

Datorsystem VT 2022

F2-F3

Talsystem, Datarepresentation

Outline

- Introduction Data representation
- Number system
- Converting Between bases
- Signed Integer Representation
- Floating- Point Representation



Introduktion Talsystem

Exponenträkning

- $3^2 = 3 * 3 = 9$
- $3^3 = 3 * 3 * 3 = 27$
- $3^1 = 3$
- $3^0 = 1$
- $3^{-1} = 1/3$
- $3^{-2} = 1/3^2 = 1/9$

Det decimala talsystemet.

- Talbas 10
- Använder siffrorna 0-9.
- Positionssystem

1	3	2	8	.	2	5	6
1000	100	10	1		1/10	1/100	1/1000
10^3	10^2	10^1	10^0		10^{-1}	10^{-2}	10^{-3}



Det binära talsystemet

- Talbas 2
- Använder siffrorna 0 och 1.
- Positionssystem

1	1	0	1	.	1	0	1
2^3	2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}
8	4	2	1		1/2	1/4	1/8



Bitar och bytes

- En bit är basenheten som en dator använder.
- En nibble är en grupp om fyra bitar.
- En byte är en grupp om åtta bitar.
- Ett ord (word) är en grupp om fyra bytes, 32 bitar.

1	0	0	1	1	0	1	1	0	1	0	0	0	0	1	0	1	0	1	1	1	0	1	0	1	0	0	0	0	0	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---



Binära prefix

Antal	Prefix
1000	kilo
1000^2	mega
1000^3	giga
1000^4	tera

Decimalt

Bitar	Prefix
1024	kilo
1024^2	mega
1024^3	giga
1024^4	tera

Bytes	Prefix
1024	kilo
1024^2	mega
1024^3	giga
1024^4	tera

Binärt

Konvertering av 100111_2 till den decimala talbasen

1	0	0	1	1	1
2^5	2^4	2^3	2^2	2^1	2^0
32	16	8	4	2	1

$$10\ 0111_2 = 39_{10}$$

Konvertering av 200_{10} till binära talbasen

Tal att konvertera	200	200	200	72	8	8	8	0	0	0
Positionsvärde	512	256	128	64	32	16	8	4	2	1
Subtrahera			72	8			0			
Binärt tal	0	0	1	1	0	0	1	0	0	0

$$200_{10} = 1100\ 1000_2$$

Konvertering av 173_{10} till binära talbasen

Tal att konvertera			173	45	45	13	13	5	1	1
Positionsvärde	512	256	128	64	32	16	8	4	2	1
Subtrahera			45		13		5	1		0
Binärt tal			1	0	1	0	1	1	0	1

$$173_{10} = 1010\ 1101_2$$

Det hexadecimala talsystemet.

- Talbas 16
- Använder siffrorna 0-9 samt bokstäverna A, B, C, D, E och F.
- Varje hexadecimal siffra kan representeras som fyra binära siffror.

Talbas 10	Talbas 2	Talbas 16
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F
16	0001 0000	10
17	0001 0001	11
18	0001 0010	12
31	0001 1111	1F
100	0110 0100	64
255	1111 1111	FF
256	1 0000 0000	100

Det hexadecimala talsystemet.

D	E	1	.	A	0
16^2	16^1	16^0		16^{-1}	16^{-2}
256	16	1		1/16	1/256



Konvertering av $102D_{16}$ till det decimala talsystemet

1	0	2	D
16^3	16^2	16^1	16^0
4096	256	16	1

$$102D_{16} = 4141_{10}$$

Konvertering av $1101\ 1011\ 1001_2$ till hexadecimala talsystemet

Binärt:	1101	1011	1001
Decimalt gruppvärde:	13	11	9
Hexadecimalt:	D	B	9

$$1101\ 1011\ 1001_2 = \text{DB9}_{16}$$

Konvertering av $12DA_{16}$ till binära talsystemet

Hexadecimal:	1	2	D	A
Decimalt gruppvärde:	1	2	13	10
Binärt:	0001	0010	1101	1010

$$12DA_{16} = 1\ 0010\ 1101\ 1010_2$$

Konvertering av 1437_{10} till hexadecimala talsystemet genom omvandling till binärt.

Tal att konvertera	1437	413	413	157	29	29	29	13	5	1	1
Positionsvärde	1024	512	256	128	64	32	16	8	4	2	1
Subtrahera	413		157	29			13	5	1		0
Binärt tal	1	0	1	1	0	0	1	1	1	0	1
Hexadecimalt tal	5			9				D			

$$1437_{10} = 59D_{16}$$

Representation av heltal i primärminnet

- 32/64-bitars int
- Signed/unsigned

Omfång:

32-bitars signed: -2147483648 - 2147483647

32-bitars unsigned: 0 - 4294967295

64-bitars signed:

-9223372036854775808 - 9223372036854775807

64-bitars unsigned: 0 - 18446744073709551615

Teckenbit



1	0	0	1	1	0	1	1	0	1	0	0	0	0	0	1	0	1	0	1	1	1	0	1	0	1	0	0	0	0	0	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---



ASCII

- Teckenkodning som använder sju bitar.
- Skapades 1963
- Har utökats till olika åttabitar-versioner.
 - Windows-1252 (ANSI)
 - MacRoman
 - Latin-1

