Digital Logic Circuit (SE273 – Fall 2020)

Lecture 5: Arithmetic Functions

Jaesok Yu, Ph.D. (jaesok.yu@dgist.ac.kr)

Assistant Professor
Department of Robotics Engineering, DGIST

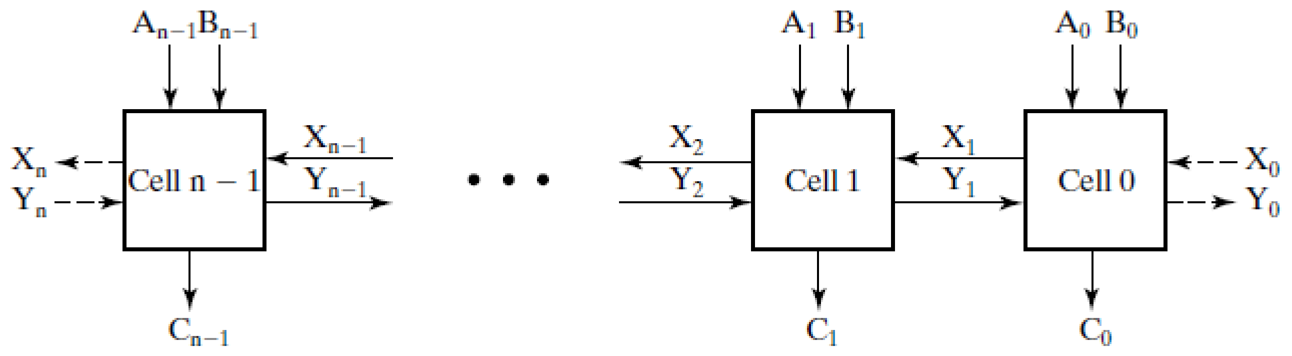
Goal

- Continue to focus on functional blocks
  - Specifically, functional blocks that perform arithmetic operations
  - Iterative circuits made up of arrays of combinational cells
  - Complement-based arithmetic (simplify the design)

Iterative Combinational Circuits

- The arithmetic blocks are designed to operate on binary input vectors and produce binary output vectors
  - The function implemented often requires the same subfunction be applied to each bit position
  - Thus, a simple functional block can be used repetitively for each bit position

[Array of cells]

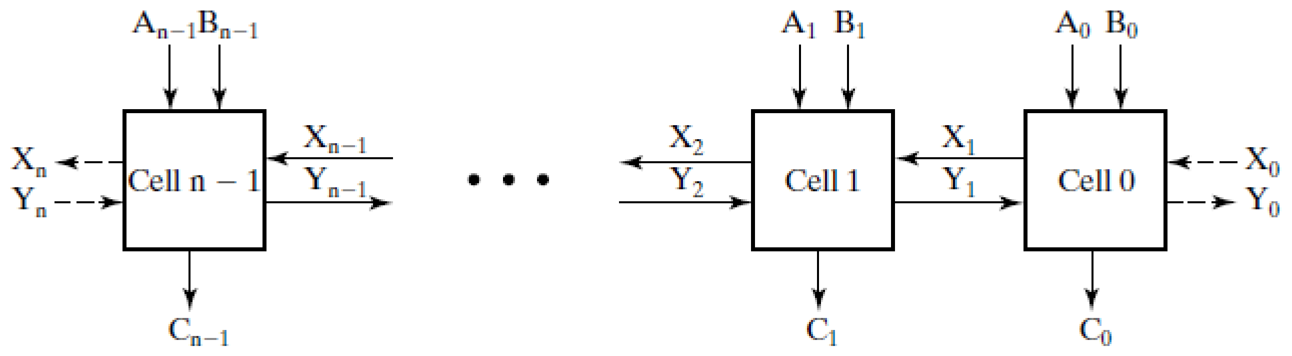


Also, called an iterative array

Iterative Combinational Circuits

- Iterative arrays are useful in handling vectors of bits
  - A circuit that adds two 32-bit binary integers (64 inputs and 32 outputs)
  - It is not easy to begin with truth tables and writing equations
  - Iterative circuits make the design process considerably simple

[Array of cells]



Binary Adders

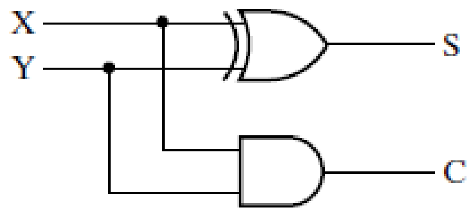
- We begin with the simple arithmetic unit that performs the addition of two binary digits
  - Four possible elementary operations: 0+0=0, 0+1=1, 1+0=1, 1+1=10
  - The case 1+1 requires two bits as an output (otherwise, 1bit is sufficient)
  - Thus, we define the carry and the sum
  - There are two types of binary adders: half adder and full adder

Half Adder

- A half adder is an arithmetic circuit that generates the sum of two binary digits (augend and addend)
  - We set two output bits: S for “sum” and C for “carry”
  - The Boolean functions for the two outputs are as follows:

Truth Table of Half Adder

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



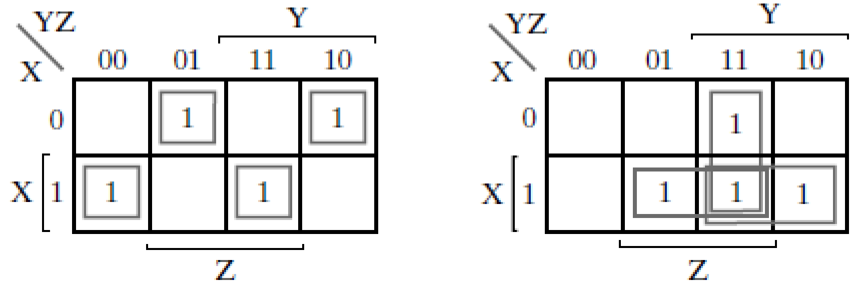
Logic diagram of half adder

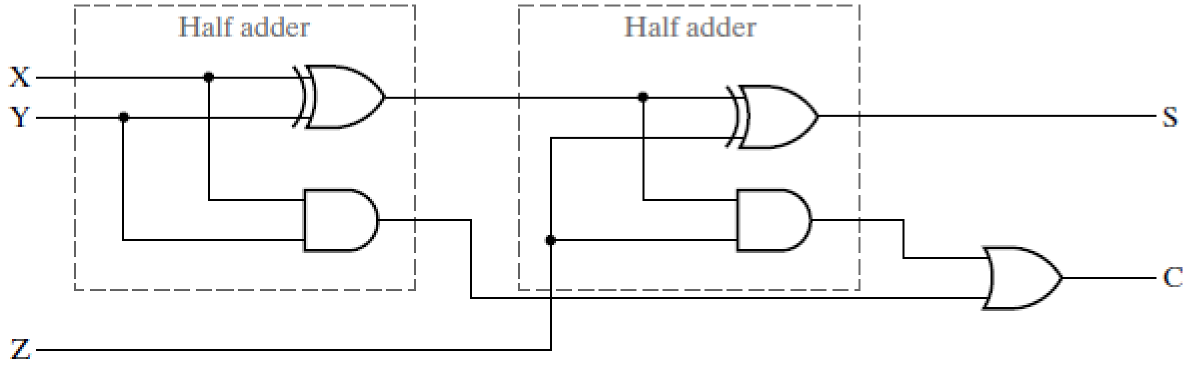
Full Adder

- A full adder is a circuit that forms the arithmetic sum of three input bits
  - Two of input variables are X and Y (two bits to be added)
  - The third input, Z, represents the carry from the previous lower significant position

