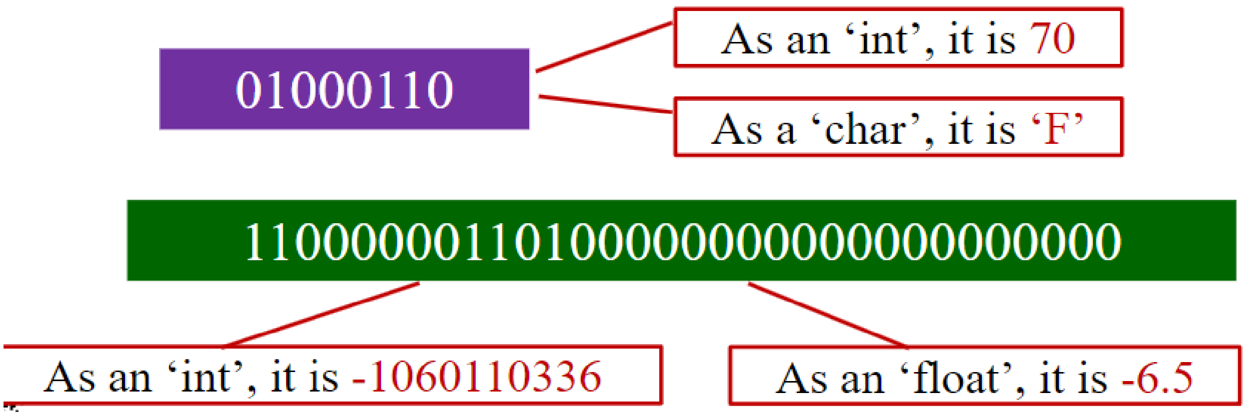
2 - Number Systems

2.1 Data Representation

- Basic data types in C:
  - `int` , with variants `short` , `long`
  - `float`
  - `double`
  - `char`
- How data is represented depends on its type:



- Data are internally represented as sequence of bits (binary digits). A bit is either 0 or 1
- Other units:
  - Byte = 8 bits
  - Nibble = 4 bits
  - Word = Multiple of bytes (eg: 1 byte, 2 bytes, 4 bytes, etc) depending on the computer architecture
- N bits can represent up to 2^n values
  - 2 bits represent up to 4 values (00, 01, 10, 11)
- To represent M values, $\lceil log_2 M \rceil$ bits required
  - 32 values requires 5 bits; 1000 values require 10 bits

2.2 Decimal (base 10) Number System

- A weighted-positional number system
- Base (also called radix) is 10
- Symbols/digits = $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$
- Each position has a weight of power of 10
  - $(7594.36)_{10} = (7 \times 10^3) + (5 \times 10^2) + (9 \times 10^1) + (4 \times 10^0) + (3 \times 10^{-1}) + (6 \times 10^{-2})$

2.3 Other Number Systems

- In some programming languages/software, special notations are used to represent numbers in certain bases
  - In C
    - prefix 0 for octal. Eg: `032` represents the octal number $(32)_8$
    - prefix 0x for hexadecimal. Eg: `0x32` represents the hexadecimal number $(32)_{16}$
  - In QTSpim (a MIPS simulator)
    - prefix 0x for hexadecimal.
  - In Verilog, the following values are the same
    - `8'b11110000` : an 8-bit binary value 11110000
    - `8'hF0` : an 8-bit binary value represented in hexadecimal F0
    - `8'd240` : an 8-bit binary value represented in decimal 240

2.4 Base-R to Decimal Conversion

- $1101.101_2 = 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-3} = 8 + 4 + 1 + 0.5 + 0.125 = 13.625_{10}$
- $572.6_8 = 5 \times 8^2 + 7 \times 8^1 + 2 \times 8^0 + 6 \times 8^{-1} = 320 + 56 + 2 + 0.75 = 378.75_8$
- $2A.8_{16} = 2 \times 16^1 + 10 \times 16^0 + 8 \times 16^{-1} = 32 + 10 + 0.5 = 42.5_{10}$

2.5 Decimal to Binary Conversion

- For whole numbers
  - Repeated Division-by-2 method
- For fractions
  - Repeated Multiplication-by-2 method

