Arithmetic Units

KECE207 Digital System Design (Spring 2020)

Prof. Seon Wook Kim
School of EE, Korea University
http://compiler.korea.ac.kr
seon@korea.ac.kr

Contents

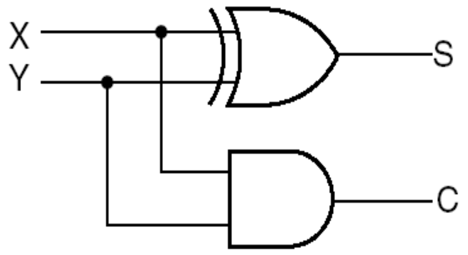
- **Binary**
  - Adders/Subtractors
  - Multipliers
- **BCD**
  - Adder
- **Floating-point (bfloat16)**
  - Multiplier
  - Adder

Half Adder (HA)

- **Add two 1-bit binary inputs**

Inputs		Outputs	
X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Truth Table of Half Adder



Logic Diagram of Half Adder

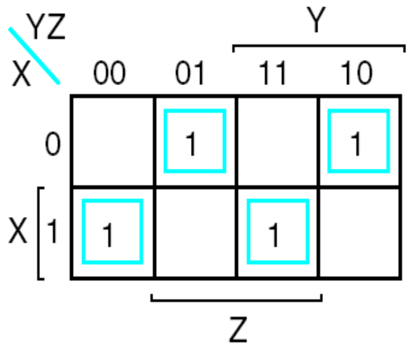
- **How about using 2-to-4 decoders & 2-to-1 multiplexers?**

Full Adder (FA)

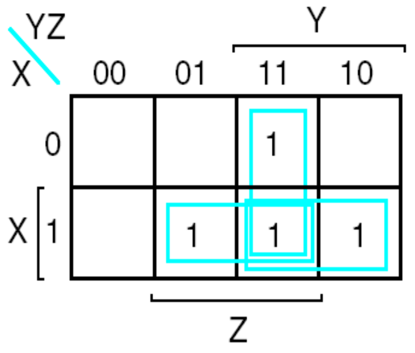
- Add three 1-bit binary inputs.

Inputs			Outputs	
X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Truth Table of Full Adder



$$S = \overline{X}\,\overline{Y}Z + \overline{X}Y\overline{Z} + X\overline{Y}\,\overline{Z} + XYZ$$
$$= X \oplus Y \oplus Z$$

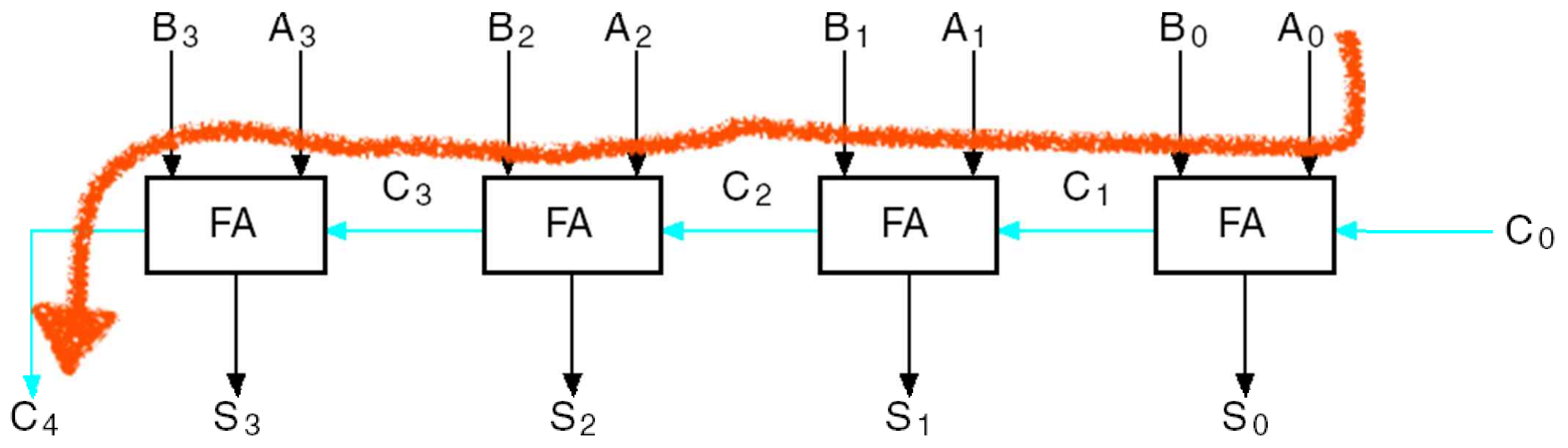


$$C = XY + XZ + YZ$$
$$= XY + Z(X\overline{Y} + \overline{X}Y)$$
$$= XY + Z(X \oplus Y)$$

Maps for Full Adder

- $X = A_i$, $Y=B_i$, $Z=C_i$, $C=C_{i+1}$, $S=S_i$

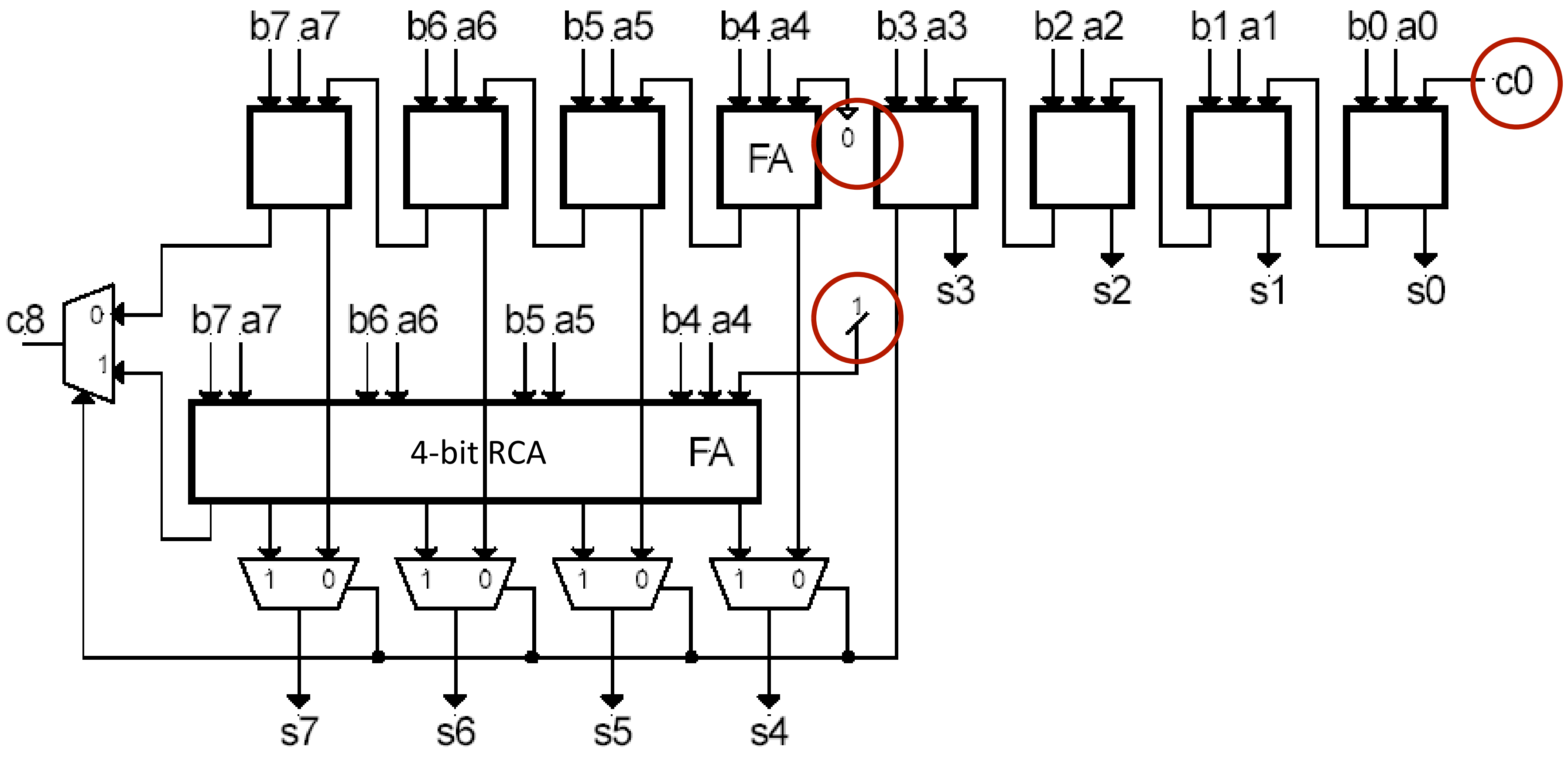
4-bit Binary Ripple Carry Adder (RCA)



