System I

Computational Operations & Units

Bo Feng, Lei Wu, Rui Chang

Zhejiang University

Disclaimer

- **Many images and resources used in this lecture are collected from the Internet, and they are used only for the educational purpose. The copyright belong to the original owners, respectively.**

- **Part of slides credit to**
  - **David Money Harris and Sarah L. Harris. Digital Design and Computer Architecture, 2nd Edition.**
  - **Morris R. Mano , Charles R. Kime and Tom Martin. Logic & Computer Design Fundamentals, Fifth Edition.**
  - **EECC 341, Prof. Muhammad Shaaban @ Rochester Institute of Technology**

Overview

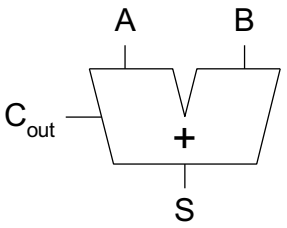
- Basic computational units
- Fixed number operations
  - Addition & Subtraction
- Arithmetic logic unit (ALU)
- Fixed number operations
  - Multiplication & Division

Overview

- Basic computational units
- Fixed number operations
  - Addition & Subtraction
- Arithmetic logic unit (ALU)
- Fixed number operations
  - Multiplication & Division

1-Bit Adders

Half Adder

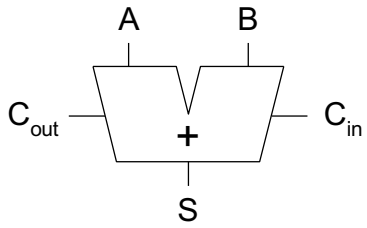


A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$S = A \oplus B$

$C_{out} = AB$

Full Adder



C_{in}	A	B	C_{out}	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

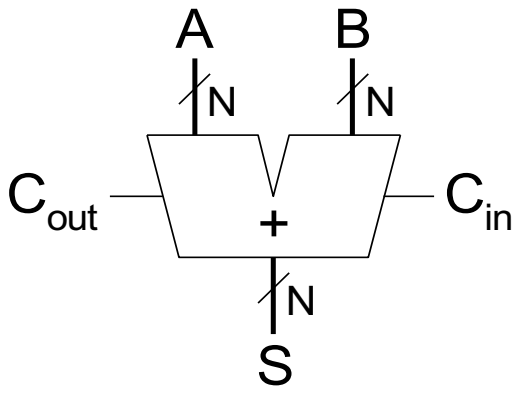
$S = A \oplus B \oplus C_{in}$

$C_{out} = AB + AC_{in} + BC_{in}$

Multibit Carry Propagate Adders (CPA)

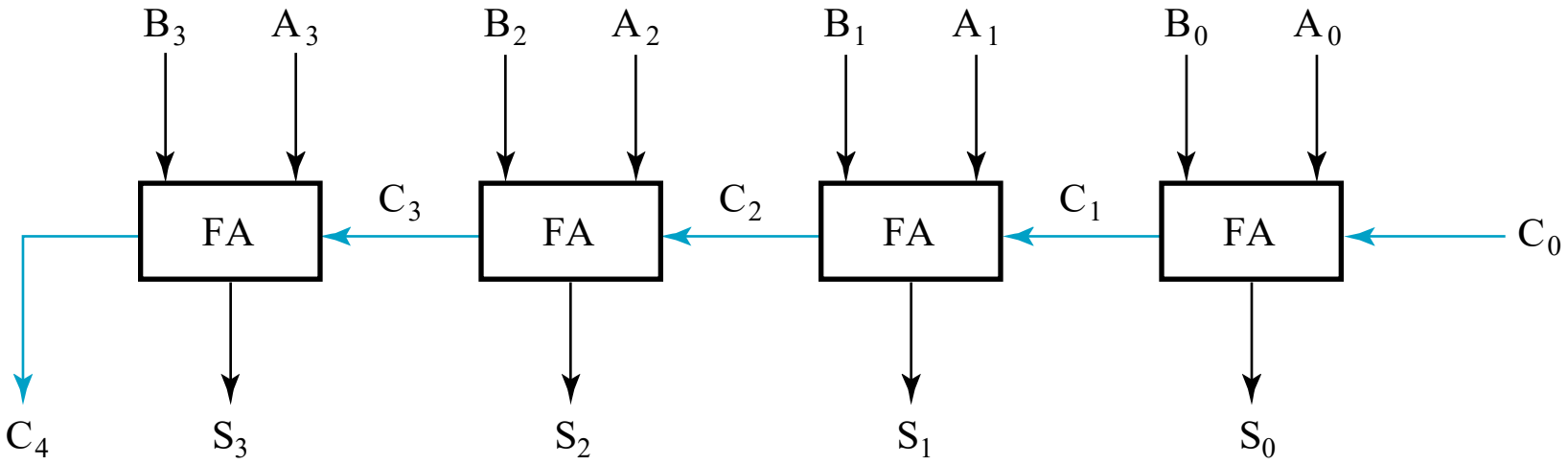
- Types of CPA
  - Ripple-carry (slow)
  - Carry-lookahead (fast)
  - Prefix (faster)
- Carry-lookahead and prefix adders faster for large adders but require more hardware

Symbol



Ripple-Carry Adder (RCA)

- Chain 1-bit adders together
- Carry ripples through entire chain
- Disadvantage: **slow**



Ripple-Carry Adder Delay

$$t_{ripple} = Nt_{FA}$$

t_{FA}: delay of a 1-bit full adder

Carry-Lookahead Adder (CLA)

- E.g., a 4-bit adder

$$C_1 = A_0B_0 + (A_0 + B_0)\, C_0$$

$$C_2 = A_1B_1 + (A_1 + B_1)\, C_1$$
$$= A_1B_1 + (A_1 + B_1)\, A_0B_0 + (A_1 + B_1)(A_0 + B_0)\, C_0$$

$$C_3 = A_2B_2 + (A_2 + B_2)\, C_2$$
$$= A_2B_2 + (A_2 + B_2)\, A_1B_1 + (A_2 + B_2)(A_1 + B_1)A_0B_0 + (A_2 + B_2)(A_1 + B_1)(A_0 + B_0)\, C_0$$

$$C_4 = A_3B_3 + (A_3 + B_3)\, C_3$$
$$= A_3B_3 + (A_3 + B_3)\, A_2B_2 + (A_3 + B_3)(A_2 + B_2)A_1B_1 + (A_3 + B_3)\, (A_2 + B_2)(A_1 + B_1)A_0B_0 + (A_3 + B_3)(A_2 + B_2)(A_1 + B_1)(A_0 + B_0)\, C_0$$

