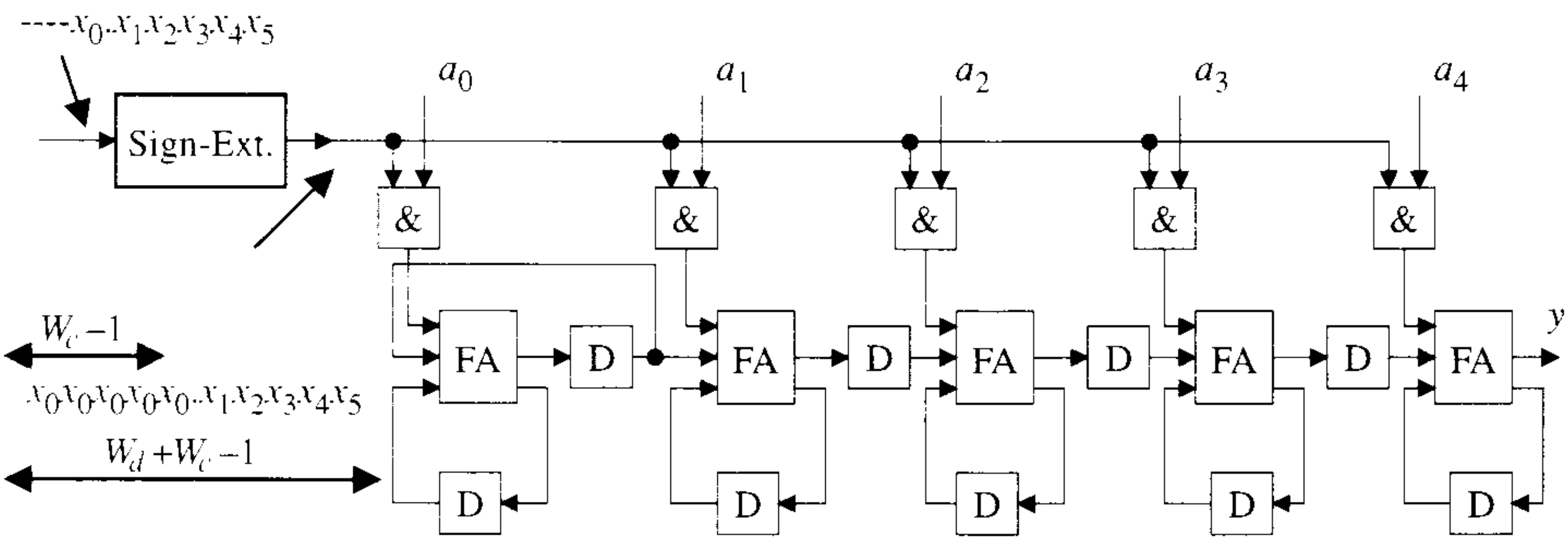
Addition



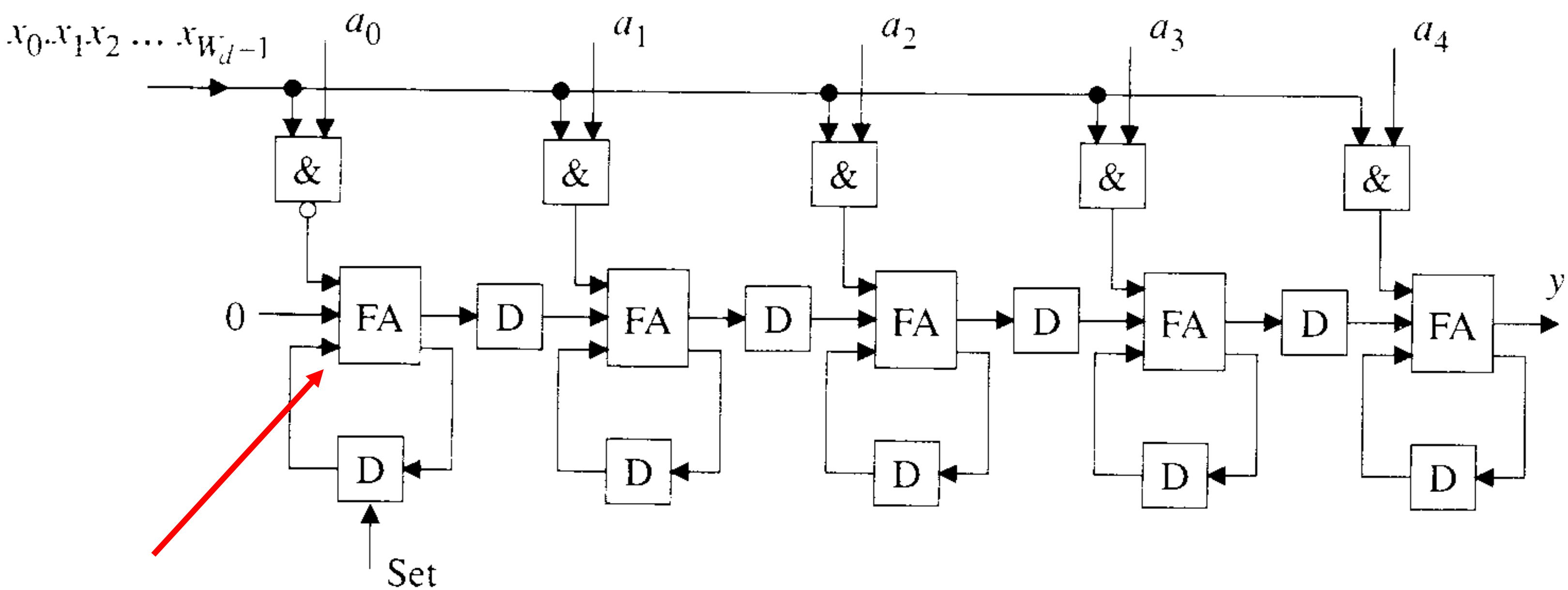
Subtraction

Serial/Parallel Multiplier

- Use carry-save adders
- Need W_d+W_c-1 cycles to compute the result

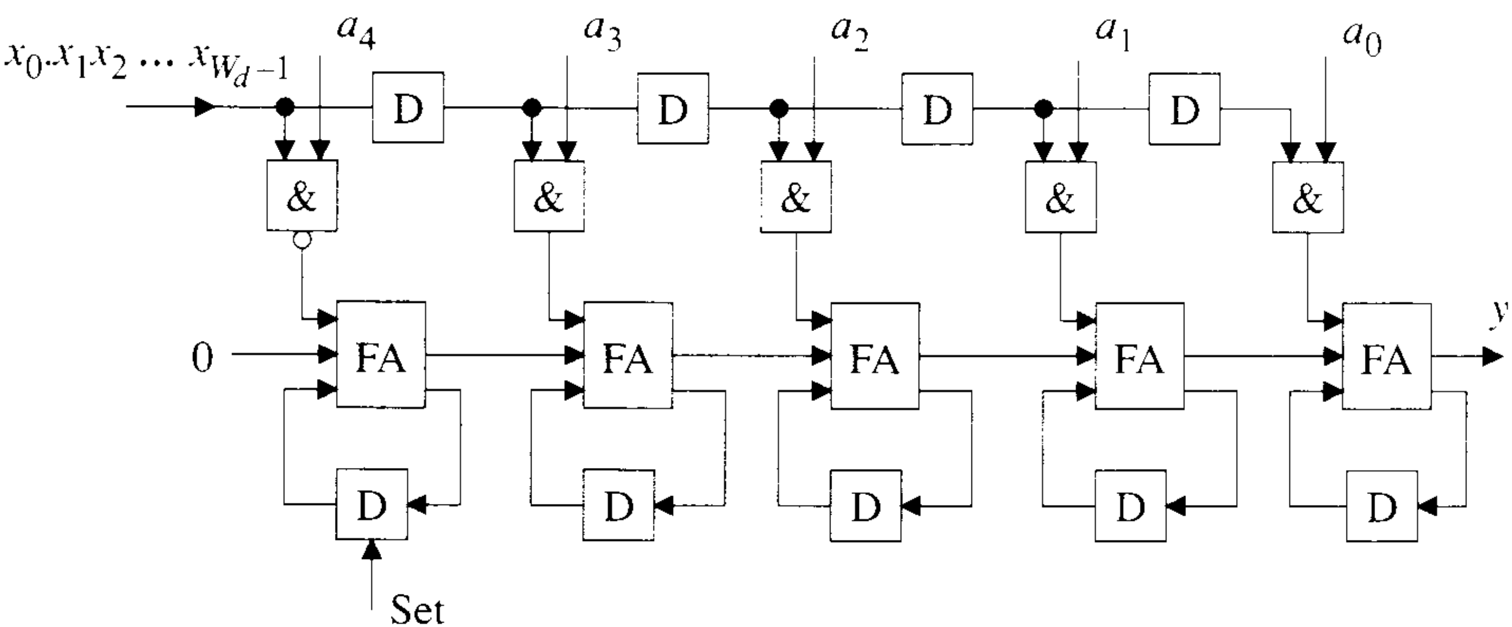


Modified Serial/Parallel Multiplier



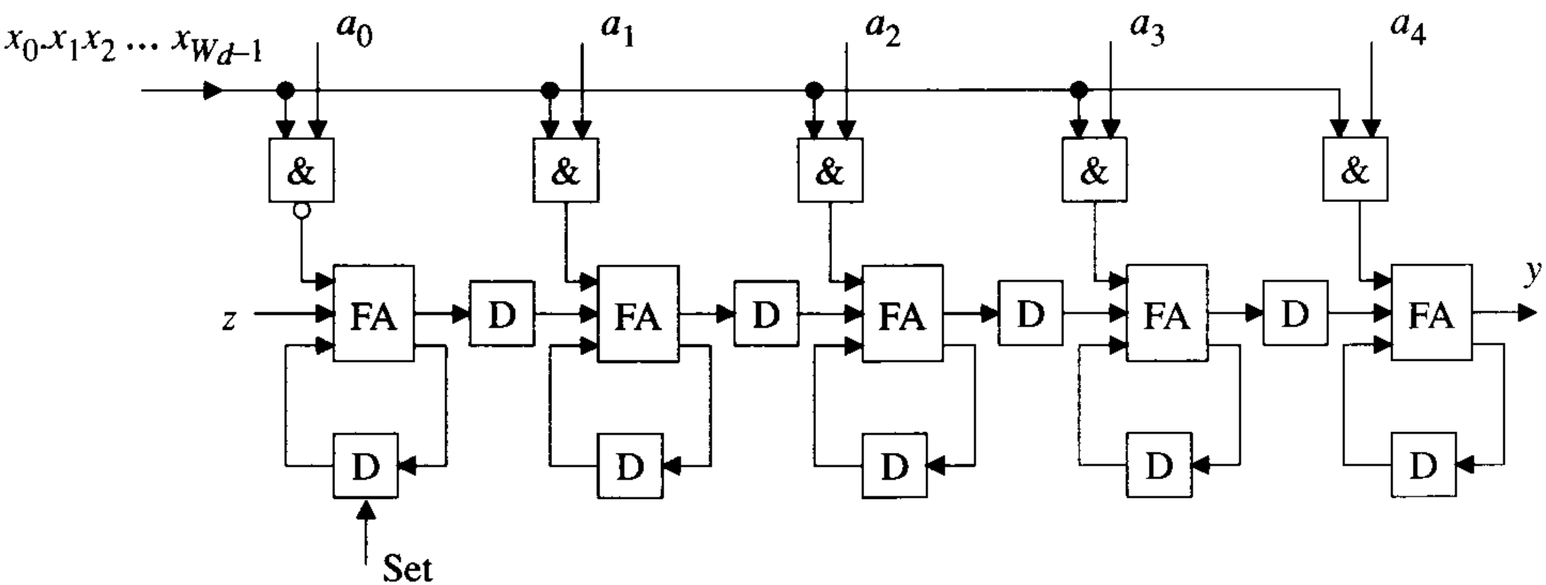
Can be implemented with a half adder

Transpose Serial/Parallel Multiplier



S/P Multiplier-Accumulator

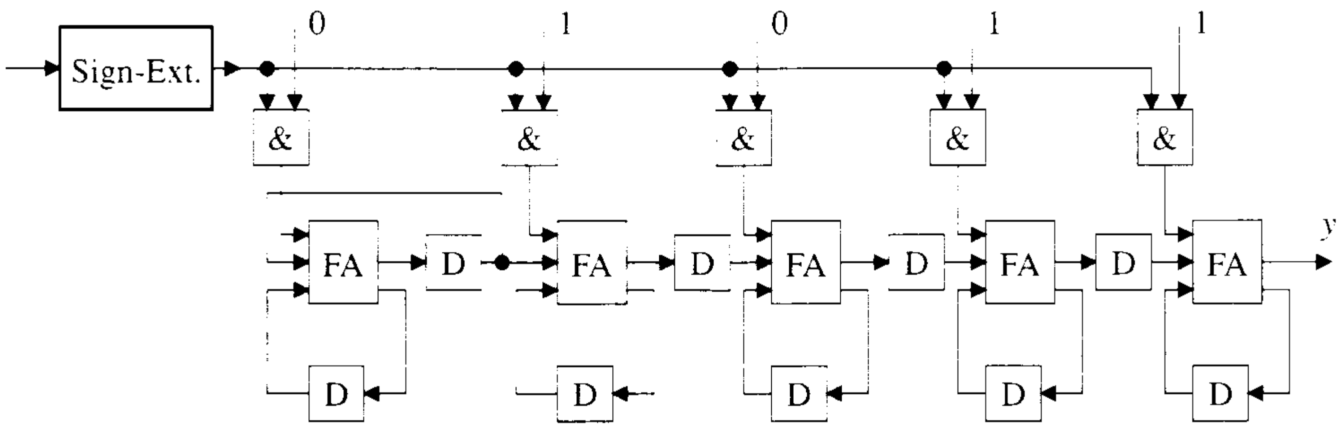
- y=a*x+z

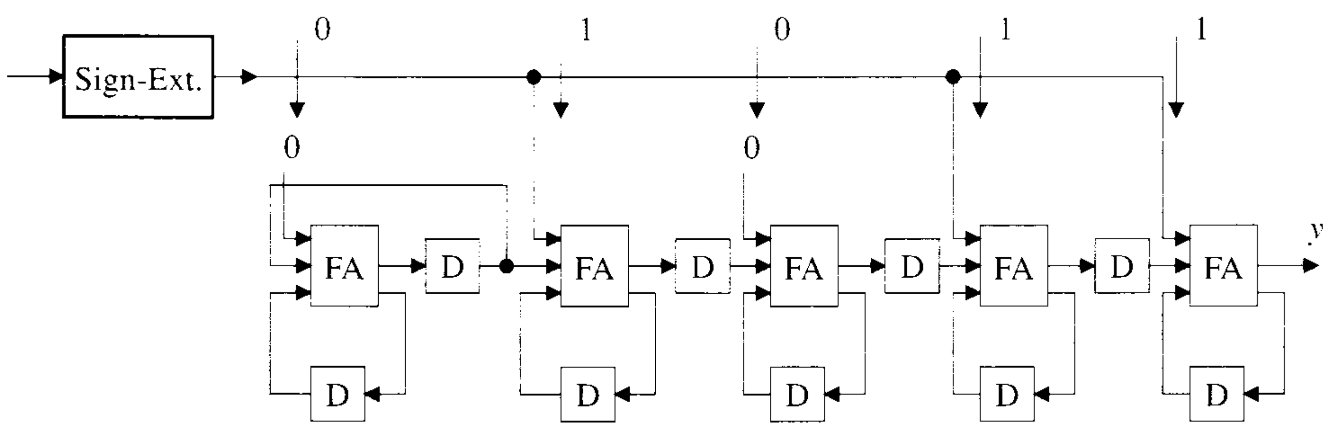


S/P Multiplier with Fixed Coefficients (1/3)

