Binary Arithmetic

COE 233

Digital Logic and Computer Organization

Dr. Muhamed Mudawar

King Fahd University of Petroleum and Minerals

Presentation Outline

- Binary and Hexadecimal Addition and Subtraction
- Binary Multiplication and Bit Shifting
- Signed Integers
- Range, Overflow, Converting Subtraction into Addition

Adding Bits

- 1 + 1 = 2, but 2 should be represented as $(10)_2$ in binary
- Adding two bits: the sum bit is S and the carry bit is C

X	0	0	1	1
+ Y	+ 0	+ 1	+ 0	+ 1
C S	0 0	0 1	0 1	1 0

- Adding three bits: the sum bit is S and the carry bit is C

0	0	0	0	1	1	1	1
0	0	1	1	0	0	1	1
+ 0	+ 1	+ 0	+ 1	+ 0	+ 1	+ 0	+ 1
0 0	0 1	0 1	1 0	0 1	1 0	1 0	1 1

Binary Addition

- ❖ Start with the least significant bit (rightmost bit)
- ❖ Add each pair of bits
- ❖ Include the carry in the addition, if present

carry		**1**	**1**	**1**	**1**				
	0	0	1	1	0	1	1	0	(54)
+	0	0	0	1	1	1	0	1	(29)
	0	1	0	1	0	0	1	1	(83)
bit position:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	

Subtracting Bits

- Subtracting 2 bits (X – Y): we get the difference (D) and the **borrow-out** (B) shown as 0 or -1

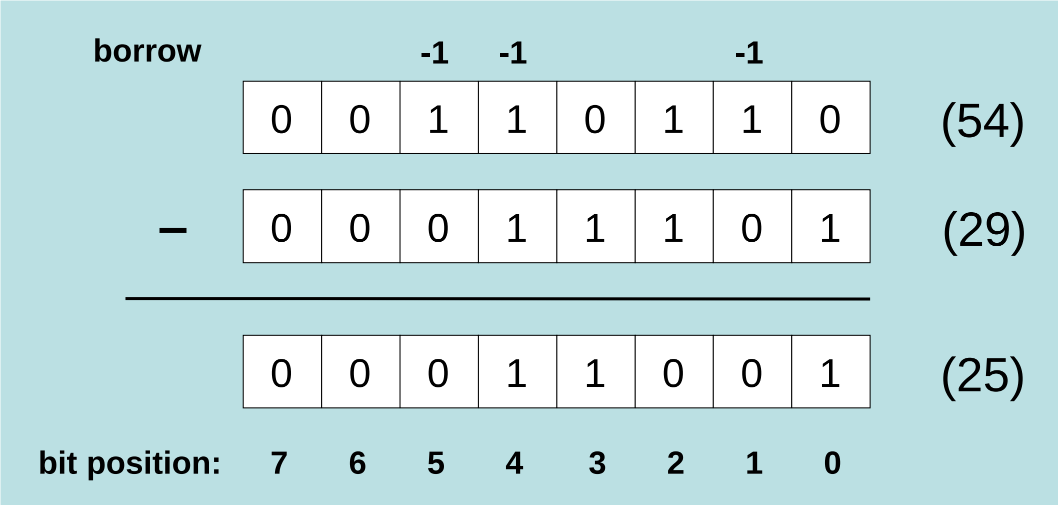
X	**0**	**0**	**1**	**1**
– Y	**– 0**	**– 1**	**– 0**	**– 1**
B D	**0 0**	**-1 1**	**0 1**	**0 0**

- Subtracting two bits (X – Y) with a **borrow-in = -1**: we get the difference (D) and the **borrow-out** (B)

borrow-in	**-1**	**-1**	**-1**	**-1**	**-1**
	X	**0**	**0**	**1**	**1**
	– Y	**– 0**	**– 1**	**– 0**	**– 1**
	B D	**-1 1**	**-1 0**	**0 0**	**-1 1**