Carry-Lookahead Adder: Bit Level

- Column i produces a carry out by either ***generating*** a carry out or ***propagating*** a carry in to the carry out
- Generate (G_i) and propagate (P_i) signals for each column:
  - Column i will generate a carry out if A_i AND B_i are both 1

    $$G_i = A_i B_i$$
  - Column i will propagate a carry in to the carry out if A_i OR B_i is 1

    $$P_i = A_i + B_i$$
  - The carry out of column i+1 (C_{i+1}) is:

    $$C_{i+1} = A_i B_i + (A_i + B_i) C_i$$
    $$= G_i + P_i C_i$$
  - The sum of column i (S_i) is:

    $$S_i = A_i \oplus B_i \oplus C_i = P_i \oplus C_i$$

Revisit the 4-Bit Adder

- Compute carry out (C_{out}) for 4-bit blocks using ***generate*** and ***propagate*** signals

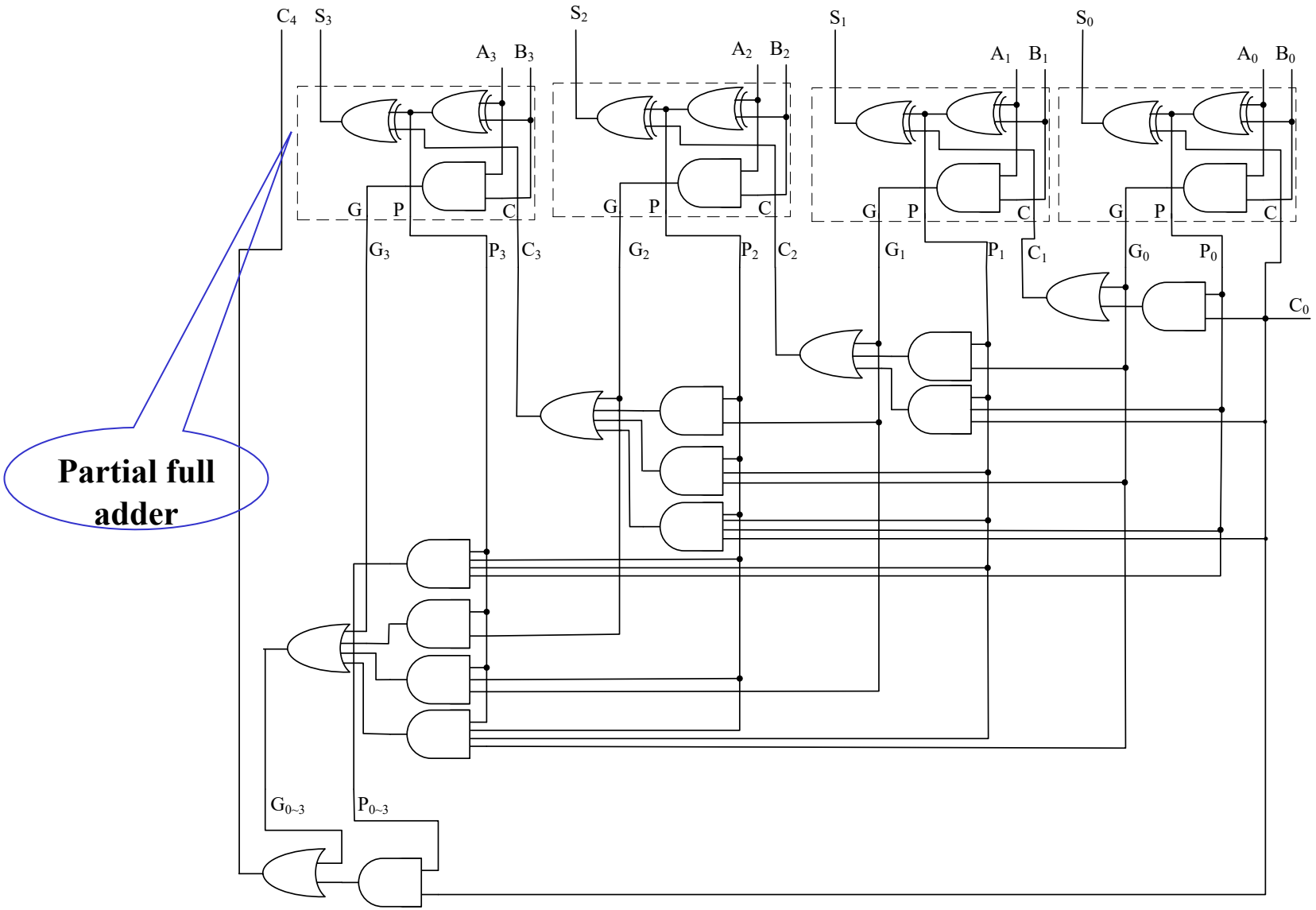
$$C_1 = G_0 + P_0C_0$$

$$\begin{aligned} C_2 &= G_1 + P_1C_1 \\ &= G_1 + P_1 (G_0 + P_0C_0) \\ &= G_1 + P_1G_0 + P_1 P_0C_0 \end{aligned}$$

$$\begin{aligned} C_3 &= G_2 + P_2C_2 \\ &= G_2 + P_2 (G_1 + P_1G_0 + P_1P_0C_0) \\ &= G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0 \end{aligned}$$

$$\begin{aligned} C_4 &= G_3 + P_3C_3 \\ &= G_3 + P_3 (G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0) \\ &= G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0 \end{aligned}$$

4-Bit CLA


C4
S3
A3
B3
S2
A2
B2
S1
A1
B1
S0
A0
B0
G
P
C
G3
P3
C3
G2
P2
C2
G1
P1
C1
G0
P0
C0
G0~3
P0~3
Partial full adder

Carry-Lookahead Addition: Block/Group Level

- The 4-bit CLA could be extended to more than four bits, however, in practice, due to limited gate fan-in, such extension is not feasible.
- Instead, the concept is extended another level by considering *group generate* ($G_{3\text{-}0}$) and *group propagate* ($P_{3\text{-}0}$) functions:
  - $\mathbf{G_{3:0} = G_3 + P_3(G_2 + P_2(G_1 + P_1G_0))}$
  - $\mathbf{P_{3:0} = P_3P_2P_1P_0}$
- Using these two equations:
  - $\mathbf{C_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0\,C_0}$
    $\mathbf{= G_{3:0} + P_{3:0}C_0}$

- Thus, it is possible to have four 4-bit adders use one of the same carry lookahead circuit to speed up 16-bit addition.