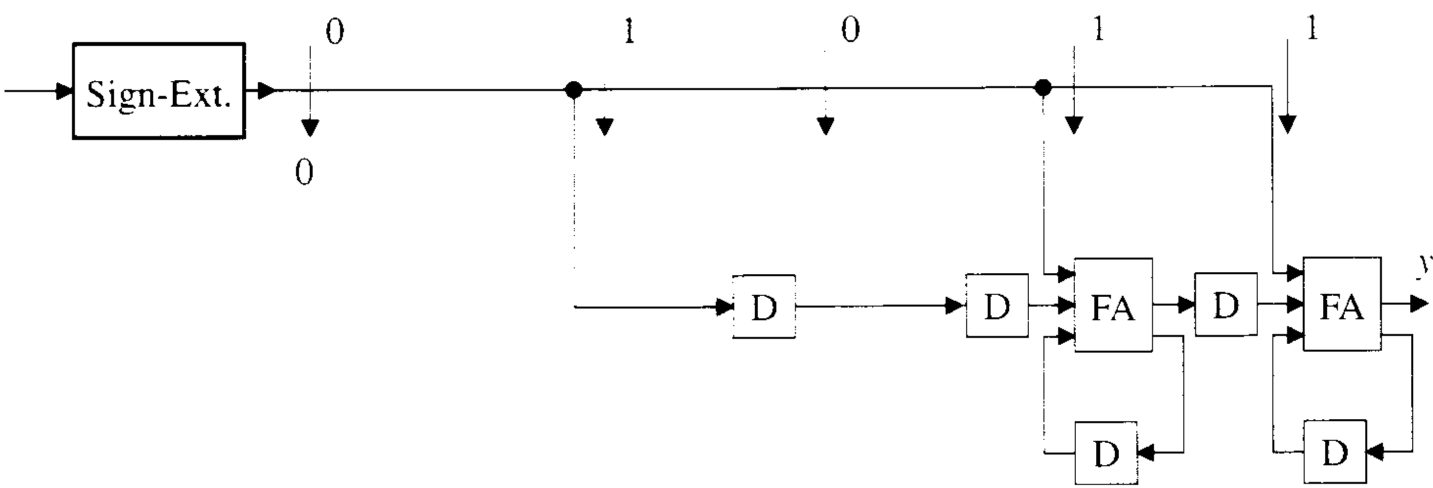
- Remove all AND gates
- Remove all FAs and corresponding D flip-flops, starting with the MSB in the coefficient, up to the first 1 in the coefficient
- Replace each FA that corresponds to a zero-bit in the coefficient with a feedthrough

S/P Multiplier with Fixed Coefficients (2/3)





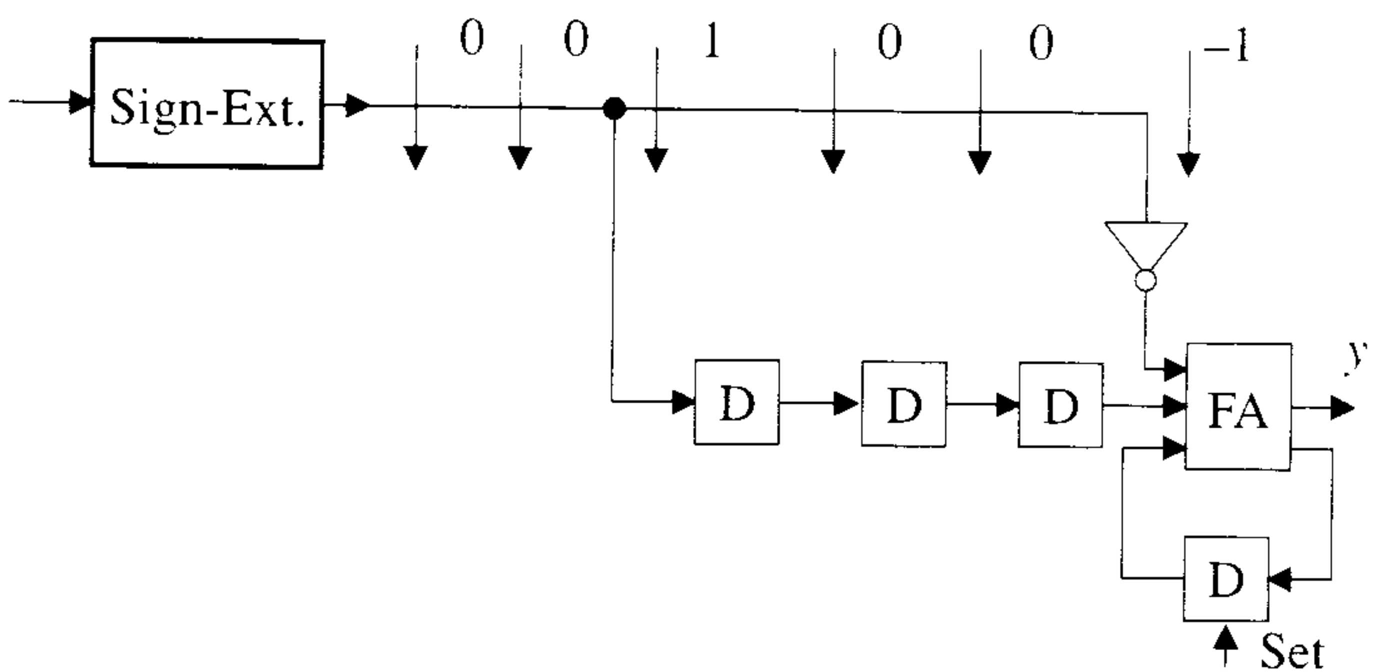
S/P Multiplier with Fixed Coefficients (3/3)



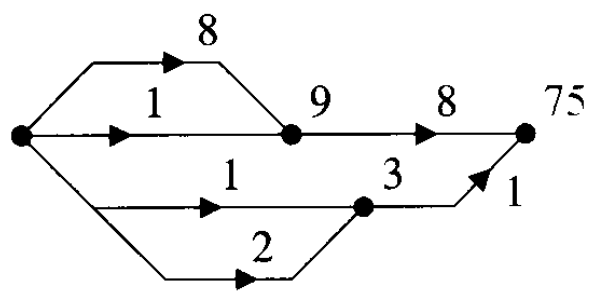
- The number of FA = (the number of 1's)-1
- The number of D flip-flops = the number of 1-bit positions between the first and last bit positions

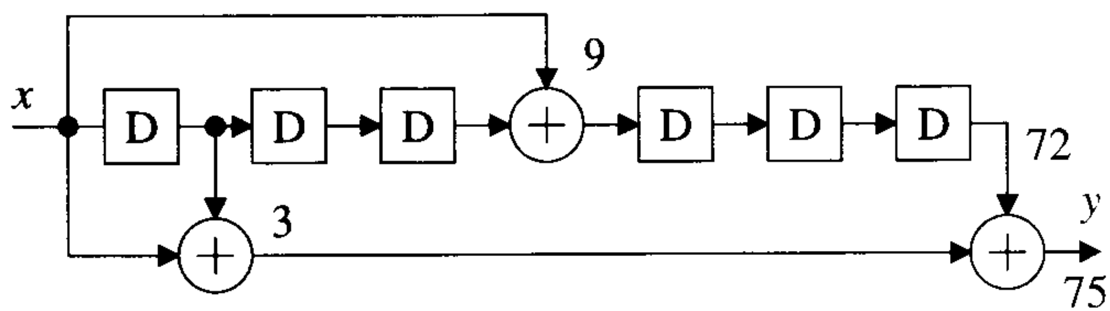
S/P Multiplier with CSDC Coefficients

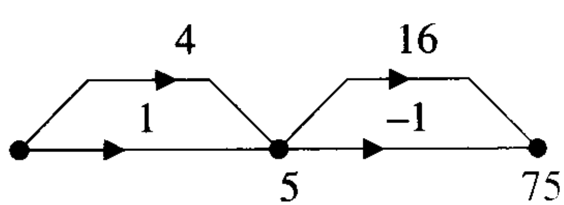
- $a=(0.00111)_{2C}=(0.0100\text{-}1)_{CSDC}$

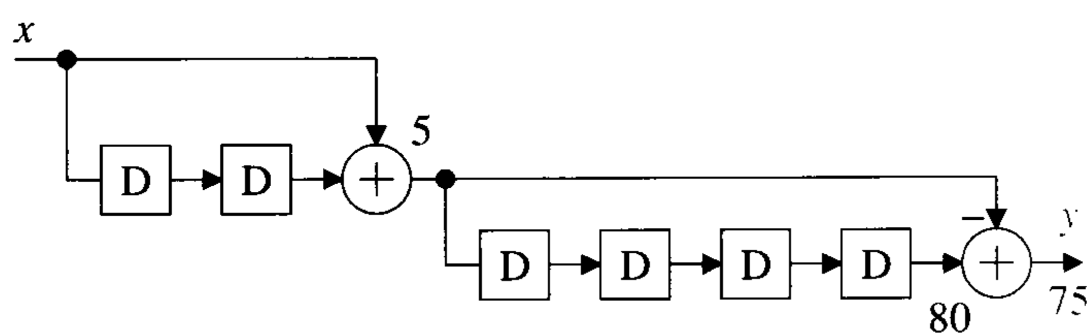


Minimum Number of Basic Operations





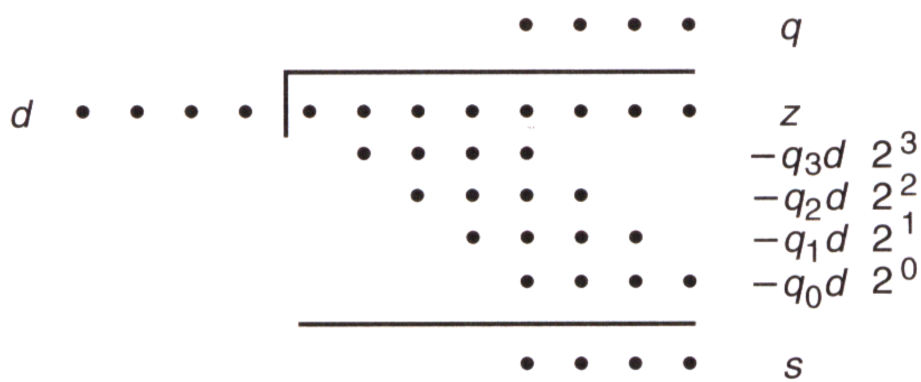




Division

Major reference:
B. Parham, *Computer Arithmetic: Algorithms and Hardware Designs*, Oxford, 2000.

- How to do binary division?



- In the following slides, we define
  - Dividend **z** = $z_{2k-1}z_{2k-2}\ldots z_1z_0$
  - Divisor **d** = $d_{k-1}d_{k-2}\ldots d_1d_0$
  - Quotient **q** = $q_{k-1}q_{k-2}\ldots q_1q_0$
  - Remainder **s** = $[z-(dxq)]=s_{k-1}s_{k-2}\ldots s_1s_0$

What’s Different?

- Added complication of requiring quotient digit selection or estimation
  - The terms to be subtracted from the dividend z are not known a priori but become known as the quotient digits are computed
  - The terms to be subtracted from the initial partial remainder must be produced from top to bottom
  - More difficult and slower than multiplication
  - Long critical path

Division

- Bit-serial division (sequential division algorithm)
- Programmed division
- Restoring bit-serial hardware divider
- Nonrestoring bit-serial hardware divider
- Division by constants
- Array divider

Bit-Serial division (Sequential Division) Algorithm

- $s^{(j)}=2s^{(j-1)}-q_{k-j}(2^k d)$ with $s^{(0)}=z$ and $s^{(k)}=2^k s$
- Or

Shift

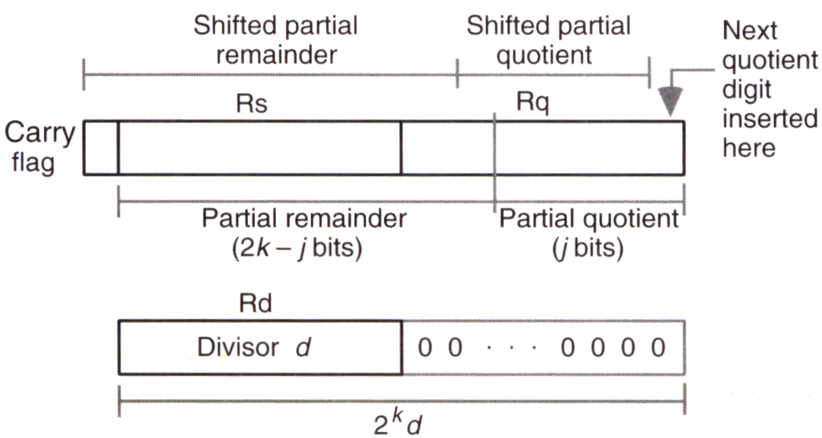
Subtract

```
For j=1 to k
{
    If(2s(j-1)>=(2kd))
    {
        qk-j=1;
        s(j)=2s(j-1)-(2kd);
    }
    Else
    {
        qk-j=0;
        s(j)=2s(j-1);
    }
}
```

Integer division

z		0 1 1 1	0 1 0 1
2^4d		1 0 1 0	
$s^{(0)}$		0 1 1 1	0 1 0 1
$2s^{(0)}$	0	1 1 1 0	1 0 1
$-q_3 2^4 d$		1 0 1 0	$\{q_3 = 1\}$
$s^{(1)}$		0 1 0 0	1 0 1
$2s^{(1)}$	0	1 0 0 1	0 1
$-q_2 2^4 d$		0 0 0 0	$\{q_2 = 0\}$
$s^{(2)}$		1 0 0 1	0 1
$2s^{(2)}$	1	0 0 1 0	1
$-q_1 2^4 d$		1 0 1 0	$\{q_1 = 1\}$
$s^{(3)}$		1 0 0 0	1
$2s^{(3)}$	1	0 0 0 1	
$-q_0 2^4 d$		1 0 1 0	$\{q_0 = 1\}$
$s^{(4)}$		0 1 1 1	
s			0 1 1 1
q			1 0 1 1