Truth Table of Full Adder

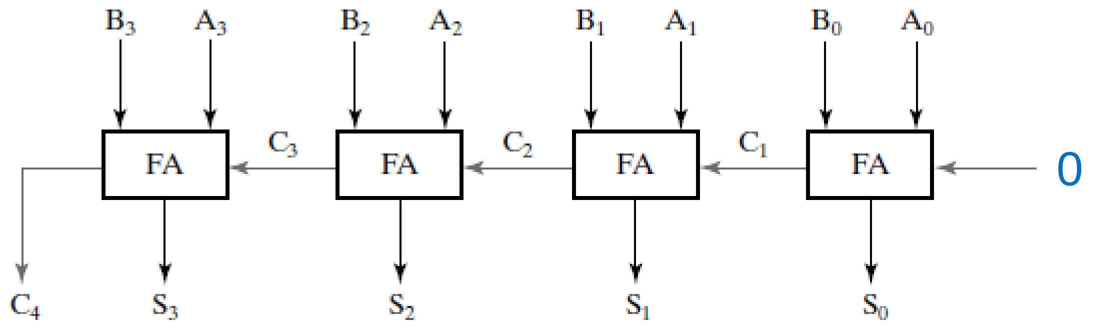
Inputs			Outputs	
X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



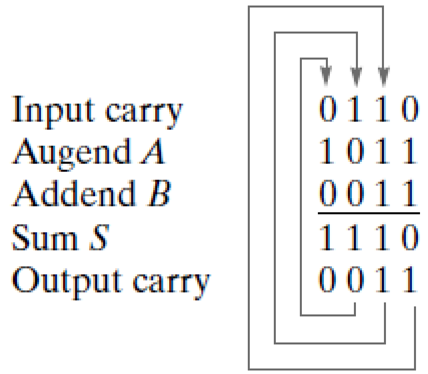


Binary Ripple Carry Adder

- A parallel binary adder is a digital circuit that produces the arithmetic sum of two (binary) numbers
  - It uses ‘n’ full adders in parallel, with all input bits applied simultaneously
  - They are connected in cascade, with **rippled carry** connections



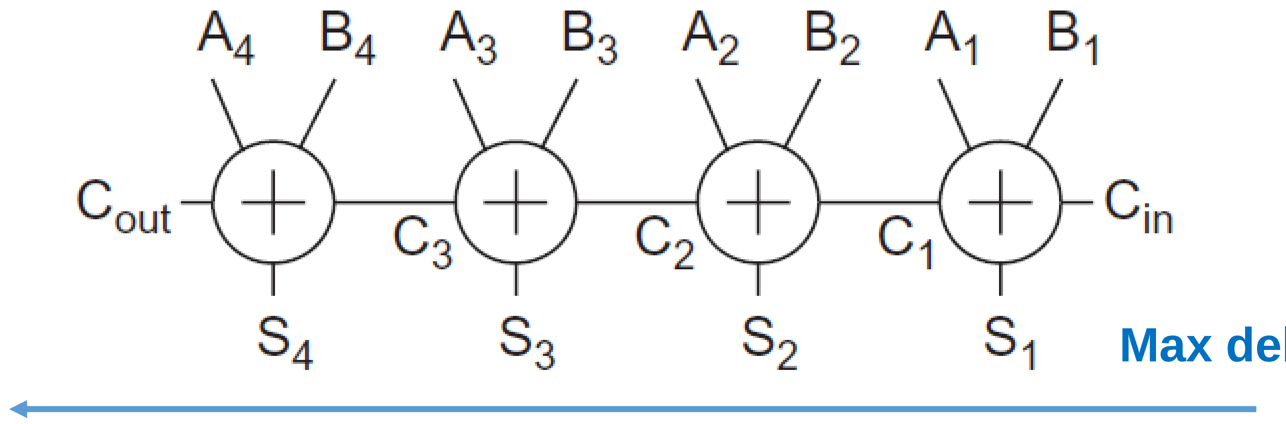
4-bit ripple carry adder (RCA)



Input carry	0 1 1 0
Augend *A*	1 0 1 1
Addend *B*	0 0 1 1
Sum *S*	1 1 1 0
Output carry	0 0 1 1

Drawback: Binary Ripple Carry Adder

- The simplest design of multi-bit adder is to connect carry-out of one bit to the carry-in to the next bit



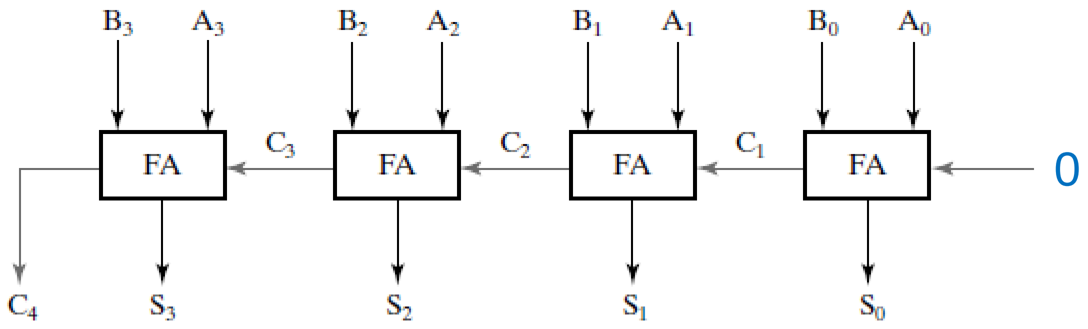
Max delay path (C_{in} to C_{out})

How to minimize?

Delay increases linearly with number of bits: $t_d = O(N)$

More on Adders

- Prior to moving to the design of efficient multipliers, let’s look at a couple of design choices for adders
  - Ripple Carry Adder (RCA): the simplest design!
  - Carry Lookahead Adder (CLA): it predicts the carry!
  - Carry Save Adder (CSA)

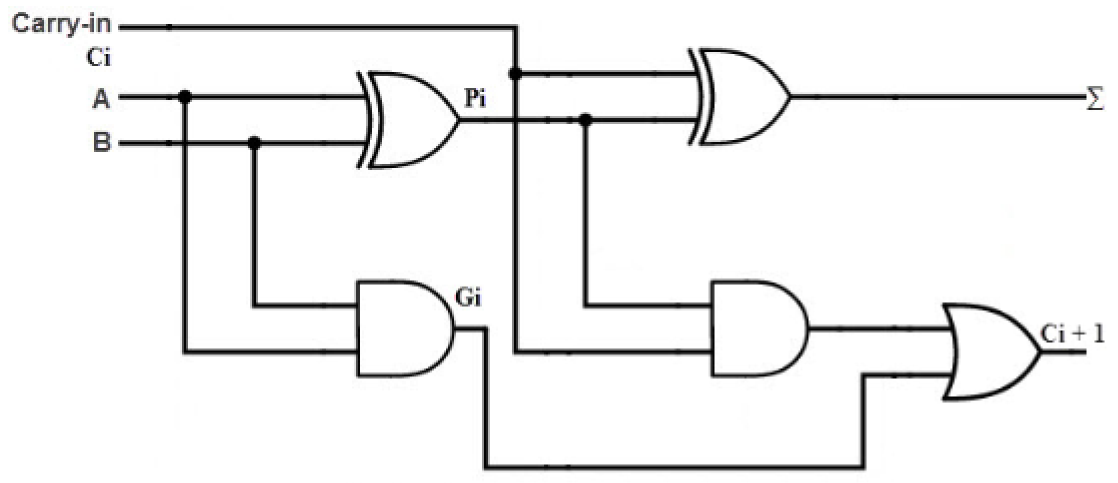


4-bit ripple carry adder (RCA)