Carry-Lookahead Addition: Block/Group Level (cont'd)

- **Step 1**: compute G_i and P_i for all columns
- **Step 2**: compute G and P for k-bit blocks
- **Step 3**: C_{in} propagates through each k-bit propagate/generate block
- E.g., 4-bit blocks ($G_{3:0}$ and $P_{3:0}$) :

$$G_{3:0} = G_3 + P_3(G_2 + P_2(G_1 + P_1G_0))$$

$$P_{3:0} = P_3P_2P_1P_0$$

- **Generally, for 4-bit blocks**

$$\mathbf{G_{i:j} = G_i + P_i\,(G_{i-1} + P_{i-1}\,(G_{i-2} + P_{i-2}G_j\,)}$$

$$\mathbf{P_{i:j} = P_iP_{i-1}P_{i-2}P_j}$$

$$\mathbf{C_{j+1} = G_{i:j} + P_{i:j}C_j}$$

Carry-Lookahead Addition: Block/Group Level (cont'd)

- **Step 1**: compute G_i and P_i for all columns
- **Step 2**: compute G and P for k-bit blocks
- **Step 3**: C_{in} propagates through each k-bit propagate/generate block
- **Generally, for 4-bit blocks**

$$G_{i:j} = G_i + P_i (G_{i-1} + P_{i-1} (G_{i-2} + P_{i-2} G_j))$$

$$P_{i:j} = P_i P_{i-1} P_{i-2} P_j$$

$$C_{j+1} = G_{i:j} + P_{i:j} C_j$$

- $C_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0 = G_{3:0} + P_{3:0}C_0$
- $C_8 = G_7 + P_7G_6 + P_7P_6G_5 + P_7P_6P_5G_4 + P_7P_6P_5P_4C_4 = G_{7:4} + P_{7:4}C_4$
- $C_{12} = G_{11} + P_{11}G_{10} + P_{11}P_{10}G_9 + P_{11}P_{10}P_9G_8 + P_{11}P_{10}P_9P_8C_8 = G_{11:8} + P_{11:8}C_8$
- $C_{16} = G_{15} + P_{15}G_{14} + P_{15}P_{14}G_{13} + P_{15}P_{14}P_{13}G_{12} + P_{15}P_{14}P_{13}P_{12}C_{12} = G_{15:12} + P_{15:12}C_{12}$

4-Bit Adder vs. 16-Bit Adder

4-bit Adder

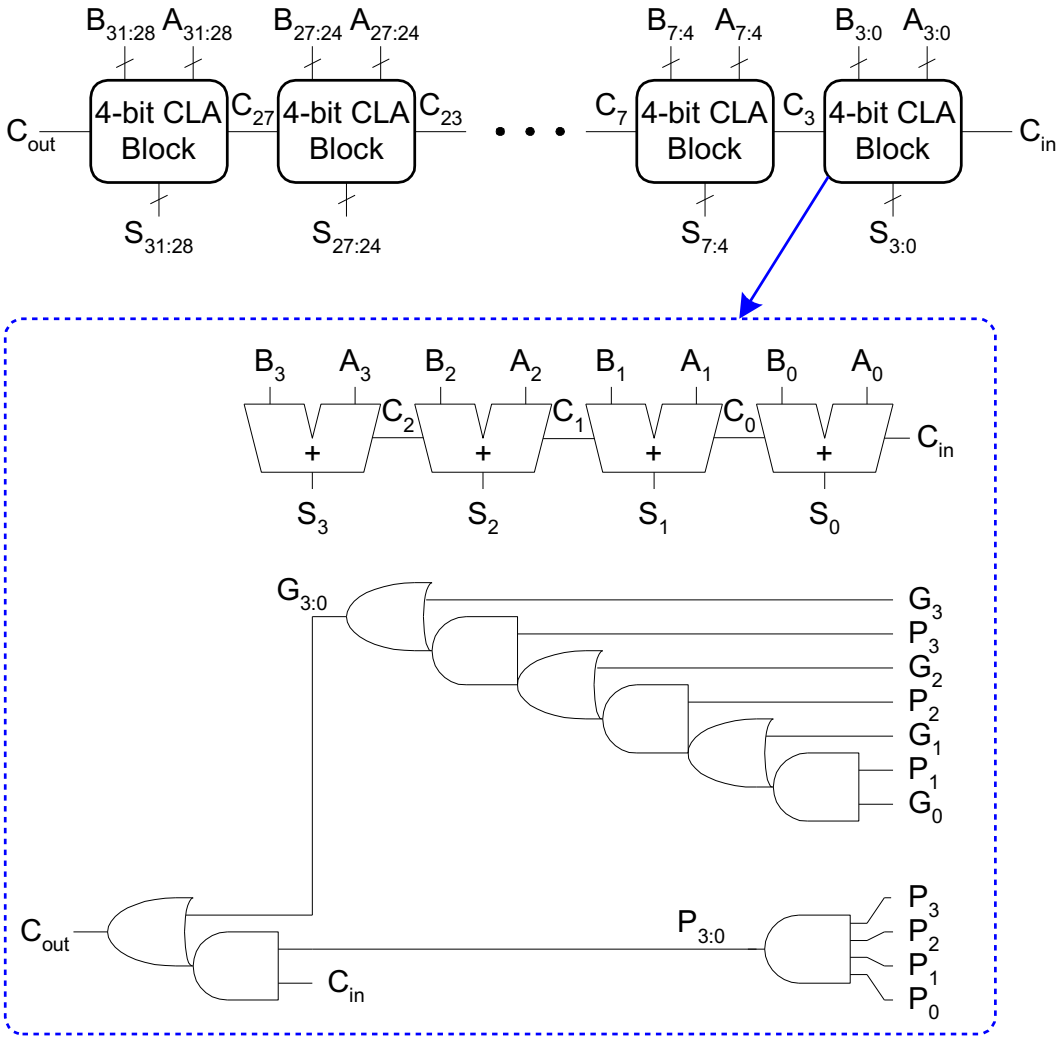
- $C_1 = G_0 + P_0C_0$
- $C_2 = G_1 + P_1C_1$
- $C_3 = G_2 + P_2C_2$
- $C_4 = G_3 + P_3C_3$

16-bit Adder

- $C_4 = G_{3:0} + P_{3:0}C_0$
- $C_8 = G_{7:4} + P_{7:4}C_4$
- $C_{12} = G_{11:8} + P_{11:8}C_8$
- $C_{16} = G_{15:12} + P_{15:12}C_{12}$

Exactly the same structure. So CLA could be used to generate Group Carry.