Programmed Division



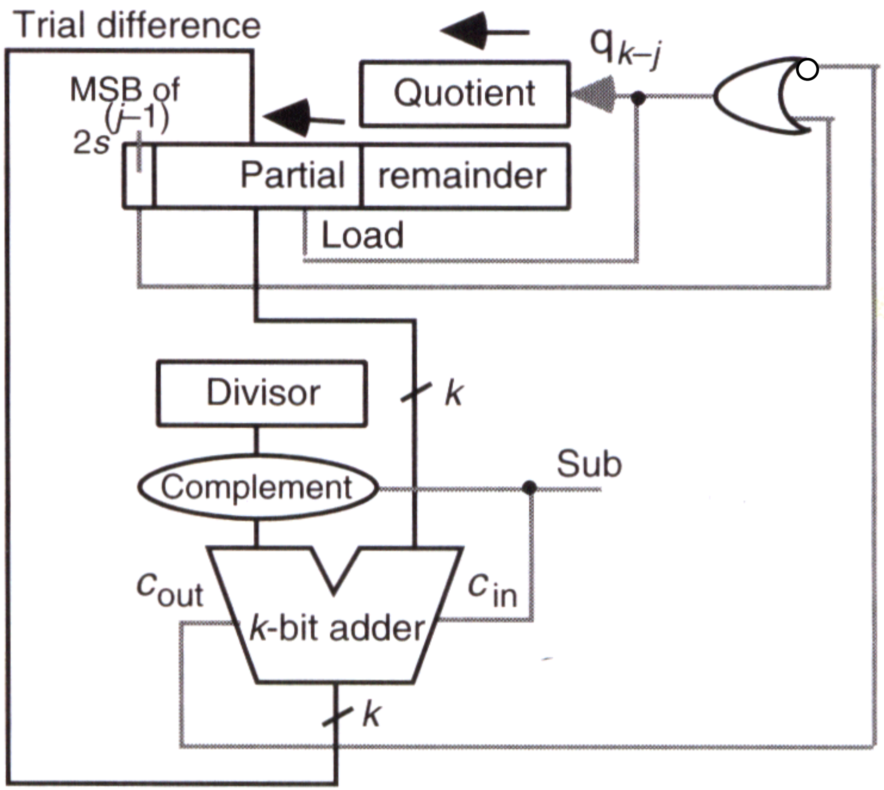

```
{Using left shifts, divide unsigned 2k-bit dividend,
z_high|z_low, storing the k-bit quotient and remainder.
Registers:    R0 holds 0              Rc for counter
              Rd for divisor          Rs for z_high & remainder
              Rq for z_low & quotient}

{Load operands into registers Rd, Rs, and Rq}

     div:     load       Rd with divisor
              load       Rs with z_high
              load       Rq with z_low

{Check for exceptions}

              branch     d_by_0 if Rd = R0
              branch     d_ovfl if Rs > Rd

{Initialize counter}

              load       k into Rc

{Begin division loop}

  d_loop:     shift      Rq left 1      {zero to LSB, MSB to carry}
              rotate     Rs left 1      {carry to LSB, MSB to carry}
              skip       if carry = 1
              branch     no_sub if Rs < Rd
              sub        Rd from Rs
              incr       Rq             {set quotient digit to 1}
  no_sub:     decr       Rc             {decrement counter by 1}
              branch     d_loop if Rc ≠ 0

{Store the quotient and remainder}

              store      Rq into quotient
              store      Rs into remainder
  d_done:     . . .
  d_by_0:     . . .
  d_ovfl:     . . .
```

Need more than 200 instructions for a 32-bit division!!

Restoring Bit-Serial Hardware Divider (1/3)

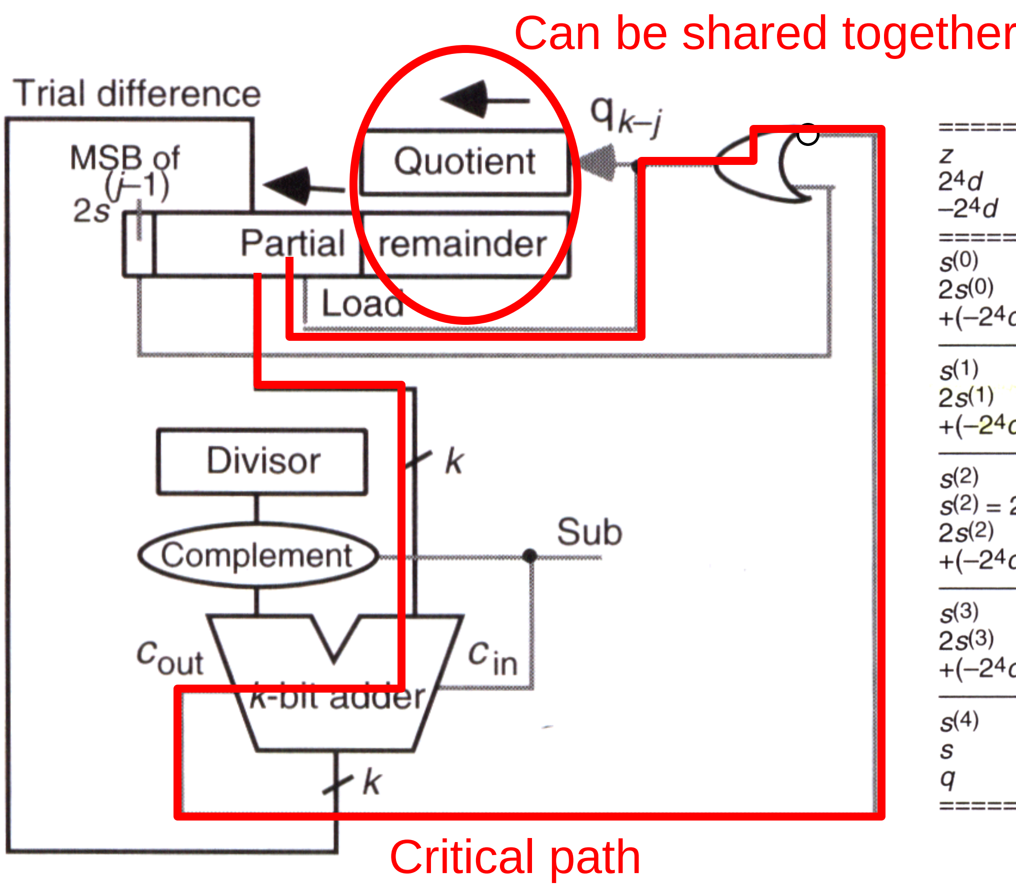
- "Restoring division"
  - Assume q=1 first, do the trial difference
  - The remainder is restored to its correct value if the trial subtraction indicates that 1 was not the right choice for q

Restoring Bit-Serial Hardware Divider (2/3)



z		0 1 1 1	0 1 0 1	No overflow, since: $(0111)_{two} < (1010)_{two}$
$2^4 d$	0	1 0 1 0		
$-2^4 d$	1	0 1 1 0		
$s^{(0)}$	0	0 1 1 1	0 1 0 1	
$2s^{(0)}$	0	1 1 1 0	1 0 1	
$+(-2^4 d)$	1	0 1 1 0		
$s^{(1)}$	0	0 1 0 0	1 0 1	Positive, so set $q_3 = 1$
$2s^{(1)}$	0	1 0 0 1	0 1	
$+(-2^4 d)$	1	0 1 1 0		
$s^{(2)}$	1	1 1 1 1	0 1	Negative, so set $q_2 = 0$ and restore
$s^{(2)} = 2s^{(1)}$	0	1 0 0 1	0 1	
$2s^{(2)}$	1	0 0 1 0	1	
$+(-2^4 d)$	1	0 1 1 0		
$s^{(3)}$	0	1 0 0 0	1	Positive, so set $q_1 = 1$
$2s^{(3)}$	1	0 0 0 1		
$+(-2^4 d)$	1	0 1 1 0		
$s^{(4)}$	0	0 1 1 1		Positive, so set $q_0 = 1$
s			0 1 1 1	
q			1 0 1 1	

Restoring Bit-Serial Hardware Divider (3/3)

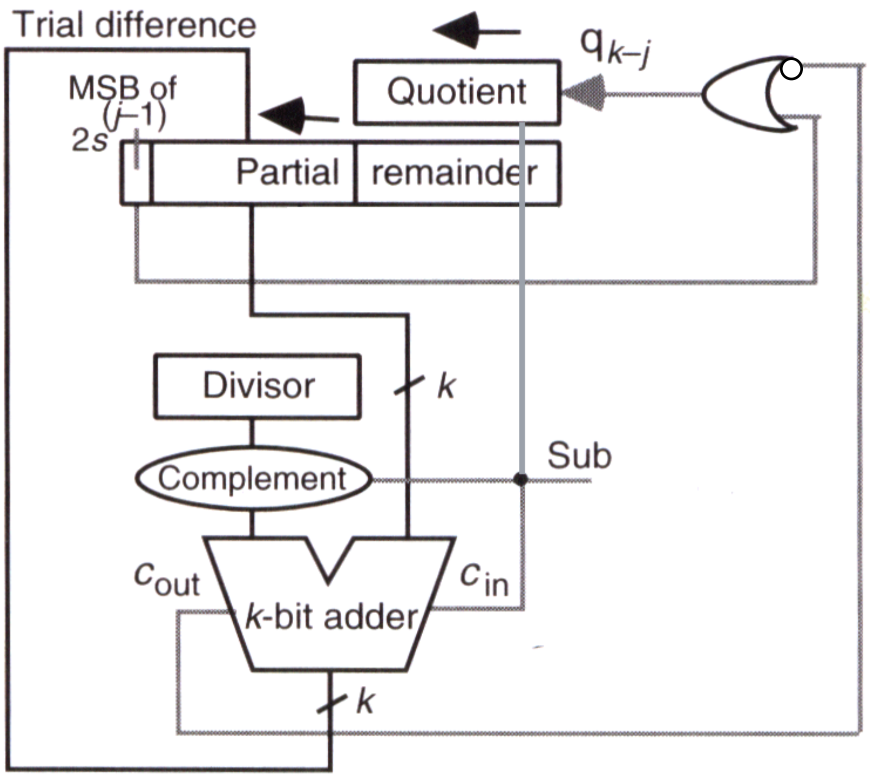


z	0 1 1 1 0 1 0 1		No overflow, since: $(0111)_{two} < (1010)_{two}$
2^4d	0 1 0 1 0		
-2^4d	1 0 1 1 0		
$s^{(0)}$	0 0 1 1 1	0 1 0 1	
$2s^{(0)}$	0 1 1 1 0	1 0 1	
$+(-2^4d)$	1 0 1 1 0		
$s^{(1)}$	0 0 1 0 0	1 0 1	Positive, so set $q_3 = 1$
$2s^{(1)}$	0 1 0 0 1	0 1	
$+(-2^4d)$	1 0 1 1 0		
$s^{(2)}$	1 1 1 1 1	0 1	Negative, so set $q_2 = 0$ and restore
$s^{(2)} = 2s^{(1)}$	0 1 0 0 1	0 1	
$2s^{(2)}$	1 0 0 1 0	1	
$+(-2^4d)$	1 0 1 1 0		
$s^{(3)}$	0 1 0 0 0	1	Positive, so set $q_1 = 1$
$2s^{(3)}$	1 0 0 0 1		
$+(-2^4d)$	1 0 1 1 0		
$s^{(4)}$	0 0 1 1 1		Positive, so set $q_0 = 1$
s		0 1 1 1	
q		1 0 1 1	

Nonrestoring Bit-Serial Hardware Divider (1/4)

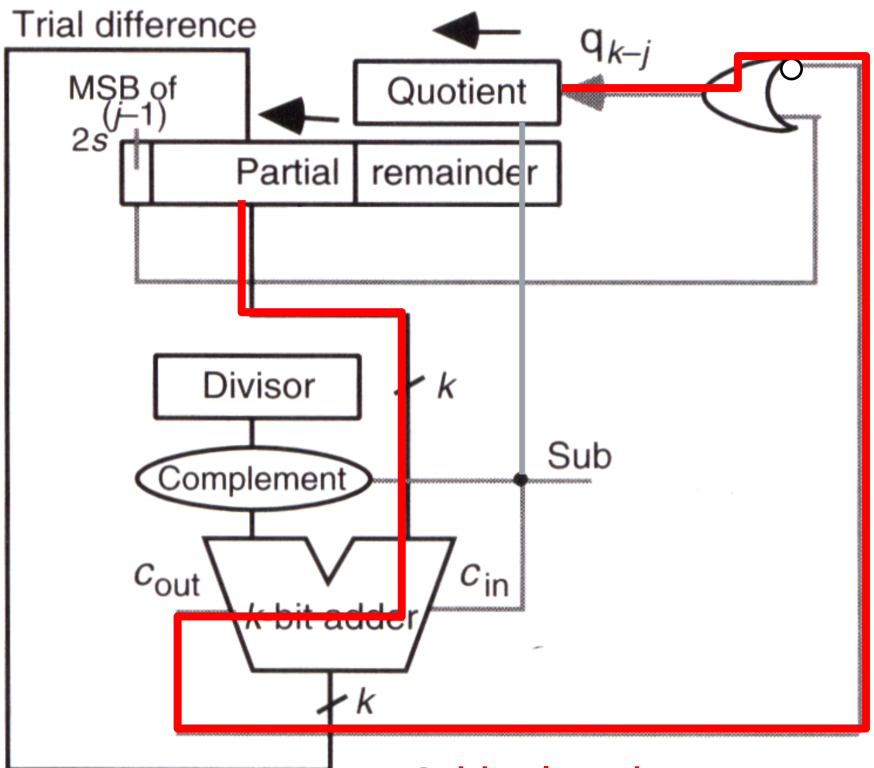
- Always store $u-2^k d$ back to the register
- If the value q in this stage is 1 ➔ correct!
  - Next stage: $2(u-2^k d)-2^k d=2u-3x2^k d$
- If the value q in this stage is 0 ➔ incorrect!
  - Next stage should be: $2u-2^k d$
  - Is equal to $2(u-2^k d)+2^k d$
- Always store the result of trail difference
  - If q=1 ➔ use subtraction; if q=0 ➔ use addition
- Can reduce critical path

Nonrestoring Bit-Serial Hardware Divider (2/4)



z		0 1 1 1	0 1 0 1	No overflow, since: $(0111)_{two} < (1010)_{two}$
2^4d	0	1 0 1 0		
-2^4d	1	0 1 1 0		
$s^{(0)}$	0	0 1 1 1	0 1 0 1	
$2s^{(0)}$	0	1 1 1 0	1 0 1	Positive, so subtract
$+(-2^4d)$	1	0 1 1 0		
$s^{(1)}$	0	0 1 0 0	1 0 1	
$2s^{(1)}$	0	1 0 0 1	0 1	Positive, so set $q_3 = 1$ and subtract
$+(-2^4d)$	1	0 1 1 0		
$s^{(2)}$	1	1 1 1 1	0 1	
$2s^{(2)}$	1	1 1 1 0	1	Negative, so set $q_2 = 0$ and add
$+2^4d$	0	1 0 1 0		
$s^{(3)}$	0	1 0 0 0	1	
$2s^{(3)}$	1	0 0 0 1		Positive, so set $q_1 = 1$ and subtract
$+(-2^4d)$	1	0 1 1 0		
$s^{(4)}$	0	0 1 1 1		Positive, so set $q_0 = 1$
s			0 1 1 1	
q			1 0 1 1	