Carry Lookahead Adder: Useful Definitions

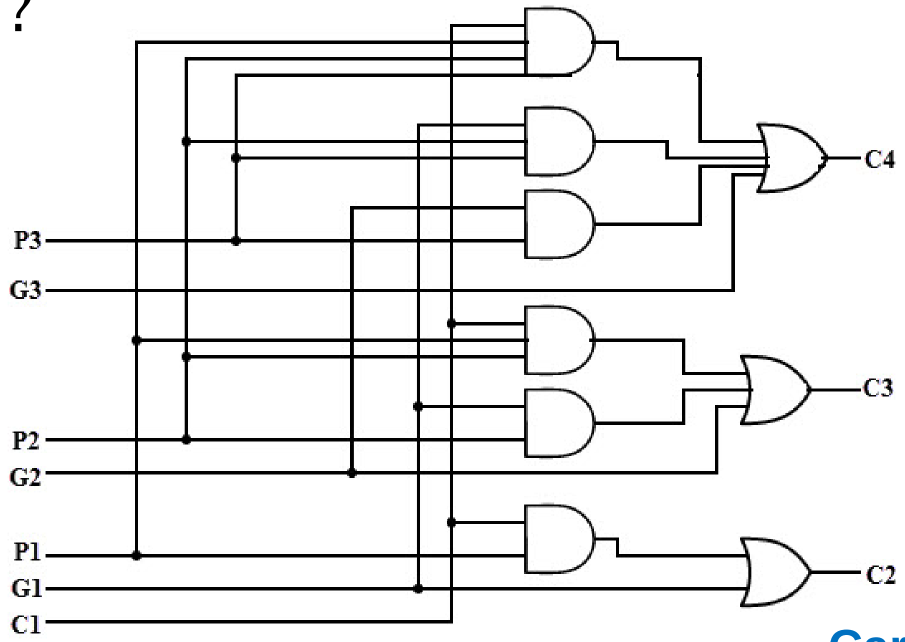
- A fast parallel adder as it reduces the propagation delay
- We first need to understand Carry **P**ropagate and **G**enerate signals
  - Propagate: $P_i = A_i \oplus B_i$
  - Generate: $G_i = A_i \cdot B_i$

A	B	C	G	P	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	0	1
		1			1	0
1	1	0	1	0	1	0
		1			1	1



Carry Lookahead Adder: C_{out} Boolean Function

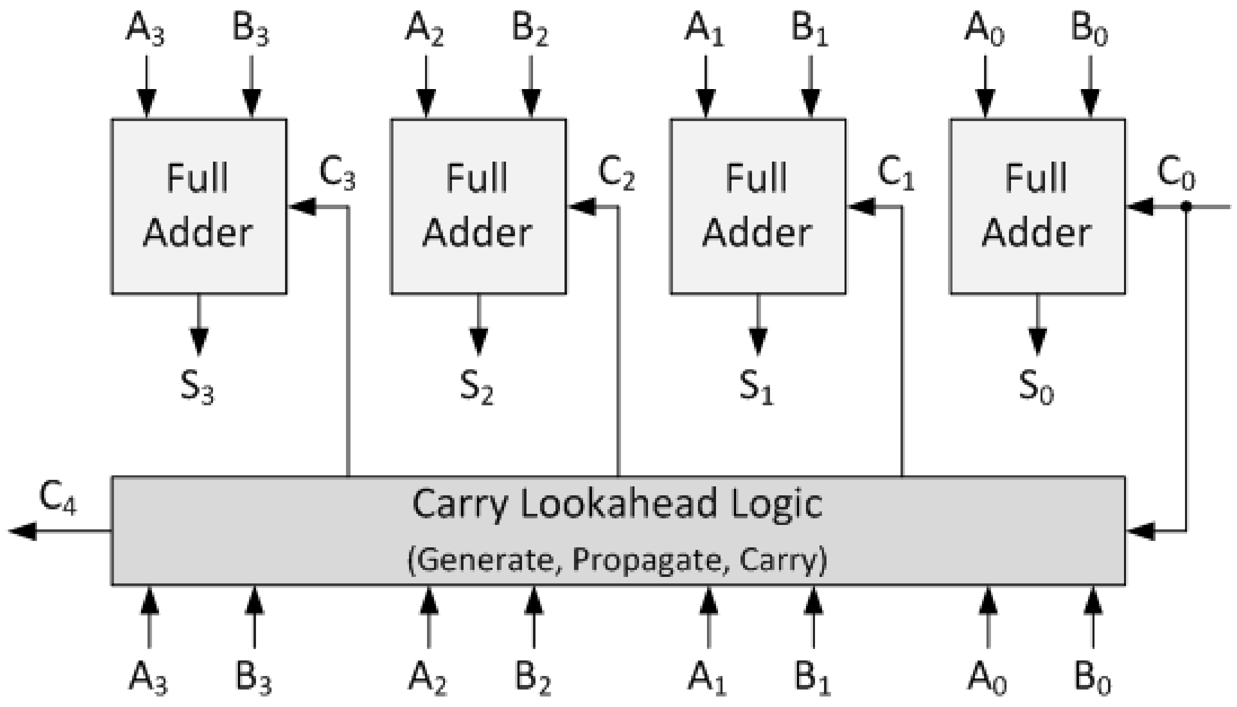
- The main idea is to predict internal carry signals by looking at only A_i and B_i (+ C_{in})
  - $C_1 = G_0 + P_0 \cdot C_{in}$
  - $C_2 = G_1 + P_1 \cdot C_1 = G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_{in}$
  - $C_3 = G_2 + P_2 \cdot C_2 = G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_{in}$
  - $C_4 = G_3 + P_3 \cdot C_3 = \ ?$



Carry generator for C_2, C_3, and C_4

4-bit Carry Lookahead Adder

- Now, the carry at intermediate nodes can be computed even without waiting for previous FAs complete their computations



Carry Save Adder (CSA)

- In CSA, three bits are added in parallel at a time
- Each CSA is simply an independent Full Adder without Carry propagation
- A Parallel Adder is required only at the last stage

```
X:      1 0 0 1 1
Y:  +   1 1 0 0 1
Z:  +   0 1 0 1 1
       ↙ ↙ ↙ ↙ ↙
-----------------
C:    1 1 0 1 1
```

```
X:      1 0 0 1 1
Y:  +   1 1 0 0 1
Z:  +   0 1 0 1 1
-----------------
S:      0 0 0 0 1
```