ASCII	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0000	N _U	S _H	S _X	E _X	E _T	E _D	A _X	R _L	R _S	H _T	L _F	V _T	F _F	C _R	S ₀	S ₁
0001	D _L	D ₁	D ₂	D ₃	D ₄	N _K	S _Y	E _Z	C _N	E _M	S _B	E _C	F _S	G _S	R _S	U _S
0010		!	"	#	\$	%	&	'	()	*	+	,	-	.	/
0011	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0100	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0101	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
0110	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0111	p	q	r	s	t	u	v	w	x	y	z	{		}	~	U _T
1000	S ₀	S ₁	S ₂	S ₃	I _N	N _L	S _S	E _S	H _S	H _J	Y _S	P _D	P _V	R _I	S ₂	S ₃
1001	D _C	P ₁	P ₂	S _E	C _C	M _M	S _P	E _P	Q _S	Q _D	Q _A	C _S	S _T	O _S	P _M	A _P
1010	A ₀	!	¢	£	¤	¥		\$	"	©	ª	«	¬	-	®	—
1011	°	±	²	³	´	µ	¶	·	,	³	°	»	¼	½	¾	¿
1100	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
1101	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
1110	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
1111	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ



F2-F3:

Positional Numbering System & Converting Between Bases

Positional Numbering Systems

- Bytes store numbers using the position of each bit to represent a power of 2.
 - The binary system is also called the base-2 system.
 - Our decimal system is the base-10 system. It uses powers of 10 for each position in a number.
 - Any integer quantity can be represented exactly using any base (or *radix*).

Positional Numbering Systems

- The decimal number 947 in powers of 10 is:

$$9 \times 10^2 + 4 \times 10^1 + 7 \times 10^0$$

- The decimal number 5836.47 in powers of 10 is:

$$\begin{aligned} 5 \times 10^3 + 8 \times 10^2 + 3 \times 10^1 + 6 \times 10^0 \\ + 4 \times 10^{-1} + 7 \times 10^{-2} \end{aligned}$$

Positional Numbering Systems

- The binary number 11001 in powers of 2 is:

$$\begin{aligned} & 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 16 + 8 + 0 + 0 + 1 = 25 \end{aligned}$$

- When the radix of a number is something other than 10, the base is denoted by a subscript.
 - Sometimes, the subscript 10 is added for emphasis:

$$11001_2 = 25_{10}$$

Converting Between Bases

- Fractional decimal values have nonzero digits to the right of the decimal point.
- Fractional values of other radix systems have nonzero digits to the right of the *radix point*.
- Numerals to the right of a radix point represent negative powers of the radix:

$$0.47_{10} = 4 \times 10^{-1} + 7 \times 10^{-2}$$

$$0.11_2 = 1 \times 2^{-1} + 1 \times 2^{-2}$$

$$= \frac{1}{2} + \frac{1}{4}$$

$$= 0.5 + 0.25 = 0.75$$

Converting Between Bases

- The calculation to the right is an example of using the subtraction method to convert the decimal 0.8125 to binary.

- Our result, reading from top to bottom is:

$$0.8125_{10} = 0.1101_2$$

- Of course, this method works with any base, not just binary.

0.8125		
- 0.5000	$= 2^{-1} \times 1$	
0.3125	1/2	
- 0.2500	$= 2^{-2} \times 1$	
0.0625	1/4	
- 0	$= 2^{-3} \times 0$	
0.0625	1/8	
- 0.0625	$= 2^{-4} \times 1$	
0	1/16	

$$1/8 = 0.125$$

Converting Between Bases

- Using the multiplication method to convert the decimal 0.8125 to binary, we multiply by the radix 2.
 - The first product carries into the units place.

$$\begin{array}{r} .8125 \\ \times \quad 2 \\ \hline 1.6250 \end{array}$$

Converting Between Bases

- **Converting 0.8125 to binary . . .**

- Ignoring the value in the units place at each step, continue multiplying each fractional part by the radix.

$$\begin{array}{r} .8125 \\ \times \quad 2 \\ \hline 1.6250 \end{array}$$

$$\begin{array}{r} .6250 \\ \times \quad 2 \\ \hline 1.2500 \end{array}$$

$$\begin{array}{r} .2500 \\ \times \quad 2 \\ \hline 0.5000 \end{array}$$

Converting Between Bases

- **Converting 0.8125 to binary . . .**

- You are finished when the product is zero, or until you have reached the desired number of binary places.
- Our result, reading from top to bottom is:
$$0.8125_{10} = 0.1101_2$$
- This method also works with any base. Just use the target radix as the multiplier.

$$\begin{array}{r} .8125 \\ \times \quad 2 \\ \hline 1.6250 \\ \\ .6250 \\ \times \quad 2 \\ \hline 1.2500 \\ \\ .2500 \\ \times \quad 2 \\ \hline 0.5000 \\ \\ .5000 \\ \times \quad 2 \\ \hline 1.0000 \end{array}$$

Converting Between Bases

- The binary numbering system is the most important radix system for digital computers.
- However, it is difficult to read long strings of binary numbers -- and even a modestly-sized decimal number becomes a very long binary number.
 - For example: $11010100011011_2 = 13595_{10}$
- For compactness and ease of reading, binary values are usually expressed using the hexadecimal, or base-16, numbering system.

Converting Between Bases

- The hexadecimal numbering system uses the numerals 0 through 9 and the letters A through F.
 - The decimal number 12 is C_{16} .
 - The decimal number 26 is $1A_{16}$.
- It is easy to convert between base 16 and base 2, because $16 = 2^4$.
- Thus, to convert from binary to hexadecimal, all we need to do is group the binary digits into groups of four.

A group of four binary digits is called a hextet

Converting Between Bases

- Using groups of hextets, the binary number 11010100011011_2 ($= 13595_{10}$) in hexadecimal (**base 16**) is:

0011	0101	0001	1011
3	5	1	B

If the number of bits is not a multiple of 4, pad on the left with zeros.

- Octal (**base 8**) values are derived from binary by using groups of three bits ($8 = 2^3$):

011	010	100	011	011
3	2	4	3	3

Octal was very useful when computers used six-bit words.