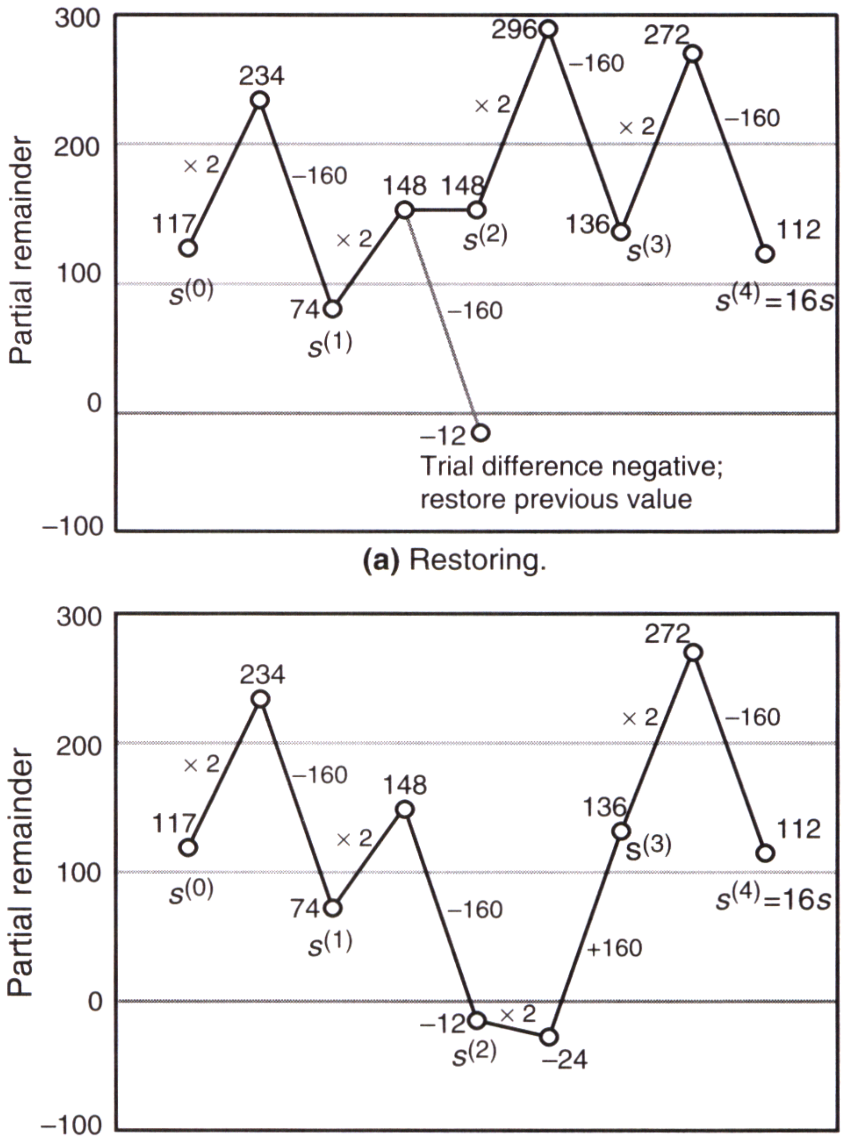
Nonrestoring Bit-Serial Hardware Divider (3/4)



Critical path

z		0 1 1 1	0 1 0 1	No overflow, since: $(0111)_{two} < (1010)_{two}$
2^4d	0	1 0 1 0		
-2^4d	1	0 1 1 0		
$s^{(0)}$	0	0 1 1 1	0 1 0 1	
$2s^{(0)}$	0	1 1 1 0	1 0 1	Positive, so subtract
$+(-2^4d)$	1	0 1 1 0		
$s^{(1)}$	0	0 1 0 0	1 0 1	
$2s^{(1)}$	0	1 0 0 1	0 1	Positive, so set $q_3 = 1$ and subtract
$+(-2^4d)$	1	0 1 1 0		
$s^{(2)}$	1	1 1 1 1	0 1	
$2s^{(2)}$	1	1 1 1 0	1	Negative, so set $q_2 = 0$ and add
$+2^4d$	0	1 0 1 0		
$s^{(3)}$	0	1 0 0 0	1	
$2s^{(3)}$	1	0 0 0 1		Positive, so set $q_1 = 1$ and subtract
$+(-2^4d)$	1	0 1 1 0		
$s^{(4)}$	0	0 1 1 1		Positive, so set $q_0 = 1$
s			0 1 1 1	
q			1 0 1 1	

Nonrestoring Bit-Serial Hardware Divider (4/4)



(a) Restoring.
(b) Nonrestoring.

Division by Constants (1/2)

- Use **lookup table** + constant multiplier
- Exploit the following equations
  - Consider **odd divisor** only since even divisor can be performed by first dividing by an odd integer and then shifting the result
  - For an odd integer d, there exists an odd integer m such that $d \times m = 2^n - 1$

Division by Constants (2/2)

- $$\frac{1}{d}=\frac{m}{2^n-1}=\frac{m}{2^n(1-2^{-n})}=\frac{m}{2^n}(1+2^{-n})(1+2^{-2n})(1+2^{-4n})\cdots$$
- For example, for 24-bit precision:

$$d=5,\Rightarrow m=3,n=4$$

Easy for hardware implementation

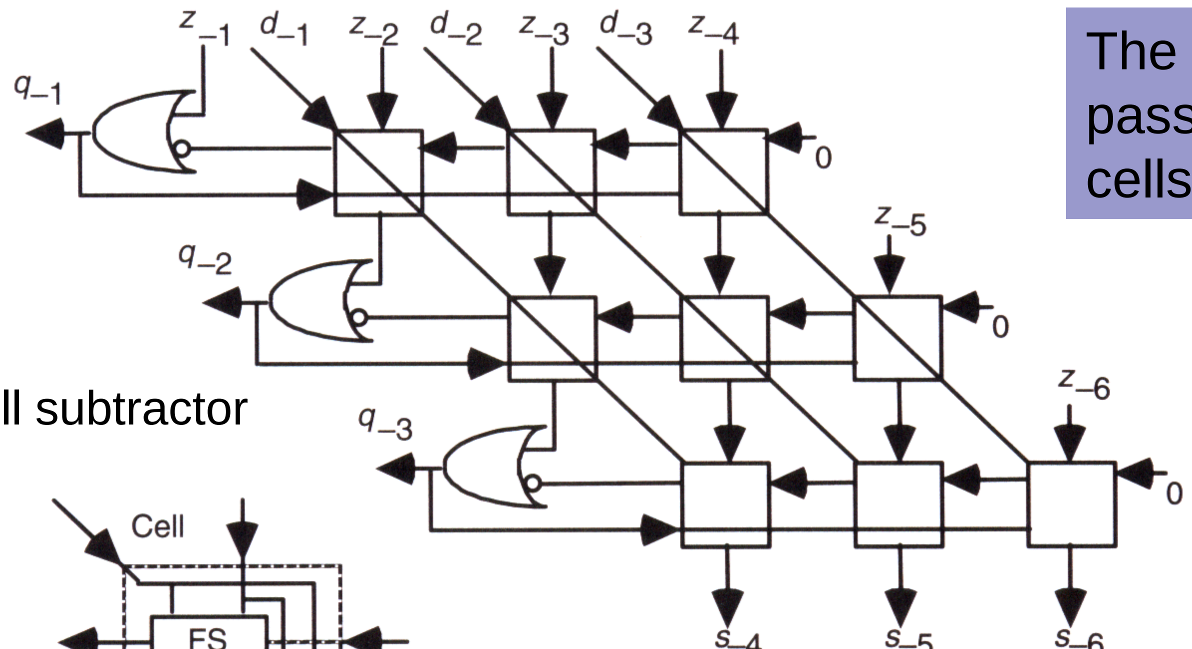
$$\frac{z}{5}=\frac{3z}{2^4-1}=\frac{3z}{16(1-2^{-4})}=\frac{3z}{16}\boxed{(1+2^{-4})(1+2^{-8})(1+2^{-16})}$$

Next term $(1+2^{-32})$ does not contribute anything to 24-bit precision

Array Divider (1/2)

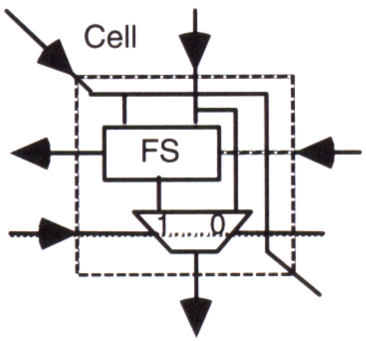
- Restoring array divider

The critical path passes through all k^2 cells



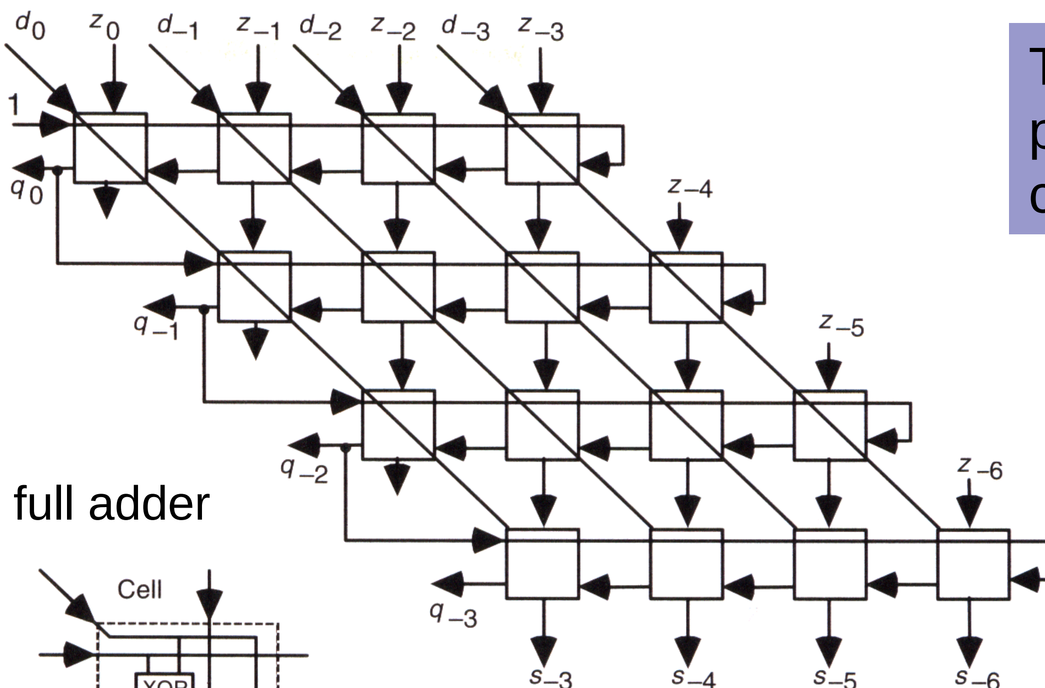
Dividend $z = .z_{-1}\, z_{-2}\, z_{-3}\, z_{-4}\, z_{-5}\, z_{-6}$
Divisor $d = .d_{-1} d_{-2} d_{-3}$
Quotient $q = .q_{-1} q_{-2}\, q_{-3}$
Remainder $s = .0 \quad 0 \quad 0 \quad s_{-4}\, s_{-5}\, s_{-6}$

FS: full subtractor



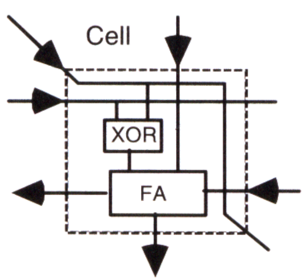
Array Divider (2/2)

- Nonrestoring array divider



The critical path passes through all k^2 cells

FA: full adder



Dividend	$z = z_0.z_{-1}\ z_{-2}\,z_{-3}z_{-4}z_{-5}\ z_{-6}$
Divisor	$d = d_0.d_{-1}d_{-2}d_{-3}$
Quotient	$q = q_0.q_{-1}q_{-2}q_{-3}$
Remainder	$s = 0\ .0\ \ 0\ \ s_{-3}s_{-4}s_{-5}s_{-6}$

Distributed Arithmetic (1/7)

- Most DSP algorithms involve sum-of-products (inner products)

$$y = \mathbf{a} \cdot \mathbf{x} = \sum_{i=1}^{N} a_i x_i$$

Fixed coefficient

- **Distributed arithmetic (DA)** is an efficient procedure for computing inner products between a fixed and a variable data vector

Distributed Arithmetic (2/7)

$$y = \sum_{i=1}^{N} a_i \left[-x_{i0} + \sum_{k=1}^{W_d - 1} x_{ik} 2^{-k} \right]$$

$$y = -\sum_{i=1}^{N} a_i x_{i0} + \sum_{k=1}^{W_d - 1} \left[\sum_{i=1}^{N} a_i x_{ik} \right] 2^{-k}$$

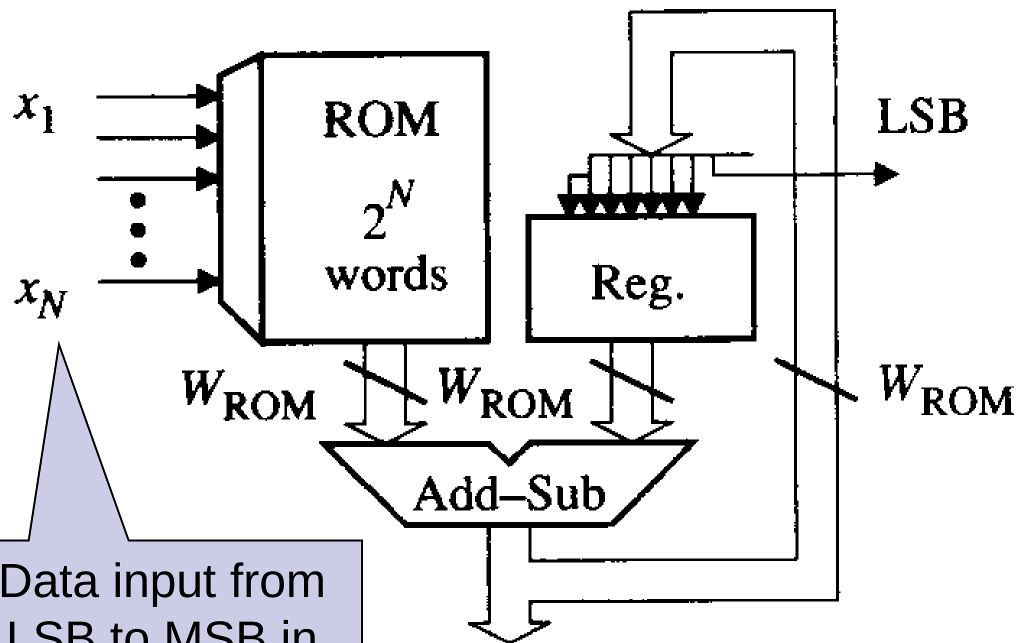
$$y = -F_0(x_{10}, x_{20}, \ldots, x_{N0}) + \sum_{k=1}^{W_d - 1} F_k\left(x_{1k}, x_{2k}, \ldots, x_{Nk}\right) 2^{-k}$$

$$\text{where} \quad F_k(x_{1k}, x_{2k}, \ldots, x_{Nk}) = \sum_{i=1}^{N} a_i x_{ik}$$

Put F_k in ROM

Distributed Arithmetic (3/7)

$$y = ((\ldots((0 + F_{W_d - 1})2^{-1} + F_{W_d - 2})2^{-1} + \ldots + F_2)2^{-1} + F_1)2^{-1} - F_0$$



Data input from LSB to MSB in bit-serial

- DA can be implemented with a ROM and a shift-accumulator
- The computation time: Wd cycles
- Word length of ROM: $W_{ROM} \le W_C + \log_2(N)$

Distributed Arithmetic (4/7)

- Example
  - $y=a_1x_1+a_2x_2+a_3x_3$
  - $a_1=(0.0100001)_{2C}$
  - $a_2=(0.1010101)_{2C}$
  - $a_3=(1.1110101)_{2C}$
- (a) The table? (b) The word length of the shift-accumulator?

