DIGITAL HARDWARE ARITHMETIC

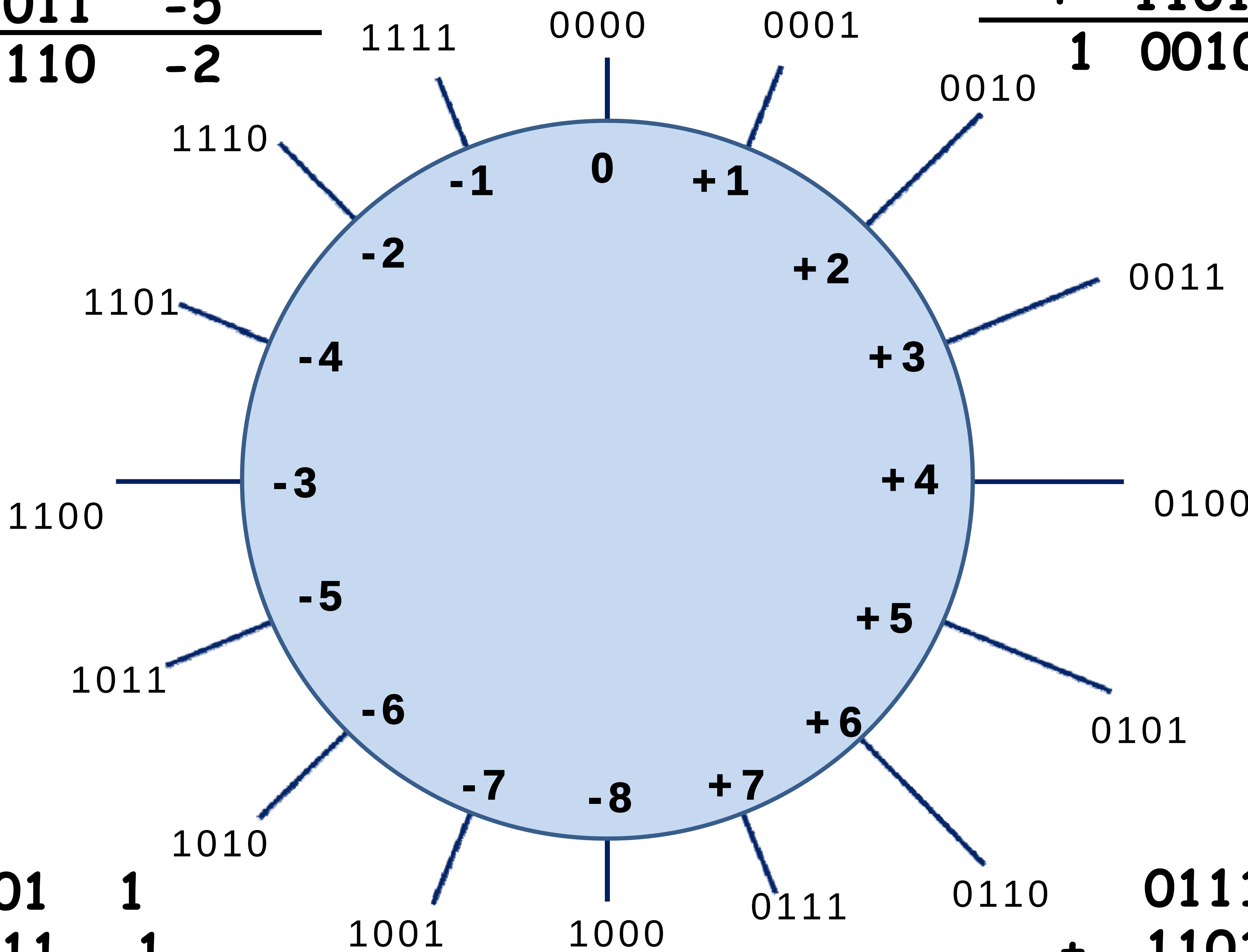
CS 2600

The Two's Complement Representation

- Range of numbers in two's complement method: $-2^{n-1} \leq X \leq 2^{n-1} - 1$
- Slightly asymmetric - one more negative number
- -2^{n-1} (represented by 10….0) does not have a positive equivalent
- A complement operation for this number will result in an overflow indication
- There is a unique representation for 0

	0011	3
+	1011	-5
	1110	-2

	0101	5
+	1101	-3
1	0010	2



	0001	1
+	1111	-1
1	0000	0

	0111	7
+	1101	-3
	0100	4

- Example - (two's complement)

```
         01001   9
         11001  -7
  --------------------
       1 00010   2
```

  - Carry-out discarded - does not indicate overflow

2's comp. binary	decimal
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7

- In general, if X and Y have opposite signs - no overflow can occur regardless of whether there is a carry-out or not

$$\begin{array}{cccccccl} & 0 & 0 & 1 & 0 & 1 & 5 & \\ + & 1 & 0 & 1 & 1 & 0 & -10 & \\ \hline & 1 & 1 & 0 & 1 & 1 & -5 & \text{No carry-out} \end{array}$$

$$\begin{array}{cccccccl} & 0 & 1 & 0 & 1 & 0 & 10 & \\ + & 1 & 1 & 0 & 1 & 1 & -5 & \\ \hline 1 & 0 & 0 & 1 & 0 & 1 & 5 & \text{Carry-out} \end{array}$$

- If X and Y have the same sign and result has different sign - overflow occurs

- Examples - (two's complement)

```
         10111   -9
         10111   -9
    ______________________
       1 01110   14 = -18 mod 32
```

  * Carry-out and overflow

```
         01001    9
         00111    7
    ______________________
       0 10000   -16 = 16 mod 32
```

  * No carry-out but overflow

	0101	5
+	0011	3
	1000	-8

* No carry-out but overflow
 -> 8 mod 16

	1011	-5
+	1100	-4
	1 0111	7

* Carry-out and overflow
 → -9 mod 16

Condition for overflow (for logic implementation):

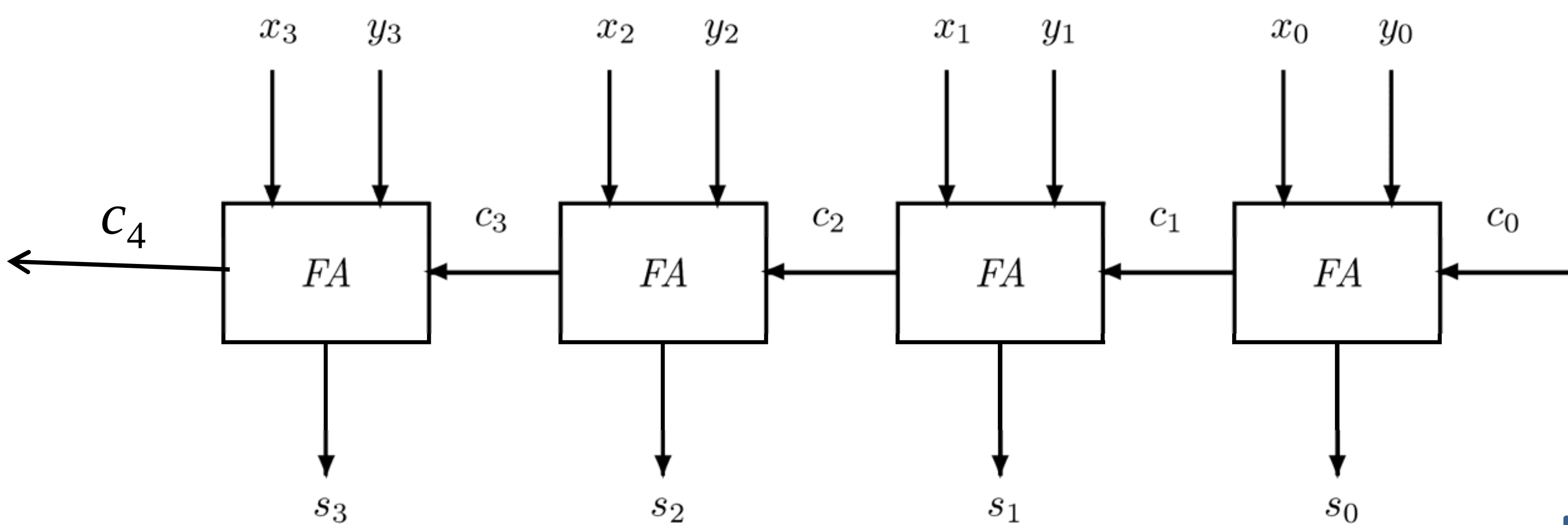
$$\text{S or C} \neq (MSB(A) = MSB(B));$$

Full adder

- $S_i = x_i \oplus y_i \oplus c_i$ $\quad T_D = 1$ (or 2);
- $C_{i+1} = x_i \cdot y_i + c_i \cdot (x_i + y_i)$ $\quad T_D = 2$;

Assume 3-i/p XOR gate for S_i and SOP form for C_{i+1}

Ripple-Carry Adder



$$O = C_n \oplus C_{n-1};$$

$$O = x_{n-1} y_{n-1} \overline{S}_{n-1} + \overline{x}_{n-1} \overline{y}_{n-1} S_{n-1}$$

$T_{D/C}$ =

$T_{D/S}$ =

$T_{D/O}$ =

Subtraction

- Subtract operation, X-Y, is performed by adding the complement of Y to X

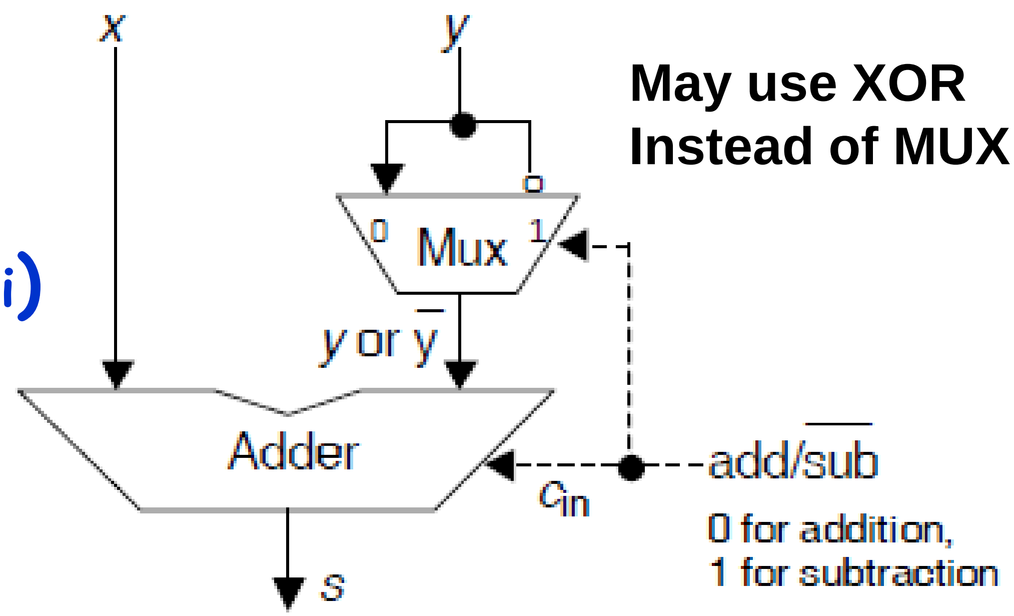
- In the two's complement system -

  $X - Y = X + (\bar{Y} + 1)$

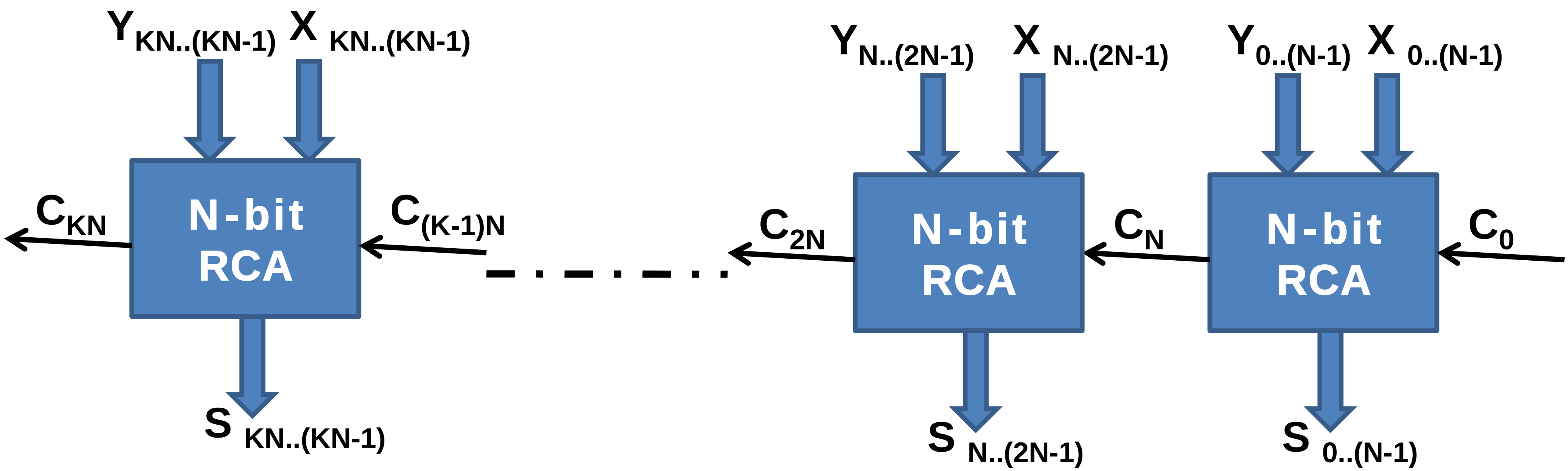
- This still requires only a single adder operation, since 1 is added through the forced carry input to the binary adder

Use EX_OR to produce complement (use one I/P bit as flag):
- Show how ??
- draw Ckt.