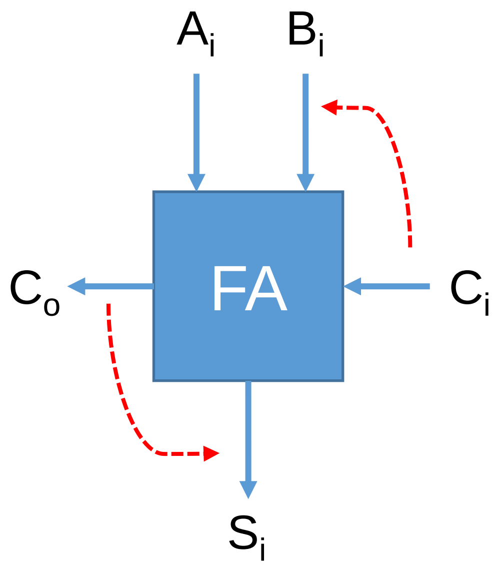
A set of full adders generate Carry and Sum bits in parallel

```
X:      1 0 0 1 1
Y:  +   1 1 0 0 1
Z:  +   0 1 0 1 1
-----------------
S:      0 0 0 0 1
C: +1 1 0 1 1
-----------------
      1 1 0 1 1 1
```

The Sum and Carry vectors are added at the last stage (with proper shifting)

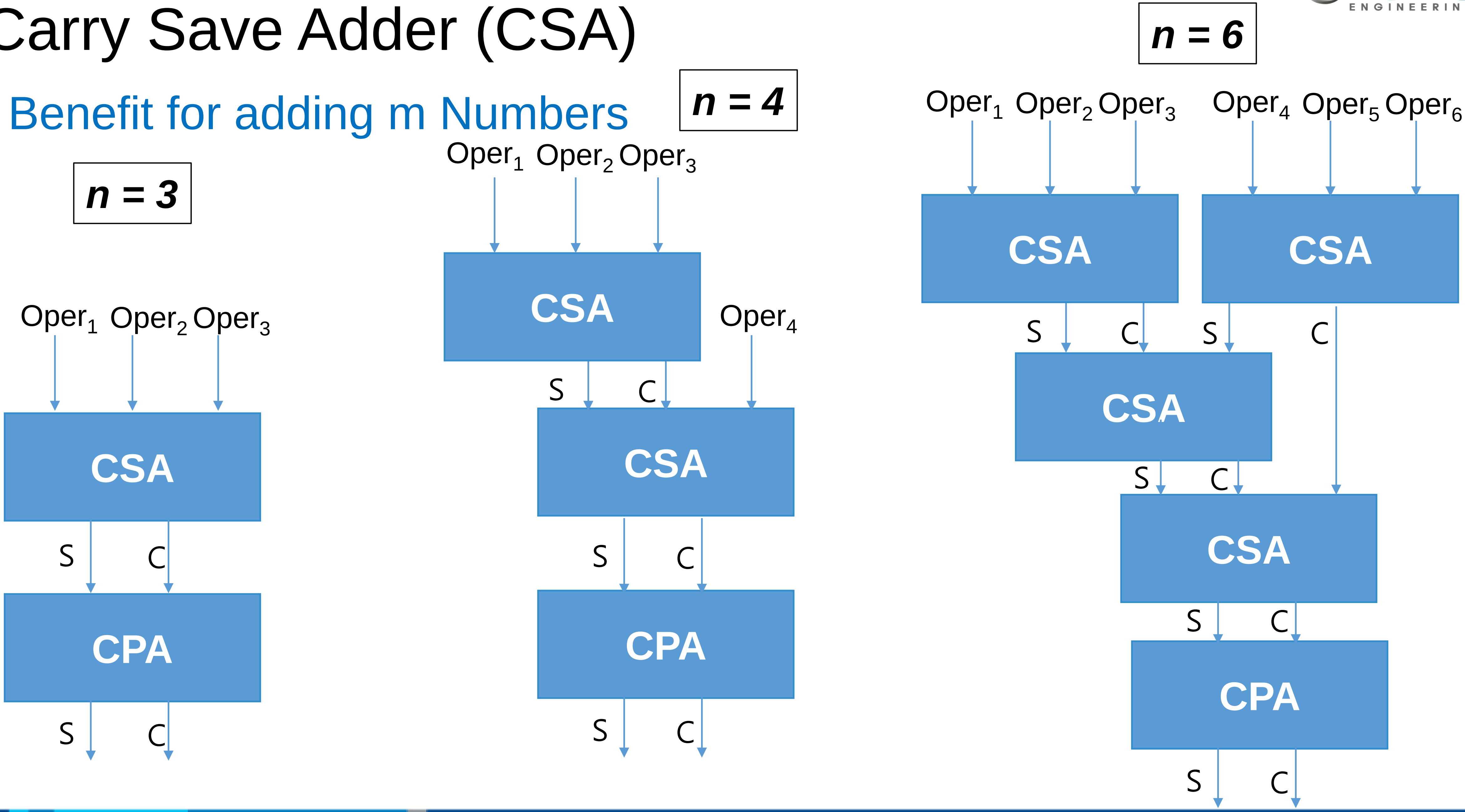
Carry Save Adder (CSA)

- In CSA, three bits are added in parallel at a time
  - The carry is not propagated through stages



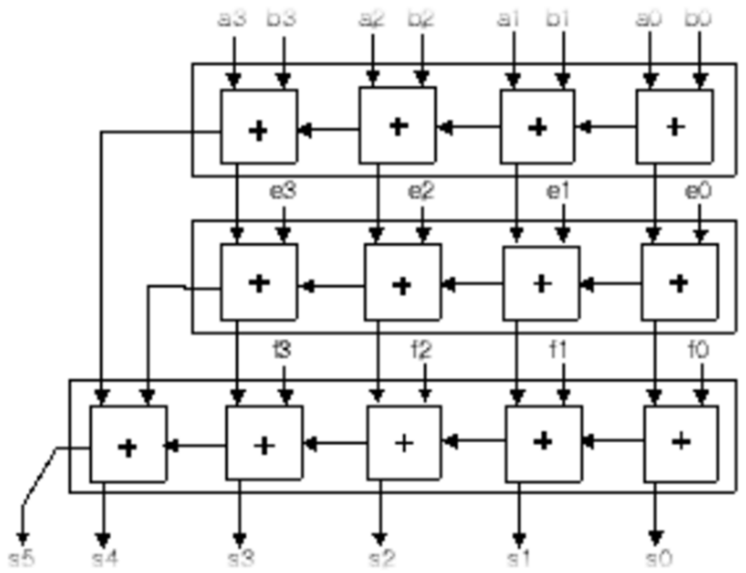
Carry Save Adder (CSA)