Distributed Arithmetic (5/7)

- Ans:
  - (a)

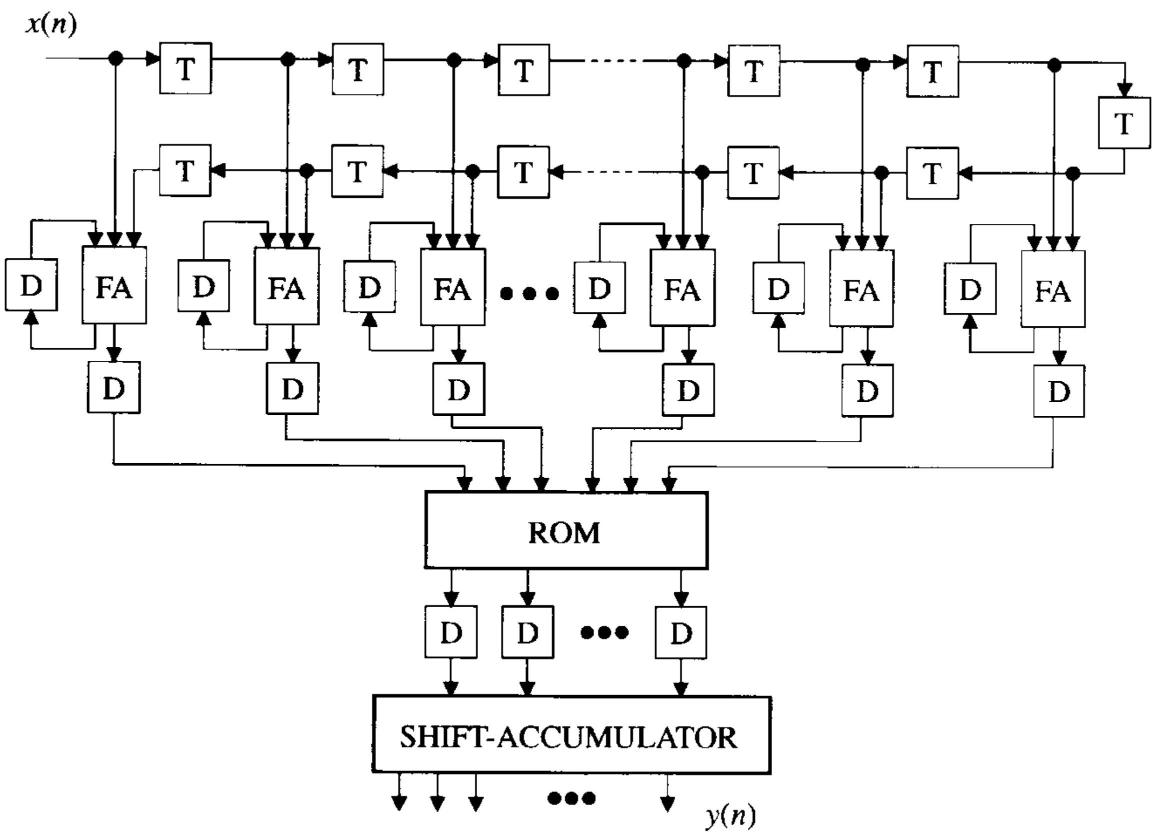
x_1 x_2 x_3	F_k	F_k	F_k
0 0 0	0	0.0000000	0.0000000
0 0 1	a_3	1.1110101	0.0859375
0 1 0	a_2	0.1010101	0.6640625
0 1 1	$a_2 + a_3$	0.1001010	0.5781250
1 0 0	a_1	0.0100001	0.2578125
1 0 1	$a_1 + a_3$	0.0010110	0.1718750
1 1 0	$a_1 + a_2$	0.1110110	0.9218750
1 1 1	$a_1 + a_2 + a_3$	0.1101011	0.8359375

  - (b) Word length=7 bits + 1 bit (sign bit) +1 bit (guard bit) = 9 bits

$$|y| = ((\ldots((0 + F_{max})2^{-1} + F_{max})2^{-1} + \ldots + F_{max})2^{-1} + F_{max})2^{-1} \le F_{max}$$

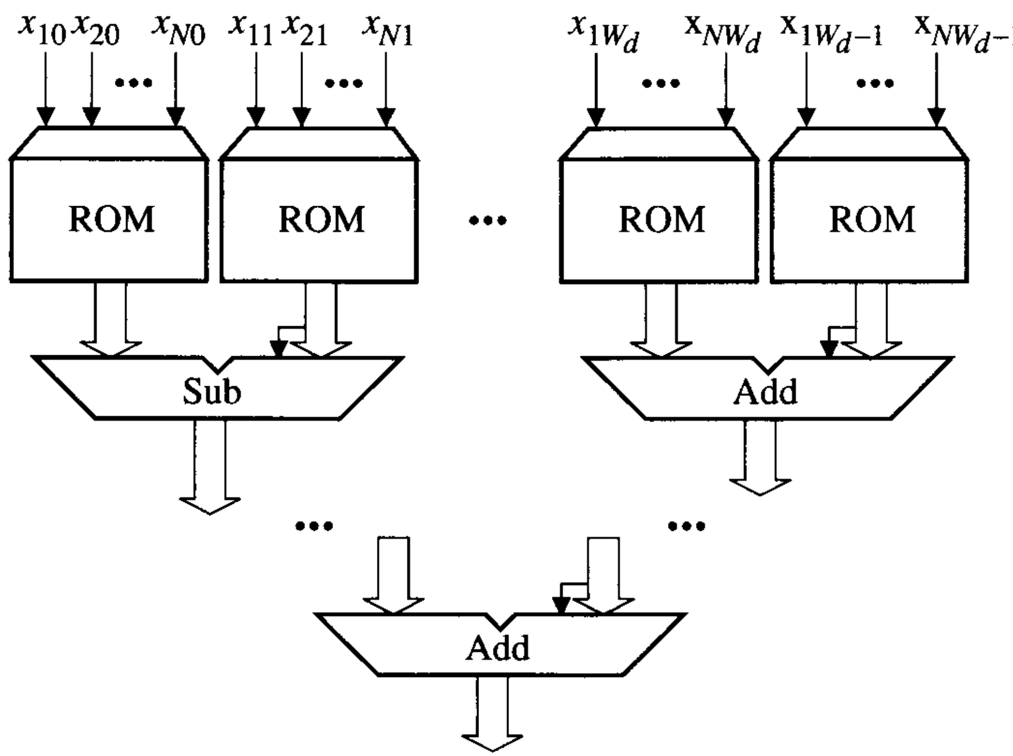
Distributed Arithmetic (6/7)

- Example: linear-phase FIR filter

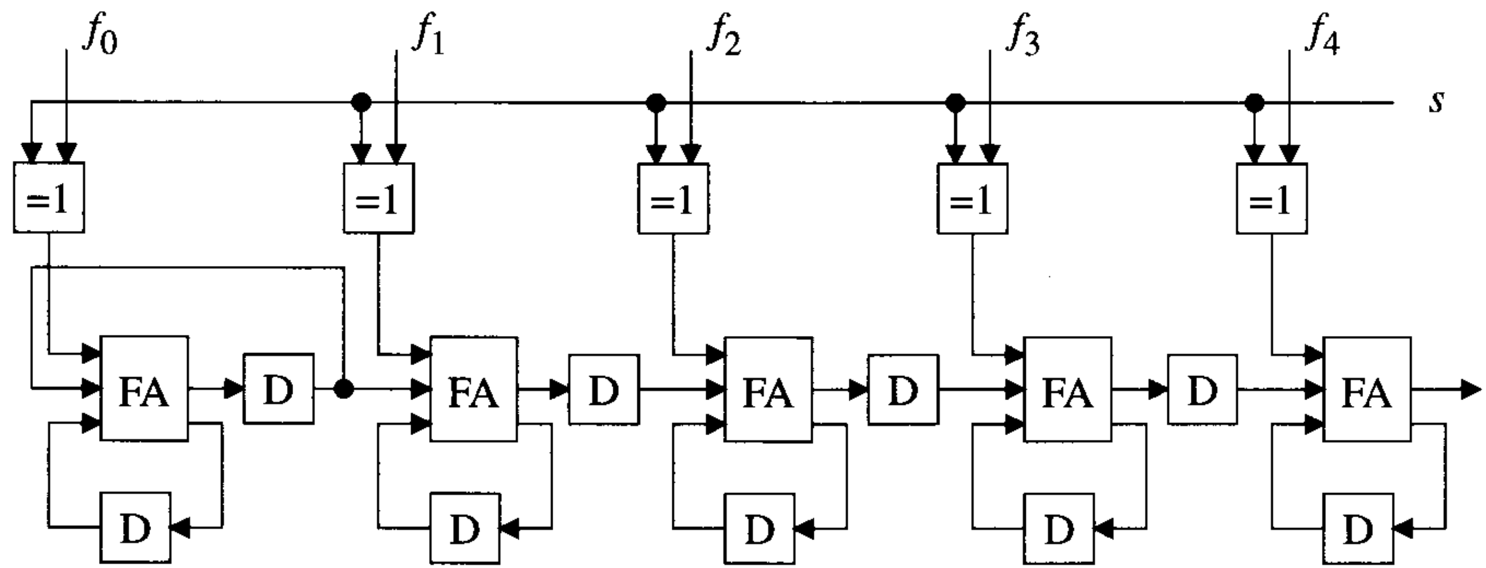


Distributed Arithmetic (7/7)

- Parallel implementation of distributed arithmetic



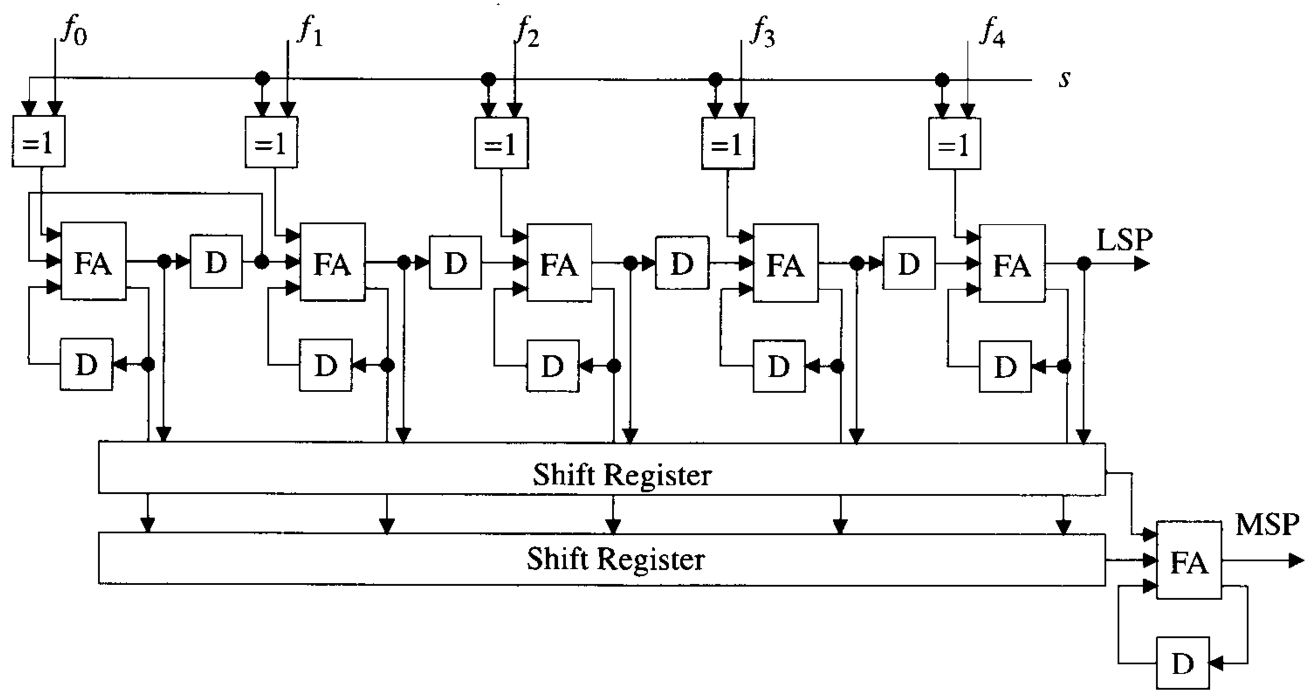
Shift-Accumulator (1/4)



- The number of cycles for one inner product is $W_d + W_{ROM}$
  - First W_d cycles: input data
  - Last W_{ROM} cycles: shift out the results