- $s_i = x_i \oplus y_i \oplus c_i$
- $c_{i+1} = x_i \cdot y_i + c_i \cdot (x_i + y_i)$



May use XOR
Instead of MUX



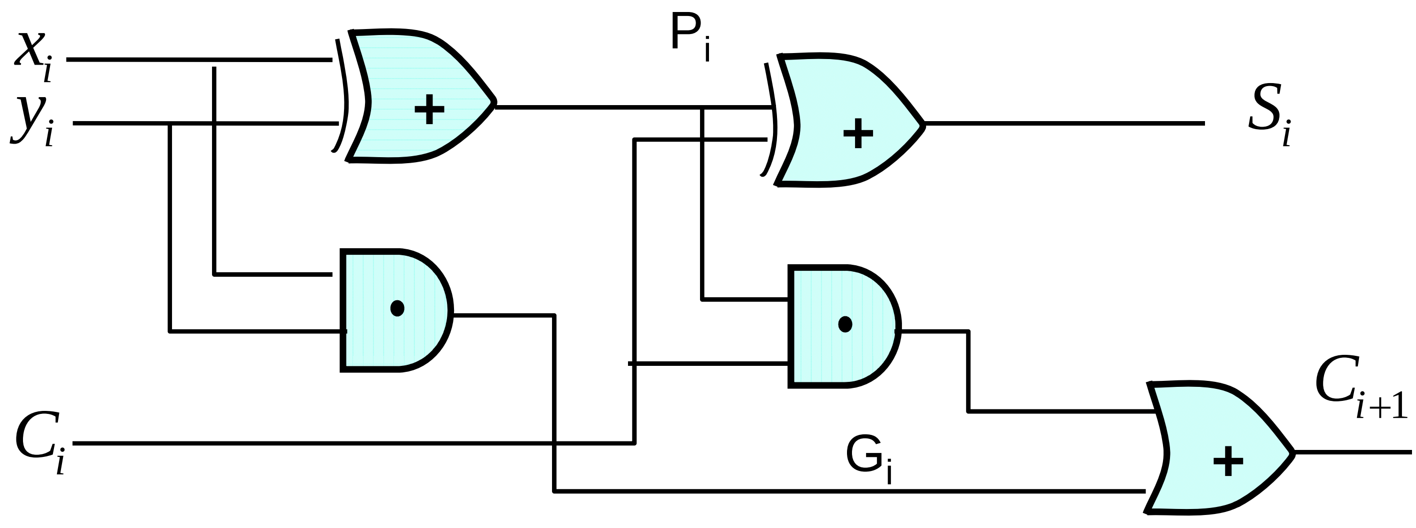
Cascade of K N-bit RCAs also possible – but delay is large

Look-Ahead Adder

Let $P_i = x_i \oplus y_i, G_i = x_i y_i;$

$S_i = P_i \oplus C_i$

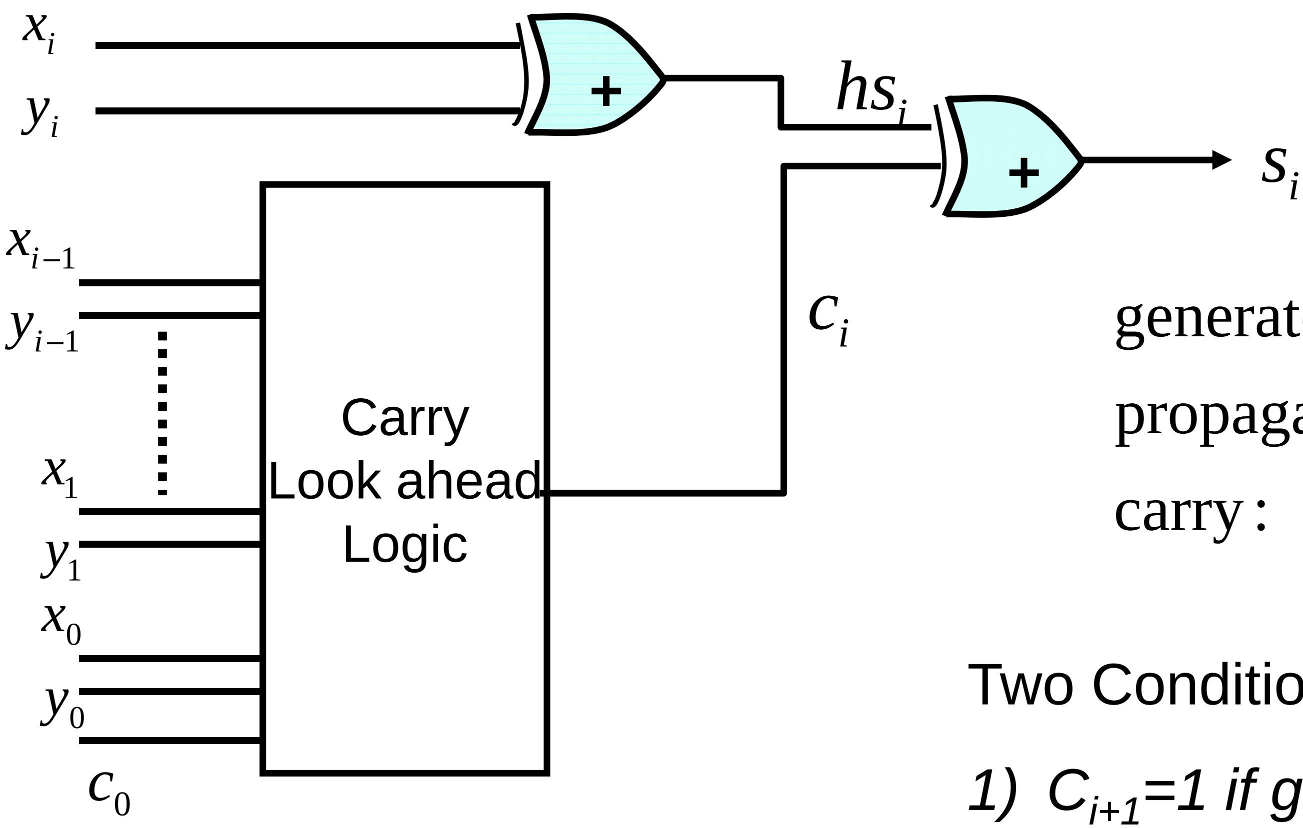
$c_{i+1} = G_i + P_i C_i$



***G_i* and *P_i* are termed:**

Carry Generate and Carry Propagate

Examples: 74283 - a 4-bit binary full adder with fast carry



generate: $g_i = x_i y_i$

propagate: $p_i = x_i + y_i$

carry: $c_{i+1} = g_i + p_i c_i$

Two Conditions:

1) *C_{i+1}=1 if g_i=1*

2) *C_{i+1}=1 if C_i=1 and p_i=1*

$$C_1 = G_0 + P_0 C_0$$

$$C_2 = G_1 + P_1 G_0 + P_1 P_0 C_0$$

74283 – contd.

$$hs_i = x_i \oplus y_i = (x_i + y_i)(\overline{x_i . y_i}) = p_i \bar{g}_i$$

$$c_{i+1} = g_i (p_i) + p_i c_i$$ i.e. when g_i=1, p_i =1

$$= p_i (g_i + c_i)$$

$$c_1 = p_0 (g_0 + c_0)$$

$$c_2 = p_1 (g_1 + c_1) = p_1 (g_1 + p_0 (g_0 + c_0))$$

$$= p_1 (g_1 + p_0)(g_1 + g_0 + c_0)$$

$$c_3 = p_2 (g_2 + c_2)$$

$$= p_2 (g_2 + p_1)(g_2 + g_1 + p_0)(g_2 + g_1 + g_0 + p_0)$$

$$c_4 = p_3 (g_3 + p_2)(g_3 + g_2 + p_1)$$

$$(g_3 + g_2 + g_1 + p_0)(g_3 + g_2 + g_1 + g_0 + c_0)$$

Carry-Look-Ahead - FAST Adders

- $G_i = x_i y_i$ - generated carry ;
- $P_i = x_i + y_i$ - propagated carry
- $c_{i+1} = x_i y_i + c_i (x_i + y_i) = G_i + c_i P_i$

- Substituting

$$c_i = G_{i-1} + c_{i-1} P_{i-1}; \Rightarrow c_{i+1} = G_i + G_{i-1} P_i + c_{i-1} P_{i-1} P_i$$

- Further substitutions -

$$\begin{aligned} c_{i+1} = & \quad G_i + G_{i-1} P_i + G_{i-2} P_{i-1} P_i + c_{i-2} P_{i-2} P_{i-1} P_i \quad = \quad \cdots \\ = & \quad G_i + G_{i-1} P_i + G_{i-2} P_{i-1} P_i + \cdots + c_0 P_0 P_1 \cdots P_i . \end{aligned}$$

- All carries can be calculated in parallel, using:
  $x_{n-1}, x_{n-2}, \ldots, x_0$, $y_{n-1}, y_{n-2}, \ldots, y_0$, and forced carry c_0

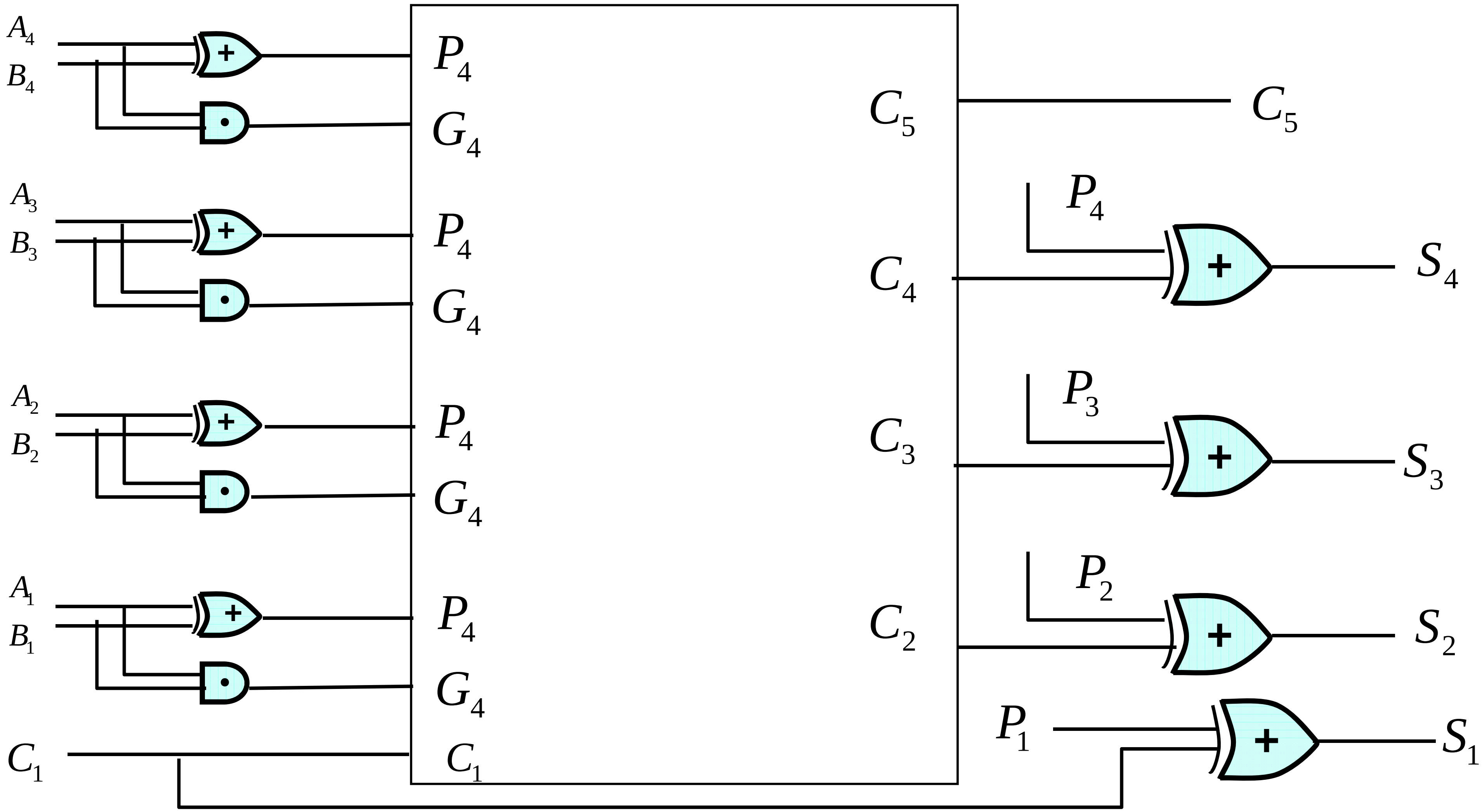
- **Method called: "Carry Look Ahead or Propagation" for Fast Adder design**