32-bit CLA with 4-bit Blocks



Carry-Lookahead Adder Delay

$$t_{CLA} = t_{pg} + t_{pg\_block} + (N/k - 1)\, t_{AND\_OR} + k t_{FA}$$

t_{pg}: delay to generate all $P_i G_i$

t_{pg_block}: delay of generate all $P_{i:j}$, $G_{i:j}$

t_{AND_OR}: delay from C_{in} to C_{out} of final AND/OR gate in k-bit CLA block

Delay: RCA vs. CLA

- Gate levels of C_i and S_i
  - RCA (2 and 1) vs. CLA (3 and 4)
- Different metric types
  - Only gate levels
    - All gates share the same cost
  - With specs
    - Delay of typical gate X # of gate levels
    - E.g.,
      - Carry propagation delay: 12 ns
      - Sum propagation delay: 15 ns
- An N-bit carry-lookahead adder is generally much faster than a ripple-carry adder for $N > 16$

Prefix Adder

- Computes carry in (C_{i-1}) for each column, then computes sum:

$$\mathbf{S_i = (A_i \oplus B_i) \oplus C_{i-1}}$$

- Computes G and P for 1-, 2-, 4-, 8-bit blocks, etc. until all G_i (carry in) known

- $\log_2 N$ stages

Prefix Adder (cont'd)

- Carry in either *generated* in a column or *propagated* from a previous column.
- Column -1 holds C_{in}, so

  $\mathbf{G_{-1} = C_{in}, P_{-1} = 0}$

- Carry in to column i == carry out of column i-1:

  $\mathbf{C_{i-1} = G_{i-1:-1}}$

  $G_{i-1:-1}$: generate signal spanning columns i-1 to -1

- Sum equation:

  $\mathbf{S_i = (A_i \oplus B_i) \oplus G_{i-1:-1}}$

- **Goal**: Quickly compute $G_{0:-1}$, $G_{1:-1}$, $G_{2:-1}$, $G_{3:-1}$, $G_{4:-1}$, $G_{5:-1}$, … (called ***prefixes***)

Prefix Adder (cont'd)

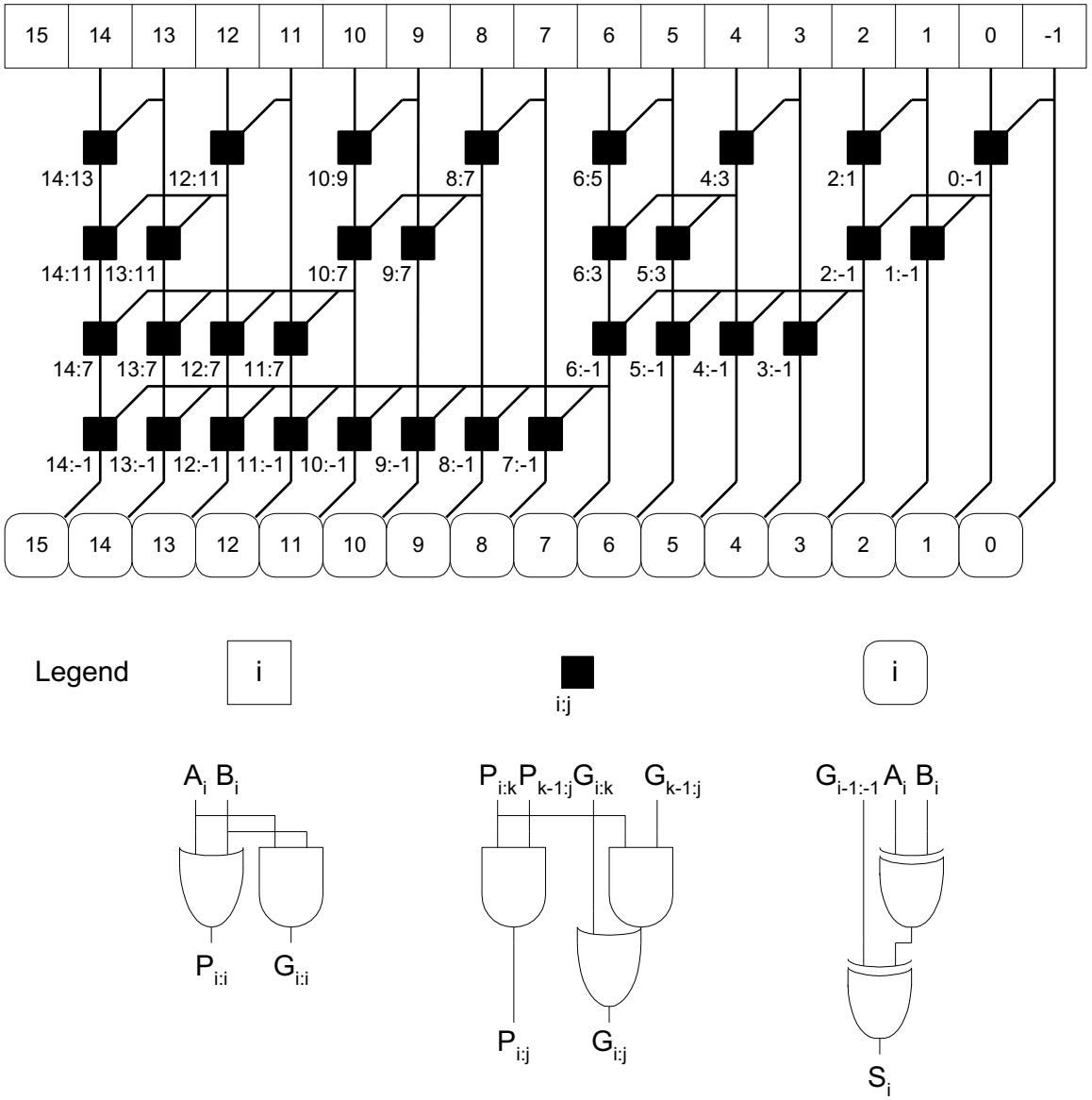
- Generate and propagate signals for a block spanning bits i:j:

$$G_{i:j} = G_{i:k} + P_{i:k} G_{k-1:j}$$

$$P_{i:j} = P_{i:k} P_{k-1:j}$$

- In words:
  - **Generate**: block i:j will generate a carry if:
    - upper part (i:k) generates a carry or
    - upper part propagates a carry generated in lower part (k-1:j)
  - **Propagate**: block i:j will propagate a carry if *both* the upper and lower parts propagate the carry

Prefix Adder Schematic



Prefix Adder Delay

$$t_{PA} = t_{pg} + log_2 N(t_{pg\_prefix}) + t_{XOR}$$

t_{pg}: delay to produce P_iG_i (AND or OR gate)

t_{pg_prefix}: delay of black prefix cell (AND-OR gate)

Half Subtractor

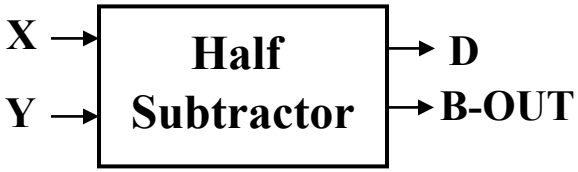
- Subtracting a single-bit binary value Y from anther X (i.e., X -Y) produces a difference bit D and a borrow out bit B-out.
- This operation is called half subtraction and the circuit to realize it is called a half subtractor.