2.5.1 Repeated Division-by-2

Repeated Divide

- To convert a whole number to binary, use **successive division by 2** until the quotient is 0. The remainders from the answer, with the first remainder as the Least Significant Bit (LSB) and the last as the Most Significant Bit (MSB)

$$(43)_{10} = (101011)_2$$

2	43		
2	21	rem 1	← LSB
2	10	rem 1	
2	5	rem 0	
2	2	rem 1	
2	1	rem 0	
	0	rem 1	← MSB

> **Repeated Division-by-2**：这种方法经常被用于将十进制数转换为二进制数。
>
> 步骤如下：
>
> 1. 用2除以给定的十进制数，记录商和余数。
> 2. 再用2除以上一步得到的商，再次记录商和余数。
> 3. 重复上述步骤，直到商变为0为止。
> 4. 从上到下读取余数，就得到了对应的二进制表示。
>
> 例如，将十进制数13转换为二进制：
>
> ```
> Copy code13 ÷ 2 = 6 商 1 余数
> 6 ÷ 2 = 3 商 0 余数
> 3 ÷ 2 = 1 商 1 余数
> 1 ÷ 2 = 0 商 1 余数
>
> 从上到下读取余数得到：1101，所以13的二进制表示是1101。
> ```

2.5.2 Repeated Multiplication-by-2

Repeated Multiply

- To convert **decimal fractions** to binary, **repeated multiplication by 2** is used, until the fractional product is 0 (or until the desired number of decimal places). The carried digits, or carries , produce the answer, with the first carry as the MSB, and the last as the LSB.

$$(0.3125)_{10} = (.0101)_2$$

	Carry	
$0.3125\times2=0.625$	0	←MSB
$0.625\times2=1.25$	1	
$0.25\times2=0.50$	0	
$0.5\times2=1.00$	1	←LSB

> **Repeated Multiplication-by-2**：这种方法经常被用于将二进制小数转换为十进制小数。

> 步骤如下：
>
> 1. 将给定的二进制小数的最高位（最左边的位）乘以2。
> 2. 记录该乘积的整数部分。
> 3. 将乘积的小数部分再乘以2。
> 4. 重复上述步骤，直到得到的小数部分为0或达到所需的精度。
>
> 例如，将二进制小数0.101转换为十进制：
>
> ```
> rustCopy code0.101 × 2 = 1.01  -> 记录整数部分 1
> 0.01 × 2  = 0.02  -> 记录整数部分 0
> 0.02 × 2  = 0.04  -> 记录整数部分 0 (如果需要更多精度则继续，否则可以停止)
>
> 从上到下读取整数部分得到：100，表示二进制的0.101等于十进制的0.5。
> ```

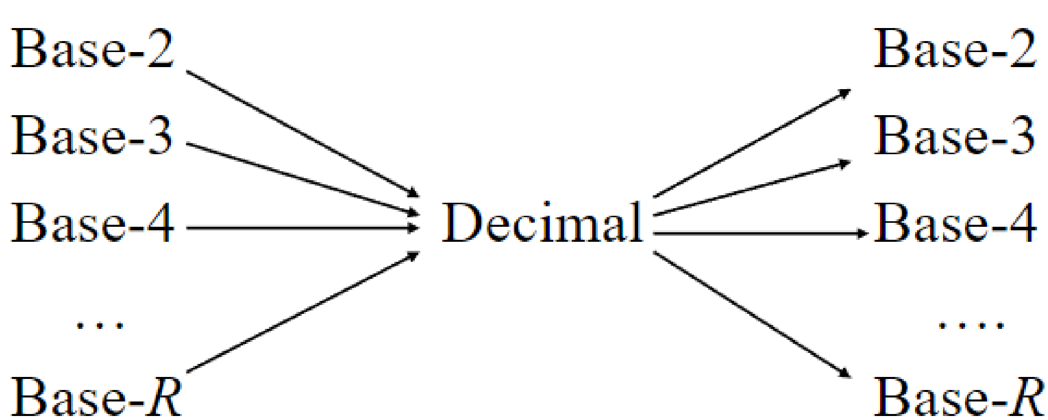
2.6 Conversion Between Decimal and Other Bases

- **Base-R to decimal:** multiply digits with their corresponding weights
- **Decimal to binary (base 2)**
  - For whole numbers: repeated division-by-2 (2.5.1)
  - For fraction numbers: repeated multiplication-by-2 (2.5.2)
- **Decimal to base-R**
  - For whole numbers: repeated division-by-R
  - For fraction numbers: repeated multiplication-by-R