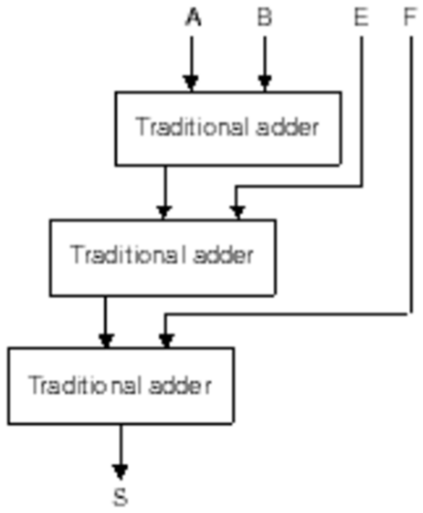
- Benefit for adding m Numbers

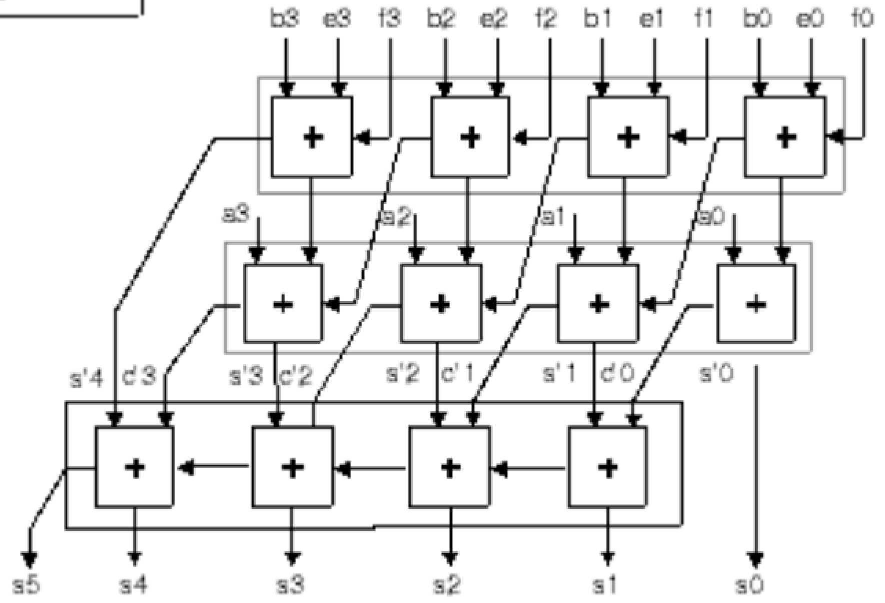


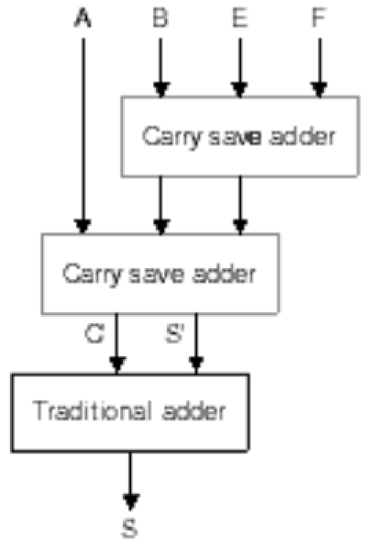
Critical Path Delay

- Let’s briefly compare critical path delay RCA and CSA



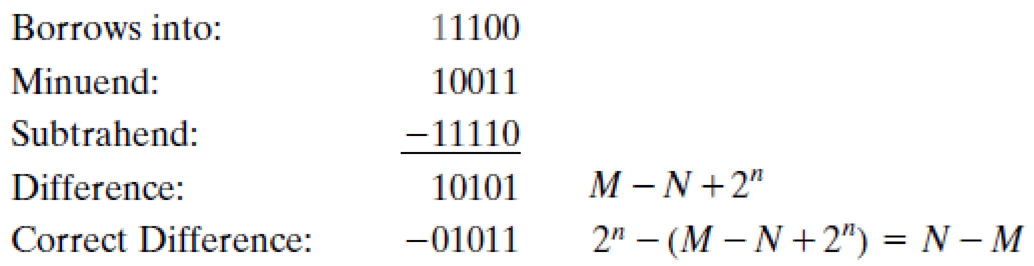



b3 e3 f3 b2 e2 f2 b1 e1 f1 b0 e0 f0
a3 a2 a1 a0
s'4 c'3 s'3 c'2 s'2 c'1 s'1 c'0 s'0
s5 s4 s3 s2 s1 s0



Binary Subtraction

- Let's consider the subtraction of unsigned binary numbers
  - It is used in signed-magnitude addition and subtraction algorithms
  - Later we will compare it with the complement arithmetic



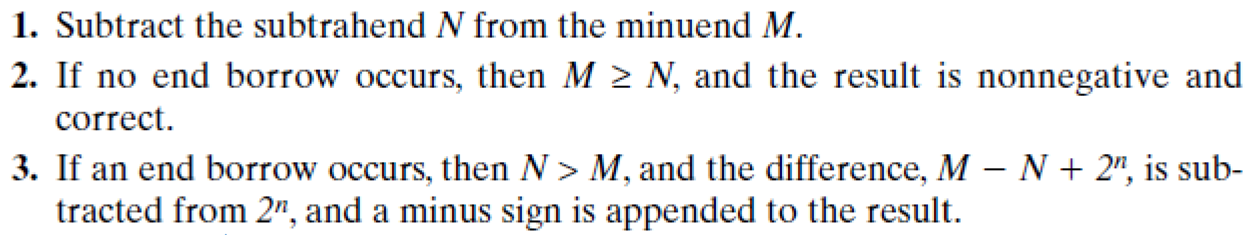
Borrows into:	11100	
Minuend:	10011	
Subtrahend:	−11110	
Difference:	10101	$M - N + 2^n$
Correct Difference:	−01011	$2^n - (M - N + 2^n) = N - M$

If **borrow occurs** into the MSB, it means that **minuend is smaller** than subtrahend

Then, the result must be negative!

Two’s Complement (Subtraction in Binary System)

- Subtraction of a binary number from 2^n to obtain an n-digit result is called taking the 2s complement of the number
  - The subtraction of two n-digit numbers in base 2 is as follows:



1. Subtract the subtrahend N from the minuend M.
2. If no end borrow occurs, then $M \geq N$, and the result is nonnegative and correct.
3. If an end borrow occurs, then $N > M$, and the difference, $M - N + 2^n$, is subtracted from 2^n, and a minus sign is appended to the result.

Taking 2s complement of $M\text{-}N\text{+}2^n$

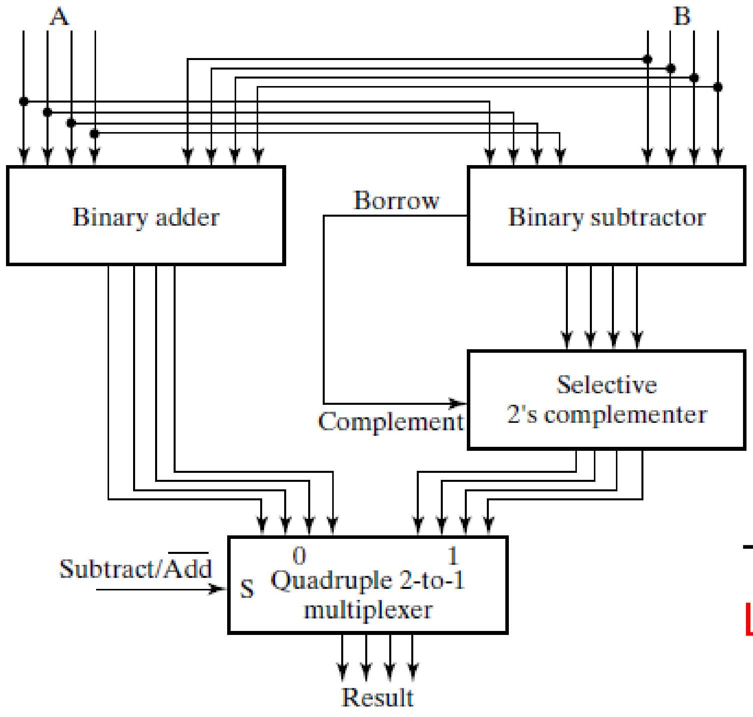
Example: Unsigned Binary Subtraction

- Perform binary subtraction 01100100 - 10010110

Borrows into:	
Minuend:	01100100
Subtrahend:	− 10010110
Initial Result	

Binary Adder-Subtractor

- As a general arithmetic unit, we can design binary adder-subtractor
  - The inputs are applied to both adder and subtractor
  - Both operations are performed in parallel