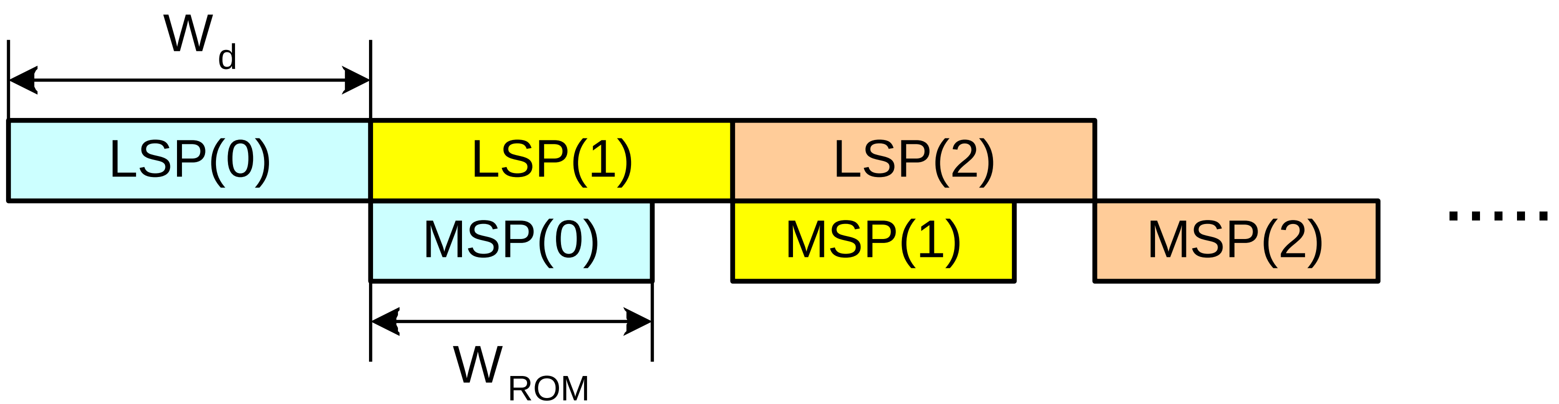
Shift-Accumulator (2/4)

- Shift-accumulator augmented with two shift registers



Shift-Accumulator (3/4)

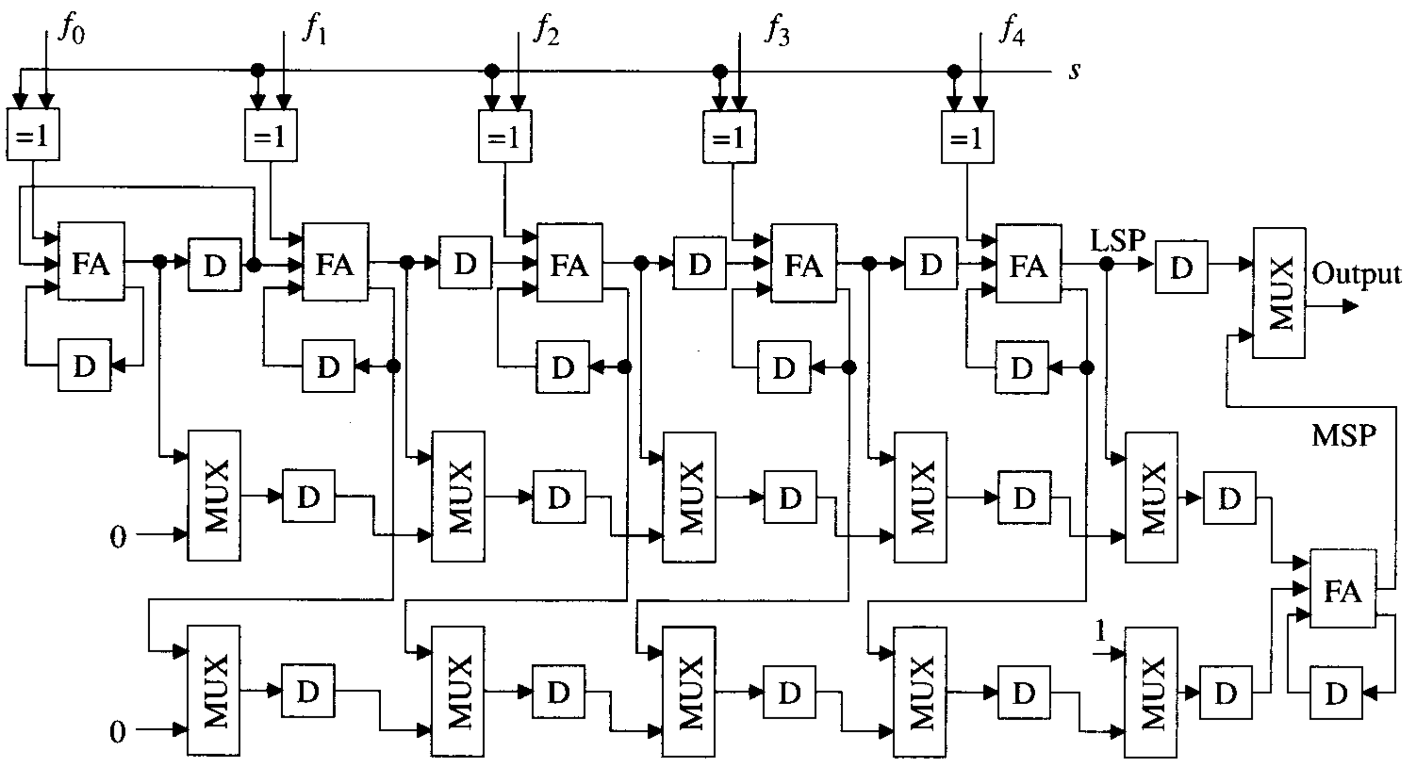
- Scheduling



- Clock cycle
  - $N_{CL}=\max\{W_{ROM}, W_d\}$

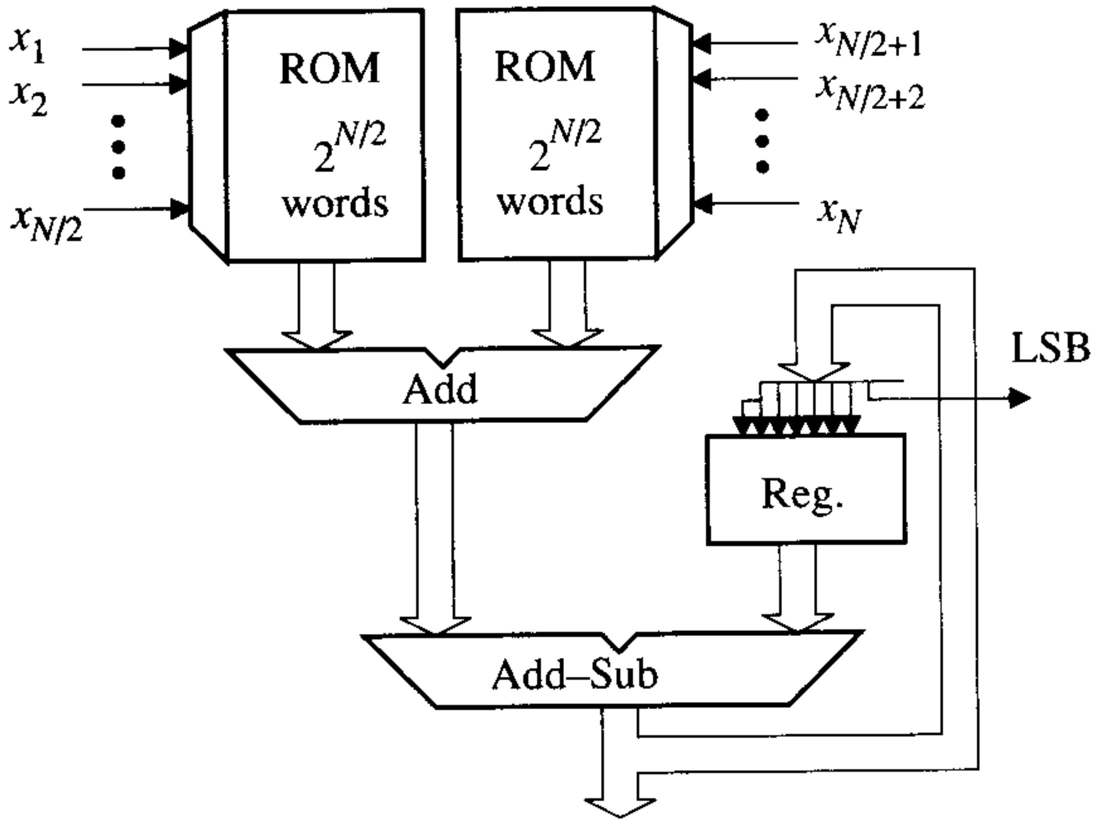
Shift-Accumulator (4/4)

- Detailed architecture



Reducing the Memory Size (1/4)

- Method 1: memory partition
  - $2*2^{N/2} < 2^N$
  - Ex: $2*2^5 = 64 < 2^{10} = 1024$



Reducing the Memory Size (2/4)

- Method 2: memory coding

$$x = \frac{1}{2}[x - (-x)]$$

$$= \frac{1}{2}\left[-x_0 + \sum_{k=1}^{W_d-1} x_k 2^{-k} - \left(-\overline{x_0} + \sum_{k=1}^{W_d-1} \overline{x_k} 2^{-k} + 2^{-(W_d-1)}\right)\right]$$

$$= -(x_0 - \overline{x_0})2^{-1} + \sum_{k=1}^{W_d-1} (x_k - \overline{x_k})2^{-k-1} - 2^{-W_d}$$

$$y = \sum_{k=1}^{W_d-1} F_k(x_{1k}, \ldots, x_{Nk})2^{-k-1} - F_0(x_{10}, \ldots, x_{N0})2^{-1} + F(0, \ldots, 0)2^{-W_d}$$

$$\text{where } F_k(x_{1k}, x_{2k}, \ldots, x_{Nk}) = \sum_{i=1}^{N} a_i(x_k - \overline{x_k})$$

Reducing the Memory Size (3/4)

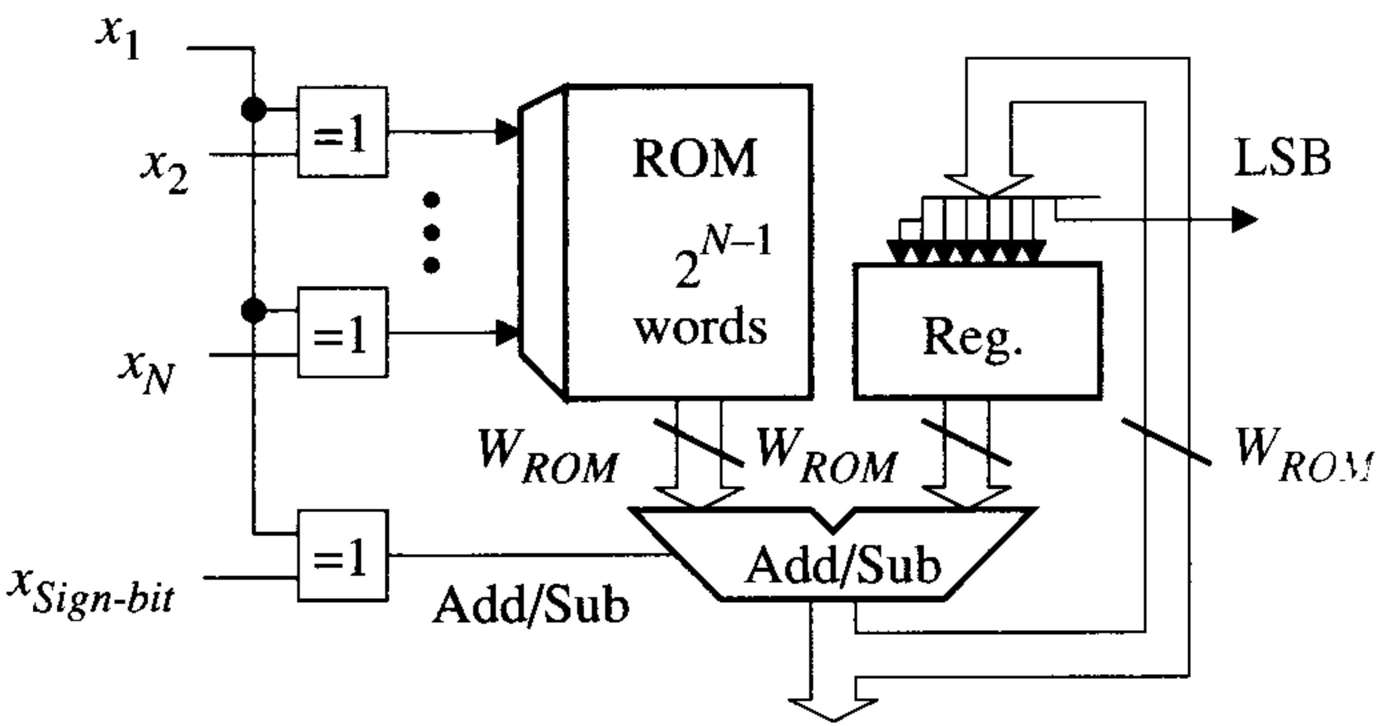
$$u_1 = x_1 \otimes x_2 \qquad A/S = x_1 \otimes x_{sign\text{-}bit}$$

$$u_2 = x_1 \otimes x_3$$

Complement

x_1	x_2	x_3	F_k	u_1	u_2	A/S
0	0	0	$-a_1 - a_2 - a_3$	0	0	A
0	0	1	$-a_1 - a_2 + a_3$	0	1	A
0	1	0	$-a_1 + a_2 - a_3$	1	0	A
0	1	1	$-a_1 + a_2 + a_3$	1	1	A
1	0	0	$+a_1 - a_2 - a_3$	1	1	S
1	0	1	$+a_1 - a_2 + a_3$	1	0	S
1	1	0	$+a_1 + a_2 - a_3$	0	1	S
1	1	1	$+a_1 + a_2 + a_3$	0	0	S

Reducing the Memory Size (4/4)

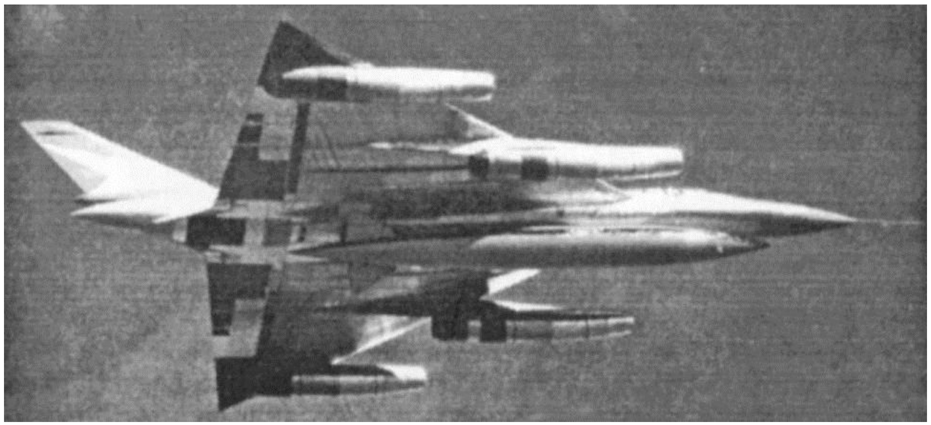


CORDIC

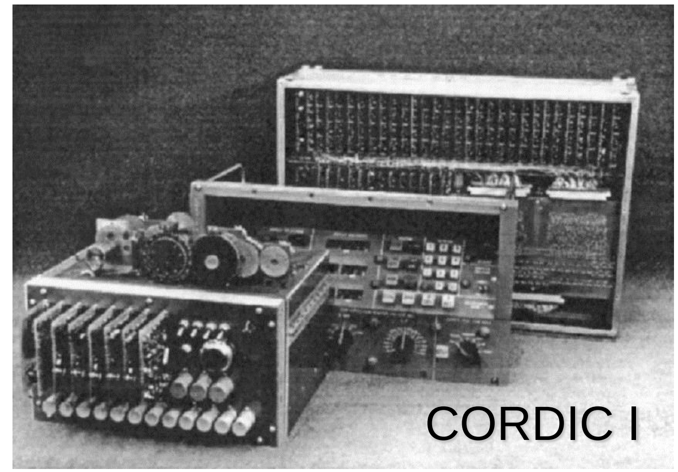
Major reference:
[1] A.-Y. Wu, "CORDIC," Slides of *Advanced VLSI*
[2] Y. H. Hu, "CORDIC-based VLSI architectures for digital signal processing," *IEEE Signal Processing Magazine*, pp. 16—35, July 1992.
[3] J. E. Volder, "The Birth of CORDIC," J. VLSI Signal Processing, vol.25, pp. 101—105, 2000.