> 总结：
>
> 不管是什么进制，均可使用2.5章内使用的方法，将除以2或乘以2替换进制数字

2.7 Conversion Between Bases

- In general, conversion between bases can be done via decimal:



2.8 Binary to Octal/Hexadecimal Conversion

- Binary -> Octal: partition in groups of 3
  - $(10\ \ 111\ \ 011\ \ 001\ .\ 101\ \ 110)_2 = (2731.56)_8$
- Octal -> Binary: reverse
  - $(2731.56)_8 = (10\ \ 111\ \ 011\ \ 001\ .\ 101\ \ 110)_2$
- Binary -> Hexadecimal: partition in groups of 4
  - $(101\ \ 1101\ \ 1001\ .\ 1011\ \ 1000)_2 = (5D9.B8)_{16}$
- Hexadecimal -> Binary: reverse
  - $(5D9.B8)_{16} = (101\ \ 1101\ \ 1001\ .\ 1011\ \ 1000)_2$

2.9 ASCII Code

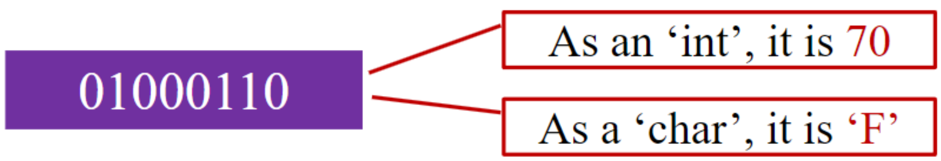
- **ASCII code** and **Unicode** are used to represent characters `('a', 'C', '?', '\0')`
- ASCII
  - American Standard Code for Information Interchange
  - 7 bits, plus 1 parity bit (odd or even parity)

Character	ASCII Code
0	0110000
1	0110001
. . .	. . .
9	0111001
:	0111010
A	1000001
B	1000010
. . .	. . .
Z	1011010
[	1011011
\	1011100

‘A’: 1000001 (or 65_{10})

	MSBs							
LSBs	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
0000	NUL	DLE	SP	0	@	P	`	p
0001	SOH	DC_1	!	1	A	Q	a	q
0010	STX	DC_2	“	2	B	R	b	r
0011	ETX	DC_3	#	3	C	S	c	s
0100	EOT	DC_4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	‘	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	O	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

- Integers (0 to 127) and characters are ‘somewhat’ interchangeable in C



CharAndInt.c

```c
int num = 65;
char ch = 'F';

printf("num (in %%d) = %d\n", num);
printf("num (in %%c) = %c\n", num);
printf("\n");

printf("ch (in %%c) = %c\n", ch);
printf("ch (in %%d) = %d\n", ch);
```

```
num (in %d) = 65
num (in %c) = A

ch (in %c) = F
ch (in %d) = 70
```

```
int main() {
int i, n = 2147483640;
for (i=1; i<=10; i++) {
  n = n + 1;
}
printf("n = %d\n", n);
}
```

对于这一段代码，其输出是什么？

这段代码中， `int` 数据类型的整数会溢出。在多数计算机系统中，一个 `int` 数据类型通常占据4个字节（32位），其范围是从 `-2,147,483,648` （即 `-2^31` ）到 `2,147,483,647` （即 `2^31 - 1` ）。

当 `n` 的值是 `2,147,483,640` ，并在循环中加了10次，它的值会变为 `2,147,483,650` 。这个值超出了 `int` 的最大值 `2,147,483,647` 。