If an end borrow value of 1 occurs in subtraction, then the selective 2’s complementer is activated

This circuit is more complex than necessary
Later, we will see how to share logic btw adder and subtractor

Complements

- There are two types of complements for base-r system
  - The radix complement: r's complement
  - The diminished radix complement: (r-1)'s complement
  - For binary number system, 2's complement and 1's complement exist

1's complement

$$(2^n - 1) - N$$

→ Subtract each digit from 1

The 1's complement of 1011001 is 0100110.
The 1's complement of 0001111 is 1110000.

2's complement

$$2^n - N \; for \; N \neq 0$$

$$0 \; for \; N = 0$$

→ Add 1 to the 1's complement

Subtraction Using 2s Complement

- Now, let's perform subtraction armed with complements
  - Add 2s complement of the subtrahend (N) to minuend (M)
    - $M + (2^n - N) = M - N + 2^n$
  - If $M \geq N$, the sum produces an end carry 2^n. Discard the end carry.
  - If $M < N$, the sum does not produce an end carry. Perform a correction, taking the 2s complement of the sum and placing a minus sign

X = 1010100
Y = 1000011

X - Y

$$\begin{array}{rr}
X = & 1010100 \\
\text{2s complement of } Y = & 0111101 \\
\text{Sum} = & 10010001 \\
\text{Discard end carry } 2^7 = & -\underline{10000000} \\
\textit{Answer: } X - Y = & 0010001
\end{array}$$

Y - X

$$\begin{array}{rr}
\text{2s complement of } X = & \underline{0101100} \\
\text{Sum} = & 1101111
\end{array}$$

There is no end carry.

Answer: $Y - X = -(\text{2s complement of } 1101111) = -0010001$

Recap: 2’s complement

- Representation of signed integer
- ex) Representation of “-7”

$$0111 \longrightarrow 1000 \longrightarrow 1001$$

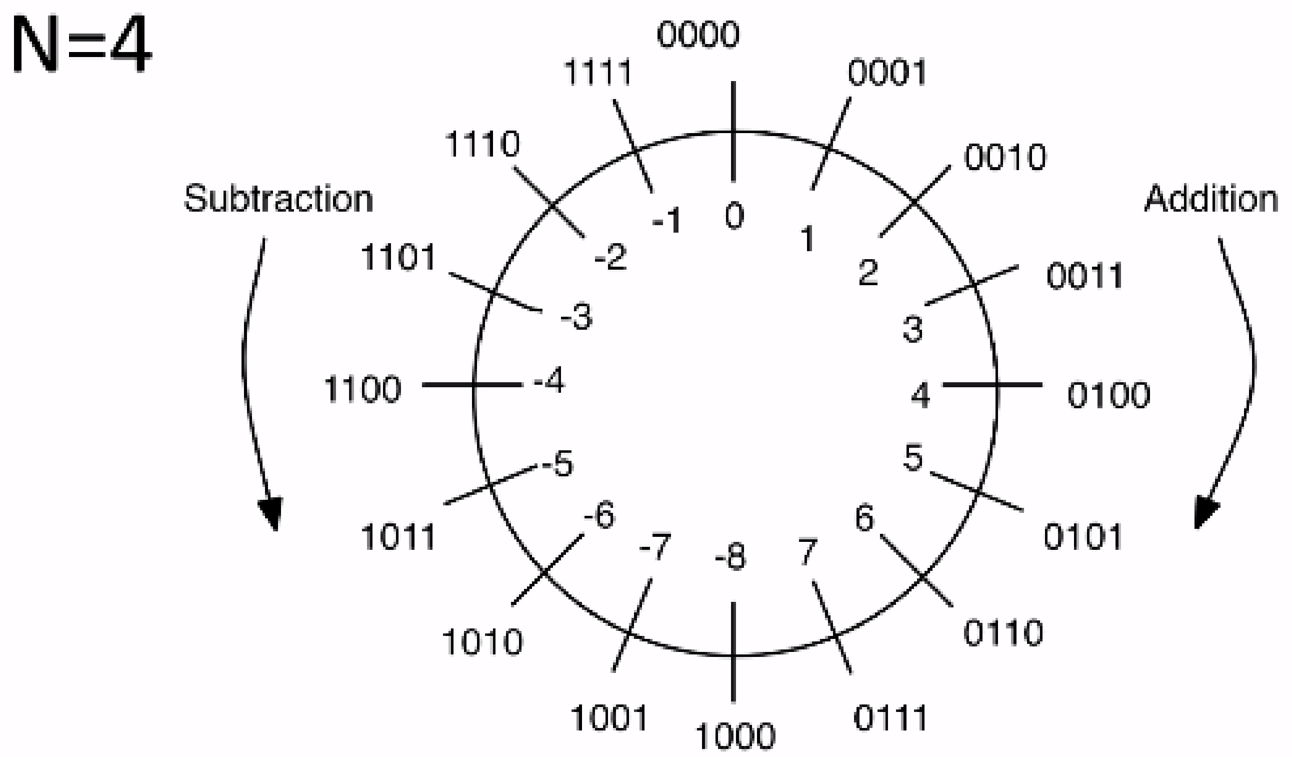
Step 1: Positive number (= 7)	Step 2: Take a 1’s complement (Invert each bit)	Step 3: +1 (if negative) 2’s Complement
0111	1000	1001

Bit pattern:	1000	1001	1010	1011	1100	1101	1110	1111	0000	0001	0010	0011	0100	0101	0110	0111
1’s complement:	-7	-6	-5	-4	-3	-2	-1	0	0	1	2	3	4	5	6	7
2’s complement:	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7

- Why? > Faster (in perspective of the hardware)

Recap: 2’s complement

- Representation of signed integer



Bit pattern:	1000	1001	1010	1011	1100	1101	1110	1111	0000	0001	0010	0011	0100	0101	0110	0111
1’s complement:	-7	-6	-5	-4	-3	-2	-1	0	0	1	2	3	4	5	6	7
2’s complement:	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7