Binary Subtraction

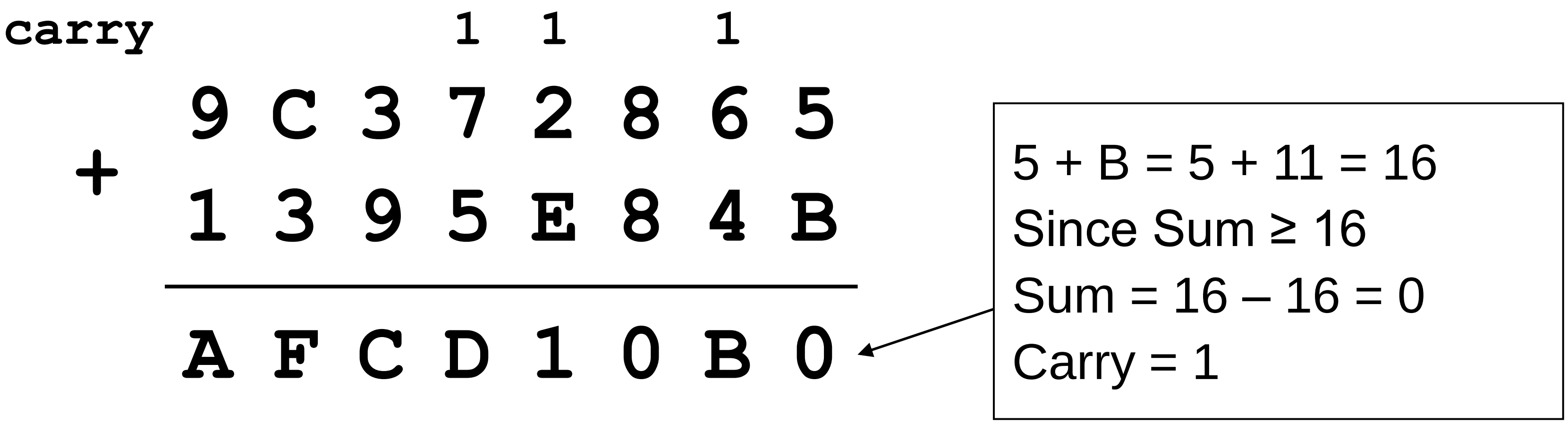
- ❖ Start with the least significant bit (rightmost bit)
- ❖ Subtract each pair of bits
- ❖ Include the borrow in the subtraction, if present



borrow			**-1**	**-1**			**-1**		
	0	0	1	1	0	1	1	0	(54)
–	0	0	0	1	1	1	0	1	(29)
	0	0	0	1	1	0	0	1	(25)
bit position:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	

Hexadecimal Addition

- Start with the least significant hexadecimal digits
- Let Sum = summation of two hex digits
- If Sum is greater than or equal to 16
  - Sum = Sum – 16 and Carry = 1
- Example:



```
carry       1 1   1
      9 C 3 7 2 8 6 5
    + 1 3 9 5 E 8 4 B
      ---------------
      A F C D 1 0 B 0
```

5 + B = 5 + 11 = 16
Since Sum ≥ 16
Sum = 16 – 16 = 0
Carry = 1

Hexadecimal Subtraction

- Start with the least significant hexadecimal digits
- Let Difference = subtraction of two hex digits
- If Difference is negative
  - Difference = 16 + Difference and Borrow = -1
- Example:

```
borrow   -1  -1    -1
        9 C 3 7 2 8 6 5
    -   1 3 9 5 E 8 4 B
       -----------------
        8 8 A 1 4 0 1 A
```

Since 5 < B, Difference < 0
Difference = 16+5–11 = 10
Borrow = -1

Next . . .

- Binary and Hexadecimal Addition and Subtraction
- Binary Multiplication and Bit Shifting
- Signed Integers
- Range, Overflow, Converting Subtraction into Addition

Binary Multiplication

- Binary Multiplication table is simple:

  `0×0=0,    0×1=0,    1×0=0,    1×1=1`

$$\begin{array}{lrl}
\text{Multiplicand} & 1100_2 & = 12 \\
\text{Multiplier} & \times\quad 1101_2 & = 13 \\
\hline
 & 1100 & \\
 & 0000 & \\
 & 1100 & \\
 & 1100 & \\
\hline
\text{Product} & 10011100_2 & = 156
\end{array}$$

Binary multiplication is easy

0 × multiplicand = 0

1 × multiplicand = multiplicand

- *n*-bit multiplicand × *n*-bit multiplier = 2*n*-bit product
- Accomplished via shifting and addition

Shifting the Bits to the Left

- What happens if the bits are shifted to the left by 1 bit position?