因此，当加1到 `2,147,483,647` ，它会溢出并回绕到 `int` 的最小值 `-2,147,483,648` ，然后再从这个值开始增加。

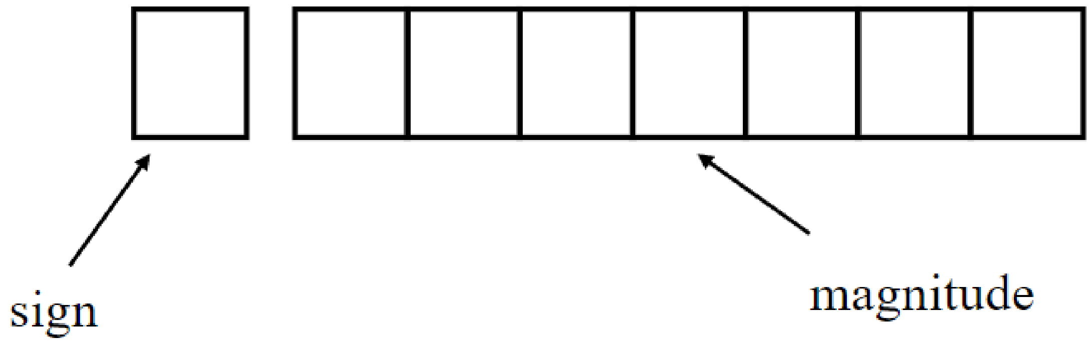
```
2,147,483,640 + 1 = 2,147,483,641
2,147,483,641 + 1 = 2,147,483,642
...
2,147,483,646 + 1 = 2,147,483,647  // 这是int的最大值
2,147,483,647 + 1 = -2,147,483,648 // 溢出，变成int的最小值
-2,147,483,648 + 1 = -2,147,483,647
-2,147,483,647 + 1 = -2,147,483,646
```

2.10 Negative Numbers

- Unsigned numbers: only non-negative values
- Signed numbers: include all values (positive and negative)
- There are 3 common representations for signed binary numbers:
  - Sign-and-magnitude
  - 1s Complement
  - 2s Complement

2.10.1 Sign-and-Magnitude

- The sign is represented by a ‘sign bit’
  - `0` for +
  - `1` for -
- For example: a 1-bit sign and 7-bit magnitude format



- Example:
  - `00110100` -> $+110100_2 = +52_{10}$
  - `10010011` -> $-10011_2 = -19_{10}$
- For 8-bit binary number:
  - Largest value: `01111111` , 127_{10}
  - Smallest value: `11111111` , -127_{10}
  - Zeros:
    - `00000000` , $+0_{10}$
    - `10000000` , -0_{10}
  - Range (for 8-bit): -127_{10} to $+127_{10}$
- For n-bit sign-and-magnitude representation, the range of values should be:
  - $-2^{n-1}+1$ to $2^{n-1}-1$
- Negate a number, just **invert the sign bit**
  - Examples:
    - Negate 00100001_2 (decimal 33)
      - 10100001_2 (decimal -33)
    - Negate 10000101_2 (decimal -5)
      - 00000101_2 (decimal 5)

2.10.2 1s Complement 一进制补码

- Given a number `x` which can be expressed as an n-bit binary number, its **negated value** can be obtained in **1's-complement** representation using:

$$-x = 2^n - x - 1$$

- Example: With an 8-bit number `00001100` (or 12_{10}), its negated value expressed in 1's-complement is:

$$\begin{aligned} -00001100_2 &= 2^8 - 12 - 1 \\ &= 243 \\ &= 11110011_{1s} \end{aligned}$$

- (This means that -12_{10} is written as `11110011` in 1s-complement representation)

- Technique to negate a value: **invert all the bits**
- Largest value: `0111 1111` = $+127_{10}$
- Smallest value: `1000 0000` = -127_{10}
- Zeros:
  - `0000 0000` = $+0_{10}$
  - `1111 1111` = -0_{10}
- Range (for 8 bits): -127_{10} to $+127_{10}$
- Range (for n bits): $-(2^{n-1}-1)$ to $2^{n-1}-1$
- The most significant bit (MSB) still represents the sign: 0 for positive, 1 for negative
- Examples:
  - $(14_{10}) = (00001110)_2 = (00001110)_{1s}$
  - $-(14)_{10} = -(00001110)_2 = (11110001)_{1s}$

> - 对于一个二进制数，它的1's complement是将该数中的每一位都取反。换句话说，将所有的0变为1，所有的1变为0。
> - 例如，考虑一个8位二进制数 `1101 0101` 。它的1's complement是 `0010 1010` 。

2.10.3 2s Complement

- Given a number x which can be expressed as an n-bit binary number, its negated value can be obtained in **2s-complement** representation using:

$$-x = 2^n - x$$

- Example: With an 8-bit number `00001100` (or 12_{10}), its negated value expressed in 2s-complement is:

$$\begin{aligned} -00001100_2 &= 2^8 - 12 \\ &= 244 \\ &= ((11110011)_{1s} + 1)_{2s} \\ &= 11110100_{2s} \end{aligned}$$