- **Critical path?**
  - 4 FA delays
- **How about longer bit adders? Like 32-bit?**

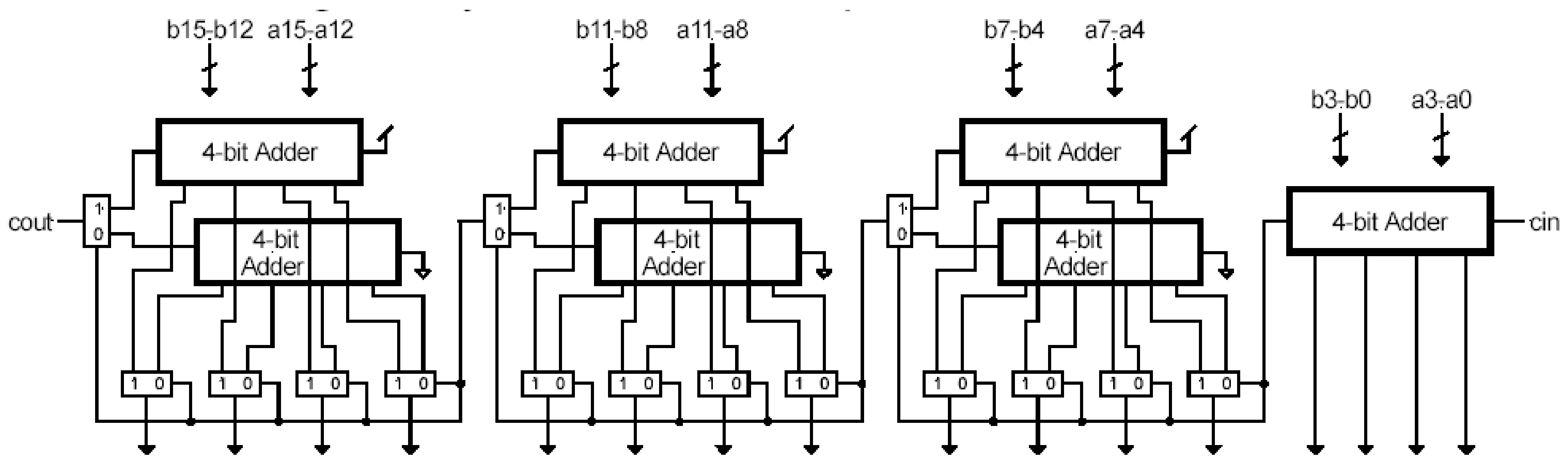
Carry Select Adder (CSA)

- **Delay = $\text{Delay}_{\text{ripple\_carry(8bit)}}/2 + \text{Delay}_{\text{MUX}}$**
- **Cost = $1.5 * \text{Cost}_{\text{ripple\_carry(4bit)}} + \text{Cost}_{\text{MUX}}$**



Extending CSA to multiple blocks

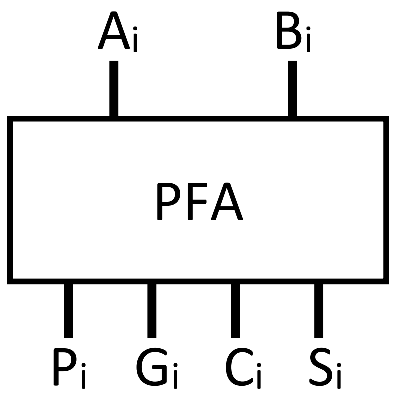
- **N-bit CSA: Sqrt(N) stages of sqrt(N) bits**



- **What is the optimal # of blocks and # of bits/block?**
  - If # blocks are too large, the delay is dominated by total MUX delay.
  - If # blocks are too small, the delay is dominated by adder delay.

Carry Lookahead Adder (CLA)

- **PFA (Partial Full Adder)**
  - Extract carry propagation path from FA's.
  - Input: A_i, B_i, C_i
  - Output: P_i, G_i, S_i



- **Propagate function: $P_i = A_i \square B_i$**
  - $P_i = 1$: an incoming carry is propagated through the bit position from C_i to C_{i+1}.
  - $P_i = 0$: carry propagation through the bit position is blocked.
- **Generate function: $G_i = A_i B_i$**
  - $G_i = 1$: the carry output from the position is 1, regardless of the value of P_i, so carry has been generated in the position.
  - $G_i = 0$: $C_{i+1} = 0$ if the carry propagated through the position from C_i is also 0.