$$C_2 = G_1 + P_1C_1$$

$$C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1C_1$$

$$C_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1C_1$$

Compare overall delay, w.r.t. previous circuits

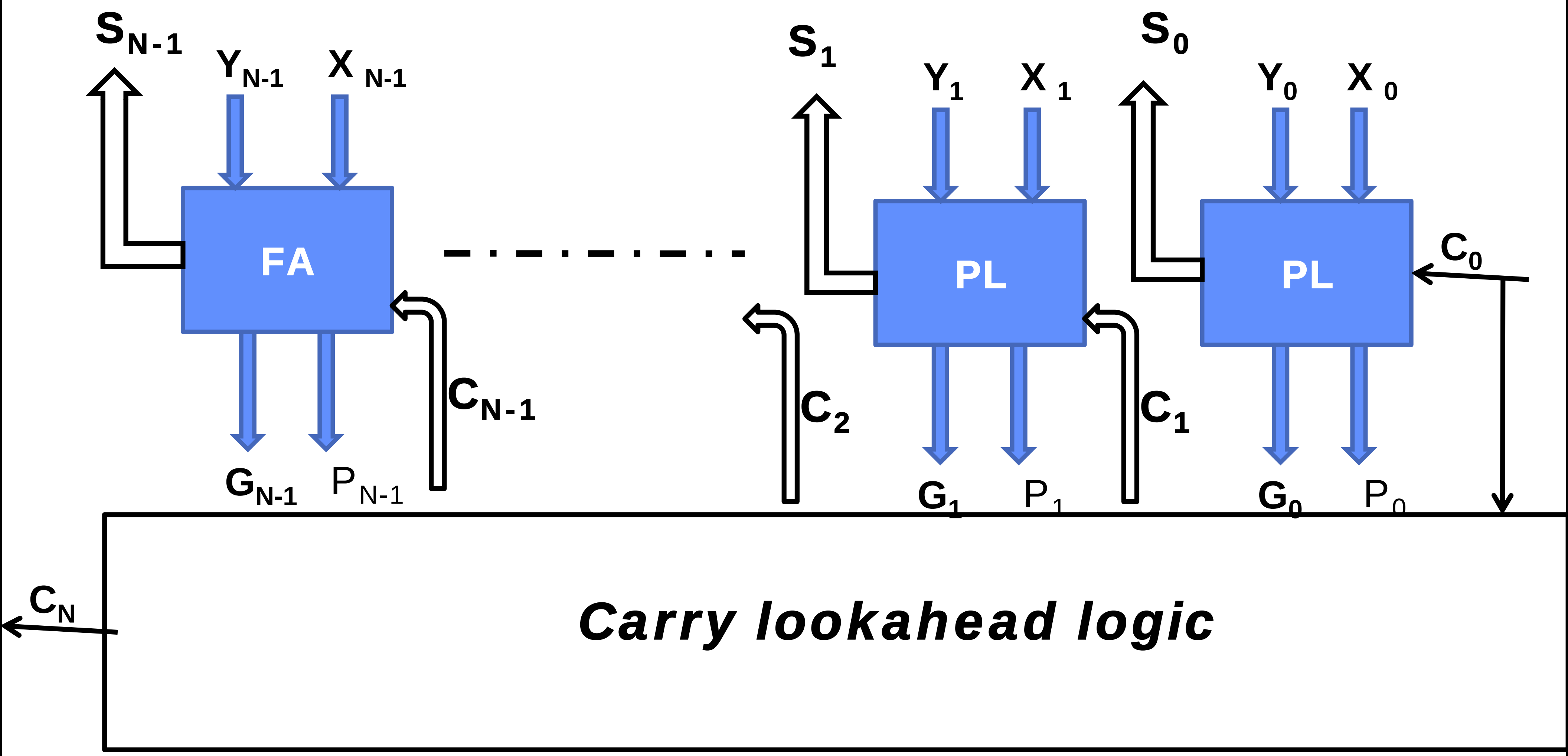


Example - 4-bit Adder

$$
\begin{aligned}
c_1 &= G_0 + c_0 P_0, \\
c_2 &= G_1 + G_0 P_1 + c_0 P_0 P_1, \\
c_3 &= G_2 + G_1 P_2 + G_0 P_1 P_2 + c_0 P_0 P_1 P_2, \\
c_4 &= G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3 + c_0 P_0 P_1 P_2 P_3
\end{aligned}
$$

- **Draw Ckt.**
- How many gates ?
- Delay: For C_i's: $T_D = 3$; $S_i = 3 + 2 = 5$.

For RCA: $T_{D/RCA} = 8$;



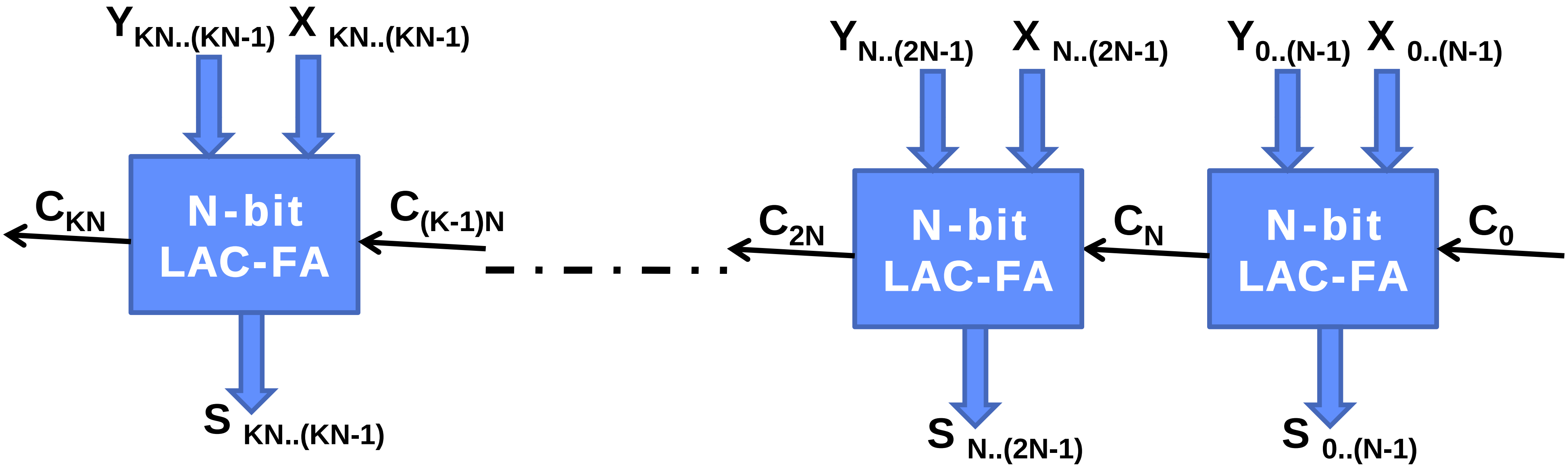
Carry lookahead Fast Adder

Delay of Carry-Look-Ahead Adders

- Δ_G - delay of a single gate
- At each stage
  - Delay of Δ_G for generating all P_i and G_i
  - Delay of $2\Delta_G$ for generating all c_i (two-level gate implementation)
  - Delay of $2\Delta_G$ for generating sum digits s_i in parallel (two-level gate implementation)
  - Total delay of $5\Delta_G$ regardless of n - number of bits in each operand
- Large n (=32) - large number of gates with large fan-in
  - Fan-in - number of gate inputs, $n+1$ here
- Span of look-ahead must be reduced at expense of speed

Reducing Span

- n stages divided into groups - separate carry-look-ahead in each group
- Groups interconnected by ripple-carry method
  - Equal-sized groups - modularity - one circuit designed
  - Commonly - group size 4 selected - $n/4$ groups
  - 4 is factor of most word sizes
  - Technology-dependent constraints (number of input/output pins)
  - ICs adding two 4 digits sequences with carry-look-ahead exist
    - ΔG needed to generate all P_i and G_i
    - $2\Delta G$ needed to propagate a carry through a group once the P_i, G_i, c_0 are available
    - $(n/4)2\Delta G$ needed to propagate carry through all groups
    - $2\Delta G$ needed to generate sum outputs
  - Total - $(2(n/4)+3)\Delta G = ((n/2)+3)\Delta G$ - a reduction of almost 75% compared to $2n\Delta G$ in a ripple-carry adder



Cascade of K N-bit LAC-FAs also possible

Comparison of the Delay of different systems

Adder	C4	C8	C12	C16	S15	C28	C32	S31
RCA	8	16	24	32	31	56	64	63
LAC-FA	3	5	7	9	10	15	17	18
??								

Speed-up for higher level carry bits

$$\begin{aligned} c_1 &= G_0 + c_0 P_0, \\ c_2 &= G_1 + G_0 P_1 + c_0 P_0 P_1, \\ c_3 &= G_2 + G_1 P_2 + G_0 P_1 P_2 + c_0 P_0 P_1 P_2, \\ c_4 &= G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3 + c_0 P_0 P_1 P_2 P_3 \end{aligned}$$

Let:

$$P_o^* = P_3 P_2 P_1 P_0; \; G_o^* = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0;$$

$$\textit{Then,} \quad C_4 = G_o^* + P_o^* C_0$$

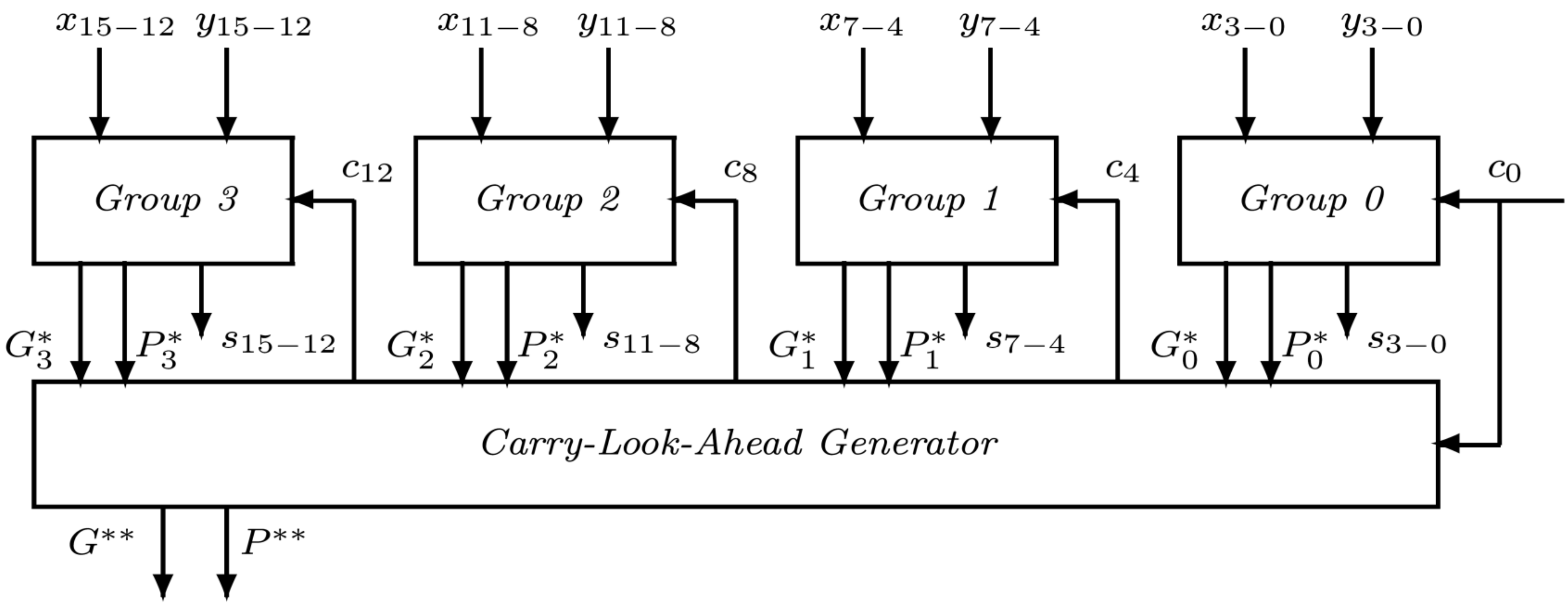
If:

$$P_1^* = P_7 P_6 P_5 P_4; \; G_1^* = G_7 + P_7 G_6 + P_7 P_6 G_5 + P_7 P_6 P_5 G_4;$$

$$\textit{Then,} \quad C_8 = G_1^* + P_1^* C_4 = G_1^* + P_1^* G_0^* + P_1^* P_0^* C_0$$

$$\textit{Similarly,} \quad C_{12} = G_2^* + P_2^* C_8; \qquad C_{16} = G_3^* + P_3^* C_{12}$$

16-bit 2-level Carry-look-ahead Adder



- **n=16 - 4 groups**
- **Outputs:** $G_o^*, G_1^*, G_2^*, G_3^*, P_0^*, P_1^*, P_2^*, P_3^*$;
- **Inputs to a carry-look-ahead generator with outputs c_4, c_8, c_{12}**

$$
\begin{aligned}
c_4 &= G_0^* + c_0 P_0^*, \\
c_8 &= G_1^* + G_0^* P_1^* + c_0 P_0^* P_1^*, \\
c_{12} &= G_2^* + G_1^* P_2^* + G_0^* P_1^* P_2^* + c_0 P_0^* P_1^* P_2^*
\end{aligned}
$$

$$C_4 = G_o^* + P_o^* C_0; \quad C_8 = G_1^* + P_1^* C_4 = G_1^* + P_1^* G_0^* + P_1^* P_0^* C_4$$

Similarly, $C_{12} = G_2^* + P_2^* C_8$

$$= G_2^* + P_2^* G_1^* + P_2^* P_1^* G_0^* + P_2^* P_1^* P_0^* C_0$$

$$C_{16} = G_3^* + P_3^* C_{12}$$

$$= G_3^* + P_3^* G_2^* + P_3^* P_2^* G_1^* + P_3^* P_2^* P_1^* G_o^* + P_3^* P_2^* P_1^* P_0^* C_0$$

Expression/Output (4-stage block)	Implementation Delay	Total Delay
$G_i;\ P_i$	1	1
$G_i^*;\ P_i^*$	2; 1	3; 2
$C_{i = 4, 8, 12, 16}$	2	**5**
S_i	2 + 1 = 3	8