Signed Integer Representation

- The conversions we have so far presented have involved **only unsigned numbers**.
- To represent **signed integers**, computer systems **allocate the high-order bit to indicate the sign** of a number.
 - The **high-order bit** *is the leftmost bit*. It is also called the **most significant bit**.
 - **0** is used to indicate a **positive number**; **1** indicates a **negative number**.
- The *remaining bits contain* the value of the number (but this can be interpreted different ways)

1 0 0 1 1 0 1 1 0 1 0 0 0 0 1 0 1 0 1 1 1 0 1 0 1 0 0 0 0 0 1 1



Tecken bit

Signed Integer Representation

- There are **three ways** in which signed binary integers may be expressed:
 - **Signed magnitude**
 - **One's complement**
 - **Two's complement**
- In an 8-bit word, *signed magnitude* representation **places the absolute value** of the number in the **7 bits** to the **right of the sign bit**.

Signed Integer Representation

- For example, in 8-bit **signed magnitude** representation:
 - +3 is: 00000011
 - 3 is: 10000011
- Computers perform arithmetic operations on signed magnitude numbers in much the same way as humans carry out pencil and paper arithmetic.
 - Humans often ignore the signs of the operands while performing a calculation, applying the appropriate sign after the calculation is complete.

Signed Integer Representation

- Binary addition is as easy as it gets. You need to know only four rules:

$$0 + 0 = 0 \qquad 0 + 1 = 1$$

$$1 + 0 = 1 \qquad 1 + 1 = 10$$

- The simplicity of this system makes it possible for digital circuits to carry out arithmetic operations.

Let's see how the addition rules work with signed magnitude numbers . . .

Signed Integer Representation

- Example:
 - Using **signed magnitude** binary arithmetic, find the sum of 75 and 46.
- First, convert 75 and 46 to binary, and arrange as a sum, but separate the (positive) sign bits from the magnitude bits.

Tecken bit
↓

0		1 0 0 1 0 1 1
0	+	0 1 0 1 1 1 0
<hr/>		

Signed Integer Representation

- Example:
 - Using **signed magnitude** binary arithmetic, find the sum of 75 and 46.
- Just as in decimal arithmetic, we find the sum starting with the rightmost bit and work left.

Tecken bit
↓

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

Signed Integer Representation

- Example:
 - Using **signed magnitude** binary arithmetic, find the sum of 75 and 46.
- In the second bit, we have a carry, so we note it above the third bit.

Tecken bit

↓

0 1 0 0 1 0 1 1

0 + 0 1 0 1 1 1 0

0 1

Signed Integer Representation

- Example:
 - Using **signed magnitude** binary arithmetic, find the sum of 75 and 46.
- The third and fourth bits also give us carries.

$$\begin{array}{r} 111 \\ 01001011 \\ 0 + 0101110 \\ \hline 1001 \end{array}$$

Signed Integer Representation

- Example:
 - Using **signed magnitude** binary arithmetic, find the sum of 75 and 46.
- Once we have worked our way through all eight bits, we are done.

$$\begin{array}{r} \\ \\ 0 \\ 0 + 0 \\ \hline 0 \end{array}$$

In this example, we were careful to pick two values whose sum would fit into 7 bits. If that is not the case, we have a problem.

Signed Integer Representation

- **Example:**
 - Using signed magnitude binary arithmetic, find the sum of 107 and 46.
- We see that the carry from the **seventh bit overflows** and is discarded, giving us the **erroneous result**: $107 + 46 = 25$.

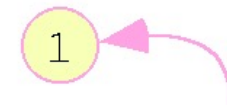


Diagram illustrating the binary addition of 107 and 46 using signed magnitude arithmetic. The carry from the seventh bit is shown as a 1, which is discarded.

$$\begin{array}{r} 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 1 \\ 0 + 0 \quad 1 \quad 0 \quad 1 \quad 1 \quad 1 \quad 0 \\ \hline 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \end{array}$$

Signed Integer Representation

- The signs in signed magnitude representation work just like the signs in pencil and paper arithmetic.

– Example: Using signed magnitude binary arithmetic, find the sum of (-46) and (-25).

$$\begin{array}{r} \text{Tecken bit} \quad \quad 1 \quad 1 \\ \boxed{1} \quad 0101110 \\ 1 \quad + \quad 0011001 \\ \hline 1 \quad 1000111 \end{array}$$

- Because the signs are the same, all we do is add the numbers and supply the negative sign when we are done.

Signed Integer Representation

- **Mixed sign addition (or subtraction)** is done the same way.
 - Example: Using signed magnitude binary arithmetic, find the **sum** of (+46) and (- 25).

$$\begin{array}{r} \\ \\ 0 \\ 1 \\ \hline 0 \end{array}$$

- The sign of the result gets the sign of the number that is larger.
 - Note the “borrows” from the second and sixth bits.

Signed Integer Representation

- Signed magnitude representation is *easy for people to understand*, but *it requires complicated computer hardware*.
- Another disadvantage of signed magnitude is that it allows two different representations for **zero: positive zero** and **negative zero**.
- For these *reasons* (among others) computers systems **employ complement systems** for numeric value representation.

Signed Integer Representation

- For example, using 8-bit **one's complement** representation:


+ 3 is: 00000011

- 3 is: 11111100

- In **one's complement** representation, as with signed magnitude, **negative values** are indicated by a **1** in the high order bit.
- Complement systems are useful because they **eliminate the need for subtraction**. The difference of two values is found by adding the minuend to the complement of the subtrahend.

Signed Integer Representation

- With **one's complement** addition, the carry bit is “***carried around***” and *added* to the sum.
 - Example: Using one's complement binary arithmetic, find the sum of 48 and - 19


$$\begin{array}{r} 11 \\ 00110000 \\ 11101100 \\ \hline 00011100 \\ +1 \\ \hline 00011101 \end{array}$$

We note that 19 in binary is 00010011,
so -19 in one's complement is: 11101100.