Recap: 2’s complement

- ex) Subtraction 6-3

Case 1: 6 - 3

```
    0 1 1 0
-   0 0 1 1
-----------
    0 0 1 1
```

Case 2: 6+(-3) by 2’s complement

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

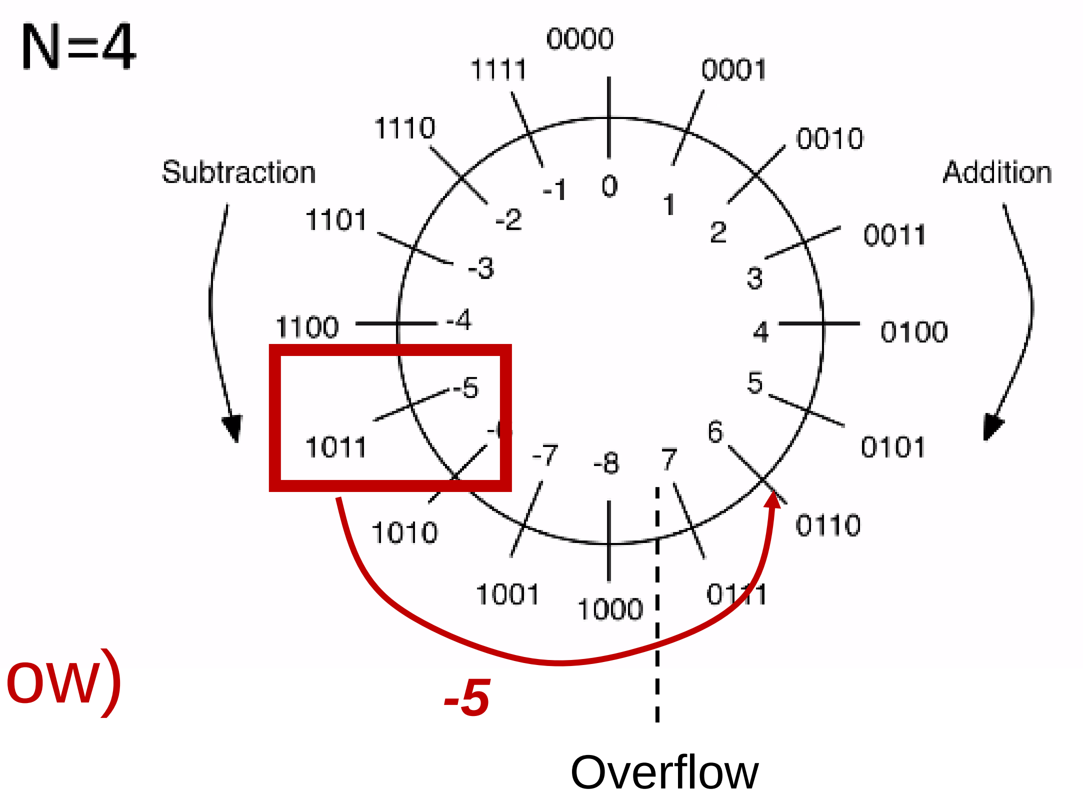
Recap: 2’s complement

- ex) Overflow in subtraction

Case 1: -5 - 5

```
     1 0 1 1    -5
+    1 0 1 1    -5
------------------
   1 0 1 1 0    6 (Overflow)
```

Overflow



Signed Binary Addition Using 2s Complement

- It does not require comparison or subtraction!
  - Simply add two numbers in signed 2s complement form, and discard any carry out if happens

$$
\begin{array}{rr}
+6 & 00000110 \\
+\underline{13} & \underline{00001101} \\
+19 & 00010011
\end{array}
\qquad
\begin{array}{rr}
-6 & 11111010 \\
+\underline{13} & \underline{00001101} \\
+7 & 00000111
\end{array}
\qquad
\begin{array}{rr}
+6 & 00000110 \\
-\underline{13} & \underline{11110011} \\
-7 & 11111001
\end{array}
\qquad
\begin{array}{rr}
-6 & 11111010 \\
-\underline{13} & \underline{11110011} \\
-19 & 11101101
\end{array}
$$

Signed Binary Subtraction Using 2s Complement

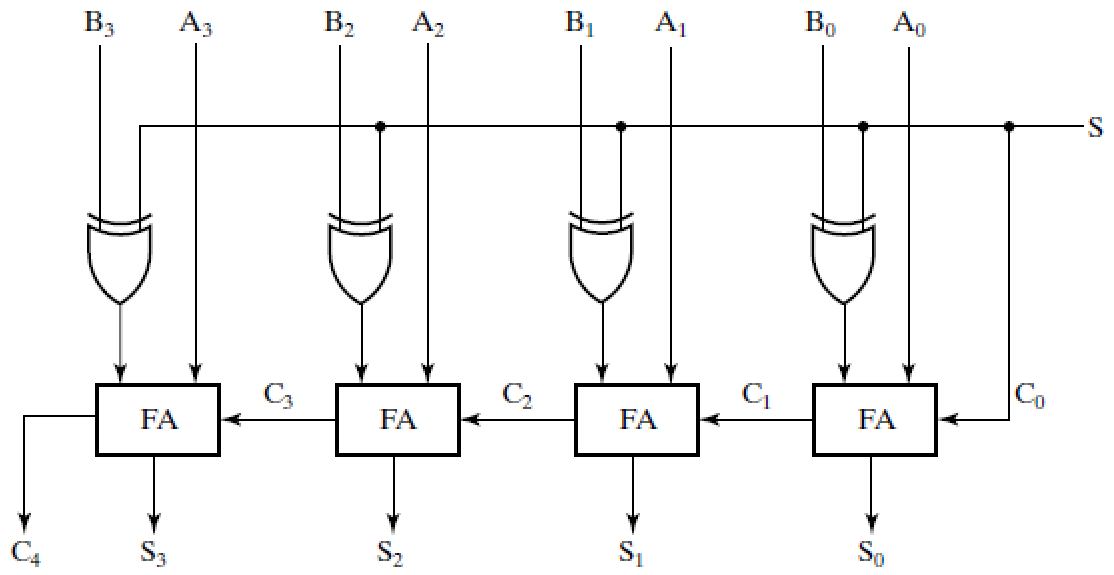
- The subtraction is simple as well
  - Take 2s complement of the subtrahend (including sign bit)
  - Add it to the minuend (including sign bit)
  - A carry out of the sign bit position is discarded

$$
\begin{array}{r}
-\ 6 \\
-\ (-13) \\
\hline
+\ 7
\end{array}
\qquad
\begin{array}{r}
11111010 \\
-\ 11110011 \\
\hline
\\
\end{array}
\qquad
\begin{array}{r}
11111010 \\
+\ 00001101 \\
\hline
00000111
\end{array}
\qquad
\begin{array}{r}
+\ 6 \\
-\ (-13) \\
\hline
+\ 19
\end{array}
\qquad
\begin{array}{r}
00000110 \\
-\ 11110011 \\
\hline
\\
\end{array}
\qquad
\begin{array}{r}
00000110 \\
+\ 00001101 \\
\hline
00010011
\end{array}
$$

Then how can we design the adder-subtractor for signed binary numbers (in 2s complement)?

Simplified Binary Adder-Subtractor

- Using 2s complement, we can eliminate the subtraction operation
  - Only the complementer and an adder are needed
  - When performing subtraction, we do complement on the subtrahend



For **subtraction**,
we need inverters placed btw each B
and the adder input, then add 1

What is the difference btw $A \geq B$ and $A < B$?