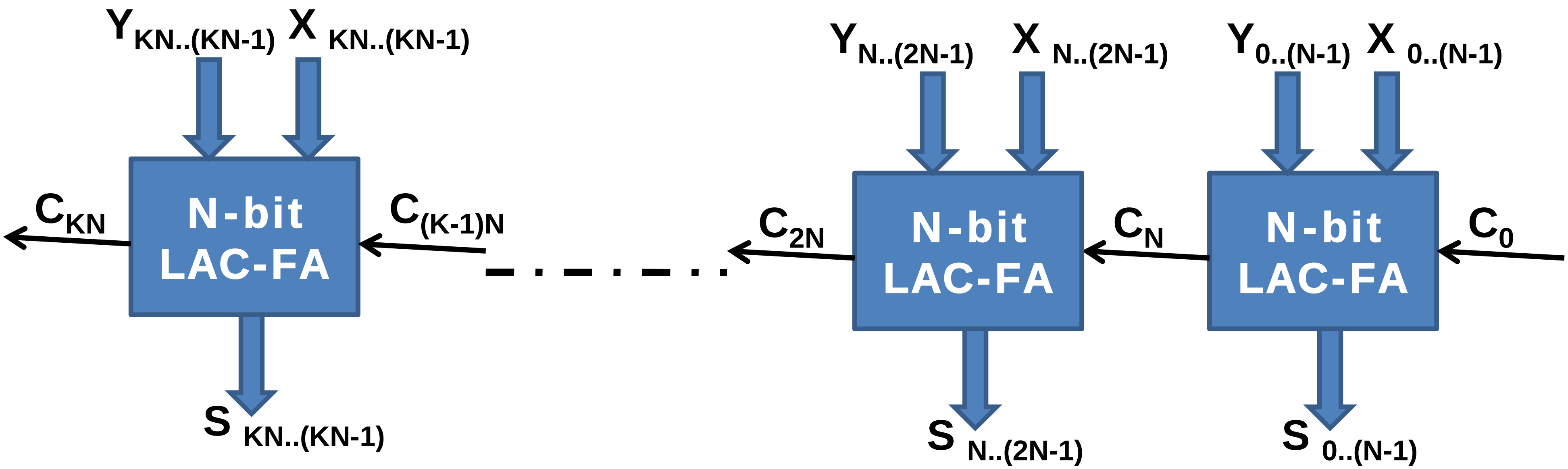
$C_{12} \rightarrow C_{15}:$

$5 + 2 = 7;$

$S_{15}:$

$7 + 1 = 8$

For LAC-L1, delays are: C_{16} -> 9, S_{15} -> 10



Cascade of K N(=16)-bit HLG&P-CAs also possible

K = 2;

Bit	Delay
(K = 1) C_{16}	5
(K = 2) C_{28}, C_{32}	5 + 2 = **7**
(K = 2) C_{28} --> C_{31}	7 + 2 = 9
(K = 2) S_{31}	9 + 1 = **10**

K = 4;

Bit	Delay
(K = 2) C_{32}	7
(K = 3) C_{44}, C_{48}	7 + 2 = 9
(K = 4) C_{60}, C_{64}	9 + 2 = **11**
(K = 4) C_{60} --> C_{63}	11 + 2 =13
(K = 4) S_{63}	13+ 1 = **14**

Comparison of the Delay of different systems

Adder	C4	C8	C16	S15	C32	S31	C64	S63
RCA	8	16	32	31	64	63	128	127
LAC-FA	3	5	9	10	17	18	33	34
HLG&P-CA	-	-	5	8	7	10	11	14
??								

L2 – HLG&P CA

$$C_{16} = G_3^* + P_3^* C_{12}$$

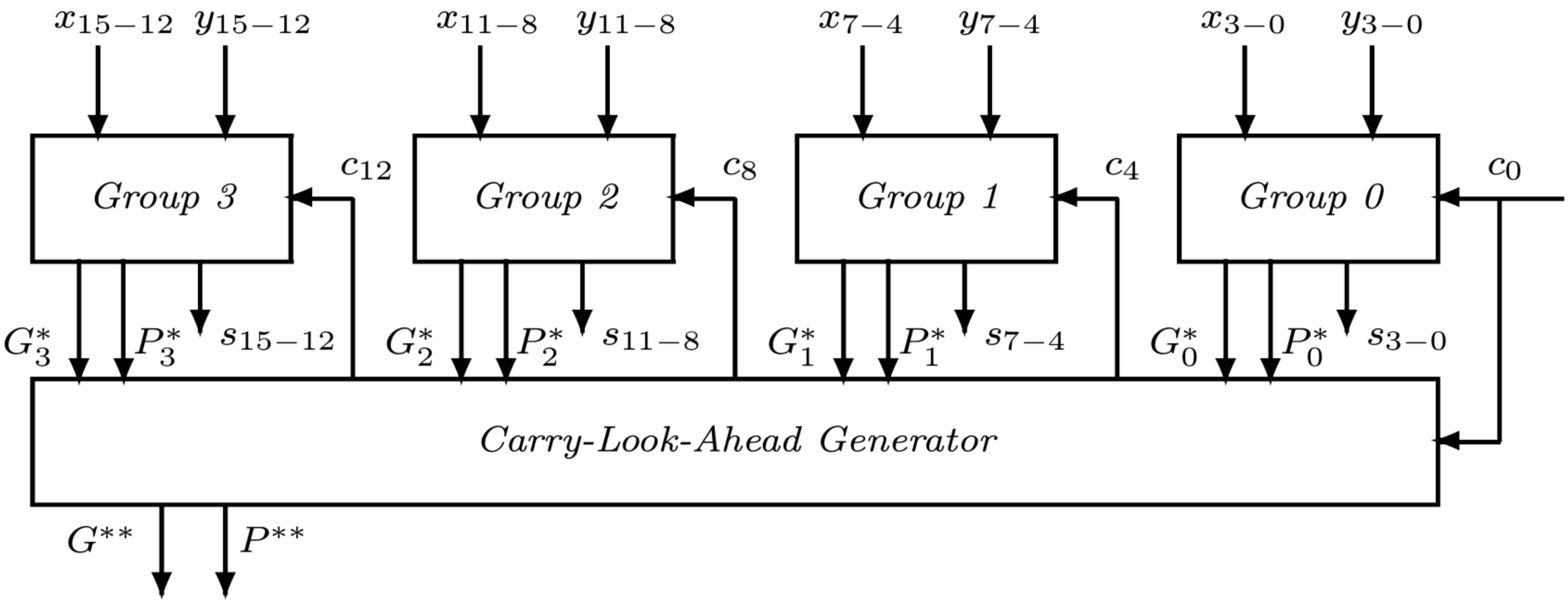
$$= G_3^* + P_3^* G_2^* + P_3^* P_2^* G_1^* + P_3^* P_2^* P_1^* G_o^* + P_3^* P_2^* P_1^* P_0^* C_0$$

$$= G_o^{**} + P_0^{**} C_0;$$

where,

$$G_o^{**} = G_3^* + P_3^* G_2^* + P_3^* P_2^* G_1^* + P_3^* P_2^* P_1^* G_o^*;$$

$$P_0^{**} = P_3^* P_2^* P_1^* P_0^*$$



$$C_{16} = G_3^* + P_3^* C_{12}$$

$$= G_3^* + P_3^* G_2^* + P_3^* P_2^* G_1^* + P_3^* P_2^* P_1^* G_o^* + P_3^* P_2^* P_1^* P_0^* C_0$$

$$= G_o^{**} + P_0^{**} C_0;$$

where ,

$$G_o^{**} = G_3^* + P_3^* G_2^* + P_3^* P_2^* G_1^* + P_3^* P_2^* P_1^* G_o^*;$$

$$P_0^{**} = P_3^* P_2^* P_1^* P_0^*$$

Delay for C_{16}: 5
Delay for S_{63}: 12
Delay for C_{64}: 7

Comparison of the Delay of different FAST ADDER systems

Adder	C4	C8	C16	S15	C32	S31	C64	S63
RCA	8	16	32	31	64	63	128	127
LAC-FA	3	5	9	10	17	18	33	34
HLG&P-CA	-	-	5	8	7	10	11	14
2nd LEVEL HLG&P-CA			5		7		7	12

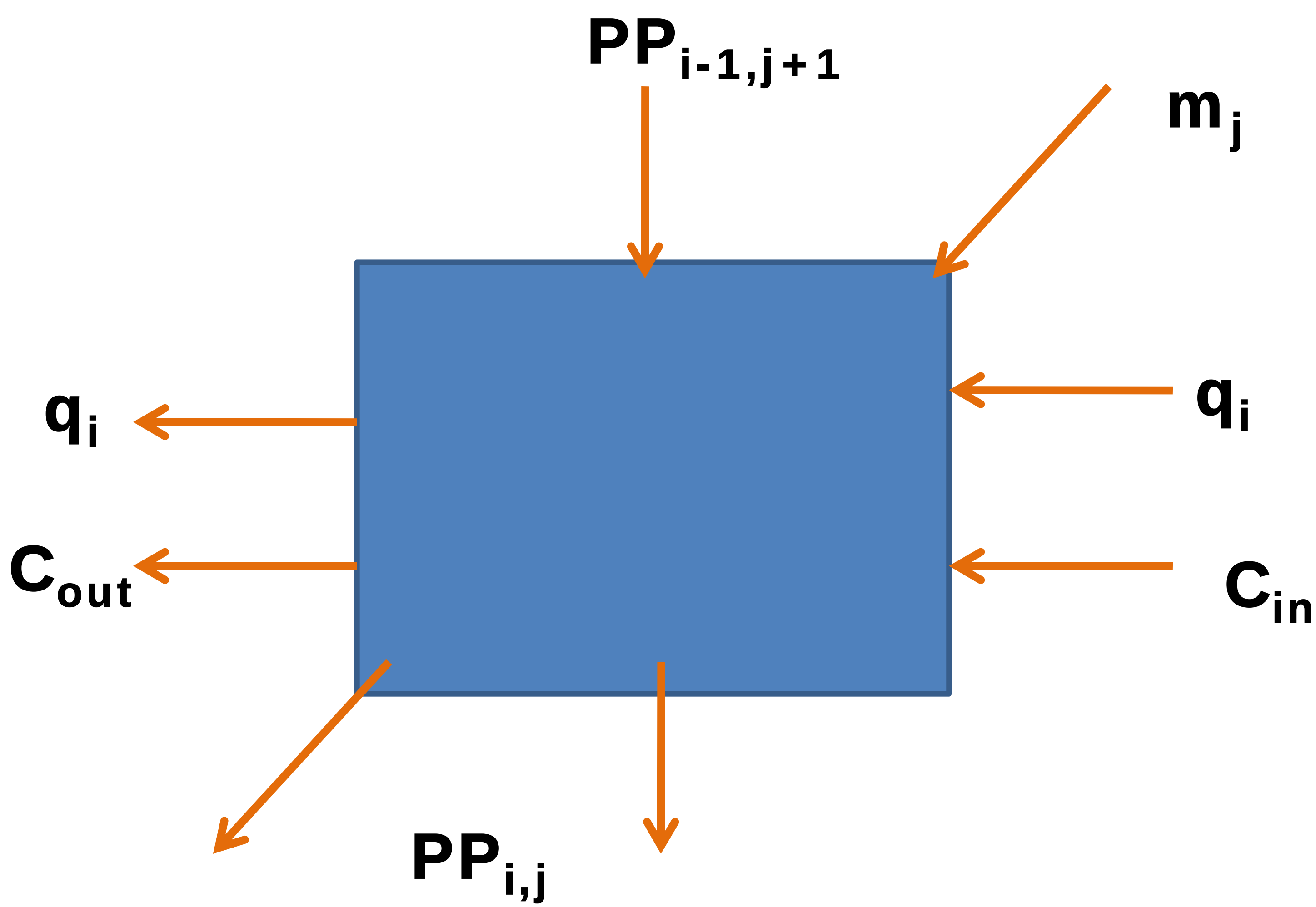
Other Advanced Adders:

- Pipelined
- Manchester Adder
- Carry-Skip / Carry Select
- Parallel Prefix
- Carry-save adders – Wallace tree
-

MULTIPLIER UNIT

```
      1 1 0 1
    x 1 0 1 1
-----------------
      1 1 0 1
    1 1 0 1
  0 0 0 0
1 1 0 1
_________________
1 0 0 1 1 1 1 1
```

- **Three** types of high-speed multipliers:
- **Sequential multiplier** - generates partial products sequentially and adds each newly generated product to previously accumulated partial product
- **Parallel multiplier** - generates partial products in parallel, accumulates using a fast multi-operand adder
- **Array multiplier** - array of identical cells generating new partial products; accumulating them simultaneously



Typical Cell of an Array Multiplier

For Sequential circuit binary Multiplier:

Need - ADDER and Shift-right Registrar (SR) modules.

The control circuit requires a clock;

Multiplixer to decide:

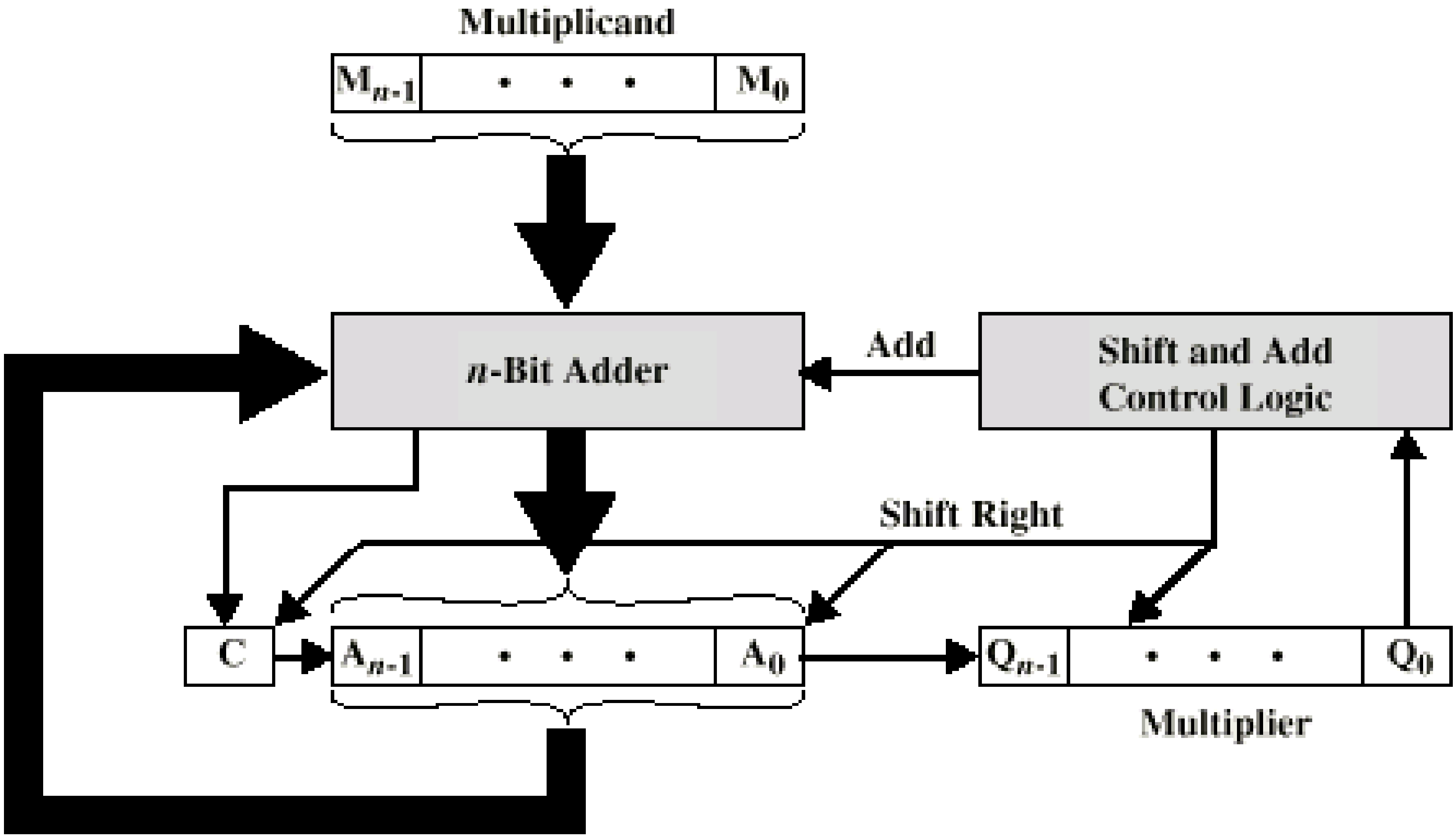
- Add zero (only SR)

Or

- Add and shift (SR).