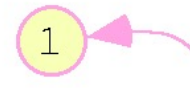
Signed Integer Representation

- To express a value in **two's complement** representation:
 - If the number is positive, just convert it to binary and you're done.
 - If the **number is negative**, find the **one's complement** of the number and **then add 1**.
- Example:
 - In 8-bit binary, 3 is:
00000011
 - -3 using one's complement representation is:
11111100
 - Adding 1 gives us -3 in two's complement form:
11111101

Signed Integer Representation

- With **two's complement arithmetic**, **all** we do is ***add our two*** binary numbers. Just **discard** any **carries emitting** from the **high order bit**.

- Example: Using one's complement binary arithmetic, find the sum of 48 and - 19.


$$\begin{array}{r} 1\ 1 \\ 00110000 \\ + 11101101 \\ \hline 00011101 \end{array}$$

We note that 19 in binary is: 00010011,
so -19 using one's complement is: 11101100,
and -19 using two's complement is: 11101101.

Signed Integer Representation

- When we use any **finite number** of bits to represent a number, we **always run the risk of the result of our calculations becoming too large or too small to be stored** in the computer.
- While we can't always **prevent overflow**, we can **always detect overflow**.
- In **complement arithmetic**, an **overflow condition is easy to detect**.

Signed Integer Representation

- Example:
 - Using **two's complement** binary arithmetic, find the sum of 107 and 46.
- We see that the **nonzero carry** from the **seventh bit overflows** into the **sign bit**, giving us the erroneous result: $107 + 46 = -103$.

$$\begin{array}{r} \overset{1}{\boxed{0}}1101011 \\ + \boxed{0}0101110 \\ \hline \boxed{1}0011001 \end{array}$$

The diagram illustrates the binary addition of 107 and 46 in two's complement. The first number, 107, is represented as 01101011 with a carry of 1 from the eighth bit. The second number, 46, is represented as 00101110. The sum is 10011001, where the leading 1 in the sign bit position indicates a negative result (-103). A red arrow points from the text "seventh bit overflows" to the carry bit 1.

But overflow into the sign bit does not always mean that we have an error.

Signed Integer Representation

- Example:

- Using **two's complement binary arithmetic**, find the sum of 23 and -9.
- We see that there is **carry into** the sign bit and **carry out**. The final result is correct: $23 + (-9) = 14$.

9		00001001
1'compl	-9	11110110
2'compl	-9	11110111

$$\begin{array}{r}
 \textcircled{1} \leftarrow \textcircled{1} 1111 \\
 00010111 \\
 + 11110111 \\
 \hline
 00001110
 \end{array}$$

Rule for detecting signed two's complement overflow: When the “carry in” and the “carry out” of the sign bit **differ**, **overflow has occurred**. If the carry into the sign bit equals the carry out of the sign bit, **no overflow has occurred**.

Signed Integer Representation

- Signed and unsigned numbers are both useful.
 - For example, **memory addresses are always unsigned.**
- Using the same number of bits, unsigned integers can express twice as many “positive” values as signed numbers.
- Trouble arises if an **unsigned value** “*wraps around.*”
 - In four bits: $1111 + 1 = 0000$.

Good programmers stay alert for this kind of problem

Signed Integer Representation

- Research into finding better arithmetic algorithms has continued for over 50 years.
- One of the many interesting products of this work is **Booth's algorithm**.
- In most cases, Booth's algorithm carries out multiplication faster and more accurately than naïve pencil-and-paper methods.
- The *general idea is to replace arithmetic operations with bit shifting* to the extent possible.

Signed Integer Representation

In **Booth's algorithm**:

- If the current multiplier bit is 1 and the preceding bit was 0, subtract the multiplicand from the product
- If the current multiplier bit is 0 and the preceding bit was 1, we add the multiplicand to the product
- **If we have a 00 or 11 pair, we simply shift.**
- Assume a mythical “0” starting bit
- Shift after each step

$$\begin{array}{r}
 \begin{array}{c} 0011 \\ \times 0110 \\ \hline \end{array} \\
 + 0000 \quad (\text{shift}) \\
 - 0011 \quad (\text{subtract}) \\
 + 0000 \quad (\text{shift}) \\
 + 0011 \quad (\text{add}) \\
 \hline
 00010010
 \end{array}$$

We see that $3 \times 6 = 18$!

Signed Integer Representation

- **Overflow** and **carry** are tricky ideas.
- Signed number overflow means nothing in the context of unsigned numbers, which **set a carry flag** instead of an **overflow flag**.
- If a carry out of the leftmost bit occurs with an *unsigned number*, overflow has occurred.
- Carry and overflow occur independently of each other.

The table on the next slide summarizes these ideas.

Signed Integer Representation

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

Tecken bit

Tecken bit

Tabell 2.2, sidan 87

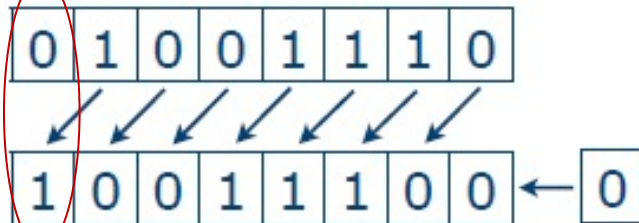
Signed Integer Representation

- We can do binary multiplication and division by 2 very easily using an arithmetic shift operation
- A left arithmetic shift inserts a 0 in for the rightmost bit and shifts everything else left one bit; in effect, it **multiplies by 2**
- A right arithmetic shift shifts everything one bit to the right, but copies the sign bit; it **divides by 2**
- Let's look at some examples.