Signed Binary Numbers

- So far, we considered arithmetic operations on unsigned numbers
  - To represent negative integers, we need a notation for negative values
  - Conventionally, ‘0’ is for positive numbers and ‘1’ is for negative numbers

0	1	0	0	1

Unsigned binary: 9
Signed binary: +9

Signed-magnitude system!

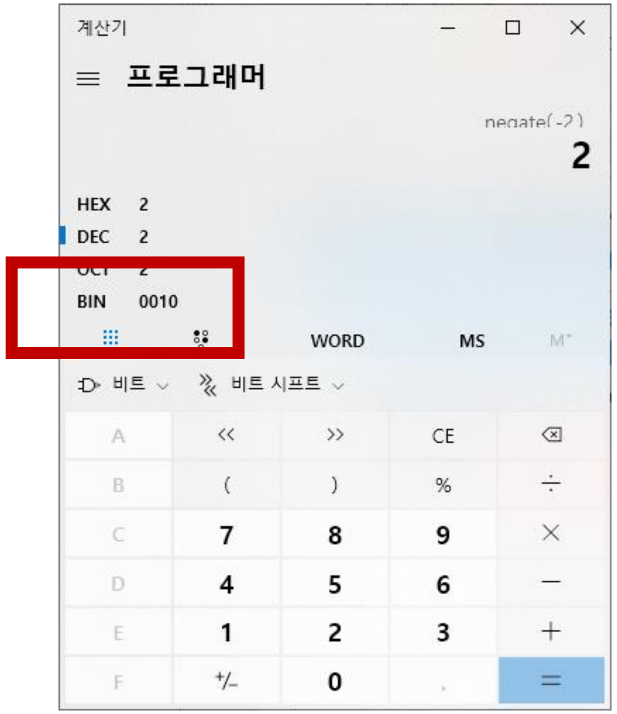
Signed-Magnitude Addition and Subtraction

- For n-bit numbers, we separate the single sign bit and (n-1) magnitude bits for processing
  - Magnitude bits are processed as unsigned binary numbers
  - To avoid correction step in subtraction, we use a signed-complement system
  - Either 1s or **2s complement** can be used

Decimal	Signed 2s Complement	Signed Magnitude
+7	0111	0111
+6	0110	0110
+5	0101	0101
+4	0100	0100
+3	0011	0011
+2	0010	0010
+1	0001	0001
+0	0000	0000
−0	—	1000
−1	1111	1001
−2	1110	1010
−3	1101	1011
−4	1100	1100
−5	1011	1101
−6	1010	1110
−7	1001	1111
−8	1000	—

Note on Signed Number System

- It is used in ordinary arithmetic, but is awkward when employed in computer arithmetic
  - Mainly due to separate handling of the sign and the correction step for subtraction
  - Thus, the **signed complement is normally used** in computers

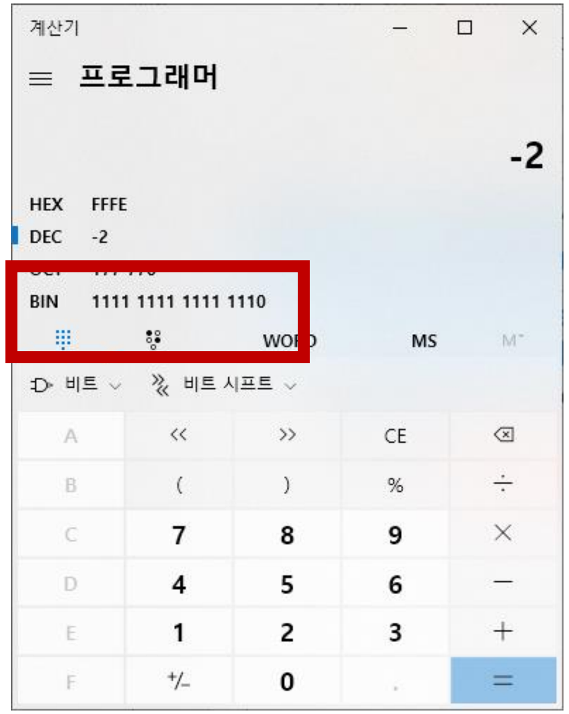

계산기

프로그래머

negate(-2)

2

HEX 2
DEC 2
OCT 2
BIN 0010


계산기

프로그래머

-2

HEX FFFE
DEC -2
BIN 1111 1111 1111 1110

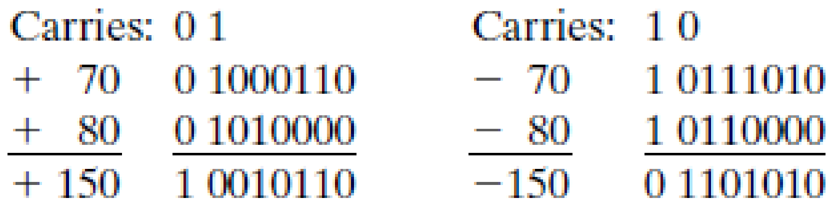
2’s complement representation

Overflow

- To obtain the correct answer, we must ensure the result has a sufficient number of bits to accommodate the sum
  - If we add two n-bit numbers, the sum may occupy n+1 bits (overflow!!)
  - In computers, the number of bits that holds the result is fixed
  - Thus, we need to detect and handle the overflow

Overflow in Binary Numbers

- For unsigned numbers,
  - An overflow is detected from the end carry out of the MSB (for addition only)
- For signed numbers,
  - An overflow may occur if two numbers added are both positive or negative
  - Also, overflow may occur for either addition or subtraction

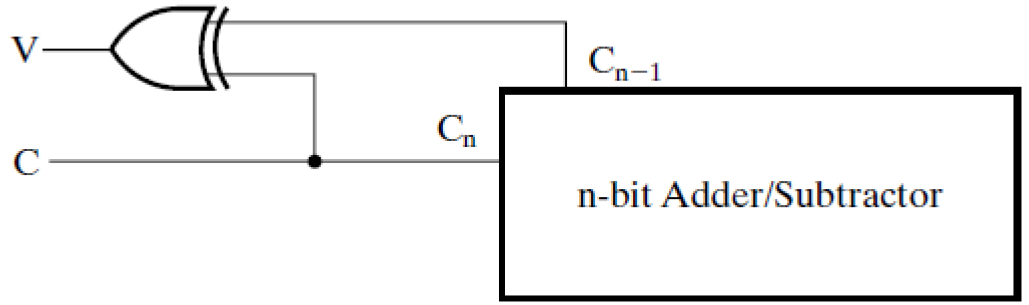


Carries:	0 1		Carries:	1 0
+ 70	0 1000110		− 70	1 0111010
+ 80	0 1010000		− 80	1 0110000
+ 150	1 0010110		−150	0 1101010

Overflow Detection

- An overflow can be detected by observing the carry into the sign bit position and the carry out of the sign bit position
  - If these two carries do not match, an overflow has occurred
  - XOR gate can be used!

Carries:	0 1		Carries:	1 0
+ 70	0 1000110		− 70	1 0111010
+ 80	0 1010000		− 80	1 0110000
+ 150	1 0010110		−150	0 1101010



What is the use of ‘V’ and ‘C’??