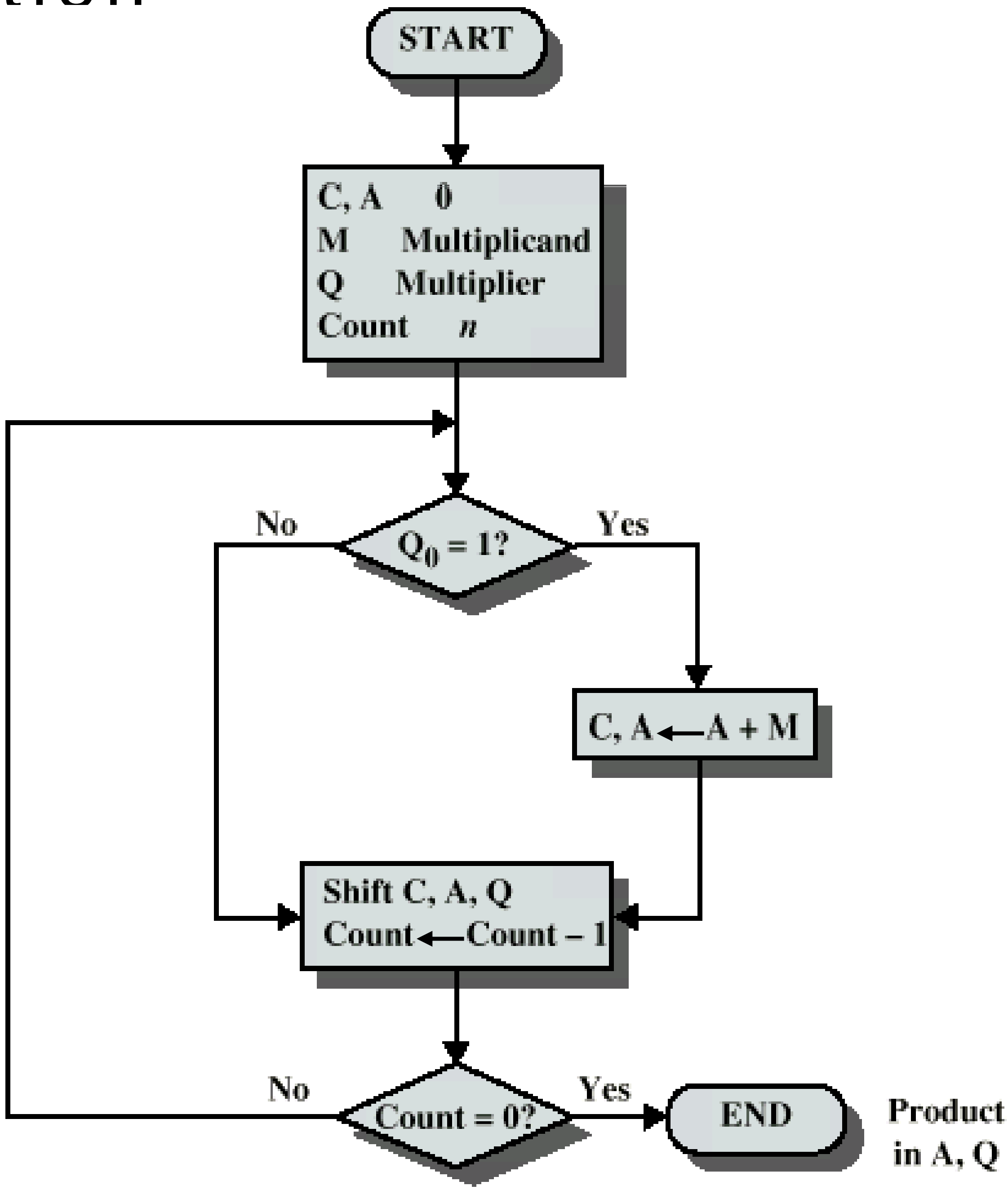
Result to be held in 2-K bit SR.

Unsigned Binary Multiplication



Block Diagram

Flowchart for Unsigned Binary Multiplication



Execution of Example

C	A	Q	M		
0	0000	1101	1011	Initial Values	
0	1011	1101	1011	Add	First Cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	Second Cycle
0	1101	1111	1011	Add	Third Cycle
0	0110	1111	1011	Shift	
1	0001	1111	1011	Add	Fourth Cycle
0	1000	1111	1011	Shift	

Booth's Algorithm for unsigned multiplication

Two basic principles:

- Strings of 0's require no addition – only shift
- String of 1's may be given special trreatment:

001110 (+14) --> 010000 - 000010 (16 - 2);

1's from K-bit posn. to M-bit posn:
Treat as: $2^{K+1} - 2^{M}$;
In the above example K = 3, M = 1;

Thus M (Multiplicand) X 14 = M X 2^4 = M X 2^1 ;

Obtain result by:
M << 4 - M << 1 // *view as C-code.*

Take a example: multiplier = 30; Multiplicand = 45

(30) --> 32 – 2 =>

0011110 → 0100000 (32)
1111110 (-2)

Change Multiplier to:

0 (+1) 0 0 0 0 0
0 0 0 0 0 (-1) 0

							0	1	0	1	1	0	1
							0	+1	0	0	0	-1	0
					-	-	-	-	-	-	-	-	-
							0	0	0	0	0	0	0
	1	1	1	1	1	1	0	1	0	0	1	1	
					0	0	0	0	0	0	0		
				0	0	0	0	0	0	0			
			0	0	0	0	0	0	0				
		0	1	0	1	1	0	1					
	0	0	0	0	0	0	0						
-	-	-	-	-	-	-	-	-	-	-	-	-	-
	0	0	1	0	1	0	1	0	0	0	1	1	0

Good multiplier for coding:

0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 0 .....

Worst case multiplier:

0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 ....

Mult	0	0	1	0	1	1	0	0	1	1	1	0	1	0	1	1	0	0
Coded Mult.	0	+1	-1	+1	0	-1	0	+1	0	0	-1	+1	-1	+1	0	-1	0	0

Three rules for Booth's algm. implementation with operation on the multiplicand:

- For the LSB = 1 in a string of 1's in a multiplier:
  Subtract the multiplicand from the partial product;

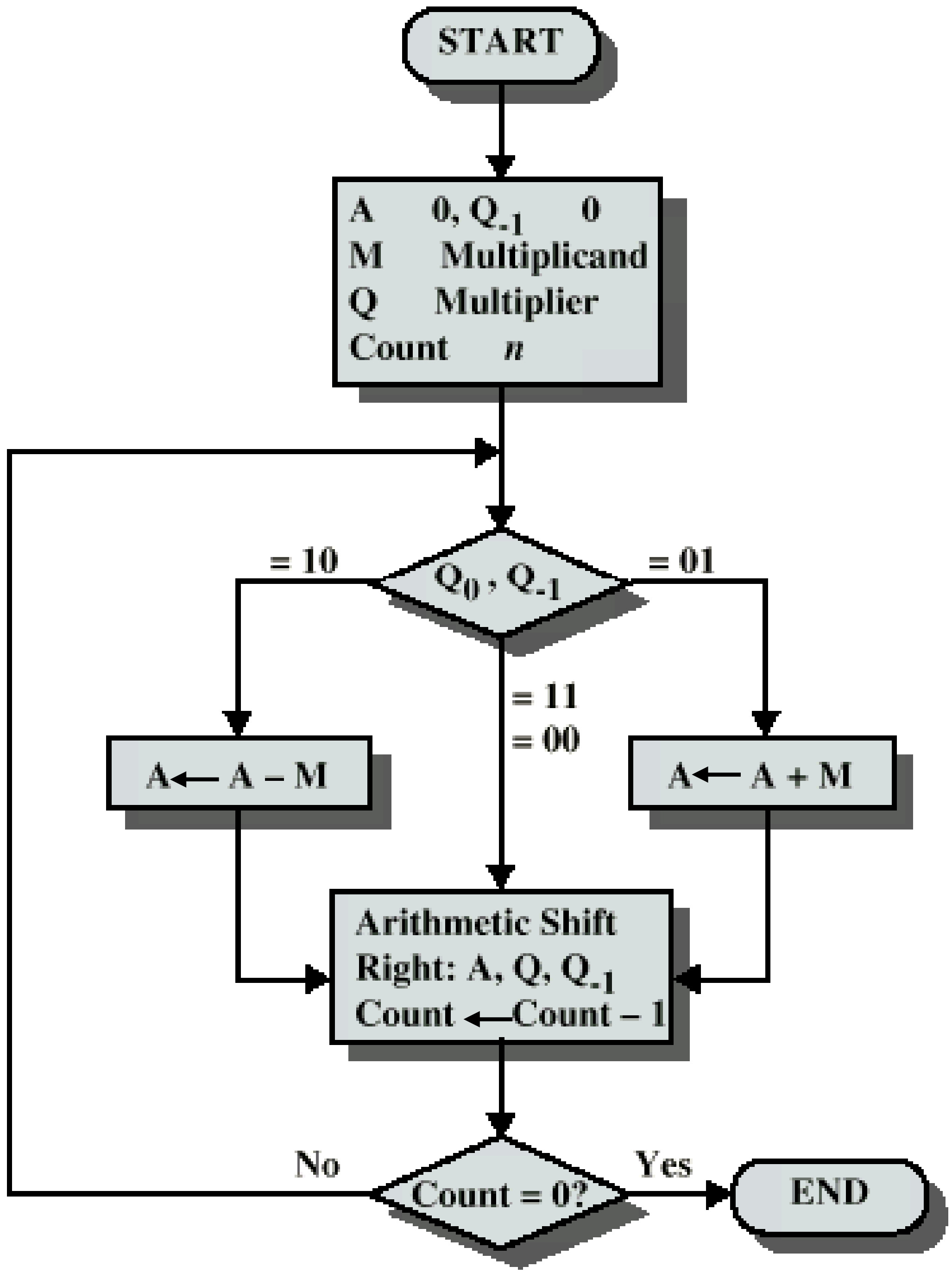
- For the first bit-0 (prior to a 1) in a string of 0's, the multiplicand is added to the partial product.

- For any pair of identical bit-pair in the mutliplier, the partial product is unchanged.

0 0 1 1 1 1 0 →
0 +1 0 0 0 -1 0

Multiplier Bits			
Bit i	Bit (i-1)		Opn. / Bit Pattern
0	0	0 x M	Shift only; String of Zeros
0	1	+1 x M	Add and Shift; End of a String of Ones
1	0	-1 x M	Subtract and Shift; Beginning of a String of Ones
1	1	0 x M	Shift only; String of Ones

Booth’s Algorithm



Example of Booth's Algorithm

A	Q	Q_{-1}	M		
0000	0011	0	0111	Initial Values	
1001	0011	0	0111	A A - M	First Cycle
1100	1001	1	0111	Shift	
1110	0100	1	0111	Shift	Second Cycle
0101	0100	1	0111	A A + M	Third Cycle
0010	1010	0	0111	Shift	
0001	0101	0	0111	Shift	Fourth Cycle

Execution of Example – Booth Multiplier

+13 → 01101

-6 → 11010 → 0 -1 +1 -1 0

							0	1	1	0	1
							0	-1	+1	-1	0
			-	-	-	-	-	-	-	-	-
					0	0	0	0	0	0	0
	1	1	1	1	1	1	0	0	1	1	
			0	0	0	1	1	0	1		
	1	1	1	1	0	0	1	1			
	0	0	0	0	0	0	0				
-	-	-	-	-	-	-	-	-	-	-	-
1	1	1	1	1	0	1	1	0	0	1	0

Fast Multiplication done by:

- Bit-pair recoding
- Carry-save Addition of Summands
- \+ Fast Look ahead Carry (with both above)
- Pipelined and Booth Array Tree
- etc.

BIT-PAIR RECODING OF MULTIPLIERS

Observe this:

-6 → 11010 → 0 -1 +1 -1 0

Consider the pair of (+1, -1):

==> (+1, -1)*M = 2xM – M = M
==> (0, +1) *M;

Thus: (+1, -1) == (0, +1);
which is also independent of the bit position "i".