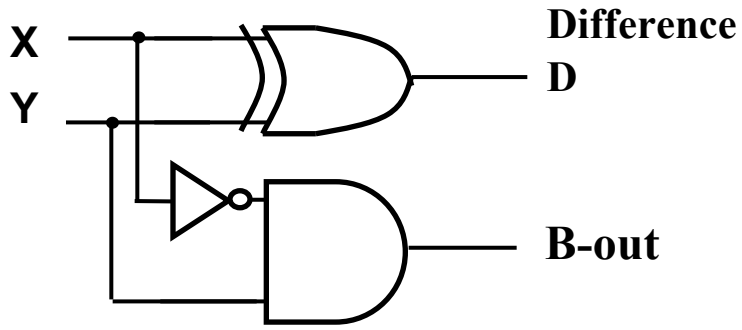
Half Subtractor Truth Table

Inputs		Outputs	
X	**Y**	**D**	**B-out**
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

$$D = X'Y + XY'$$
$$= X \oplus Y$$

$$B\text{-}out = X'Y$$





Full Subtractor

- Subtracting two single-bit binary values, Y, B-in from a single-bit value X produces a difference bit D and a borrow out B-out bit. This is called full subtraction.

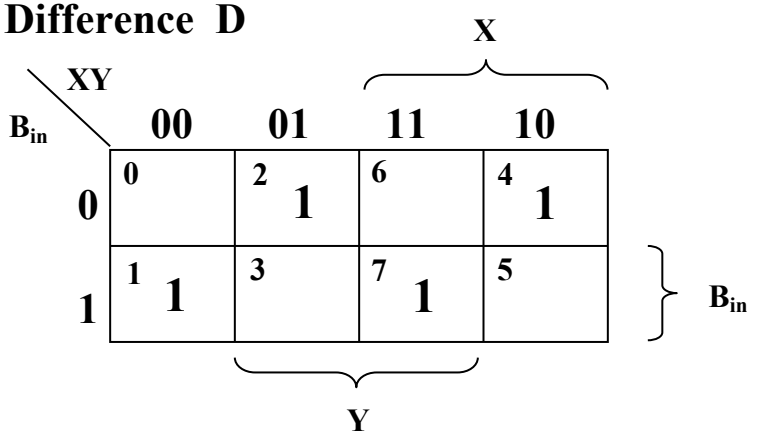
Full Subtractor Truth Table

Inputs			Outputs	
X	Y	B-in	D	B-out
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

$D(X,Y,B_{in}) = \Sigma\,(1,2,4,7)$

$B_{out}(X,Y,B_{in}) = \Sigma\,(1,2,3,7)$

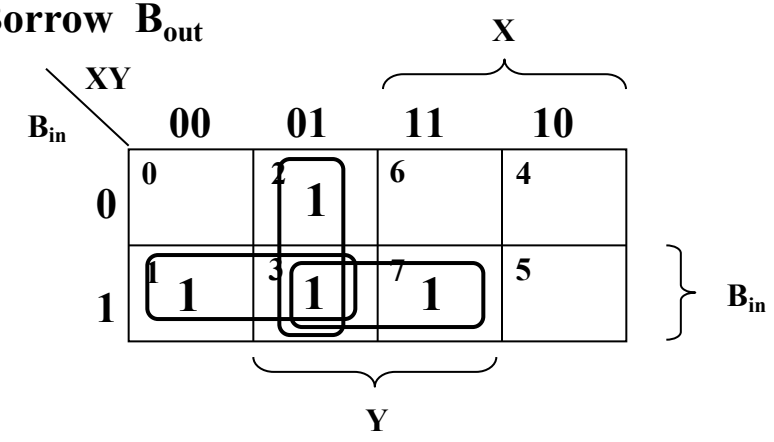
Difference D



$D = X'Y'(B_{in}) + XY'(B_{in})' + XY'(B_{in})' + XY(B_{in})$

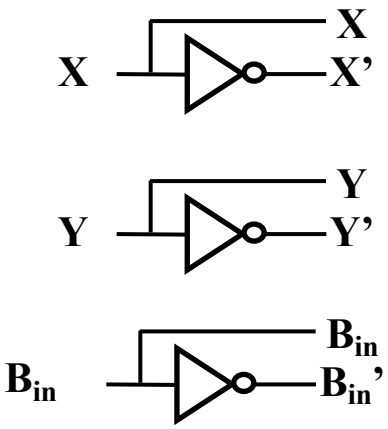
$D = X \oplus Y \oplus (B_{in})$

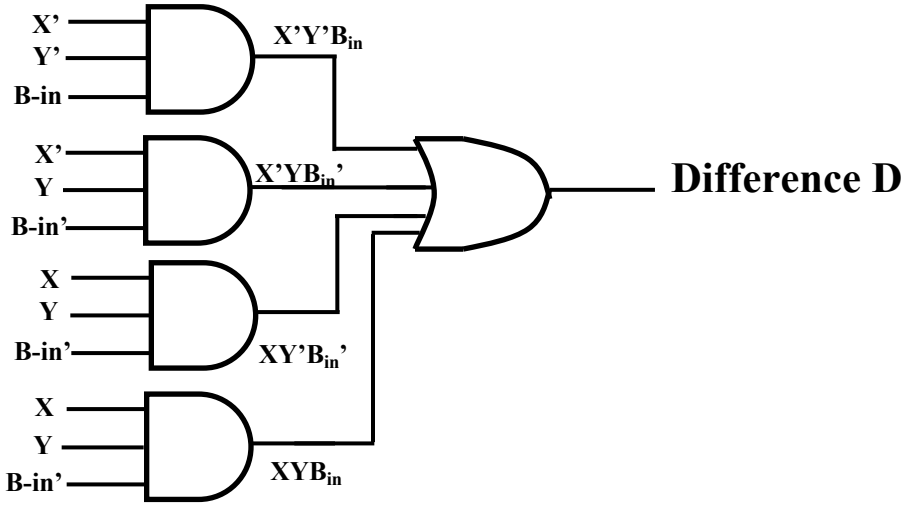
Borrow B_{out}

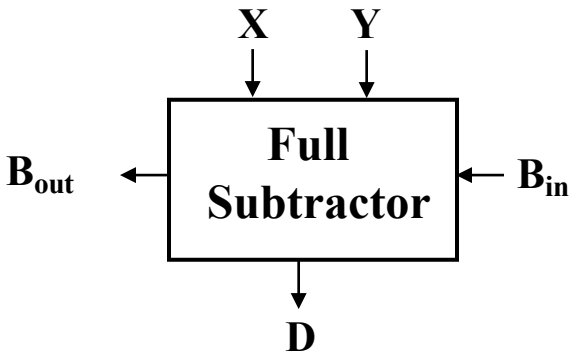


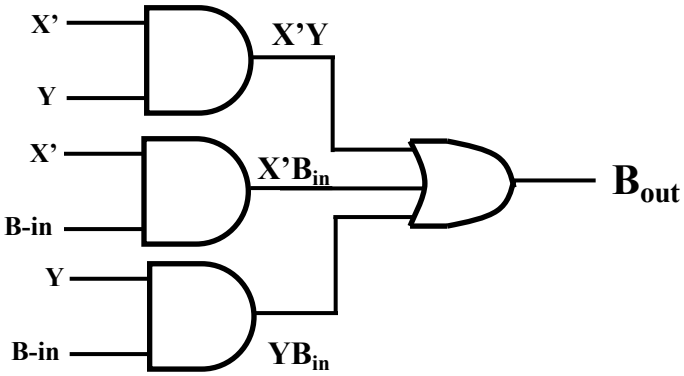
$B_{out} = X'Y + X'(B_{in}) + Y(B_{in})$