Before	0	0	0	0	0	1	0	1	= 5
After	0	0	0	0	1	0	1	0	= 10

Multiplication

By 2

- What happens if the bits are shifted to the left by 2 bit positions?

Before	0	0	0	0	0	1	0	1	= 5
After	0	0	0	1	0	1	0	0	= 20

Multiplication

By 4

- Shifting the Bits to the Left by *n* bit positions is multiplication by 2^n
- As long as we have sufficient space to store the bits

Shifting the Bits to the Right

- What happens if the bits are shifted to the right by 1 bit position?

Before	0	0	1	0	0	1	1	**0**	= 38
After	0	0	0	1	0	0	1	1	= 19, **r=0**

Division By 2

- What happens if the bits are shifted to the right by 2 bit positions?

Before	0	0	1	0	0	1	**1**	**0**	= 38
After	0	0	0	0	1	0	0	1	= 9, **r=2**

Division By 4

- Shifting the Bits to the Right by *n* bit positions is division by 2^n
- The **remainder r** is the value of the bits that are **shifted out**

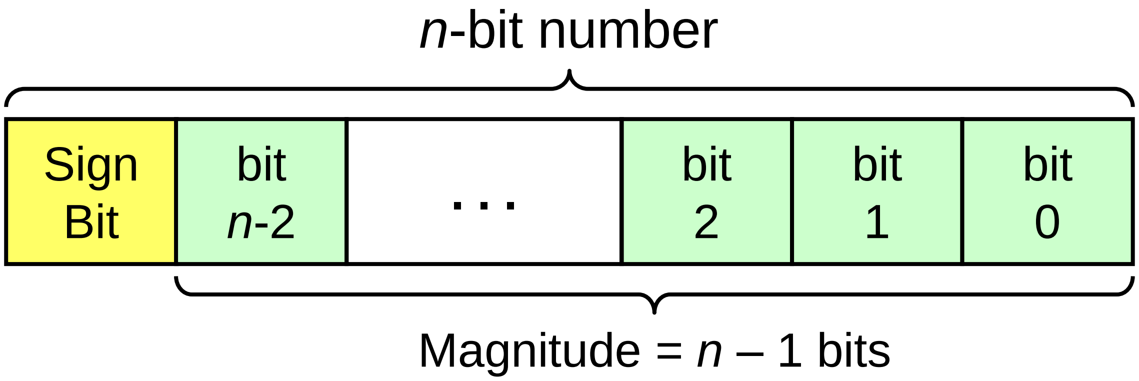
Next . . .

- Binary and Hexadecimal Addition and Subtraction
- Binary Multiplication and Bit Shifting
- Signed Integers
- Range, Overflow, Converting Subtraction into Addition

Signed Integers

- Several ways to represent a signed integer
  - Sign-Magnitude
  - 1's complement
  - 2's complement
- Divide the range of values into two parts
  - First part corresponds to the positive numbers (≥ 0)
  - Second part correspond to the negative numbers (< 0)
- The 2's complement representation is widely used
  - Has many advantages over other representations

Sign-Magnitude Representation


n-bit number

Sign Bit	bit *n*-2	. . .	bit 2	bit 1	bit 0

Magnitude = *n* – 1 bits

- ❖ Independent representation of the sign and magnitude
- ❖ Leftmost bit is the sign bit: 0 is positive and 1 is negative
- ❖ Using *n* bits, largest represented magnitude = $2^{n-1} – 1$

Sign-magnitude
8-bit representation of +45

0	0	1	0	1	1	0	1

Sign-magnitude
8-bit representation of -45

1	0	1	0	1	1	0	1

Properties of Sign-Magnitude

❖ Symmetric range of represented values:

For *n*-bit register, range is from -(2^{n-1} – 1) to +(2^{n-1} – 1)

For example, if *n* = 8 bits then range is -127 to +127

❖ Two representations for zero: +0 and -0 **NOT Good!**

❖ Two circuits are needed for addition & subtraction **NOT Good!**

- ✧ In addition to an adder, a second circuit is needed for subtraction
- ✧ Sign and magnitude parts should be processed independently
- ✧ Sign bit should be examined to determine addition or subtraction
- ✧ Addition of numbers of different signs is converted into subtraction
- ✧ Increases the cost of the add/subtract circuit