Booth Pair	Equiv. to	Recoded pair
(+1, 0)	→	(0, +2)
(-1, +1)	→	(0, -1)
(0, 0)	→	(0, 0)
(0, 1)	→	(0, 1)
(+1, 1)	→	--
(-1, 0)	→	(0, -2)

BIT-PAIR RECODING OF MULTIPLIERS

Multiplier Bits		
Bit i	Bit (i-1)	
0	0	0 x M
0	1	+1 x M
1	0	-1 x M
1	1	0 x M

Booth Pair	Equiv. to	Recoded pair
(+1, 0)	→	(0, +2)
(-1, +1)	→	(0, -1)
(0, 0)	→	(0, 0)
(0, 1)	→	(0, 1)
(+1, 1)	→	--
(+1, -1)	→	(0, +1)
(-1, 0)	→	(0, -2)

-6 → 1 1 1 0 1 0 →

0 0 -1 +1 -1 0

0 0 -1 0 -2

Execution of Example – Booth's Recoded Multiplier

+13 → 01101

-6 → 11010 → 0 -1 +1 -1 0 → 0 0 -1 0 -2

+13 → 01101; +26 → 011010; -26 → 100110;

-13 → 10011;

							0	1	1	0	1
							0	0	-1	0	-2
			-	-	-	-	-	-	-	-	-
	1	1	1	1	1	1	0	0	1	1	0
	0	0	0	0	0	0	0	0	0	0	
	1	1	1	1	1	0	0	1	1		
	0	0	0	0	0	0	0	0			
	0	0	0	0	0	0	0				
-	-	-	-	-	-	-	-	-	-	-	-
1	1	1	1	1	0	1	1	0	0	1	0

Unsigned Binary Division

+13 → 1101 ← DIVISOR;

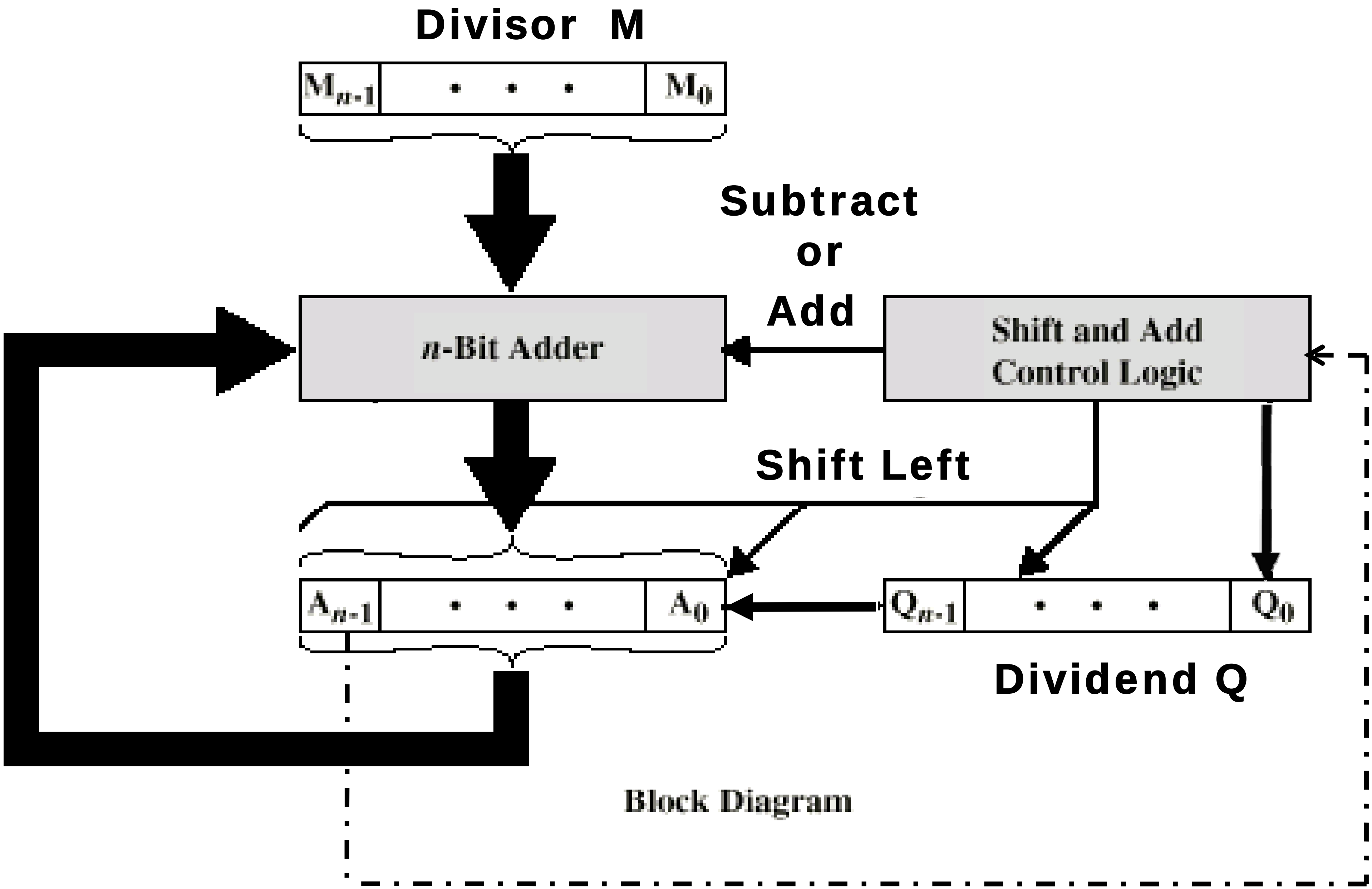
274 → 100010010 ← DIVIDEND

```
       03
63 ) 243
   − 189
   -----
      54
```

```
       21
13 ) 274
   −  26
   -----
      14
    - 13
   -----
      01
```

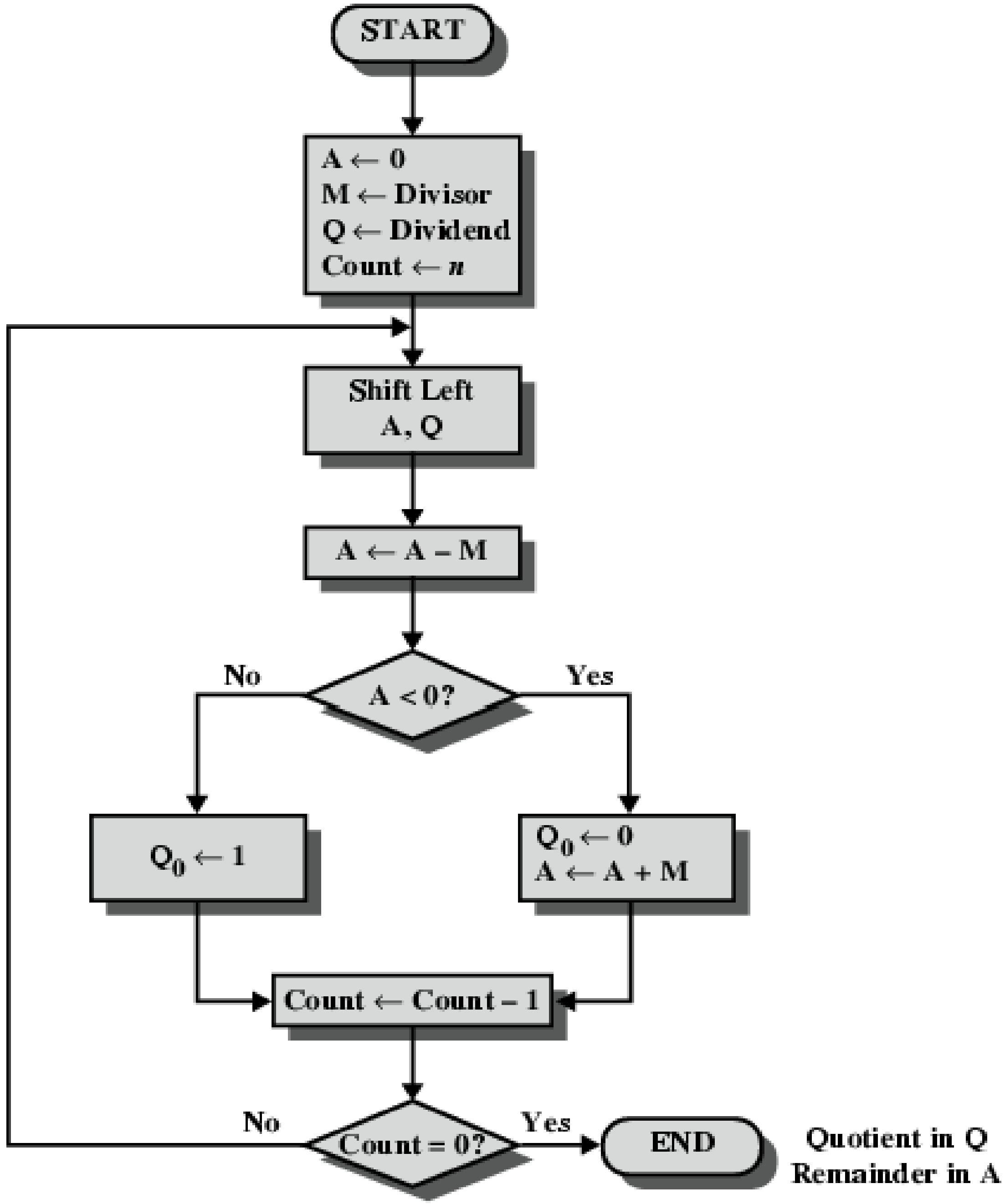
									1	0	1	0	1
1	1	0	1	√	1	0	0	0	1	0	0	1	0
					-	1	1	0	1				
					_	_	_	_	_				
							1	0	0	0	0		
							-	1	1	0	1		
							_	_	_	_	_		
										1	1	1	0
									-	1	1	0	1
									_	_	_	_	_
													1

Unsigned Binary Division



Block Diagram

Flowchart for Unsigned Binary Division



ALGO. for RESTORING DIVISION

Load Divisor in Reg. M;
Load Dividend in Reg. Q;
Set Reg. A = 0;

Repeat n times:

- **Shift (left) A & Q (one bit posn.)**
- **A = A – M;**
- **If sgn(A) == 1 (-ve A)**
  - **Set Q_0 = 0;**
  - **A = A + M;**

  Else
  - **Set Q_0 = 1;**

End

ANS:

- Quotient is in Reg. Q;

- Rem. is in Reg. A.

Dividend, Q = 1000;

Divisor, M = 11.

- M = 11101

Result, in decimal ??

Quotient = 02 ;

Rem = 02.

OPN.		REG. A					REG. Q			
INIT.		0	0	0	0	0	1	0	0	0
I	S. L.	0	0	0	0	1	0	0	0	
	SUB	1	1	1	1	0				
	Rstr	0	0	0	0	1	0	0	0	0
II	S. L.	0	0	0	1	0	0	0	0	
	SUB	1	1	1	1	1				
	Rstr	0	0	0	1	0	0	0	0	0
III	S. L.	0	0	1	0	0	0	0	0	
	SUB	0	0	0	0	1				
	Set-Q_0	0	0	0	0	1	0	0	0	1
IV	S. L.	0	0	0	1	0	0	0	1	
	SUB	1	1	1	1	1				
	Rstr	0	0	0	1	0				0
Final Result		0	0	0	1	0	0	0	1	0
		REM					Q			

How to improve the Algo. ??

If A is +ve,
Opns. :

- S.L. (A);
- Subtract M;

→ (2A – M)

Repeat n times:

- **Shift (left) A & Q (one bit posn.)**
- **A = A – M;**
- **If sgn(A) == 1 (-ve A)**
  - o **Set Q_0 = 0;**
  - o **A = A + M;**

  Else
  - o **Set Q_0 = 1;**

End

If A is -ve,
Opns. :

- A + M
- S.L. (A+M);
- Subtract M;

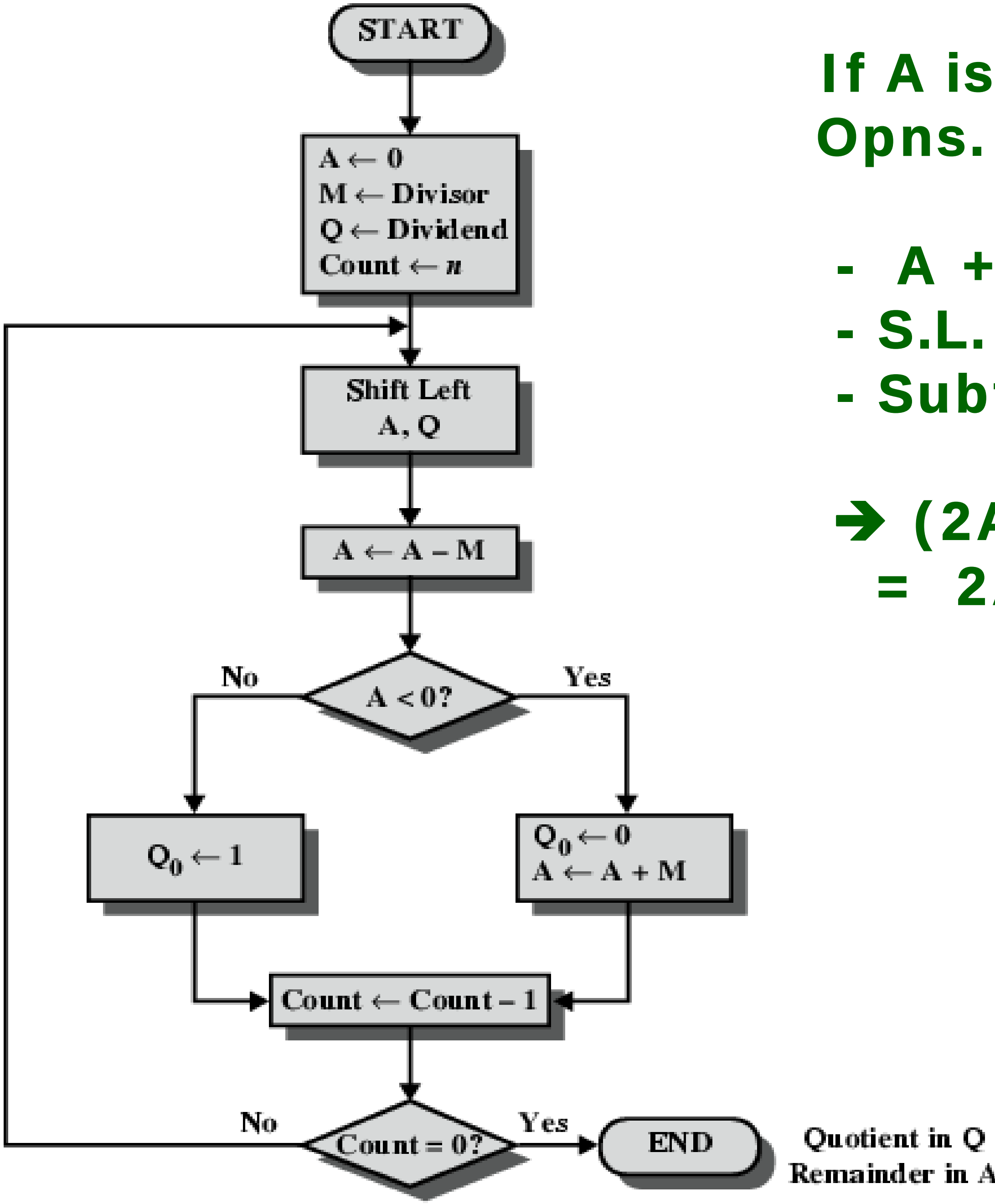
→ (2A + 2M) – M
= 2A + M

How to improve the Algo. ??