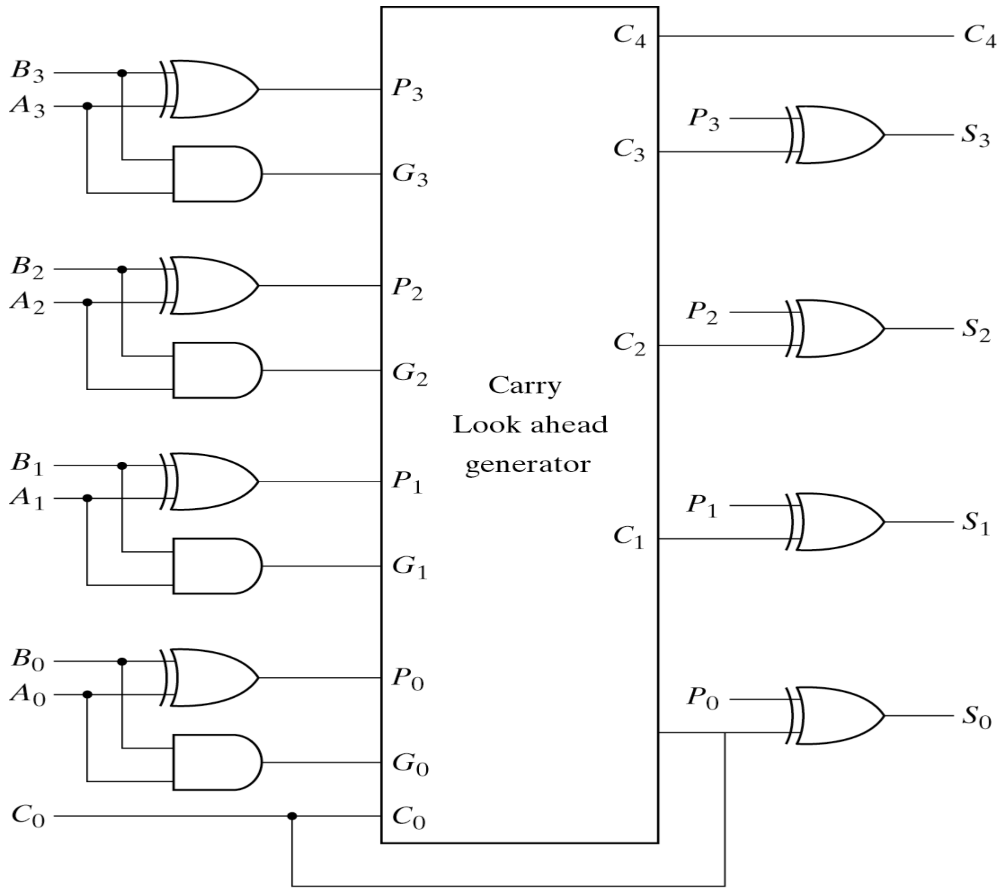
CLA Boolean Expression

- $C_1 = A_0B_0 + C_0(A_0 \square B_0) = G_0 + C_0P_0$
- $C_2 = A_1B_1 + C_1(A_1 \square B_1) = G_1 + C_1P_1$

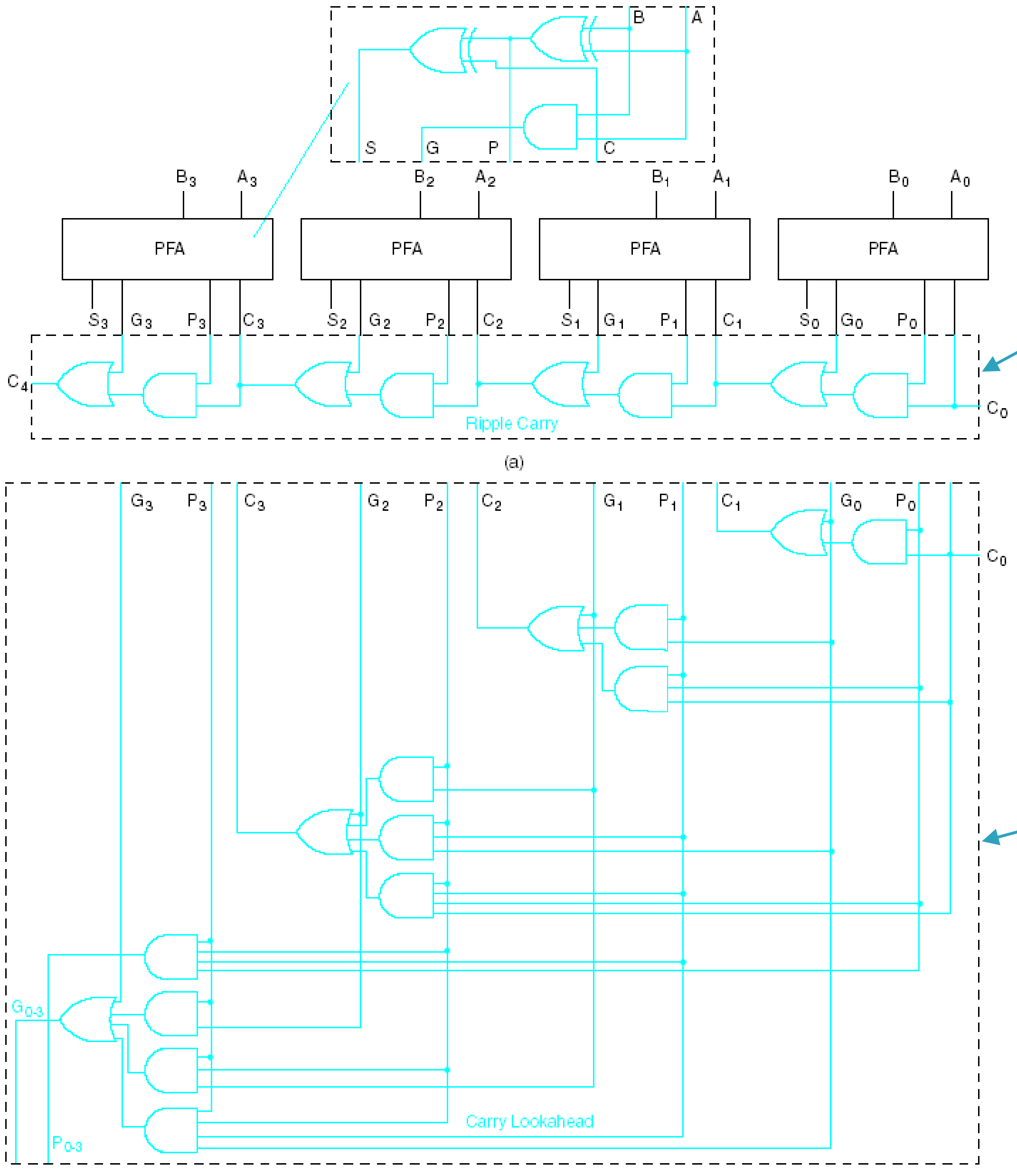
  $= G_1 + (G_0 + C_0P_0)\ P_1$

  $= G_1 + P_1\ G_0 + P_1P_0C_0$
- ……
- $C_n = G_{n-1} + P_{n-1}\ G_{n-2} + P_{n-1}\ P_{n-2}\ G_{n-3} + \ldots\ldots + P_{n-1}\ P_{n-2} \ldots P_1G_0 + P_{n-1}\ P_{n-2} \ldots P_1P_0C_0$