Sign-Magnitude Addition / Subtraction

Eight cases for Sign-Magnitude Addition / Subtraction

Operation	ADD Magnitudes	Subtract Magnitudes	
		A >= B	A < B
(+A) + (+B)	+(A+B)		
(+A) + (-B)		+(A–B)	-(B–A)
(-A) + (+B)		-(A–B)	+(B–A)
(-A) + (-B)	-(A+B)		
(+A) – (+B)		+(A–B)	-(B–A)
(+A) – (-B)	+(A+B)		
(-A) – (+B)	-(A+B)		
(-A) – (-B)		-(A–B)	+(B–A)

1's Complement Representation

- Given a binary number *A*

  The 1's complement of *A* is obtained by inverting each bit in *A*

- Example: 1's complement of $(01101001)_2 = (10010110)_2$

- If *A* consists of *n* bits then:

  A + (1's complement of A) = $(2^n - 1) = (1\ldots111)_2$ (all bits are 1's)

- Range of values is $-(2^{n-1} - 1)$ to $+(2^{n-1} - 1)$

  For example, if $n = 8$ bits, range is -127 to +127

- Two representations for zero: +0 and -0 **NOT Good!**

  1's complement of $(0\ldots000)_2 = (1\ldots111)_2 = 2^n - 1$

  $-0 = (1\ldots111)_2$ **NOT Good!**

2's Complement Representation

- ❖ Standard way to represent signed integers in computers
- ❖ A simple definition for 2's complement:

  Given a binary number *A*

  The 2's complement of *A* = (1's complement of *A*) + 1
- ❖ Example: 2's complement of $(01101001)_2$ =

  $(10010110)_2 + 1 = (10010111)_2$
- ❖ If *A* consists of *n* bits then

  A + (2's complement of A) = 2^n

  2's complement of $A = 2^n – A$

Computing the 2's Complement

starting value	00100100_2 = +36
step1: Invert the bits (1's complement)	11011011_2
step 2: Add 1 to the value from step 1	+ 1_2
sum = 2's complement representation	11011100_2 = -36

2’s complement of 11011100_2 (-36) = 00100011_2 + 1 = 00100100_2 = +36

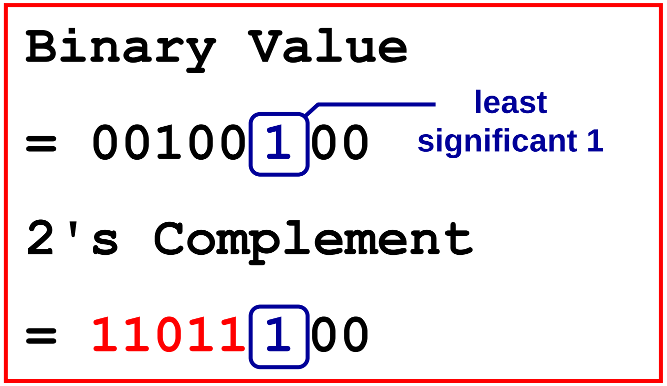
The 2’s complement of the 2’s complement of *A* is equal to *A*

Another way to obtain the 2's complement:

Start at the least significant 1

Leave all the 0s to its right unchanged

Complement all the bits to its left



Binary Value

= 00100$\boxed{1}$00

least significant 1

2's Complement

= 11011$\boxed{1}$00

Properties of the 2's Complement

- Range of represented values: -2^{n-1} to $+(2^{n-1} - 1)$

  For example, if n = 8 bits then range is -128 to +127

- There is only **one zero** = $(0\ldots000)_2$ (all bits are zeros)

- The 2’s complement of *A* is the **negative of *A***

- The sum of *A* + (2’s complement of *A*) **must be zero**

  The final carry is ignored

- Consider the 8-bit number $A = 00101100_2 = +44$

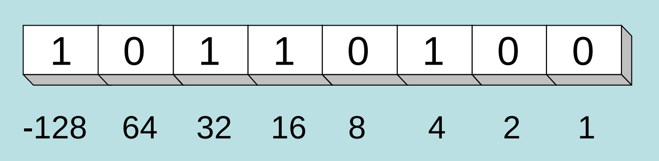
  2’s complement of $A = 11010100_2 = -44$

  $00101100_2 + 11010100_2 = \mathbf{1}\ 00000000_2$ (8-bit sum is 0)