If A is +ve,
Opns. :

- **S.L. (A);**
- **Subtract M;**

➔ (2A – M)



If A is -ve,
Opns. :

- **A + M**
- **S.L. (A+M);**
- **Subtract M;**

➔ (2A + 2M) – M
= 2A + M

Step 1: Repeat n times:

1. **If sgn(A) == 0**
   - **- Shift (left) A & Q (1-bit posn.)**
   - **- A = A – M;**

   else
   - **- Shift (left) A & Q (1-bit posn.)**
   - **- A = A + M;**

2. **If sgn(A) == 0**
   - o **Set Q_0 = 1;**
   - o **else Q_0 = 0;**

End

Step 2: If sgn(A) == 1

A = A + M;

NON_RESTORING DIVISION

RESTORING DIVISION

Repeat n times:

- **Shift (left) A & Q (1 bit posn.)**
- **A = A – M;**
- **If sgn(A) == 1 (-ve A)**
  - o **Set Q_0 = 0;**
  - o **A = A + M;**

  Else
  - o **Set Q_0 = 1;**

End

Dividend, Q = 1000;

Divisor, M = 11.

- M = 11101

OPN.		REG. A					REG. Q			
INIT.		0	0	0	0	0	1	0	0	0
I	S. L.	0	0	0	0	1	0	0	0	
	SUB	1	1	1	1	0				
	Set Q_0	1	1	1	1	0	0	0	0	0
II	S. L.	1	1	1	0	0	0	0	0	
	ADD	1	1	1	1	1				
	Set Q_0	1	1	1	1	1	0	0	0	0
III	S. L.	1	1	1	1	0	0	0	0	
	ADD	0	0	0	0	1				
	Set-Q_0	0	0	0	0	1	0	0	0	1
IV	S. L.	0	0	0	1	0	0	0	1	
	SUB	1	1	1	1	1				
	Set-Q_0	1	1	1	1	1	0	0	1	0
	ADD	0	0	0	1	0	0	0	1	0
Final Result		0	0	0	1	0	0	0	1	0
		REM					Q			

Floating Point Numbers

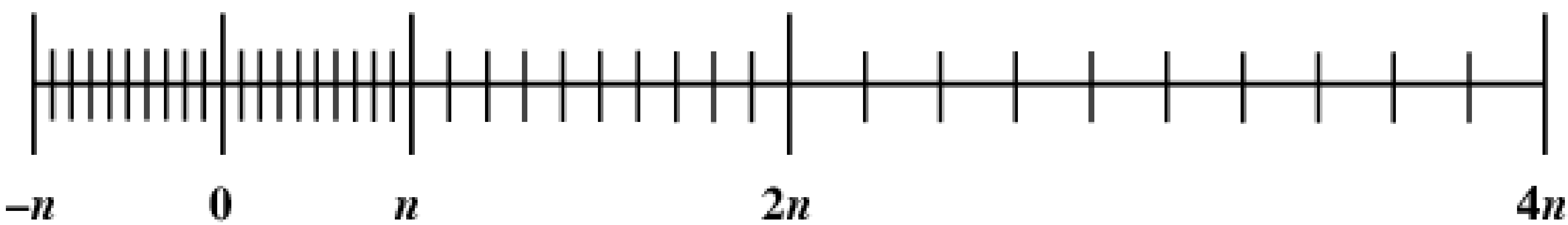
IEEE 32-bit single precision

S	E' (excess-127 Rep.)	M
0/1	8	23

IEEE 64-bit single precision

|E| = 11 **|M| = 52**

Density of Floating Point Numbers



SEM Field

- Common case - signed-magnitude fraction
- Floating-point format - sign bit S, e bits of exponent E, m bits of unsigned fraction M ($m+e+1=n$)

S	Exponent E	Unsigned Significand M

- Value of (S,E,M) : $F = (-1)^S \cdot M \cdot \beta^E$

  ($(-1)^0 = 1$; $(-1)^1 = -1$)

- Maximal value - $M_{max} = 1 - ulp$
- ulp -Unit in the last position - weight of the least-significant bit of the fractional significand
- Usually (not always) $ulp = 2^{-m}$

In IEEE 32-bit single precision;

E is a "signed exponent", E' is in excess-127 Representation

$$E' = E + 127;$$

The end values are reserved for special use:

$$0 \le E' \le 255$$

$$1 \le E' \le 254; \quad E = E' - 127;$$

$$-127 \le E \le 128;$$

$$-126 \le E \le 127;$$

Range,

for Exponent:

$$[2^{-126} \ldots . 2^{+127}] \Rightarrow 10^{\pm 38}$$

For Mantissa:

$$[2^{-23}] \Rightarrow 10^{-7}$$

In IEEE 64-bit single precision;

$$E' = E + 1023 \; ; \quad 1 \le E' \le 2046$$

$$-1022 \le E \le 1023 \; ;$$

Range,

for Exponent:

$$[2^{-1022} \ldots . 2^{+1023}] \Rightarrow 10^{\pm 308}$$

For Mantissa:

$$[2^{-53}] \Rightarrow 10^{-16}$$

Floating-Point Formats of Three Machines

	IBM/370	DEC/VAX	Cyber 70
Word length (double)	32 (64) bits	32 (64) bits	60 bits
Significand+{hidden bit}	24 (56) bits	$23 + 1$ $(55 + 1)$ bits	48 bits
Exponent	7 bits	8 bits	11 bits
Bias	64	128	1024
Base	16	2	2
Range of M	$\frac{1}{16} \leq M < 1$	$\frac{1}{2} \leq M < 1$	$1 \leq M < 2$
Representation of M	Signed-magnitude	Signed-magnitude	One's complement
Approximate range	$16^{63} \approx 7 \cdot 10^{75}$	$2^{127} \approx 1.9 \cdot 10^{38}$	$2^{1023} \approx 10^{307}$
Approximate resolution	$2^{-24} \approx 10^{-7}\,(10^{-17})$	$2^{-24} \approx 10^{-7}\,(10^{-17})$	$2^{-48} \approx 10^{-14}$

Binary Normalized Mantissa (IEEE):

1.<xxxx ..23 bits....xxxx>

0	10001000	.0010110 ....

Numerical Value (unnormalized): = $+0.0010110.... \times 2^{9}$

0	10000101	.0110 ....

Numerical Value (Normalized): = $+1.0110.... \times 2^{6}$

1	00101000	.001010 ....

Numerical Value (Normalized): = $-1.001010.... \times 2^{-87}$

Special Values :

E' = 0 AND M = 0 → ZERO VALUE;

E' = 255 AND M = 0 → INFINITY;

Both ± 0 and $\pm\infty$ are possible

E'= 0 and M <> 0 → denormal numbers; $\pm 0.M \times 2^{-126}$

E'= 255 and M <> 0 → NaN; $0/0$; $\sqrt{-1}$

Exceptions :

Underflow, Overflow, Divide by zero and

Inexact → Result that requires rounding in order to be represented in one of the normal formats;

Invalid → 0/0 and sqrt(-1) are attempted.

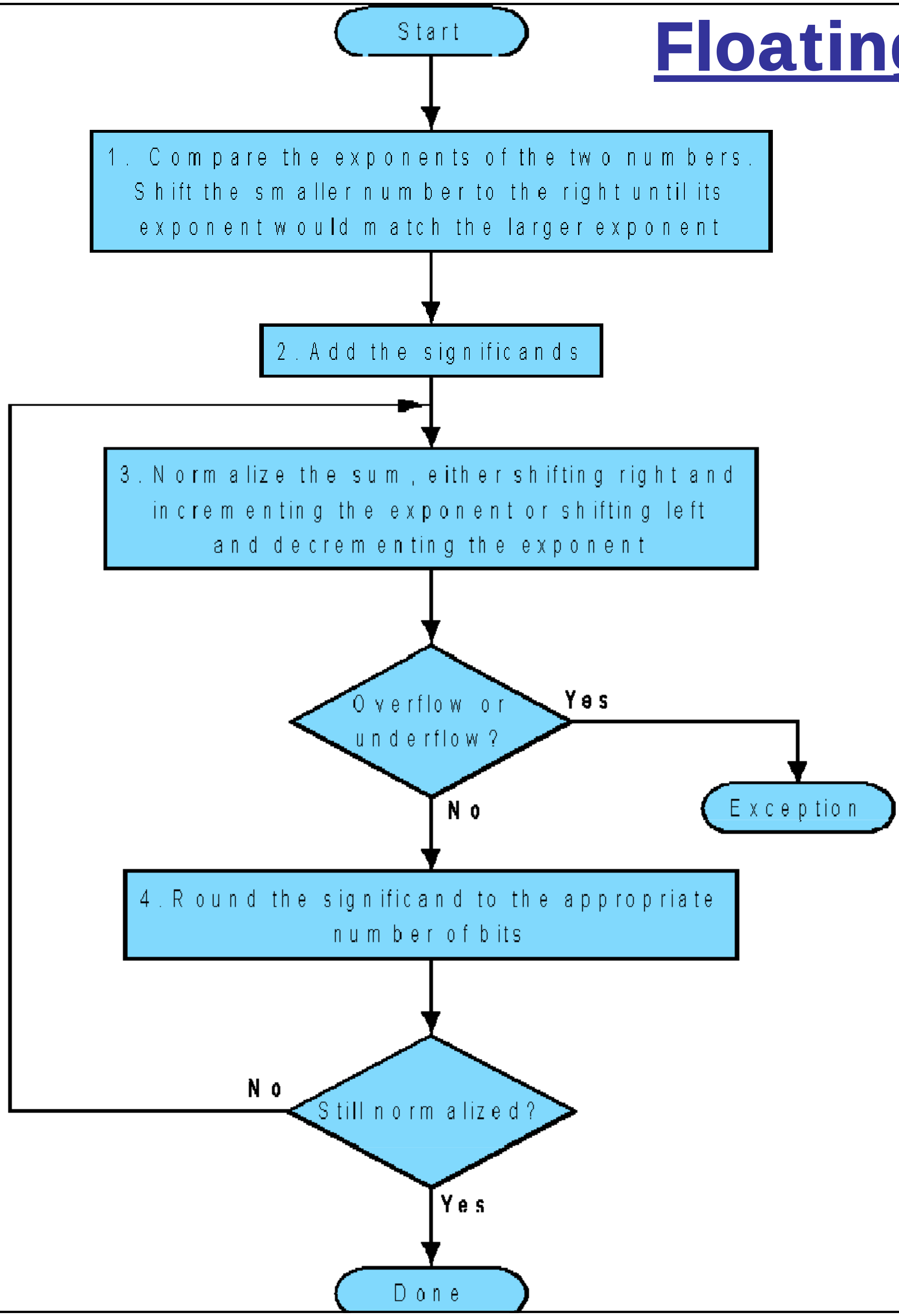
Special values are set to the results.

Algms. For ADD / SUB, MULT / DIV, in normalized floating point opens.