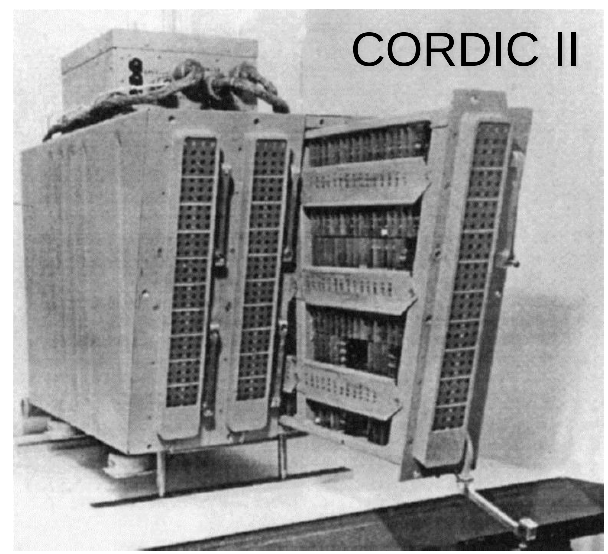
- CORDIC (**CO**ordinate **R**otation **DI**gital **C**omputer)
  - An **iterative arithmetic algorithm** introduced by Volder in 1956
  - Can handle many elementary functions, such as trigonometric, exponential, and logarithm **with only shift-and-add** arithmetic
  - For these functions CORDIC based architecture is much efficient than multiplier and accumulator (MAC) based architecture
  - Suitable for transformations and matrix based filters

The Birth of CORDIC



B-58 Supersonic Bomber

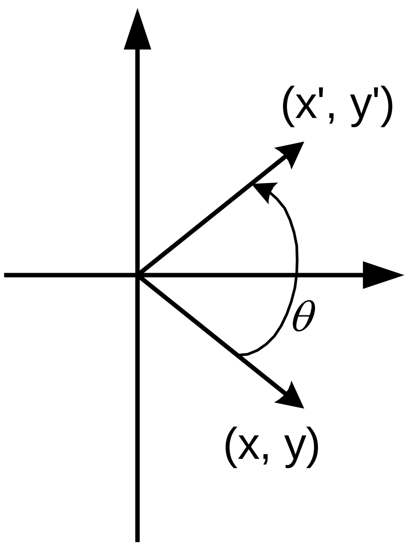




Simple Concepts of CORDIC (1/2)

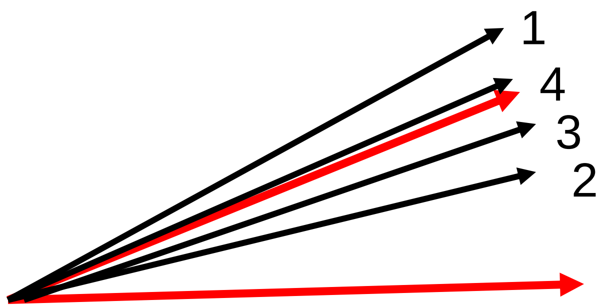
- Originally, CORDIC is invented to deal with rotation problem with shift-and-add arithmetic

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$



Simple Concepts of CORDIC (2/2)

- How to make it with shift-and-add?
- Decompose the desired rotation angle into small rotation angles (micro-rotation)
- Rotate finite times (by "elementary angles" $\{a_i \mid 0 \le i \le n-1\}$) to achieve the desired rotation θ



Conventional CORDIC Algorithm (1/2)

$$\begin{bmatrix} x(i+1) \\ y(i+1) \end{bmatrix} = \begin{bmatrix} \cos a_i & -\sin a_i \\ \sin a_i & \cos a_i \end{bmatrix} \begin{bmatrix} x(i) \\ y(i) \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} x(i+1) \\ y(i+1) \end{bmatrix} = \cos a_i \begin{bmatrix} 1 & -\tan a_i \\ \tan a_i & 1 \end{bmatrix} \begin{bmatrix} x(i) \\ y(i) \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} x(i+1) \\ y(i+1) \end{bmatrix} = \cos a_i \begin{bmatrix} 1 & -2^{-i} \\ 2^{-i} & 1 \end{bmatrix} \begin{bmatrix} x(i) \\ y(i) \end{bmatrix}$$

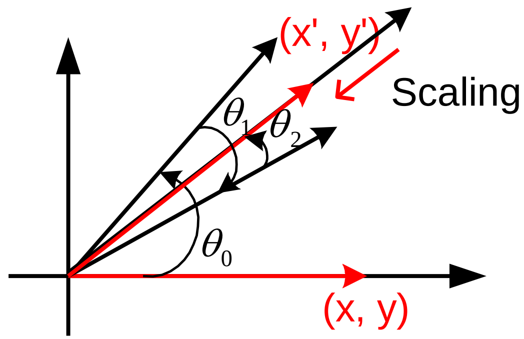
$$a_i = \tan^{-1} 2^{-i}, \cos a_i = \frac{1}{\sqrt{1+2^{-2i}}}$$

Conventional CORDIC Algorithm (2/2)

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

$$= S \times \begin{bmatrix} 1 & -\mu_0 2^{-0} \\ \mu_0 2^{-0} & 1 \end{bmatrix} \times \cdots$$

$$\times \begin{bmatrix} 1 & -\mu_i 2^{-i} \\ \mu_i 2^{-i} & 1 \end{bmatrix} \times \cdots \times \begin{bmatrix} 1 & -\mu_{n-1} 2^{-(n-1)} \\ \mu_{n-1} 2^{-(n-1)} & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix}$$



$$\theta_0 = 1 \cdot \tan^{-1} 2^{-0} = \mu_0 a_0$$
$$\theta_1 = -1 \cdot \tan^{-1} 2^{-1} = \mu_1 a_1$$
$$\theta_2 = 1 \cdot \tan^{-1} 2^{-2} = \mu_2 a_2$$

$$\text{Scaling factor}: S = \frac{1}{\prod_{i=0}^{n-1} \sqrt{1+\mu_i^2 2^{-2i}}}$$

$$\text{Mode of rotation}: \mu_i \in \{-1, 1\}$$

Can be implemented with shift-and-add arithmetic

Generalized CORDIC (1/2)

- Target: $\theta = \sum_{i=0}^{n-1} \mu_i a_m(i)$
- i-th elementary rotation angle is defined by

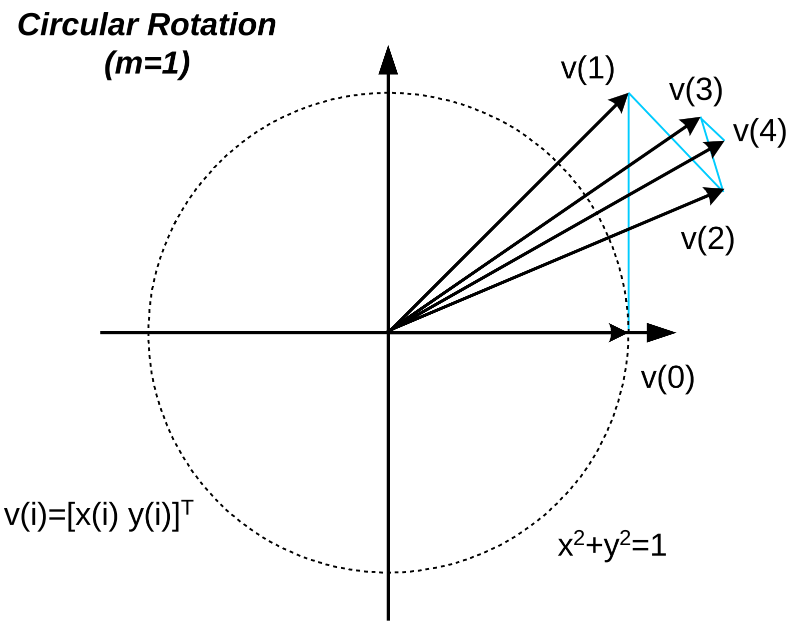
$$a_m(i) = \frac{1}{\sqrt{m}} \tan^{-1}\left[\sqrt{m}\, 2^{-s(m,i)}\right] = \begin{cases} -2^{s(0,i)} & m \to 0 \quad \text{Linear coordinate} \\ \tan^{-1} 2^{-s(1.i)} & m = 1 \quad \text{Circular coordinate} \\ \tanh^{-1} 2^{-s(-1,i)} & m = -1 \quad \text{Hyperbolic coordinate} \end{cases}$$

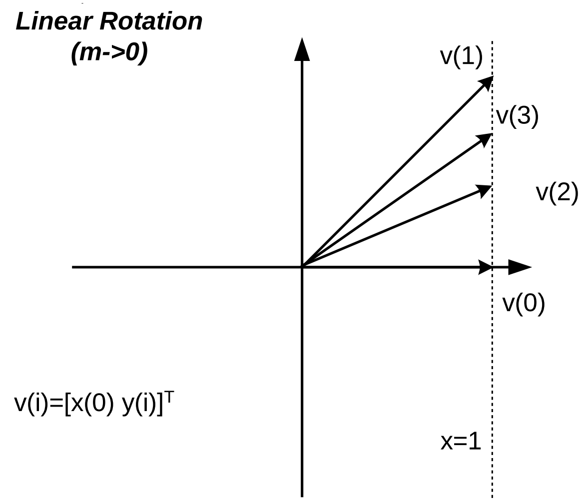
$\text{norm of a vector } [\mathrm{x\ y}]^T \text{ is } \sqrt{\mathrm{x}^2 + my^2}$

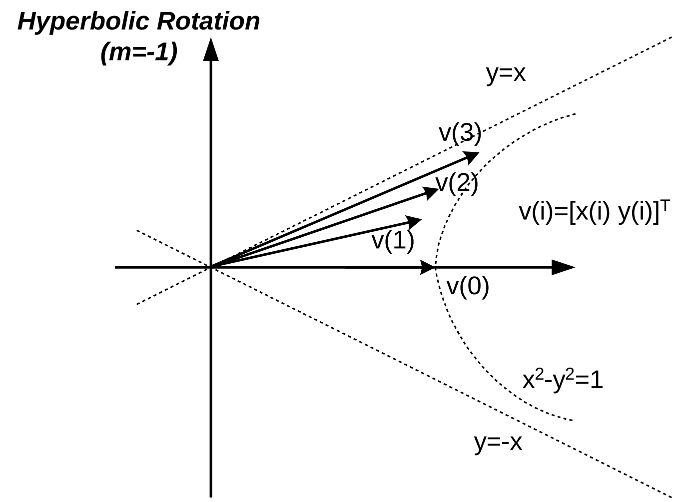
$\mu_i \in \{-1,1\}$: mode of rotation

$s(m,i)$: non-descreasing integer shift sequence

Generalized CORDIC (2/2)







CORDIC Algorithm

Initiation : Given $x(0), y(0), z(0)$

For i = 0 to n -1, Do

/* CORDIC iteration equation */

$$\begin{bmatrix} x(i+1) \\ y(i+1) \end{bmatrix} = \begin{bmatrix} 1 & -\mu_i 2^{-s(m,i)} \\ \mu_i 2^{-s(m,i)} & 1 \end{bmatrix} \begin{bmatrix} x(i) \\ y(i) \end{bmatrix}$$

/* Angle updating equation */

$$z(i+1) = z(i) - \mu_i a_m(i)$$

End i - loop

/* Scaling operation (required for $m = \pm 1$ only) */

$$\begin{bmatrix} x_f \\ y_f \end{bmatrix} = \frac{1}{K_m(n)} \cdot \begin{bmatrix} x(n) \\ y(n) \end{bmatrix} = \frac{1}{\prod_{i=0}^{n-1} \sqrt{1 + m\mu_i^2 2^{-2s(m,i)}}} \cdot \begin{bmatrix} x(n) \\ y(n) \end{bmatrix}$$

Remained problems:

μ_i

$s(m,i)$

Scaling

Mode of Operation (1/2)

- Vector rotation mode (θ is given)

  $$z(0) = \theta$$

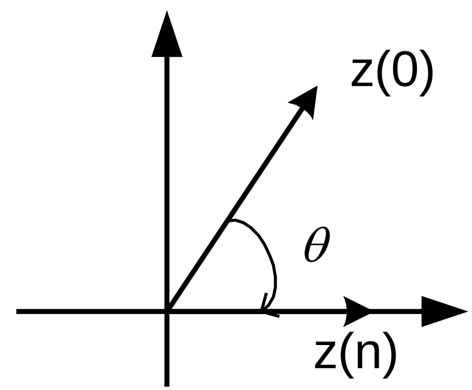
  After n iterations, the total angle rotated is :

  $$z(0) - z(n) = \theta - z(n) = \sum_{i=0}^{n-1} \mu_i a_m(i)$$

  we want to make $|z(n)| \to 0$

  $$\mu_i = \text{sign of } z(i)$$

  - For many DSP problems, θ is know in advance, and sequence $\{\mu_i\}$ can be stored instead



Mode of Operation (2/2)

- Angle accumulation mode (θ is not given)
  - The objective is to rotate the given initial vector $[x(0)\ y(0)]^T$ back to the x-axis

$$\text{set } z(0) = 0$$

$$\mu_i = -\text{sign of } x(i) \cdot y(i)$$

- Summary

$$\mu_i = \begin{cases} \text{sign of } z(i) & \text{Vector rotation mode} \\ -\text{sign of } x(i) \cdot y(i) & \text{Angle accumulation mode} \end{cases}$$