4-bit Adder with Carry Lookahead



CLA Generator

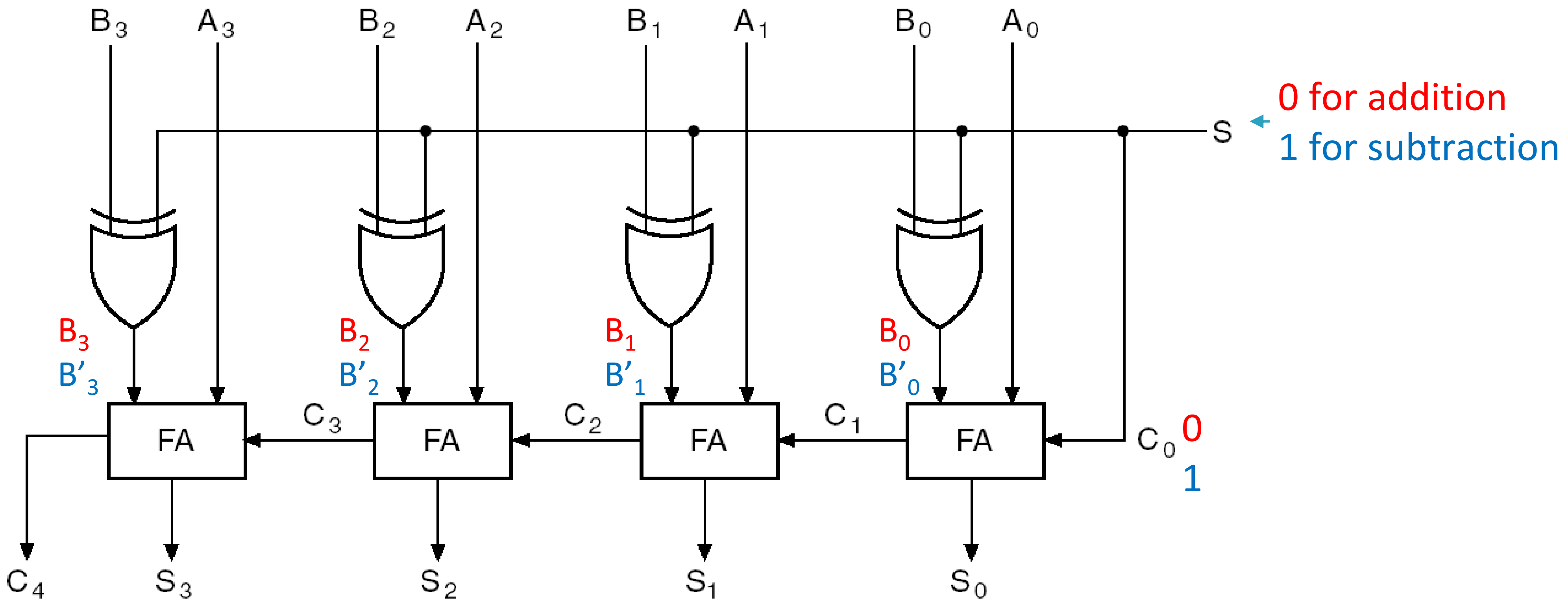


Ripple.
Slow but Simple

Fast, but Complex

Development of a Carry Lookahead Adder

4-bit Binary Adder & Subtractor

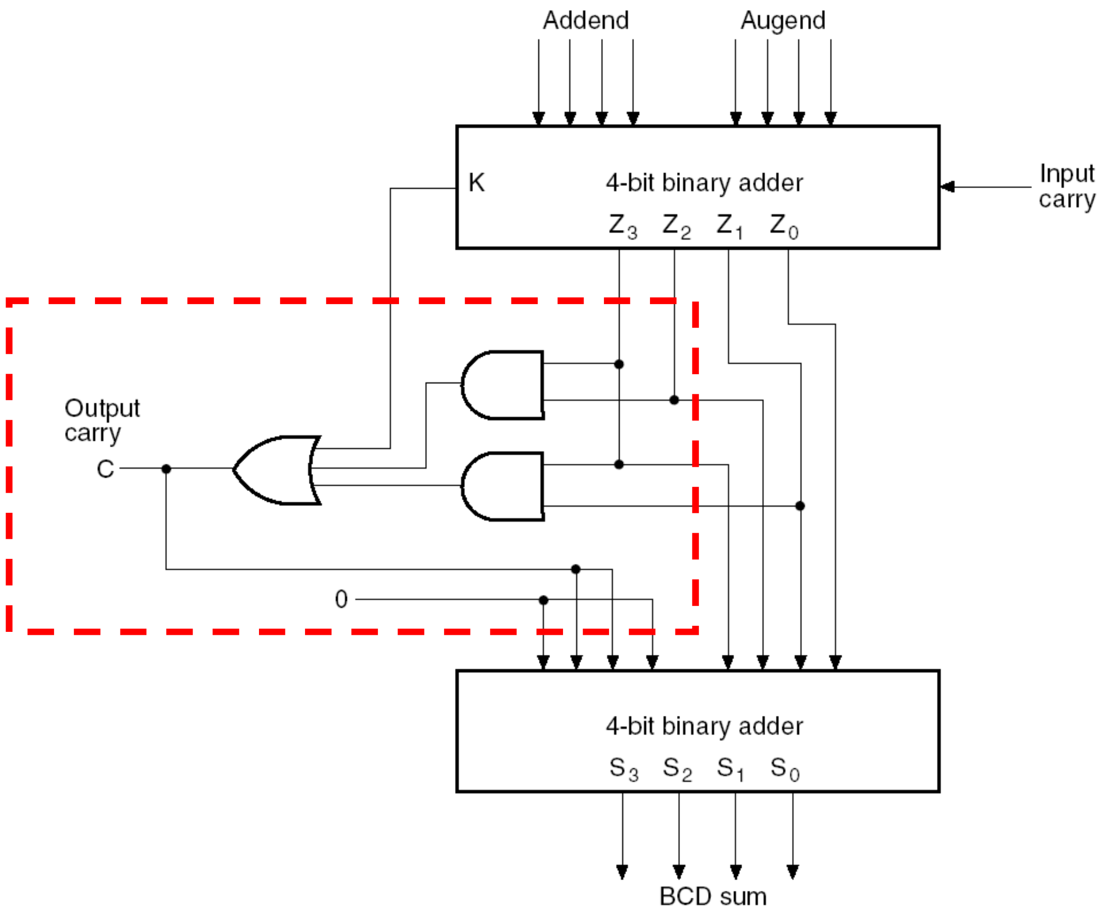


- $A_3A_2A_1A_0 - (B_3B_2B_1B_0) = A_3A_2A_1A_0 + (-B_3B_2B_1B_0)$

  $= A_3A_2A_1A_0 + (B'_3B'_2B'_1B'_0 + 1)$

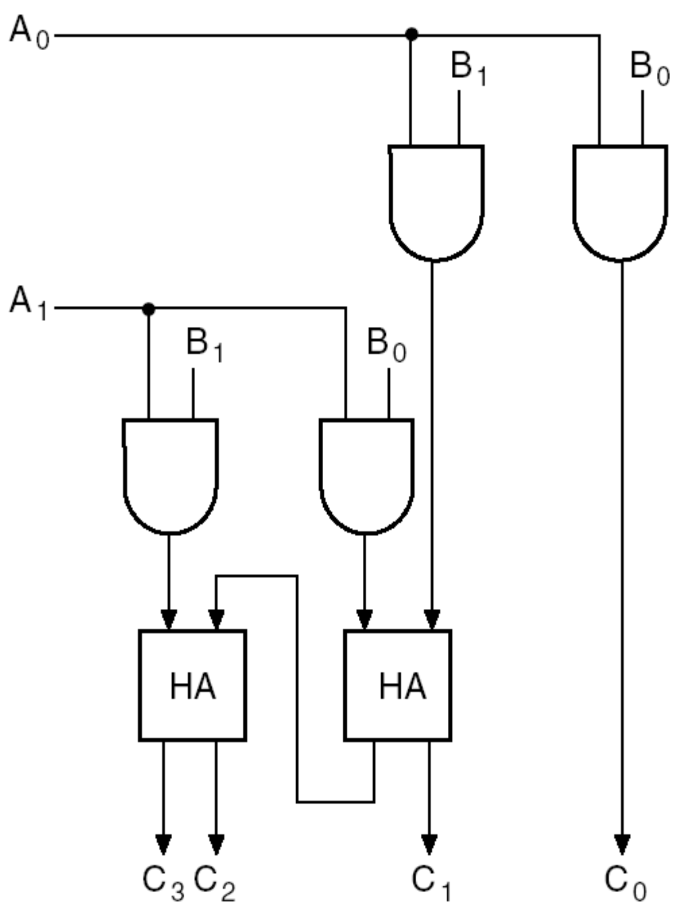
Decimal Adder (BCD Adder)