  Ignore final carry = 2^8

2's Complement Signed Decimal Value

- Positive numbers (sign-bit = 0)
  - Signed value = Unsigned value
- Negative numbers (sign-bit = 1)
  - Signed value = Unsigned value – 2^n
  - n = number of bits
- Negative weight for sign bit
  - The 2's complement representation assigns a negative weight to the sign bit (most-significant bit)



1	0	1	1	0	1	0	0
-128	64	32	16	8	4	2	1

-128 + 32 + 16 + 4 = -76

8-bit Binary	Unsigned Value	Signed Value
00000000	0	0
00000001	1	+1
00000010	2	+2
. . .	. . .	. . .
01111101	125	+125
01111110	126	+126
01111111	127	+127
10000000	128	-128
10000001	129	-127
10000010	130	-126
. . .	. . .	. . .
11111101	253	-3
11111110	254	-2
11111111	255	-1

Values of Different Representations

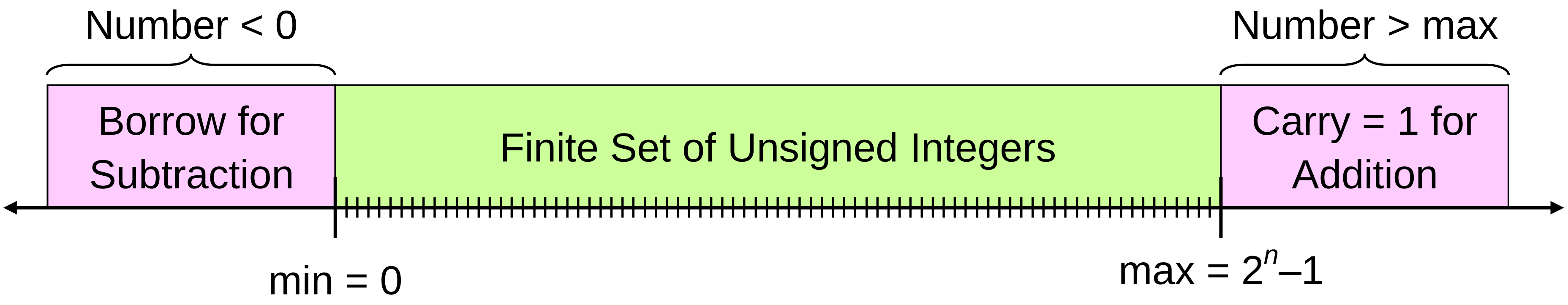
8-bit Binary Representation	Unsigned Value	Sign Magnitude Value	1's Complement Value	2's Complement Value
00000000	0	+0	+0	0
00000001	1	+1	+1	+1
00000010	2	+2	+2	+2
. . .	. . .	. . .	. . .	. . .
01111101	125	+125	+125	+125
01111110	126	+126	+126	+126
01111111	127	+127	+127	+127
10000000	128	-0	-127	-128
10000001	129	-1	-126	-127
10000010	130	-2	-125	-126
. . .	. . .	. . .	. . .	. . .
11111101	253	-125	-2	-3
11111110	254	-126	-1	-2
11111111	255	-127	-0	-1

Next . . .

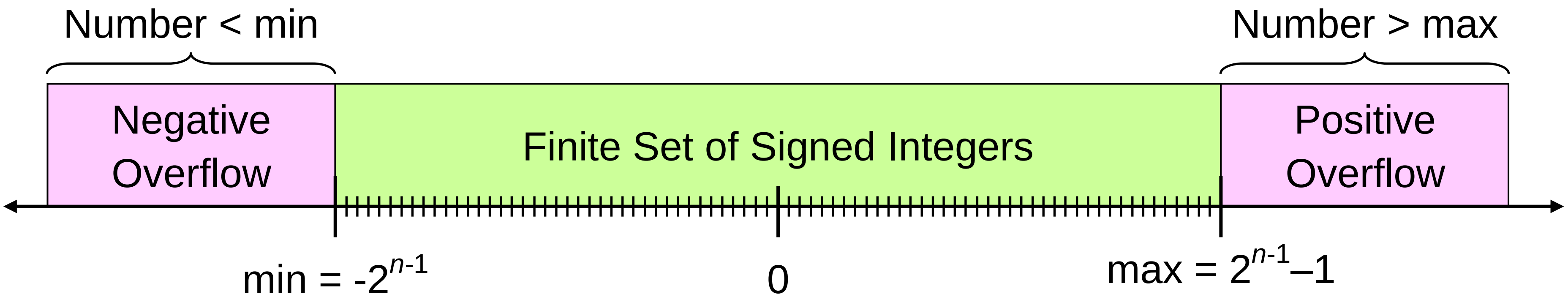
- Binary and Hexadecimal Addition and Subtraction
- Binary Multiplication and Bit Shifting
- Signed Integers
- Range, Overflow, Converting Subtraction into Addition