Full Subtractor Circuit Using AND-OR

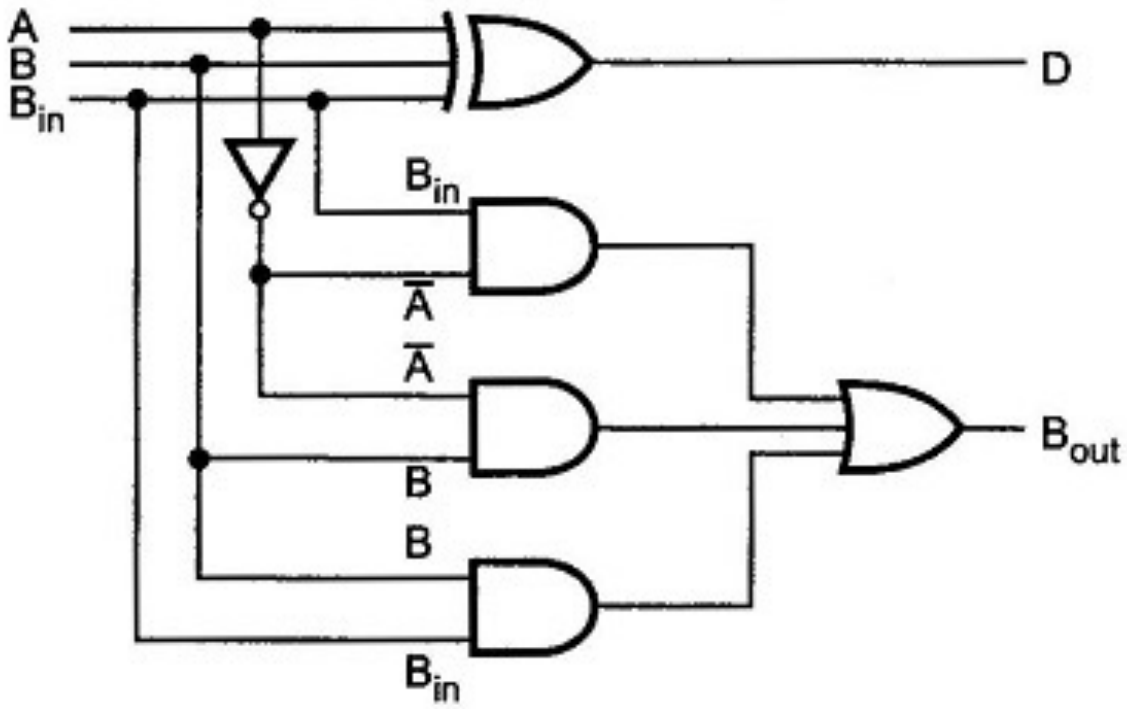








Circuit Using XOR



Implementation of N-bit Subtractors

- An n-bit subtractor used to subtract an n-bit number Y from another n-bit number X (i.e., X-Y) can be built in one of two ways:
  - By using n full subtractors and connecting them in series, creating a borrow ripple subtractor:
    - Each borrow out B-out from a full subtractor at position j is connected to the borrow in B-in of the full subtractor at the higher position j+1.
  - By using an n-bit adder and n inverters:
    - Find 2's complement of Y by:
      - Inverting all the bits of Y using the n inverters.
      - Adding 1 by setting the carry in of the least significant position to 1
    - The original subtraction (X - Y) now becomes an addition of X to two's complement of Y using the n-bit adder.