- **Binary numbers 1010 ~ 1111 need to be corrected**
- **1010, 1011, 1100, 1101, 1110, 1111**
- **If $C = K + Z_1Z_3 + Z_2Z_3$, add 0110 to the binary sum**



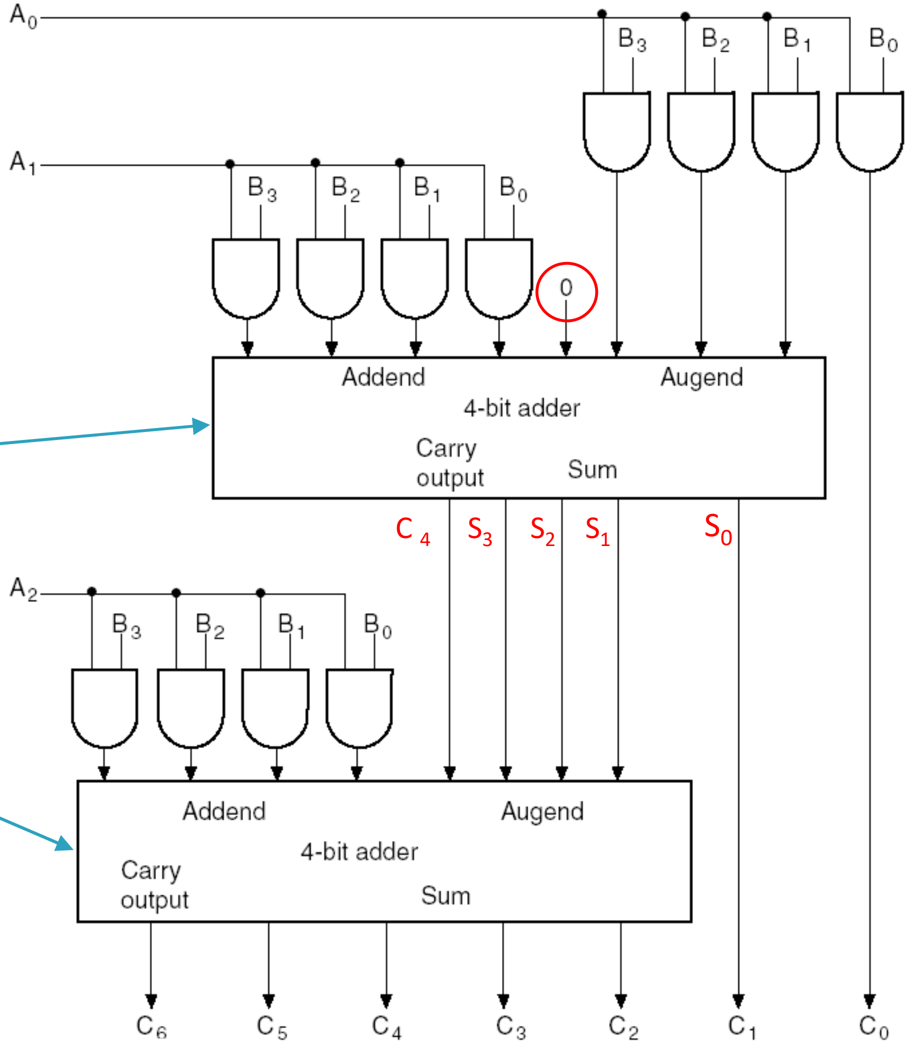
2-bit x 2-bit Unsigned Binary Multiplier

$$
\begin{array}{cccc}
 & & B_1 & B_0 \\
 & & A_1 & A_0 \\
\cline{3-4}
 & & A_0B_1 & A_0B_0 \\
 & A_1B_1 & A_1B_0 & \\
\hline
C_3 & C_2 & C_1 & C_0
\end{array}
$$



4-bit x 3-bit Unsigned Binary Multiplier

$$
\begin{array}{cccccccc}
 & & & & B_3 & B_2 & B_1 & B_0 \\
X & & & & & A_2 & A_1 & A_0 \\
\hline
 & & & & B_3A_0 & B_2A_0 & B_1A_0 & B_0A_0 \\
 & & & B_3A_1 & B_2A_1 & B_1A_1 & B_0A_1 & \\
 & & C_4 & S_3 & S_2 & S_1 & S_0 & \\
 & & B_3A_2 & B_2A_2 & B_1A_2 & B_0A_2 & & \\
\hline
 & C_6 & C_5 & C_4 & C_3 & C_2 & C_1 & C_0
\end{array}
$$