- This means that -12_{10} is written as `1111 0100` in 2s-complement representation
- Technique to negate a value: **invert all the bits, then add 1**
- Largest value: `0111 1111` = $+127_{10}$
- Smallest value: `1000 0000` = -128_{10}
- Zero: `0000 0000` = $+0_{10}$
- Range (for 8 bits): -128_{10} to $+127_{10}$
- Range (for n bits): $-(2^{n-1})$ to $2^{n-1}-1$
- The most significant bit (MSB) still represents the sign: 0 for positive, 1 for negative
- Examples:
  - $(14)_{10} = (00001110)_2 = (00001110)_{2s}$

- $-(14)_{10} = -(00001110)_2 = (11110010)_{2s}$

2.10.4 Comparisons

4-bit system

Positive values

Value	Sign-and-Magnitude	1s Comp.	2s Comp.
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000

Negative values

Value	Sign-and-Magnitude	1s Comp.	2s Comp.
-0	1000	1111	-
-1	1001	1110	1111
-2	1010	1101	1110
-3	1011	1100	1101
-4	1100	1011	1100
-5	1101	1010	1011
-6	1110	1001	1010
-7	1111	1000	1001
-8	-	-	1000

2.10.5 Complement on Fractions

- We can extend the idea of complement on fractions
- Examples:
  - Negate `0101.01` in 1s-complement
    - Answer: `1010.10`
  - Negate `111000.101` in 1s-complement
    - Answer: `000111.010`
  - Negate `0101.01` in 2s-complement
    - Answer: `1010.11`

2.10.6 2s Complement Addition/Subtraction

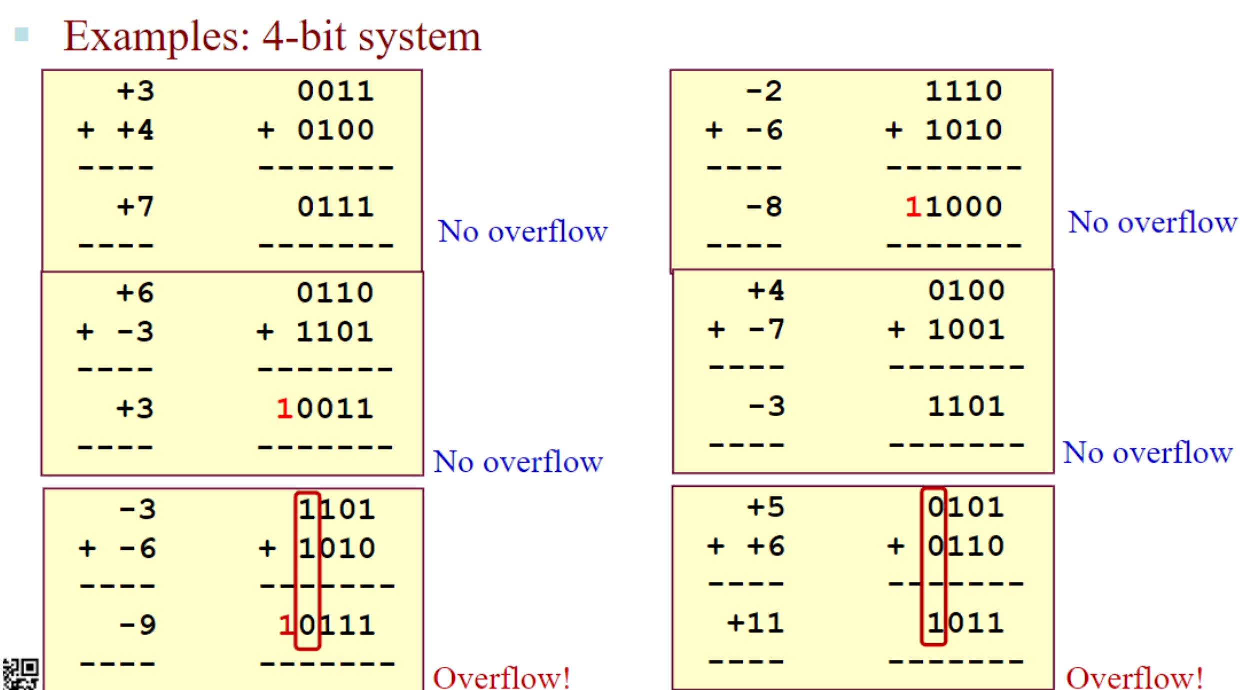
- Algorithm for addition of integers, $A + B$:
  1. Perform binary addition on the two numbers
  2. Ignore the carry out of the MSB
  3. Check for overflow. Overflow occurs if the 'carry in' and 'carry out' of the MSB are different, or if result is opposite sign of A and B
- Algorithm for subtraction of integers, $A - B$:

$$A - B = A + (-B)$$

  1. Take 2s-complement of B
  2. Add the 2s-complement of B to A

Overflow

- Signed numbers are of a fixed range
- If the result of addition/subtraction goes beyond this range, an **overflow** occurs
- Overflow can be easily detected:
  - positive add positive -> negative
  - negative add negative -> positive
- Example: 4-bit 2s-complement system
  - Range of value: -8_{10} to 7_{10}
  - $0101_{2s} + 0110_{2s} = 1011_{2s}$
    $5_{10} + 6_{10} = -5_{10}$ (Overflow!)
  - $1001_{2s} + 1101_{2s} = 10110_{2s}$ (discard end-carry) $= 0110_{2s}$
    $-7_{10} + -3_{10} = 6_{10}$ (Overflow!)



- Examples: 4-bit system
  - 4 – 7
  - Convert it to 4 + (-7)

```
 +4        0100
+ -7     + 1001
----     -------
 -3        1101
----     -------
```

No overflow

  - 6 – 1
  - Convert it to 6 + (-1)

```
 +6        0110
+ -1     + 1111
----     -------
 +5       10101
----     -------
```

No overflow

  - -5 – 4
  - Convert it to -5 + (-4)

```
 -5        1011
+ -4     + 1100
----     -------
 -9       10111
----     -------
```

Overflow!

> 在二进制的2's-complement加减法中，判断溢出的情况是基于加法的结果与两个操作数的关系来确定的。下面我将为您解释如何判断正溢出和负溢出。
>
> 1. **正溢出**：
>   - 当两个正数相加得到一个负数结果时，就发生了正溢出。
>   - 具体判断方式为：两个操作数的最高位（符号位）都是0，但结果的最高位是1。
> 2. **负溢出**：
>   - 当两个负数相加得到一个正数结果时，就发生了负溢出。
>   - 具体判断方式为：两个操作数的最高位都是1，但结果的最高位是0。
>
> 加减法的关系：减法可以看作加上一个数的2's complement。所以，如果你知道如何检测加法的溢出，你也可以应用这个知识来检测减法的溢出。

2.10.7 1s Complement Addition/Subtraction

- Algorithm for addition of integers, $A+B$:
  1. Perform binary addition on the two numbers
  2. If there is a carry out of the MSB, add 1 to the result
  3. Check for overflow. Overflow occurs if result is opposite sign of A and B
- Algorithm for subtraction of integers, $A-B$:

$$A-B=A+(-B)$$

  1. Take 1s-complement of B
  2. Add the 1s-complement of B to A

- Examples: 4-bit system

```
  +3         0011
+ +4       + 0100
----       -------
  +7         0111
----       -------
```

No overflow

```
  +5         0101
+ -5       + 1010
----       -------
  -0         1111
----       -------
```

No overflow

```
  -2        1101
+ -5      + 1010
----      -------
  -7       10111
----      +     1
          -------
            1000
          -------
```

No overflow

```
  -3         1100
+ -7       + 1000
----       -------
 -10        10100
----       +     1
           -------
             0101
           -------
```

Overflow!

> 在二进制加减法中，判断溢出是否发生取决于你是否在执行有符号的运算。溢出的概念主要适用于有符号数，通常是使用`2's complement`表示法。
>
> 对于加法：
>
> 1. **正溢出**：当你将两个正数相加并得到一个负结果时，发生正溢出。
> 2. **负溢出**：当你将两个负数相加并得到一个正结果时，发生负溢出。
>
> 对于减法： 由于减法可以被视为加法（减去一个数等同于加上它的负数），因此溢出条件与上述相同。
>
> 判断溢出的实际方法：
>
> 1. 检查操作数的符号和结果的符号。
> 2. 如果两个正操作数的加法得到一个负结果，或者两个负操作数的加法得到一个正结果，则发生溢出。
> 3. 更具体地说，可以检查进位到符号位和从符号位的进位。如果它们不同，就发生了溢出。例如，对于8位数，如果从第7位到第8位有进位，但从第8位向上没有进位（或相反），则发生溢出。
>
> 这种基于进位的方法在硬件加法器中更为实用，因为可以直接从加法器的输出中获得进位信号，用于溢出检测。