Range, Carry, Borrow, and Overflow

- Unsigned Integers: *n*-bit representation



- Signed Integers: 2's complement representation



Range of Unsigned Integers

For *n*-bit unsigned integers: Range is 0 to (2^n – 1)

There are NO negative values

Storage Size	Unsigned Range	Powers of 2
Byte = 8 bits	0 to 255	0 to (2^8 – 1)
Half Word = 16 bits	0 to 65,535	0 to (2^{16} – 1)
Word = 32 bits	0 to 4,294,967,295	0 to (2^{32} – 1)
Double Word = 64 bits	0 to 18,446,744,073,709,551,615	0 to (2^{64} – 1)

Practice: What is the range of values for unsigned 20-bit integers?

Range of Signed Integers

For *n*-bit signed integers: Range is -2^{n-1} to $(2^{n-1} - 1)$

Positive range: 0 to $2^{n-1} - 1$

Negative range: -2^{n-1} to -1

Storage Type	Signed Range	Powers of 2
Byte = 8 bits	–128 to +127	-2^{7} to $(2^{7} - 1)$
Half Word = 16 bits	–32,768 to +32,767	-2^{15} to $(2^{15} - 1)$
Word = 32 bits	–2,147,483,648 to +2,147,483,647	-2^{31} to $(2^{31} - 1)$
Double Word = 64 bits	–9,223,372,036,854,775,808 to +9,223,372,036,854,775,807	-2^{63} to $(2^{63} - 1)$

Practice: What is the range of values for signed 20-bit integers?

Carry and Overflow Examples

- We can have carry without overflow and vice-versa
- Four cases are possible (Examples on 8-bit numbers)

				1					
	0	0	0	0	1	1	1	1	15
+	0	0	0	0	1	0	0	0	8
	0	0	0	1	0	1	1	1	23

Carry = 0   Overflow = 0

1	1	1	1	1					
	0	0	0	0	1	1	1	1	15
+	1	1	1	1	1	0	0	0	248 (-8)
	0	0	0	0	0	1	1	1	7

Carry = 1   Overflow = 0

	1								
	0	1	0	0	1	1	1	1	79
+	0	1	0	0	0	0	0	0	64
	1	0	0	0	1	1	1	1	143 (-113)

Carry = 0   Overflow = 1

1			1	1					
	1	1	0	1	1	0	1	0	218 (-38)
+	1	0	0	1	1	1	0	1	157 (-99)
	0	1	1	1	0	1	1	1	119

Carry = 1   Overflow = 1

Carry versus Overflow

- Carry is important when …
  - Adding **unsigned integers**
  - Indicates that the **unsigned sum** is out of range
  - Sum > 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 …
  - Adding two positive numbers and the sum is negative
  - Adding two negative numbers and the sum is positive
- Simplest way to detect Overflow: $V = C_{n-1} \oplus C_n$
  - C_{n-1} and C_n are the carry-in and carry-out of the most-significant bit

Converting Subtraction into Addition

- When computing **A – B**, convert **B** to its 2's complement

  A – B = A + (2’s complement of B)

- **Same adder** is used for **both addition and subtraction**

  This is the biggest advantage of 2's complement

```
borrow:      -1 -1     -1              carry:  1  1        1  1

            0 1 0 0 1 1 0 1                   0 1 0 0 1 1 0 1
         –  0 0 1 1 1 0 1 0       →        +  1 1 0 0 0 1 1 0     (2's complement)
            _______________                   _______________
            0 0 0 1 0 0 1 1                   0 0 0 1 0 0 1 1     (same result)
```

- Final carry is **ignored**, because

  A + (2's complement of B) = A + (2^n – B) = (A – B) + 2^n

  Final carry = 2^n, for *n*-bit numbers