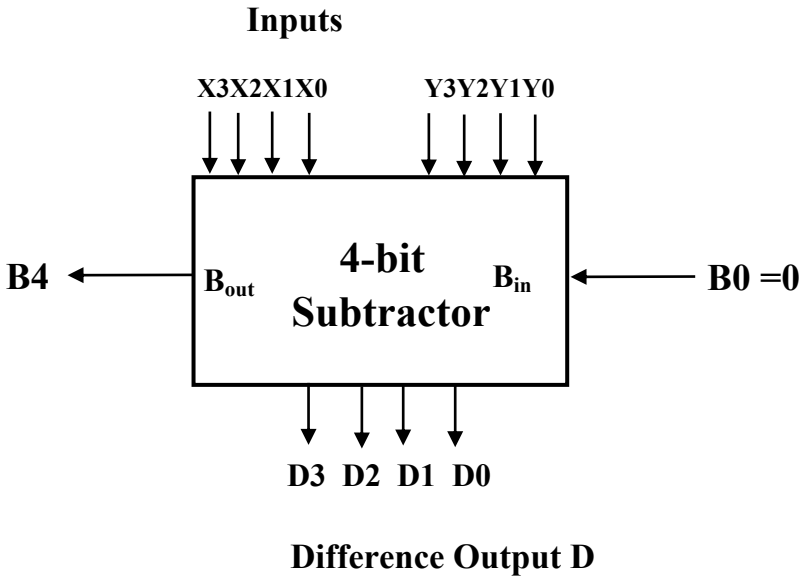
4-bit Borrow Ripple Subtractor

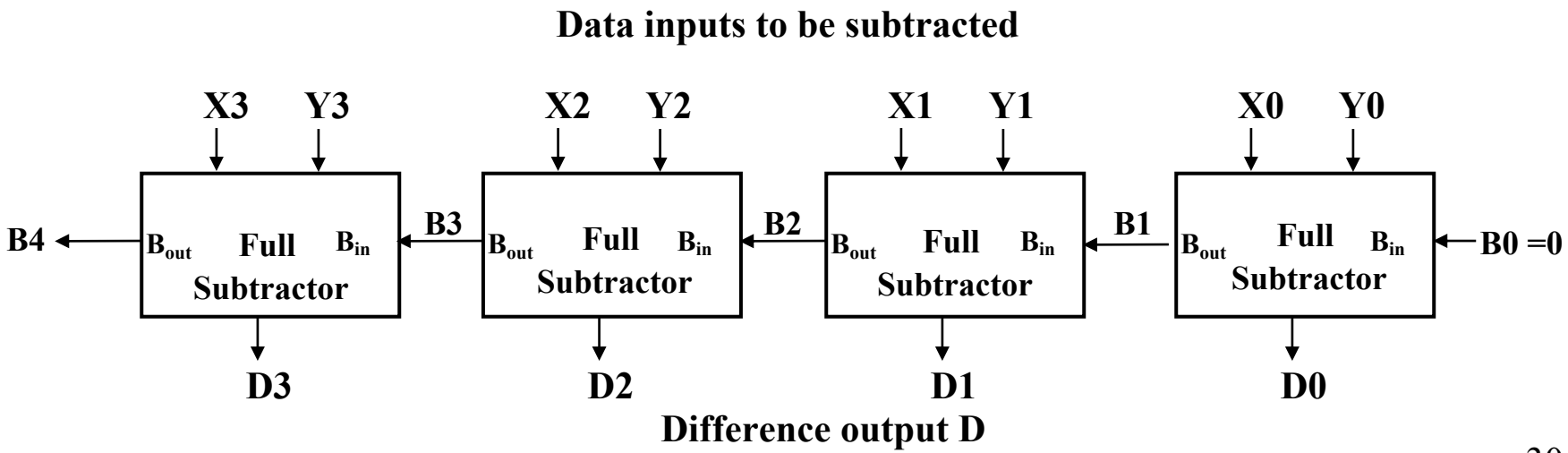
- Subtracts two 4-bit numbers:

  Y = Y3 Y2 Y1 Y0 from

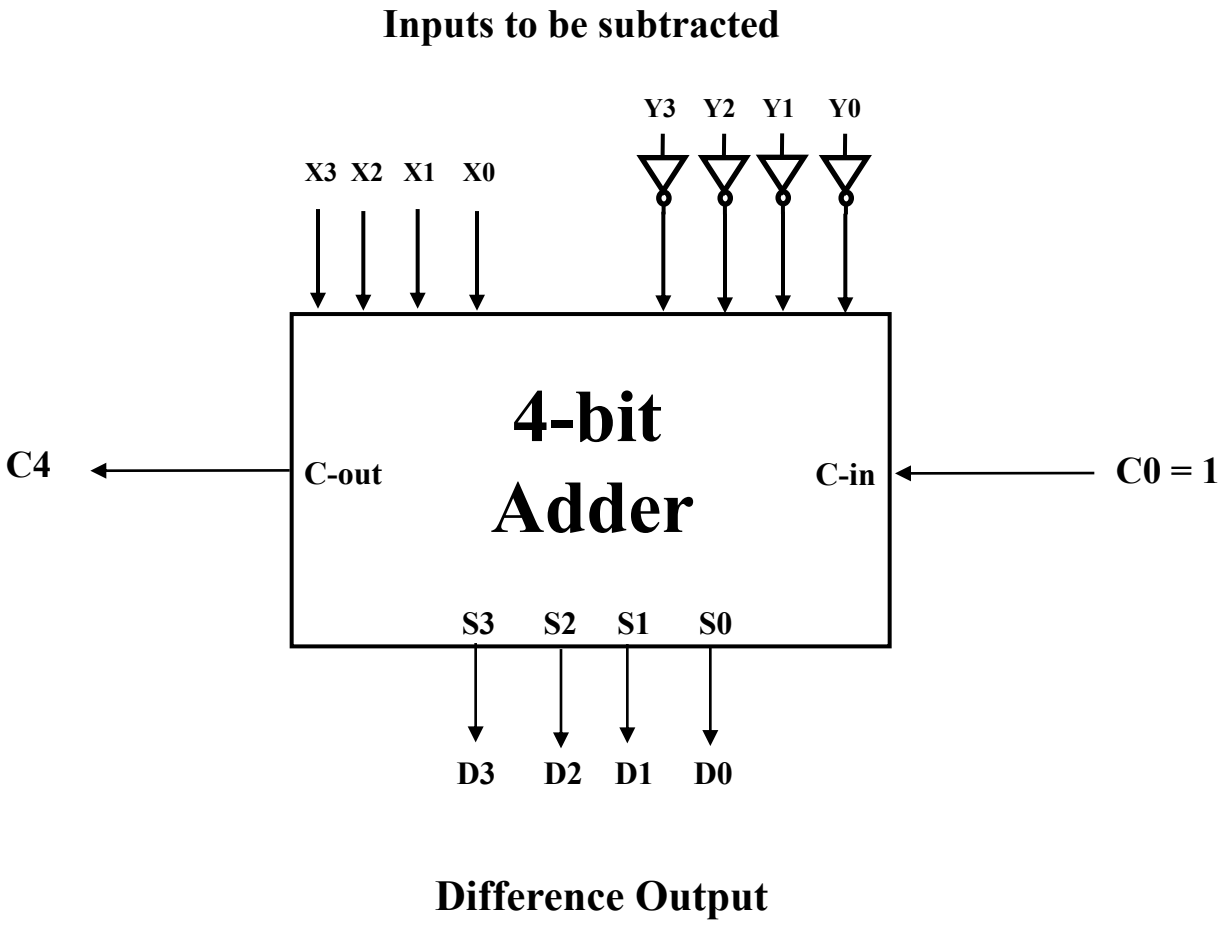
  X = X3 X2 X1 X0

  producing the difference D = D3 D2 D1 D0 , B_{out} = B4 from the most significant position j=3