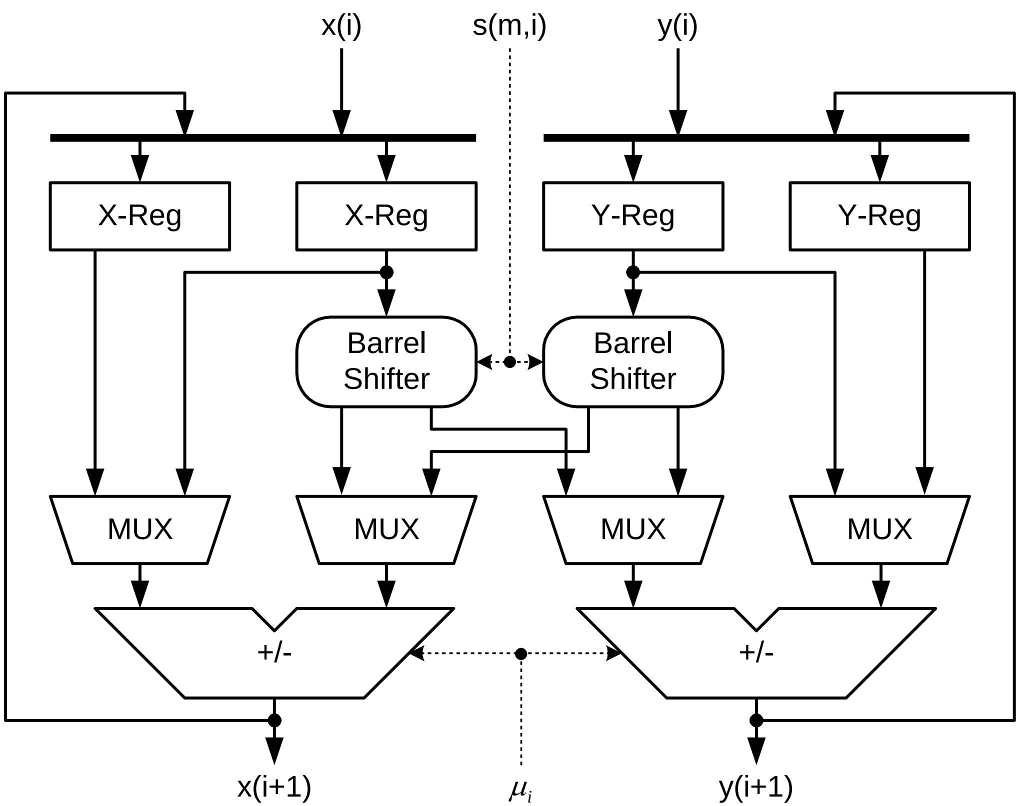
Shift Sequence

- Usually defined in advance
- Walther has proposed a set of shift sequence for each of the three coordinate systems
  - For m=0 or 1, s(m,i)=i
  - For m=-1, s(-1, i)=1, 2, 3, 4, 4, 5, …, 12, 13, 13, 14, …

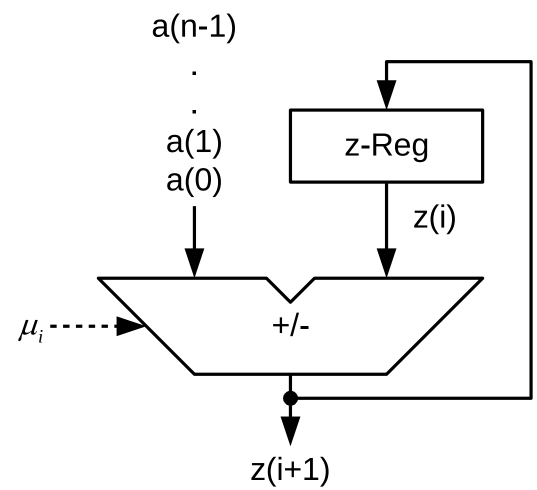
Scaling Operation $\frac{1}{K_m(n)}$

- Significant computation overhead of CORDIC
- Fortunately, since $|\mu_i| = 1$, and assume $\{s(m,i)\}$ is given, $K_m(n)$ can be computed in advance
- Two approaches to compute scaling
  - CSD representation $$\frac{1}{K_m(n)} = \sum_{p=1}^{P} \kappa_p 2^{-i_p}$$
    $\kappa_q = \pm 1$
  - Project of factors $$\frac{1}{K_m(n)} = \prod_{q=1}^{Q} (1 + \kappa_q 2^{-i_q}) + \varepsilon_q$$

Basic CORDIC Processor (1/3)



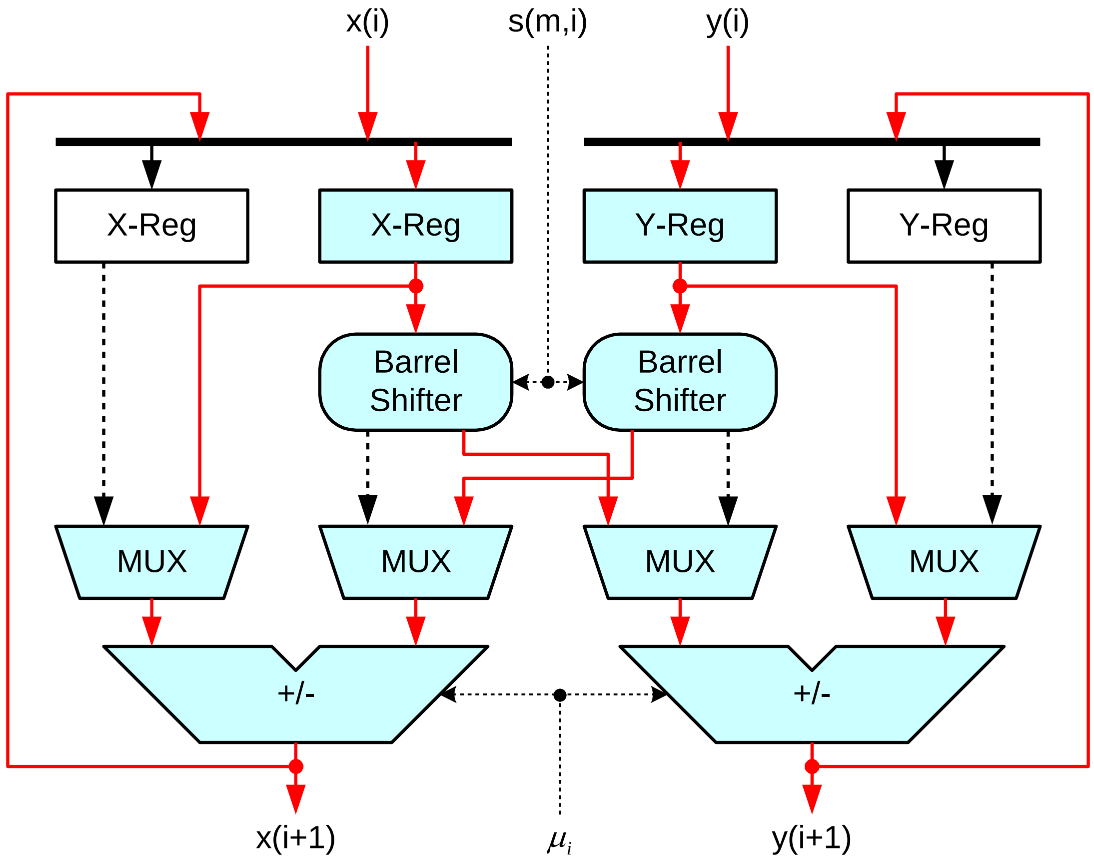
For CORDIC Iteration and Scaling



For Angle Update

Basic CORDIC Processor (2/3)

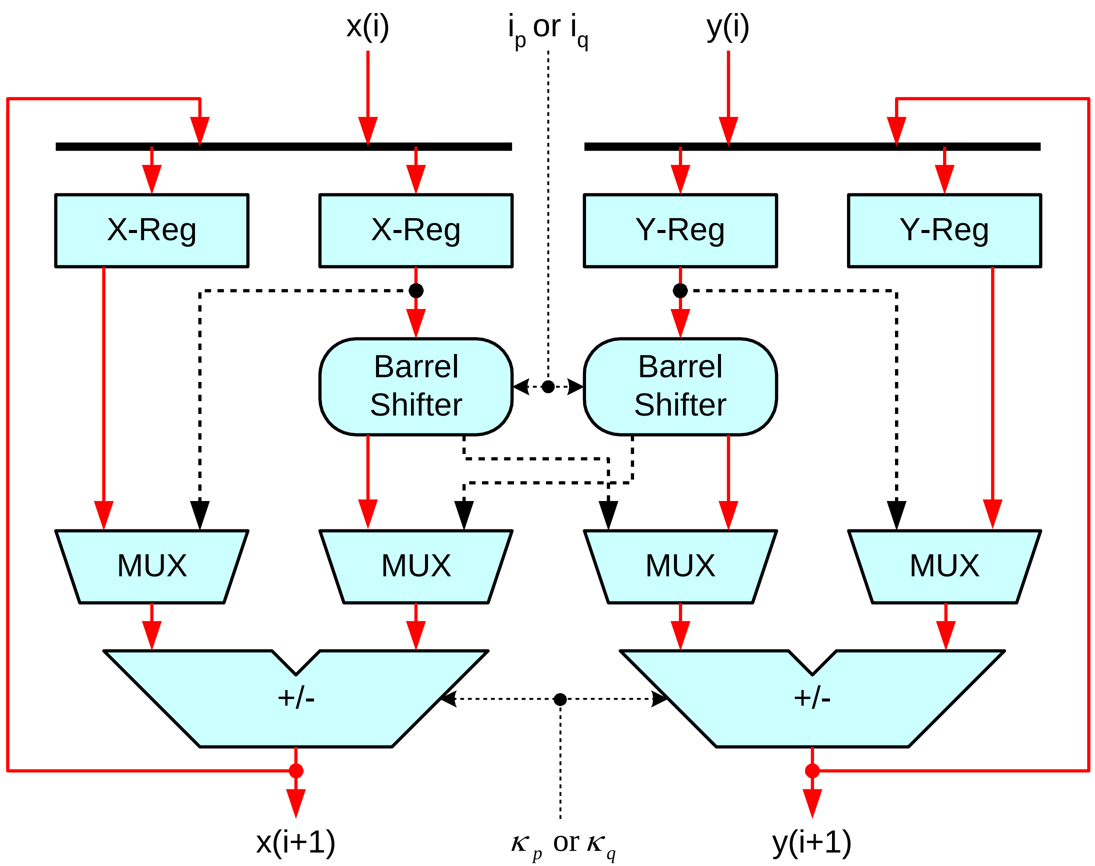
- CORDIC Iteration



$$\begin{bmatrix} x(i+1) \\ y(i+1) \end{bmatrix} = \begin{bmatrix} 1 & -\mu_i 2^{-s(m,i)} \\ \mu_i 2^{-s(m,i)} & 1 \end{bmatrix} \begin{bmatrix} x(i) \\ y(i) \end{bmatrix}$$

Basic CORDIC Processor (3/3)

- Scaling



I: $$\frac{1}{K_m(n)} = \sum_{p=1}^{P} \kappa_p 2^{-i_p}$$

II: $$\frac{1}{K_m(n)} = \prod_{q=1}^{Q} (1+\kappa_q 2^{-i_q})$$

Given $x'(0)=x(n), y'(0)=y(n)$

TypeI:

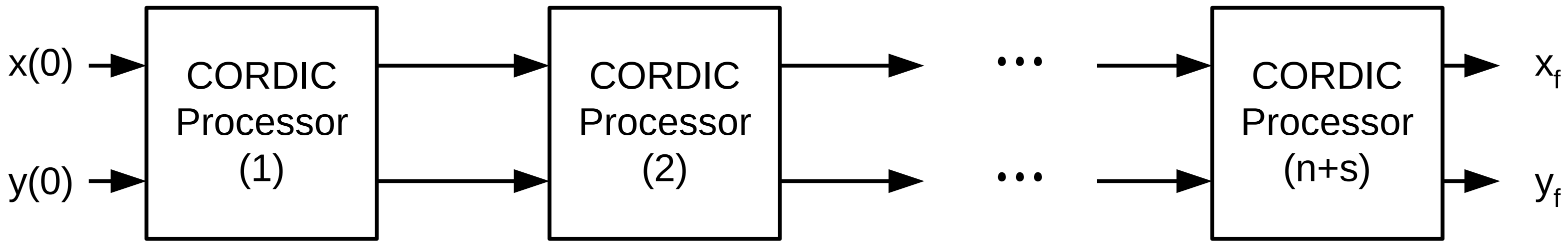
$$\begin{cases} x'(p+1) = x'(p) + \kappa_p 2^{-i_p} x(n) \\ y'(p+1) = y'(p) + \kappa_p 2^{-i_p} x(n) \end{cases}$$

TypeII:

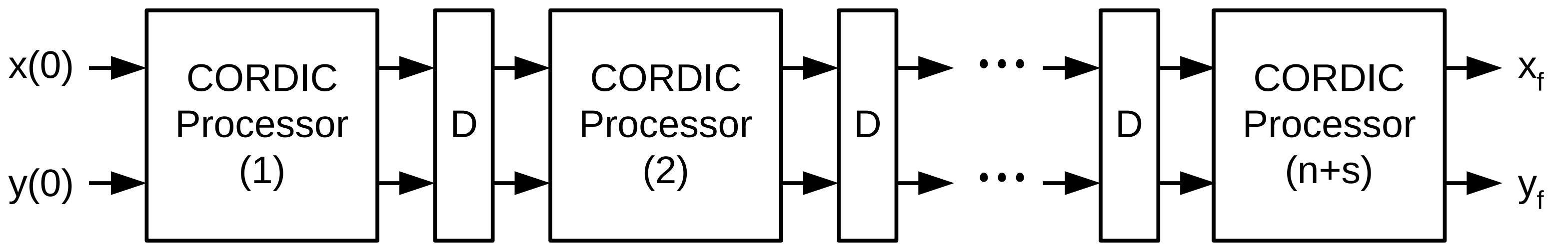
$$\begin{cases} x'(q+1) = x'(q) + \kappa_q 2^{-i_q} x'(q) \\ y'(q+1) = y'(q) + \kappa_q 2^{-i_q} x'(q) \end{cases}$$

Parallel and Pipelined Arrays

- n stages for CORDIC, and s stages for scaling
- Parallel



- Pipelined



Discrete Fourier Transform (DFT) with CORDIC (1/2)

- DFT

$$Y(K) = X(0)e^{\frac{-j2\pi k0}{N}} + X(1)e^{\frac{-j2\pi k\cdot 1}{N}} + \cdots X(N-1)e^{\frac{-j2\pi k\cdot(N-1)}{N}}$$

- DFT with CORDIC

$\text{Initiation}: Y(0,k) = 0 \text{ for } 0 \le k \le N-1$

$\text{For k} = 0 \text{ to N-1, Do}$

$\text{For m} = 0 \text{ to N-1, Do}$

$$\begin{bmatrix} Y_r(m+1,k) \\ Y_i(m+1,k) \end{bmatrix} = K_1(n)\cdot\begin{bmatrix} \cos\frac{-2\pi mk}{N} & -\sin\frac{-2\pi mk}{N} \\ \sin\frac{-2\pi mk}{N} & \cos\frac{-2\pi mk}{N} \end{bmatrix}\begin{bmatrix} x_r(m) \\ x_i(m) \end{bmatrix} + \begin{bmatrix} Y_r(m,k) \\ Y_i(m,k) \end{bmatrix}$$

End m-loop

$\text{/* Scaling operation */}$

$$Y(k) = \frac{Y(N,k)}{K_1(n)}$$

End k-loop

Discrete Fourier Transform (DFT) with CORDIC (2/2)