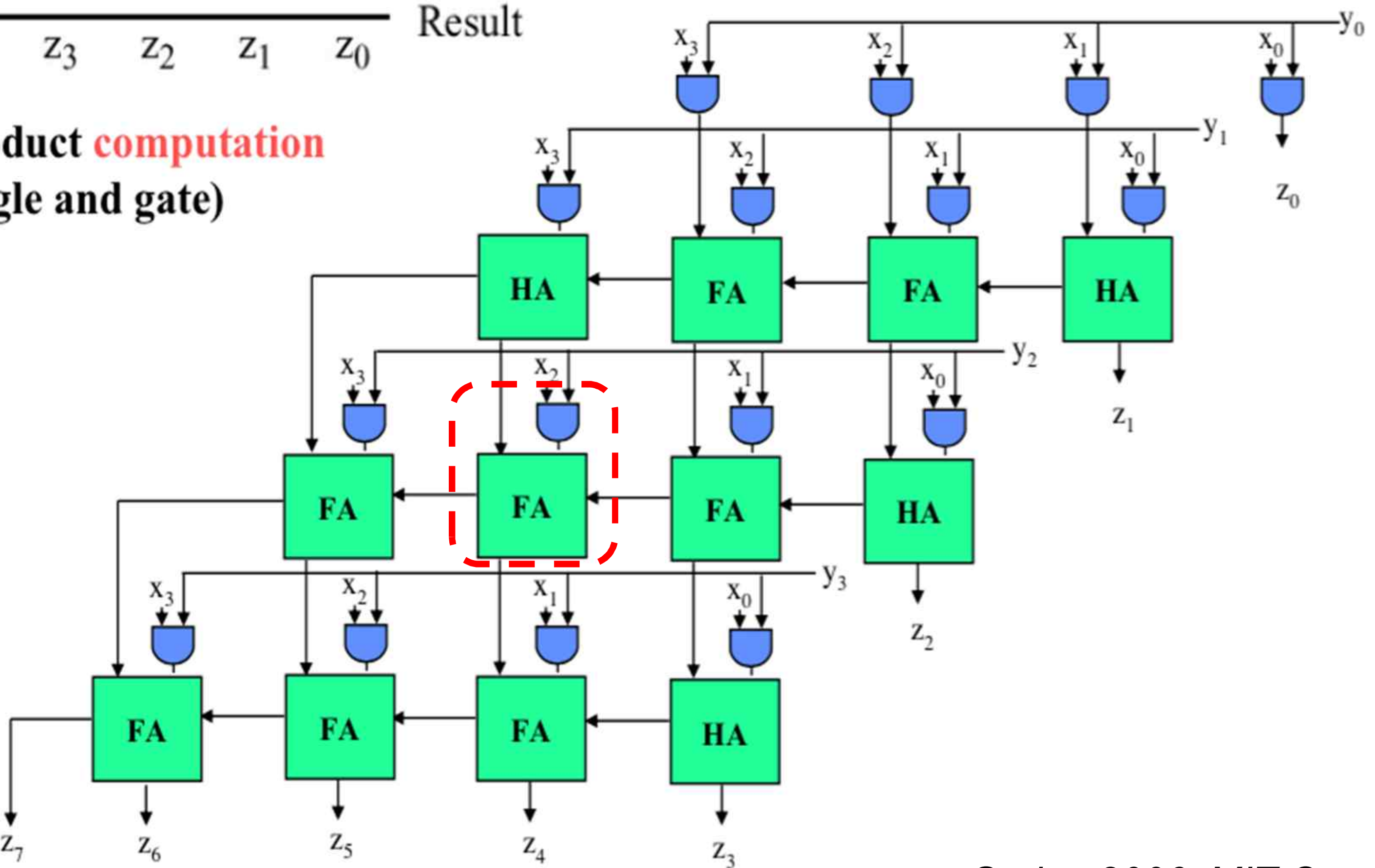


A 4-Bit by 3-Bit Binary Multiplier

4-bit x 4-bit Unsigned Array Multiplier

$$
\begin{array}{cccccccc|l}
 & & & & x_3 & x_2 & x_1 & x_0 & \text{Multiplicand} \\
 & & & \times & y_3 & y_2 & y_1 & y_0 & \text{Multiplier} \\
\hline
 & & & & x_3y_0 & x_2y_0 & x_1y_0 & x_0y_0 & \\
 & & & x_3y_1 & x_2y_1 & x_1y_1 & x_0y_1 & & \text{Partial Product} \\
 & & x_3y_2 & \boxed{x_2y_2} & x_1y_2 & x_0y_2 & & & \\
+ & x_3y_3 & x_2y_3 & x_1y_3 & x_0y_3 & & & & \\
\hline
z_7 & z_6 & z_5 & z_4 & z_3 & z_2 & z_1 & z_0 & \text{Result}
\end{array}
$$

- **Partial product computation is simple (single and gate)**



Spring 2006, MIT Courseware

Signed Binary Multiplier

- Two steps
  ① Sign extend both integers to twice as many bits.
  ② Then take the correct number of result bits from the least significant portion of the result.
- **4-bit examples**

-1 x -7 = 7

```
        1111 1111
      x 1111 1001
  ---------------
         11111111
        00000000
       00000000
      11111111
     11111111
    11111111
   11111111
  11111111
  ---------------
```

$1 \ldots 00\underline{00000111}$

3 x -5 = -15

```
        0000 0011
      x 1111 1011
  ---------------
         00000011
        00000011
       00000000
      00000011
     00000011
    00000011
   00000011
  00000011
  ---------------
```

$10\underline{11110001}$

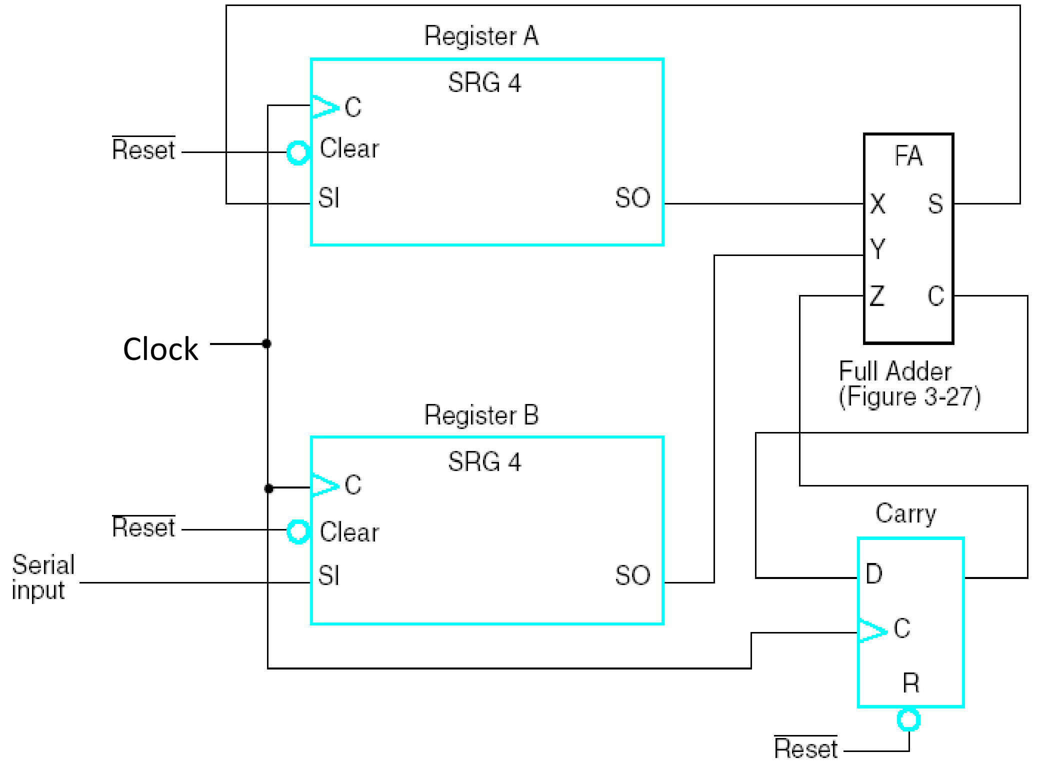
Signed Binary Multiplier (why?)

- (-3) x (-2): Ignore the overflow & Expand the dimension!

```
              111 101
            x 111 110
    ---------------------
              000 000
            1111 01
          11110 1
        111101
        ....
    ---------------------
         .... 000 110
```

Bit Serial Adder: A+B

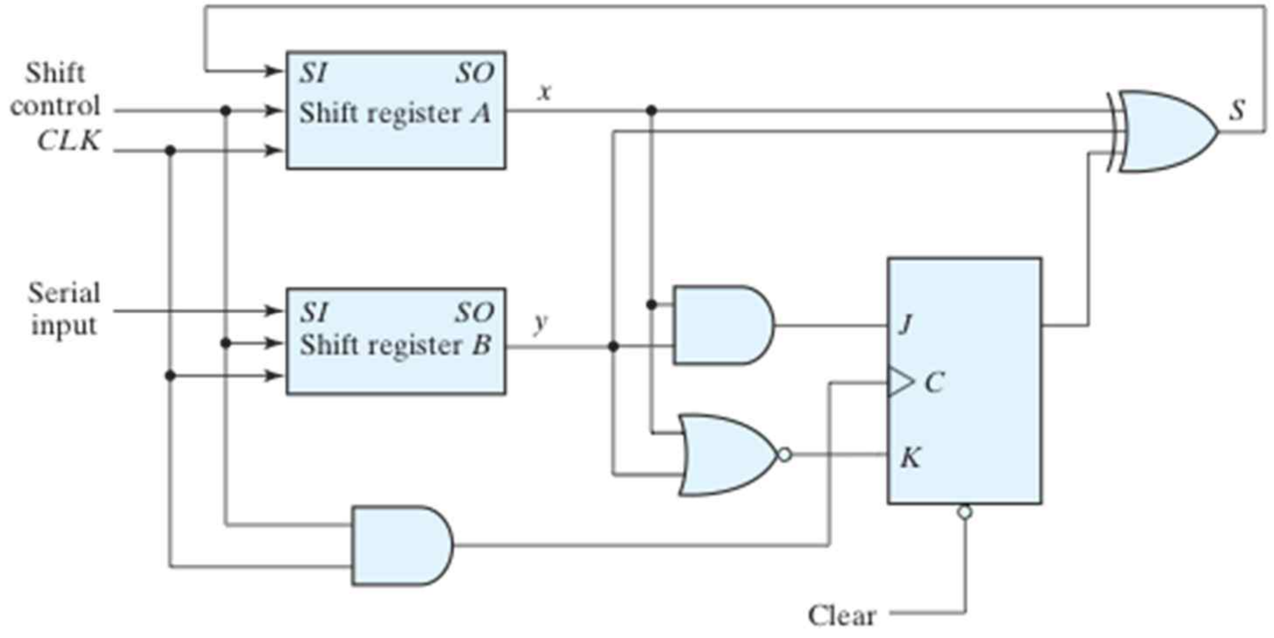
- A, B are held in shift registers.
- Initially, Carry FF is set to 0.
- Shift A and B right once per clock cycle.
- Take n clock cycles.