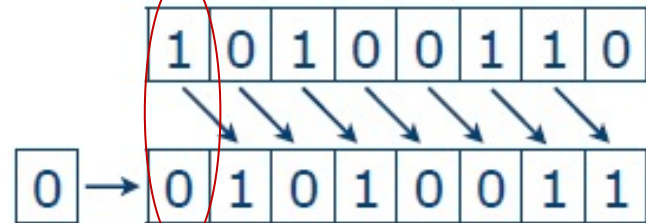
Shift

Logical shift

Left shift

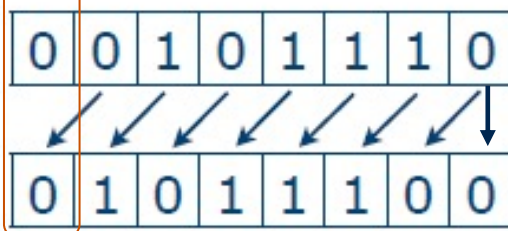


Right shift

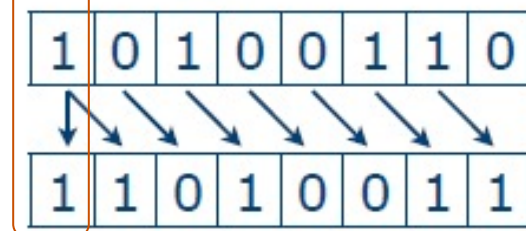


Arithmetic shift

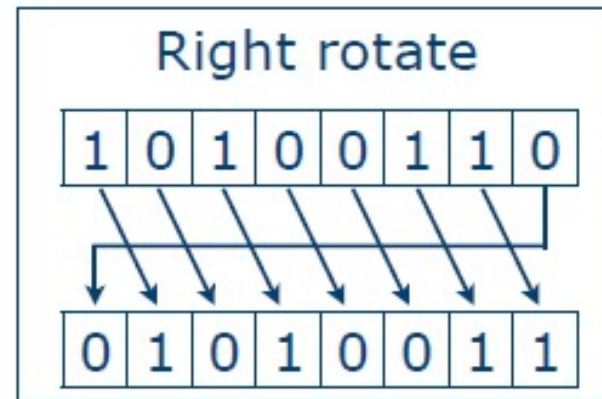
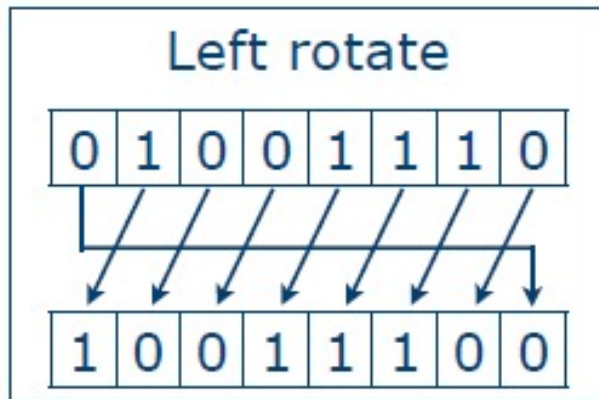
Left shift



Right shift



Rotate



Add-operationen

$$\begin{array}{r} \underline{1} \\ 4 \\ + 6 \\ \hline 10 \end{array}$$

$$\begin{array}{r} \underline{1} \\ 0100 \\ + 0110 \\ \hline 1010 \end{array}$$

$$\begin{array}{r} \underline{1} \quad \underline{1} \\ 1101 \\ + 1110 \\ \hline 11011 \end{array}$$

$$\begin{array}{r} 13 \\ + 14 \\ \hline 27 \end{array}$$

Subtraktion

$$\begin{array}{r} \underline{10} \\ 10 \\ - 6 \\ \hline 4 \end{array}$$

Decimal Talsystemet

$$\begin{array}{r} \underline{10} \\ 1010 \\ - 0110 \\ \hline 0100 \end{array}$$

Binära Talsystemet

$$\begin{array}{r} \underline{10} \ \underline{10} \ \underline{10} \ \underline{10} \ \underline{10} \ \underline{10} \\ 1000000 \\ - 000001 \\ \hline 011111 \end{array}$$