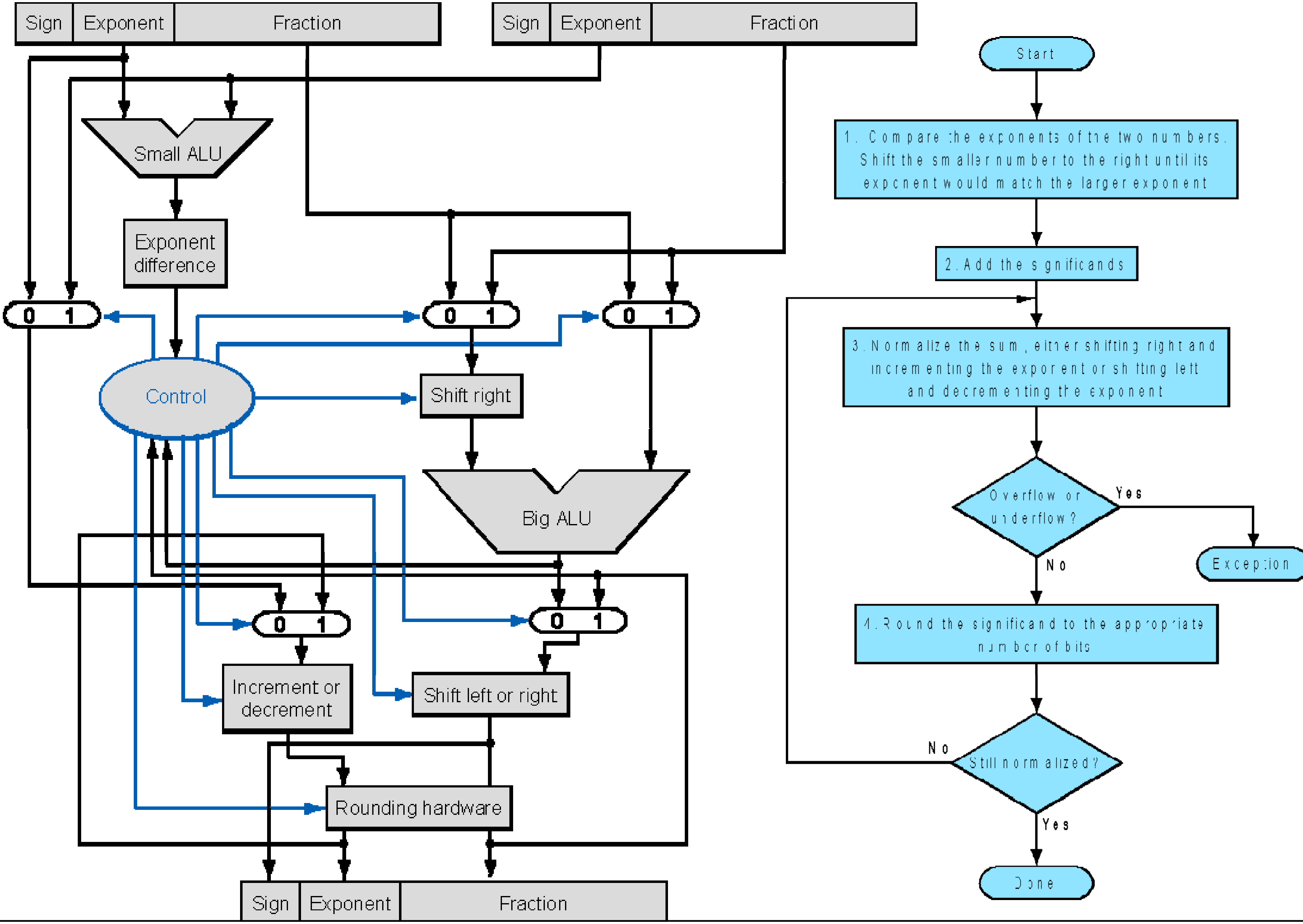
ADD / SUB	MULT.	DIV.
• Make smaller exp = large exp., by shifting opn. • Perform ADD / SUB on mantissas (get result with sign)	• Add the E's and subtract 127 • Mult. the Mantissas and get the sign	• Subtract the E's and add 127 • Div. The Mantissas and get the sign

Normalize the result value, if necessary, in all the three cases above.

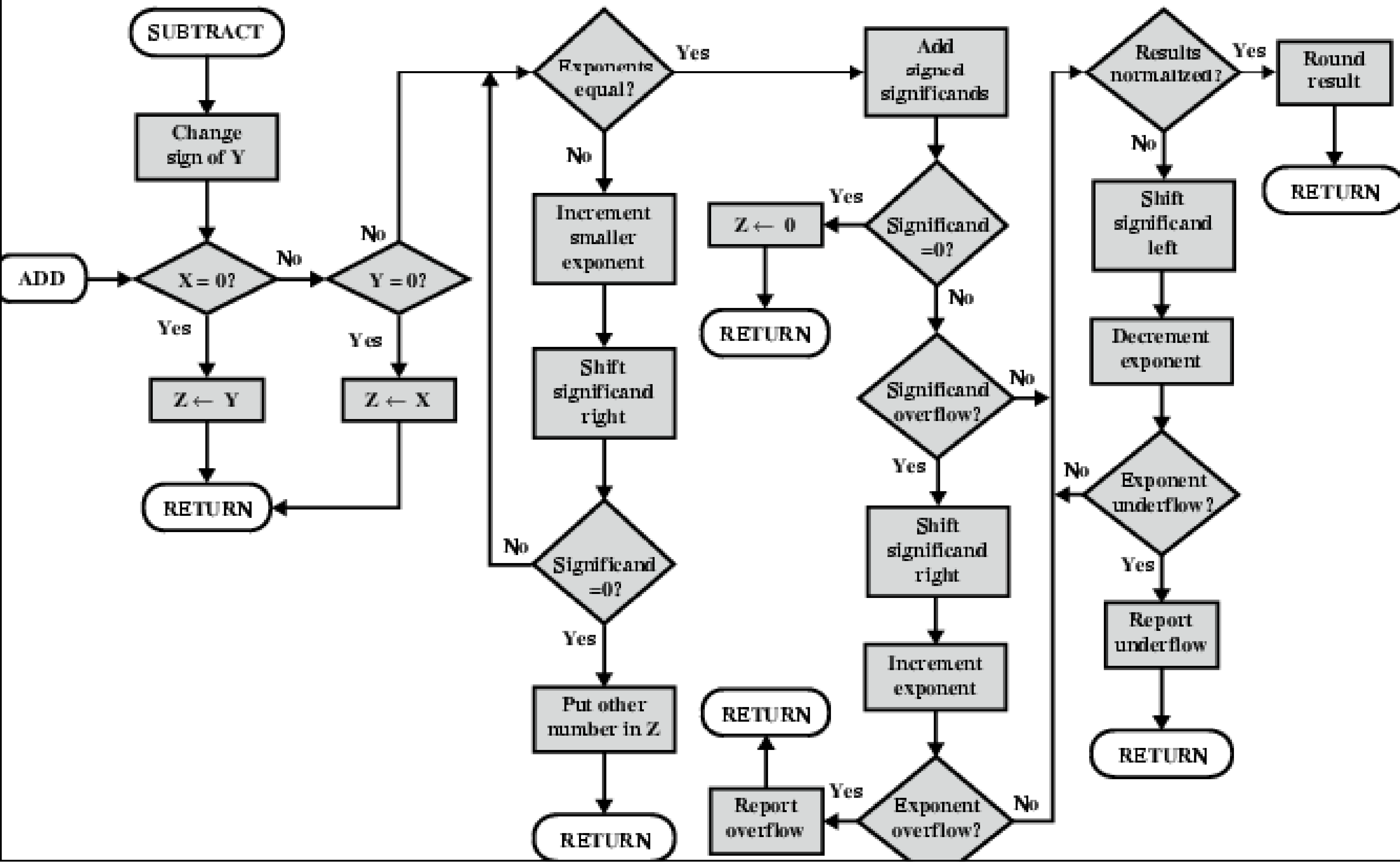
Floating point addition



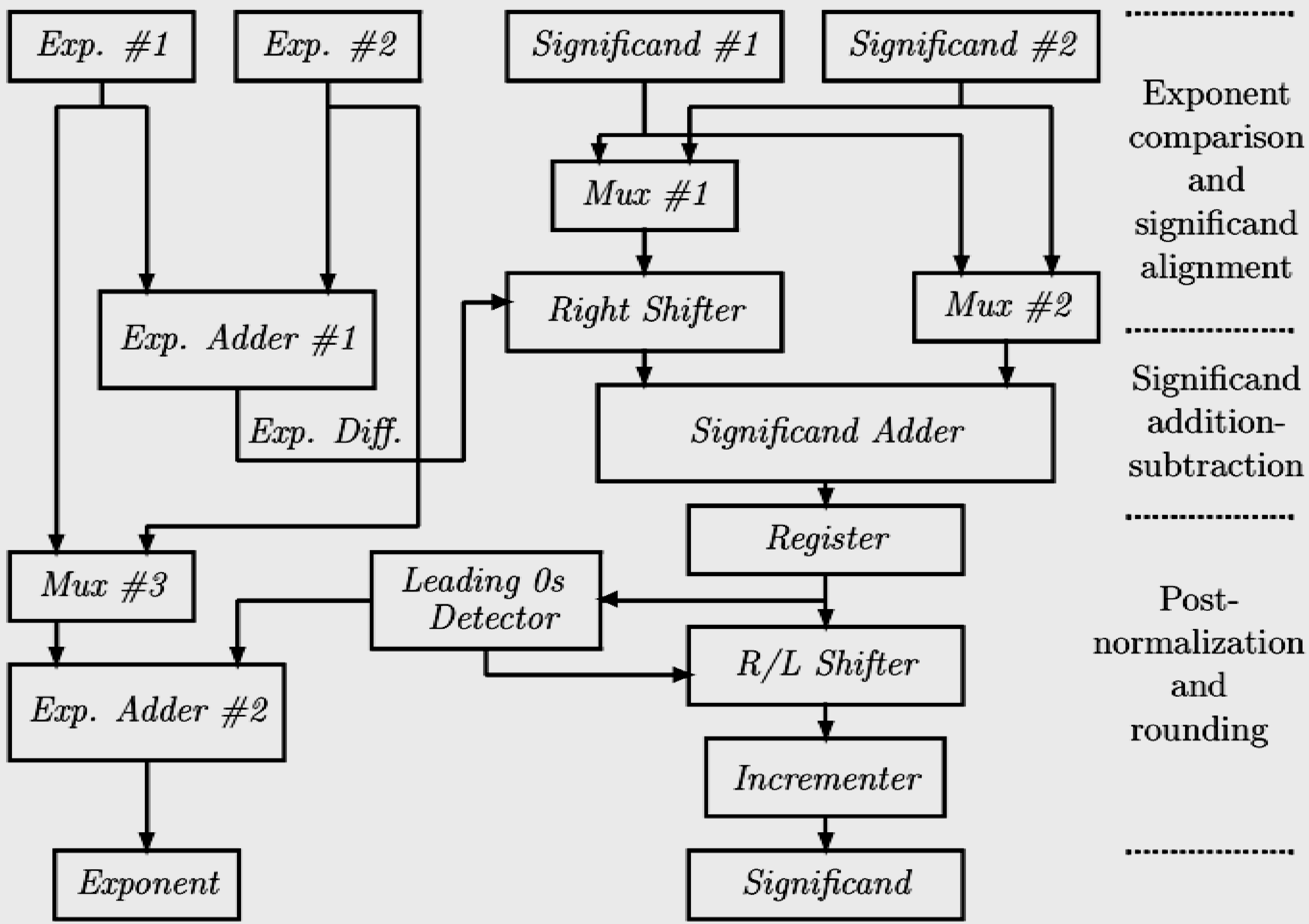
Floating point addition



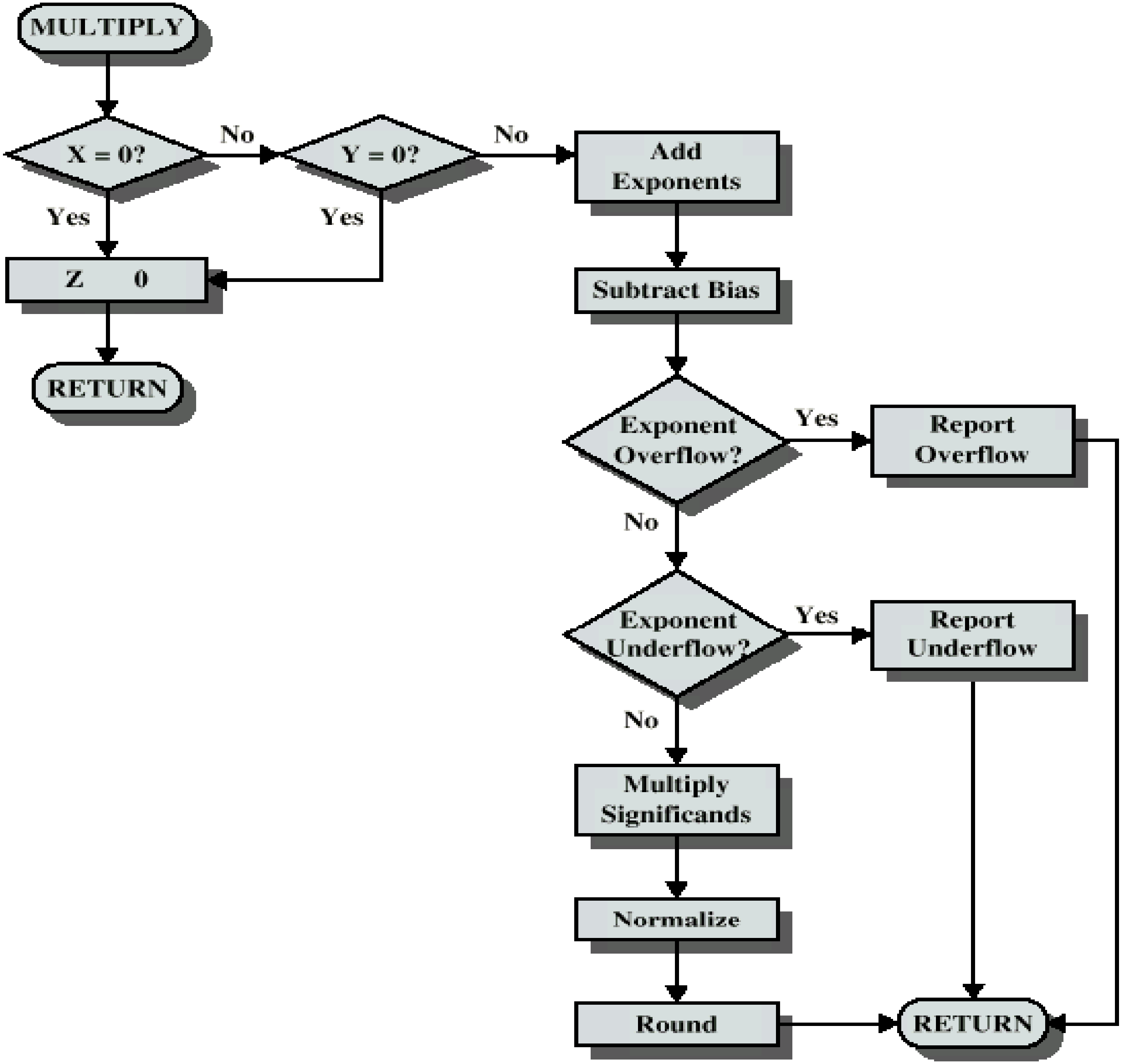
FP Addition & Subtraction Flowchart



Circuitry for Addition/Subtraction



Floating Point Multiplication



Floating Point Division