Bit Serial Adder (Another Approach)

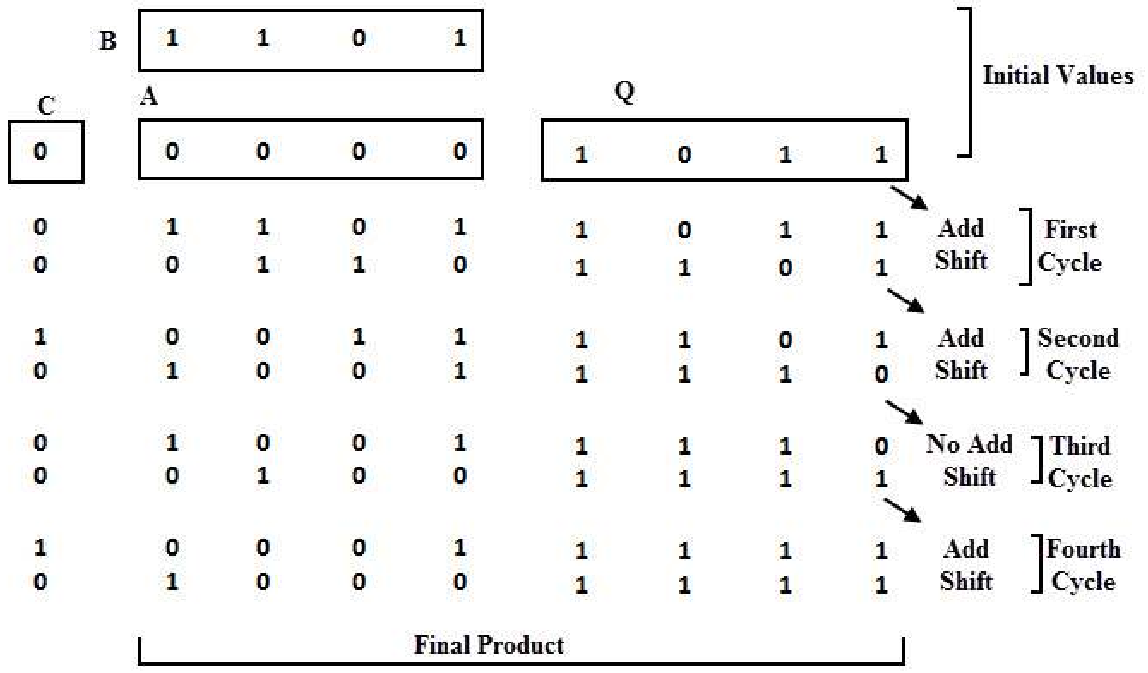
State Table for Serial Adder

Present State	Inputs		Next State	Output	Flip-Flop Inputs	
Q	x	y	Q	S	J_Q	K_Q
0	0	0	0	0	0	X
0	0	1	0	1	0	X
0	1	0	0	1	0	X
0	1	1	1	0	1	X
1	0	0	0	1	X	1
1	0	1	1	0	X	0
1	1	0	1	0	X	0
1	1	1	1	1	X	0



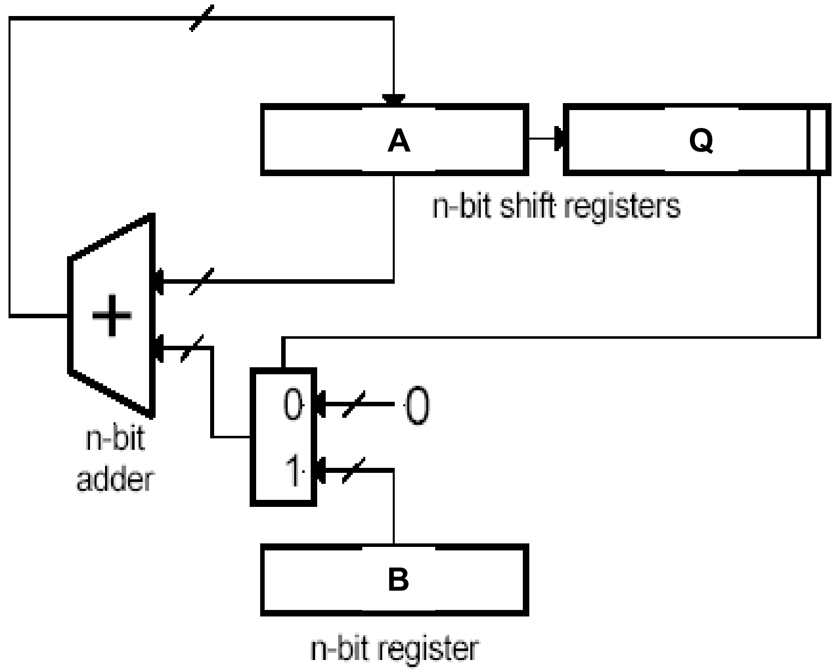
Shift-and-Add Multiplier

- **Algorithm**
  ① Step 1: A ←0, B ←multiplicand, Q ←multiplier
  ② Step 2: If Q_0 == 1 then add B to A
  ③ Step 3: Shift A|Q right once.
  ④ Step 4: Repeat Steps 2 and 3 n-1 times.
  ⑤ Step 5: A|Q has product.