Bitwise and och or

0 1 0 0 1 1 1 0

&

1 0 1 0 1 0 1 0

=

0 0 0 0 1 0 1 0

0 1 0 0 1 1 1 0

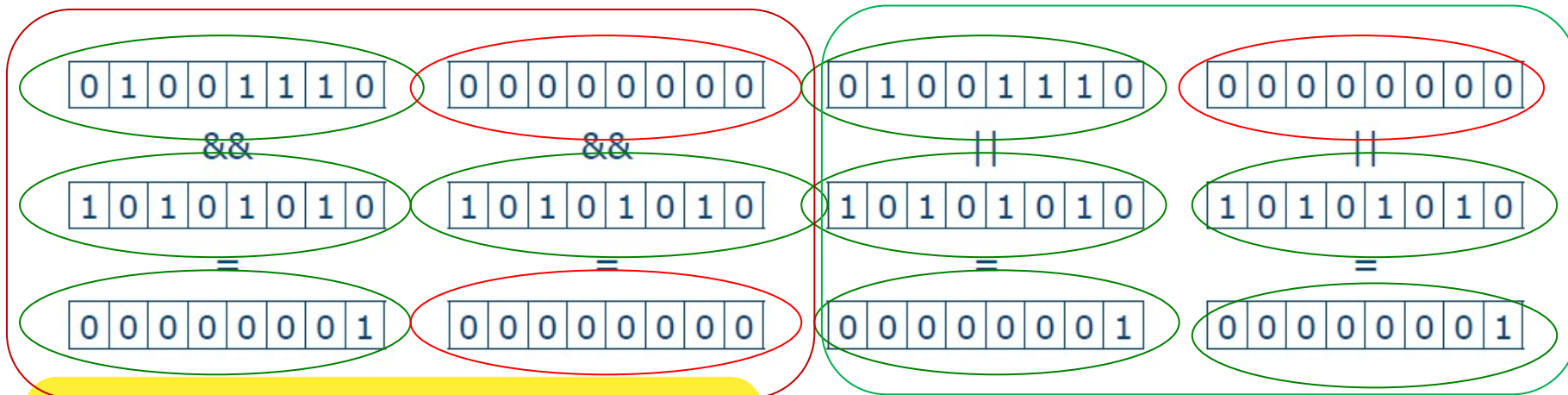
|

1 0 1 0 1 0 1 0

=

1 1 1 0 1 1 1 0

Logiskt and och or



If ((sant) && (Falskt)) { }

// här ska båda villkoren vara sant

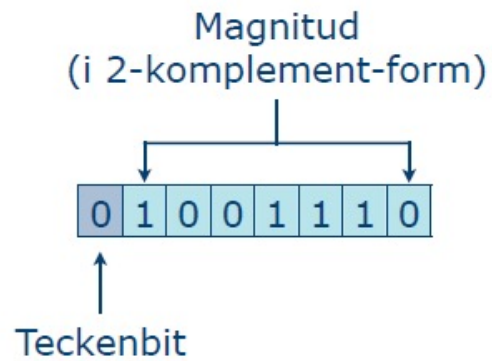
för att returnera (Sant)

If ((Sant) || (Falskt)) { }

Gröna ringar = sant

röda ringar = falskt

2-komplementsrepresentation



1. Invertera talet
2. Addera 1

0	1	0	0	1	1	1	0
---	---	---	---	---	---	---	---

 = 78

1	0	1	1	0	0	1	0
---	---	---	---	---	---	---	---

 = -78

0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

 = 0

1	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

 = -128

Subtraktion genom negerad addition

$$2 - 1 = 2 + (-1) = 1$$

<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr></table>	0	0	0	0	0	0	1	0		<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr></table>	0	0	0	0	0	0	1	0						
0	0	0	0	0	0	1	0																	
0	0	0	0	0	0	1	0																	
	-	=	+	=	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr></table>	0	0	0	0	0	0	0	1											
0	0	0	0	0	0	0	1																	
<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr></table>	0	0	0	0	0	0	0	1						<table border="1"><tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	1	1	1	1	1	1	1	1		
0	0	0	0	0	0	0	1																	
1	1	1	1	1	1	1	1																	

Multiplikation

$$\begin{array}{r}
 13 \\
 * 123 \\
 \hline
 39 \\
 26 \\
 + 13 \\
 \hline
 1599
 \end{array}$$

				1	1	0	1
			*	1	0	0	1
			<u>1</u>	1	1	0	1
			0	0	0	0	
		0	0	0	0		
+	1	1	0	1			
	1	1	1	0	1	0	1

Signed Integer Representation

Example:

Multiply the value 11 (expressed using 8-bit signed two's complement representation) by 2.

We start with the binary value for 11:

00001011 (+11)

We shift left one place, resulting in:

00010110 (+22)

The sign bit has not changed, so the value is valid.

To multiply 11 by 4, we simply perform a left shift twice.

Signed Integer Representation

Example:

Divide the value 12 (expressed using 8-bit signed two's complement representation) by 2.

We start with the binary value for 12:

00001100 (+12)

We shift right one place, resulting in:

00000110 (+6)

(Remember, we carry the sign bit to the left as we shift.)

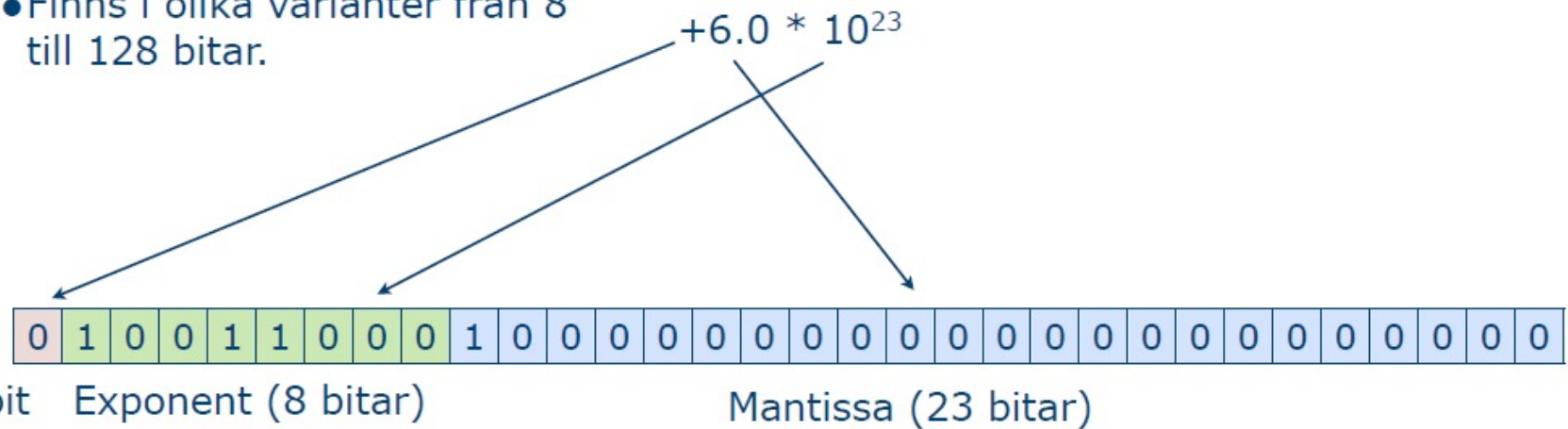
To divide 12 by 4, we right shift twice.



Part II: Floating-Point Representation

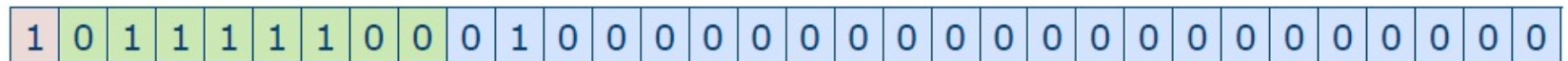
IEEE 754

- Standard för flyttal.
- Finns i olika varianter från 8 till 128 bitar.