2.10.8 Excess Notation (Excess Representation)

- Besides sign-and-magnitude and complement schemes, the **excess representation** is another scheme
- It allows the range of values to be distributed **evenly** between the positive and negative values, by a simple translation (addition/subtraction)
- Example: Excess+4 (Excess-(-4)) representation on 3-bit numbers. See table below

Excess-4 Representation	*Value*
000	-4
001	-3
010	-2
011	-1
100	0
101	1
110	2
111	3

- Example: Excess+8 (Excess-(-7)) representation on 4-bit numbers

Excess-8 Representation	*Value*
0000	-8
0001	-7
0010	-6
0011	-5
0100	-4
0101	-3
0110	-2
0111	-1

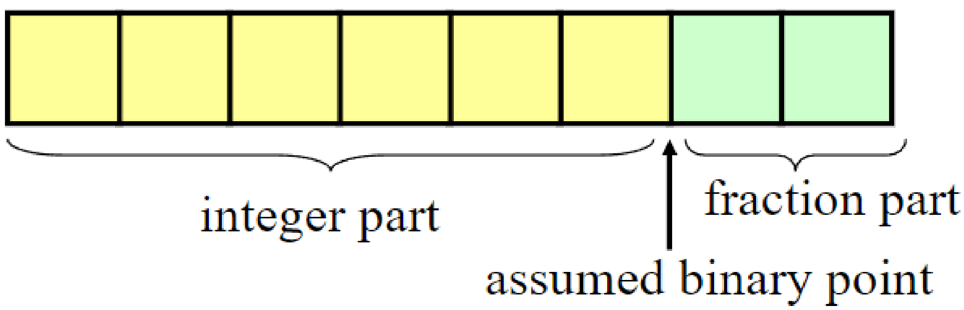
Excess-8 Representation	*Value*
1000	0
1001	1
1010	2
1011	3
1100	4
1101	5
1110	6
1111	7

2.11 Real Numbers

- Many applications involve computations not only on integers but also on real numbers
- How are real numbers represented in a computer system?
- Due to the finite number of bits, real number are often represented in their approximate values

2.11.1 Fixed-point Representation

- In **fixed-point representation**, the number of bits allocated for the whole number part and fractional part are fixed
- For example, given an 8-bit representation, 6 bits are for whole number part and 2 bits for fractional parts



- If 2s-complement is used, we can represent values like:
  $011010.11_{2s} = 26.75_{10}$
  $111110.11_{2s} = -000001.01_2 = -1.25_{10}$

2.11.2 Floating-point Representation

- Floating-point representation has limited range
- Alternative: **Floating points numbers** allow us to represent very large or very small numbers
- Examples:
  - 0.23×10^{23} (very large positive number)
  - 0.5×10^{-37} (very small positive number)
  - -0.2397×10^{-18} (very small negative number)
- 3 components: **sign, exponent and mantissa (fraction)**
- The base (radix) is assumed to be 2
- Two formats:
  - Single-precision (32-bit): 1-bit sign, 8-bit exponent with bias 127 (excess-127), 23-bit mantissa
  - Double-precision (64-bit): 1-bit sign, 11-bit exponent with bias 1023 (excess-1023), 53-bit mantissa

- 3 components: **sign**, **exponent** and **mantissa (fraction)**

sign	exponent	mantissa

- Sign bit: 0 for positive, 1 for negative
- Mantissa is **normalized** with an implicit leading bit 1
  - 110.1_2 -> normalized -> $1.101_2 \times 2^2$ -> only 101 is stored in the mantissa field
  - 0.00101101_2 -> normalized -> $1.01101_2 \times 2^{-3}$ -> only 01101 is stored in the mantissa field

- Example: How is -6.5_{10} represented in IEEE 754 single-precision floating-point format?

$$-6.5_{10} = -110.1_2 = -1.\boxed{101_2} \times 2^2$$

Exponent = 2 + 127 = 129 = 10000001_2

1	10000001	10100000000000000000000
sign	exponent (excess-127)	mantissa

- We may write the 32-bit representation in hexadecimal:

$$1\ 10000001\ 10100000000000000000000_2 = \text{C0D00000}_{16}$$

(Slide 4) 11000000110100000000000000000000

As an 'int', it is -1060110336

As an 'float', it is -6.5