	B									
	1	1	0	1						
C	A				Q					
0	0	0	0	0	1	0	1	1		Initial Values
0	1	1	0	1	1	0	1	1	Add	First Cycle
0	0	1	1	0	1	1	0	1	Shift	
1	0	0	1	1	1	1	0	1	Add	Second Cycle
0	1	0	0	1	1	1	1	0	Shift	
0	1	0	0	1	1	1	1	0	No Add	Third Cycle
0	0	1	0	0	1	1	1	1	Shift	
1	0	0	0	1	1	1	1	1	Add	Fourth Cycle
0	1	0	0	0	1	1	1	1	Shift	

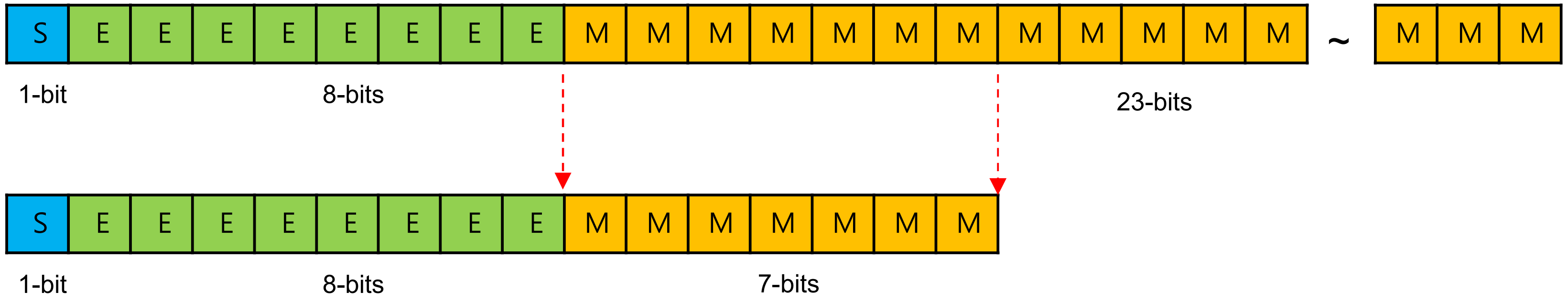
Final Product



Floating-Point (bfloat16)

In 1938, Konrad Zuse of Berlin completed the Z1, the first binary, programmable mechanical computer; it uses a 24-bit binary floating-point number representation with a 7-bit signed exponent, a 17-bit significand (including one implicit bit), and a sign bit.

- **fp32 ↔ bfloat16**



- **Bfloat16**
  - $E_{min} = 01_H - 7F_H = -126$
  - $E_{max} = FE_H - 7F_H = 127$
  - Exponent bias = $7F_H = 127$

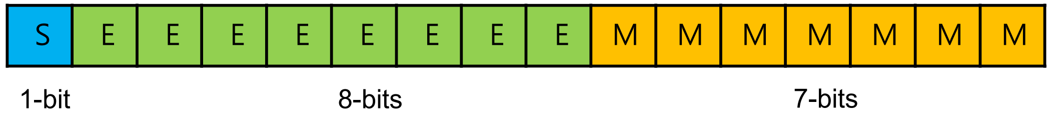
Examples:

① 0 01111111 0000000 = 1

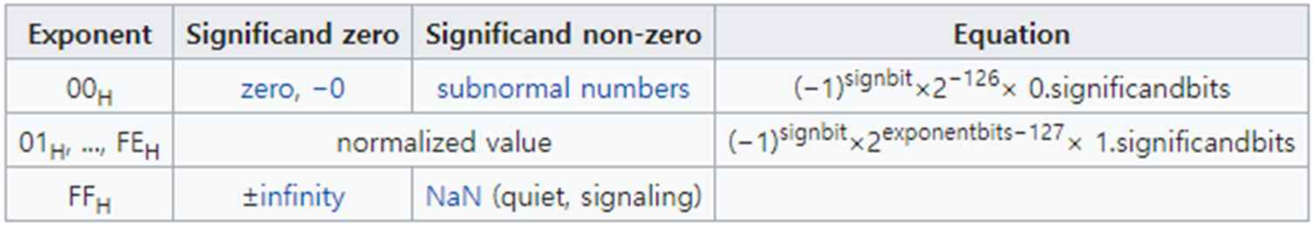
② 1 10000000 0000000 = -2

Exponent	Significand zero	Significand non-zero	Equation
00_H	zero, −0	subnormal numbers	$(-1)^{signbit} \times 2^{-126} \times$ 0.significandbits
01_H, ..., FE_H	normalized value		$(-1)^{signbit} \times 2^{exponentbits-127} \times$ 1.significandbits
FF_H	±infinity	NaN (quiet, signaling)	

bloat16



- **Positive and negative infinity**
  - +inf = 0 11111111 0000000
  - -inf = 1 11111111 0000000
- **Not a Number (NaN)** where at least one of *k, l, m, n, o, p,* or *q* is 1.
  - +NaN = 0 11111111 klmnopq
  - -NaN = 1 11111111 klmnopq
- **Zeros**
  - +0 = 0 0000000 0000000
  - -0 = 1 0000000 0000000
- **Positive**
  - Max = 0 11111110 1111111 = $(2^8-1) \times 2^{-7} \times 2^{127} = 3.38953139 \times 10^{38}$
  - Min = 0 00000001 0000000 = $2^{-126} = 1.175494351 \times 10^{-38}$



Exponent	Significand zero	Significand non-zero	Equation
00_H	zero, −0	subnormal numbers	$(-1)^{\text{signbit}} \times 2^{-126} \times 0.\text{significandbits}$
$01_H, \dots, FE_H$	normalized value		$(-1)^{\text{signbit}} \times 2^{\text{exponentbits}-127} \times 1.\text{significandbits}$
FF_H	±infinity	NaN (quiet, signaling)	

From wikipedia

Multiplication: Algorithm

- **$X = (-1)^{Sx} M_x * 2^{Ex}$,$Y = (-1)^{Sy} M_y * 2^{Ey}$**
- **$X * Y = ((-1)^{Sx} * (-1)^{Sy}) * (M_x * M_y) * 2^{Ex+Ey}$**
- **Steps**

  ① Check Zeros
  - If one or both operands is equal to zero, return the result as zero.

  ② Compute the sign of the result, S_x ☐ S_y

  ③ Multiply mantissa, $M_x * M_y$

  ④ Add exponents: $E_x + E_y – 127$

  ⑤ Normalize the result
  - Left shift the result mantissa & decrease the result exponent (e.g., 0.001xx…)
  - Right shift the result mantissa & increase the result exponent (e.g.,10.1xx…)

  ⑥ Check result
  - If larger/smaller than maximum exponent allowed return exponent overflow/underflow

Multiplication: Example

- **Suppose**
  - X = 0 01111111 0000000 = 1
  - Y = 1 10000000 0000000 = -2
- **Steps**

  ① N/A

  ② S_x □ S_y = 1

  ③ M_x = 1.0000000, M_y = 1.0000000. Don’t forget the hidden bit.

  - M_x* M_y = 1.00000000000000

  ④ E_x = 01111111, Ey = 10000000: $E_x + E_y - 127$ = 10000000

  ⑤ Normalize the result

  - N/A

  ⑥ N/A

- **Result: 1 10000000 0000000 = -2**

Addition and Subtraction: Algorithm

① **Compare the exponent of the two numbers & shift the smaller number to the right until its exponent matches the larger exponent.**

② **Add the mantissas**

③ **Normalize the result**

- Left shift the result mantissa & decrease the result exponent (e.g., 0.001xx…)
- Right shift the result mantissa & increase the result exponent (e.g.,10.1xx…)

④ **Check result**

- If larger/smaller than maximum exponent allowed return exponent overflow/underflow

⑤ **If the mantissa is zero, then set the exponent to zero**

Addition: Example

- **Suppose**
  - X = 0 01111111 0100000 = 1.25
  - Y = 0 01111101 0000000 = 0.25
- **Steps**
  ① Set Y = 0 01111111 0100000
  ② 1.0100000 + 0.0100000 = 1.1000000
  ③ N/A
  ④ N/A
  ⑤ N/A
- **Result: 0 01111111 1000000 = 1.5**

Q/A

- **More? Divisor……**