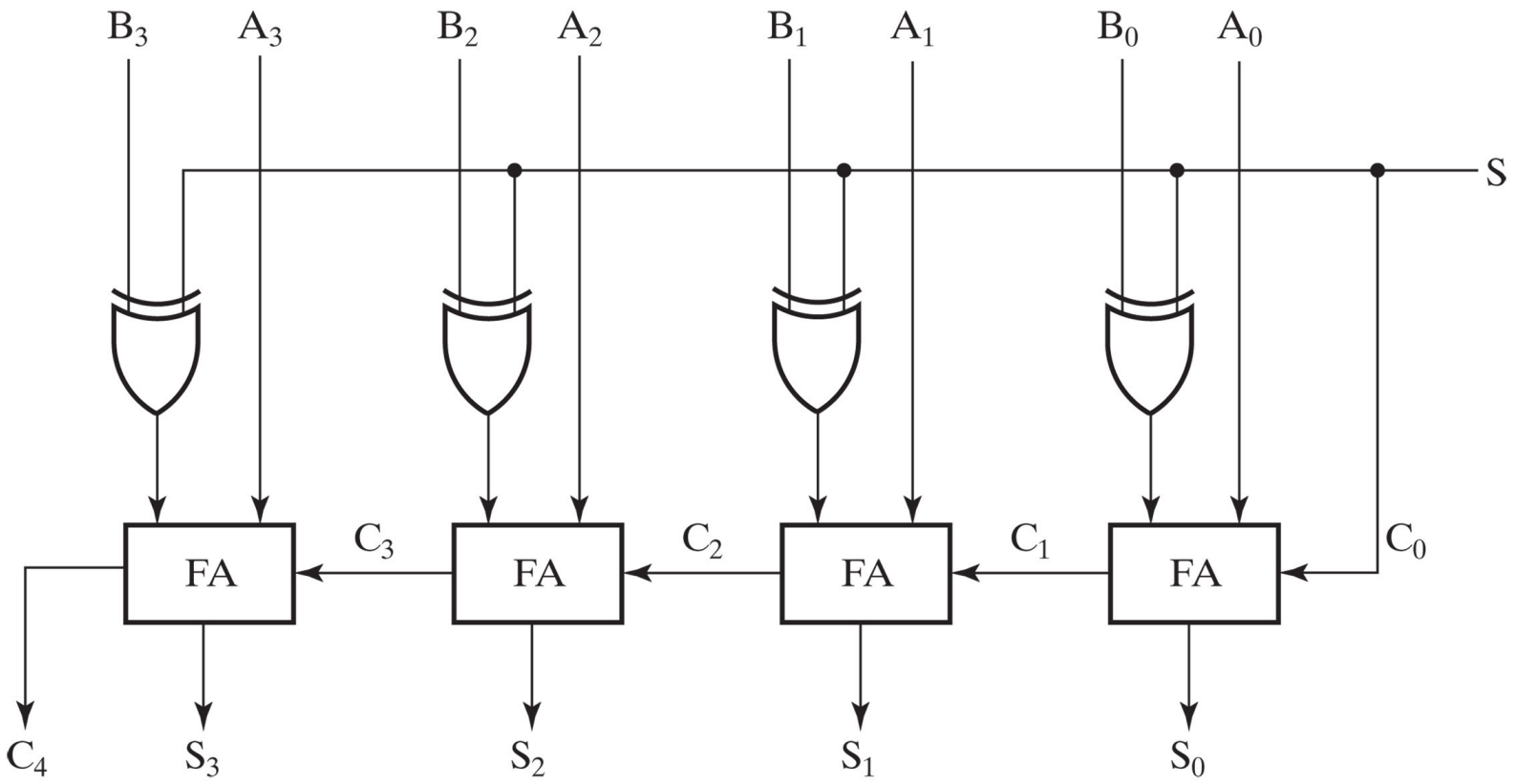
4-bit Subtractor Using 4-bit Adder



Overview

- Basic computational units
- Fixed number operations
  - Addition & Subtraction
- Arithmetic logic unit (ALU)
- Fixed number operations
  - Multiplication & Division

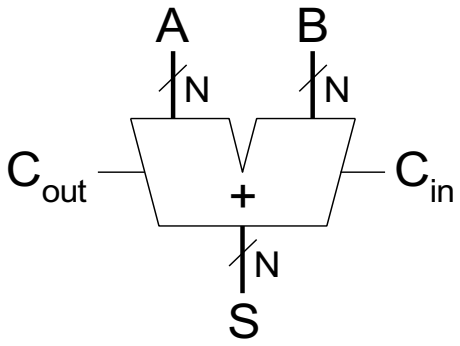
4-Bit Binary Adder-Subtractors

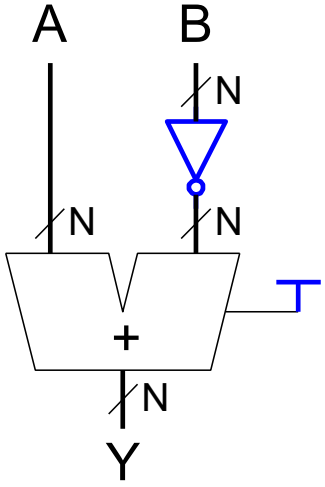


- When S=0: Addition (A+B)
- When S=1: Subtraction (A+2's complement of B)
- Can be used to add/subtract unsigned numbers and signed 2's complement numbers

Addition/Subtraction

- Both can be handled by using 2's complement representation
- Can achieve a unified implementation
  - Addition vs. subtraction
  - Unsigned vs. signed





- Corner cases shall be considered
  - Some important flags

Carry & Overflow

- Carry is important when…
  - Adding or subtracting ***unsigned integers***
  - Indicates that the unsigned sum is out of range
  - Either < 0 or > maximum unsigned n-bit value
- Overflow is important when…
  - Adding or subtracting ***signed integers***
  - Indicates that the signed sum is out of range

- Overflow occurs when?

Signed Overflow

- With two's complement and a 4-bit adder, for example, the largest representable decimal number is +7, and the smallest is -8.
- What if you try to compute 4 + 5, or (-4) + (-5)?

```
  0100   (+4)          1100   (-4)
+ 0101   (+5)        + 1011   (-5)
---------------      ---------------
 01001   (-7)         10111   (+7)
```

- We cannot just include the carry out to produce a five-digit result, as for unsigned addition. If we did, (-4) + (-5) would result in +23!
- Also, unlike the case with unsigned numbers, the carry out cannot be used to detect overflow, by itself
  - In the example on the left, the carry out is 0 but there is overflow.
  - Conversely, there are situations where the carry out is 1 but there is no overflow.

How to Detect Signed Overflow?

- The impact of carry and overflow

Expression	Result	Carry?	Overflow?	Correct Result?
0100(+4)+0010(+2)	0110(+6)	No	No	Yes
0100(+4)+0110(+6)	1010(−6)	No	Yes	No
1100(−4)+1110(−2)	1010(−6)	Yes	No	Yes
1100(−4)+1010(−6)	0110(+6)	Yes	Yes	No

Examples of four *signed* additions

- The easiest way to detect signed overflow is to look at all the sign bits.

```
  0100   (+4)
+ 0101   (+5)
-------------
 01001   (-7)
```

```
  1100   (-4)
+ 1011   (-5)
-------------
 10111   (+7)
```

Detecting Signed Overflow

- Overflow occurs only in the two situations:
  - Adding two positive numbers and the sum is negative
  - Adding two negative numbers and the sum is positive
  - Can happen because of the fixed number of sum bits
- Overflow cannot occur if you add a positive number to a negative number. Do you see why?

- In two's complement addition/subtraction
  - If the two numbers have the same sign bit and the sum/difference has a different sign bit, then overflow
  - Or, if the carry out flags of the sign bit and the highest value bit are different

Important Flags

- Zero flag (ZF)
  - ZF = 1 means the result is 0
  - Valid for both unsigned and signed operations
- Sign flag (SF/NF)
  - The sign of the result, i.e., S_{n-1}
  - Valid for signed operations
- Carry/borrow flag (CF)
  - If CF = 1
    - Carry for addition, i.e., C_{out}
    - Borrow for subtraction, i.e., $\sim C_{out}$
  - Valid for unsigned operations
- Overflow flag (OF)
  - Valid for signed operations

Adders with Flags

- ZF, SF, CF and OF

Symbol