Flyttalsformatet

$$(-1)^{\text{teckenbit}} * 2^{\text{exponent}-127} * 1.\text{mantissa}$$



Teckenbit Exponent (8 bitar)

Mantissa (23 bitar)

Efter normalisering:

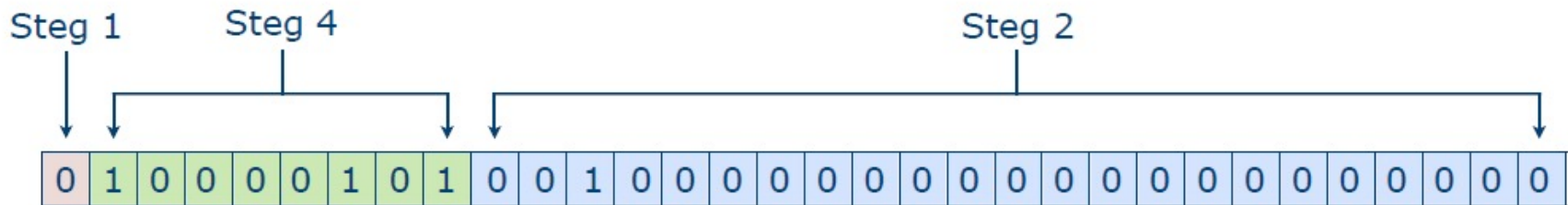
1.0010000₂

Konvertering till IEEE 754

1. Bestäm teckenbiten
2. Normalisera mantissan
3. Beräkna exponenten
4. Addera 127₁₀ till exponenten

Steg 3 $\xrightarrow{72_{10} = 100\ 1000.0_2}$ $\uparrow\uparrow\uparrow\uparrow\uparrow\uparrow$

$$6_{10} + 127_{10} = 133_{10} = 1000\ 0101_2$$



IEEE 754 - Specialvärden

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	= 0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	= -0
0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	= ∞
1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	= -∞
X	1	1	1	1	1	1	1	1	≠0																		= NaN					

$$X = \{0, 1\}$$

Floating-Point Representation

- The signed magnitude, one's complement, and two's complement representation that we have just presented deal with **signed integer values only**.
- Without **modification**, these formats are not useful in **scientific** or **business applications** that deal with real number values.
- Floating-point representation solves this problem.

Floating-Point Representation

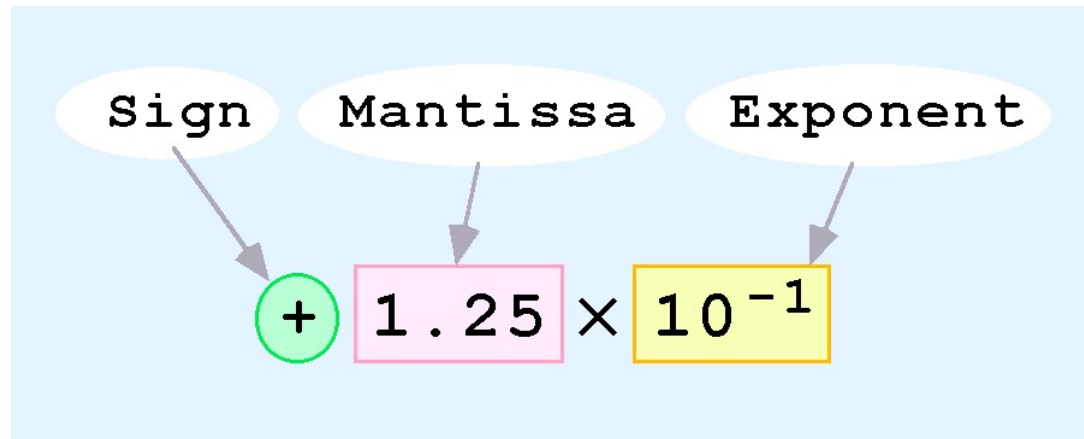
- If we are clever programmers, we can perform **floating-point calculations** using any integer format.
- This is called ***floating-point emulation***, because floating point values **aren't stored as such**; we just **create programs** that make it **seem** as if floating-point values are being used.
- Most of today's computers are equipped with specialized hardware that performs floating-point arithmetic with no special programming required.

Floating-Point Representation

- Floating-point numbers allow an arbitrary number of decimal places to the right of the decimal point.
 - For example: $0.5 \times 0.25 = 0.125$
- They are often expressed in scientific notation.
 - For example:
 $0.125 = 1.25 \times 10^{-1}$
 $5,000,000 = 5.0 \times 10^6$

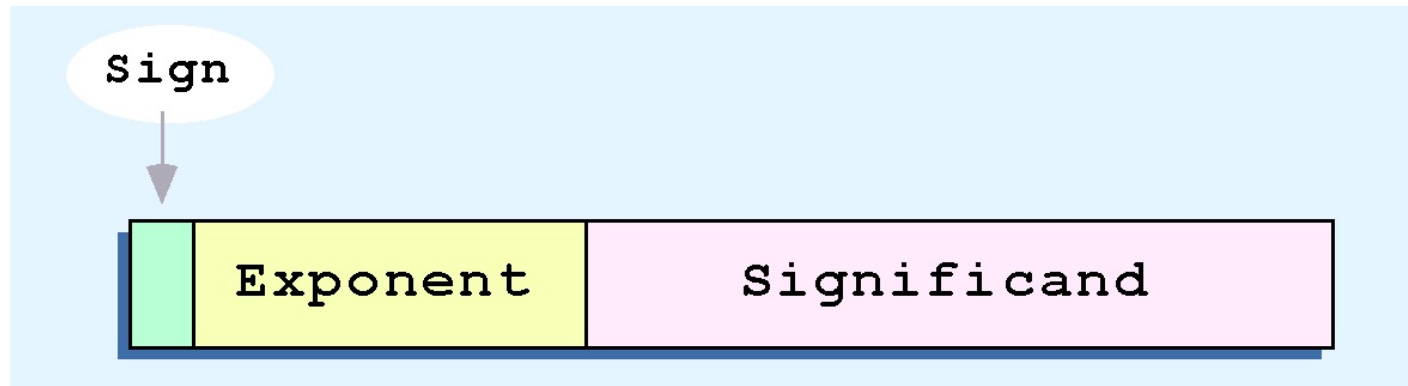
Floating-Point Representation

- Computers use a form of scientific notation for floating-point representation
- Numbers written in scientific notation have three components:



Floating-Point Representation

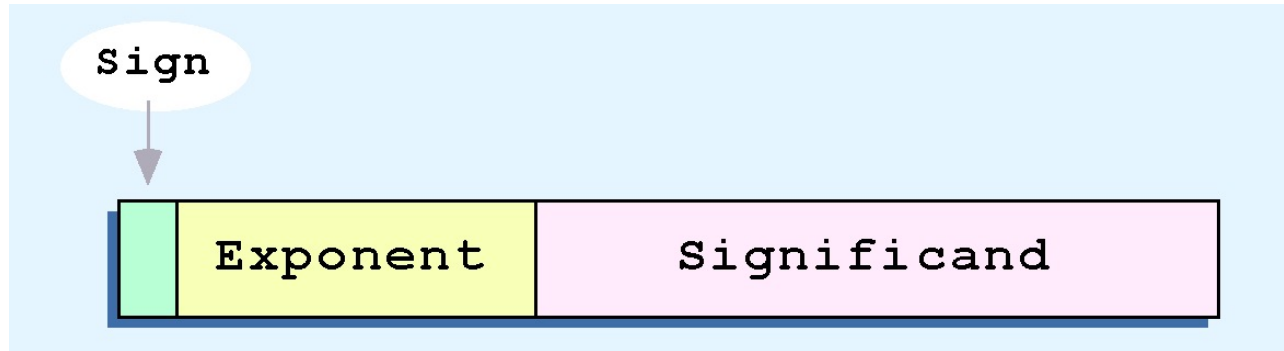
- Computer representation of a floating-point number consists of three fixed-size fields:



- This is the standard arrangement of these fields.

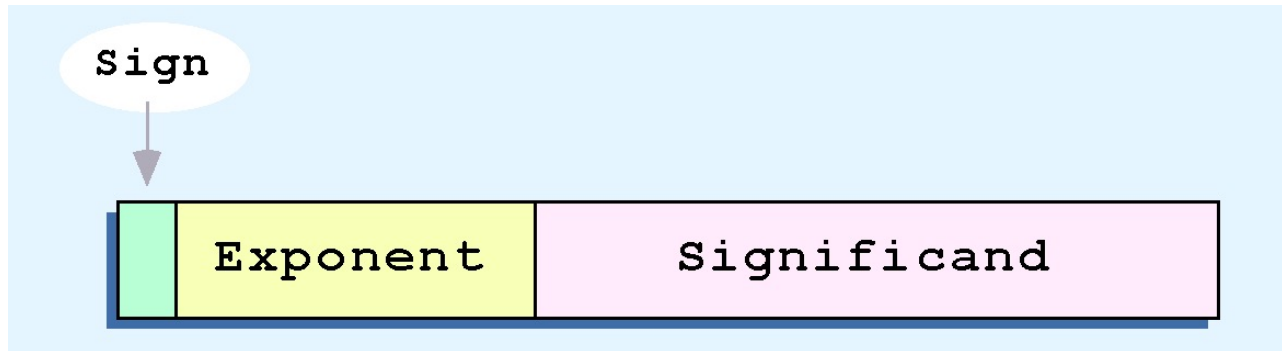
Note: Although “significand” and “mantissa” do not technically mean the same thing, many people use these terms interchangeably. We use the term “significand” to refer to the fractional part of a floating point number.

Floating-Point Representation



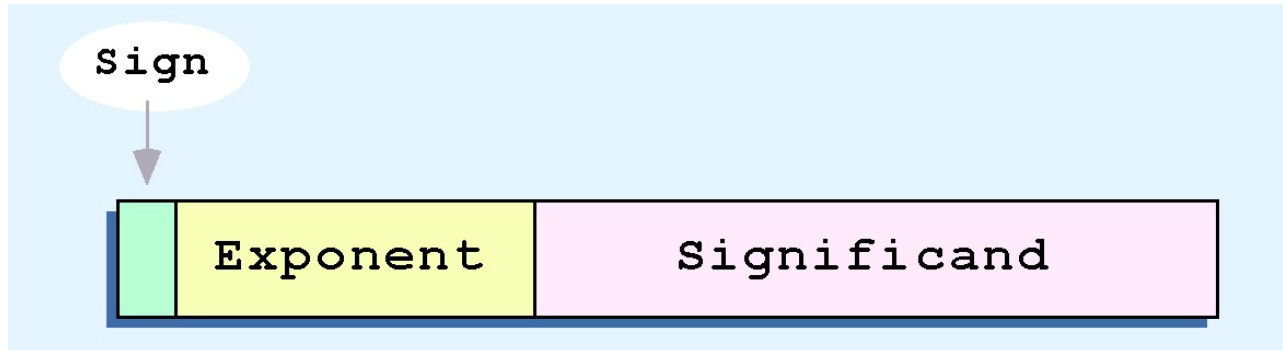
- The one-bit sign field is the sign of the stored value.
- The size of the exponent field determines the range of values that can be represented.
- The size of the significand determines the precision of the representation.

Floating-Point Representation



- We introduce a hypothetical “Simple Model” to explain the concepts
- In this model:
 - A floating-point number is **14 bits** in length
 - The **exponent** field is **5 bits**
 - The **significand** field is **8 bits**

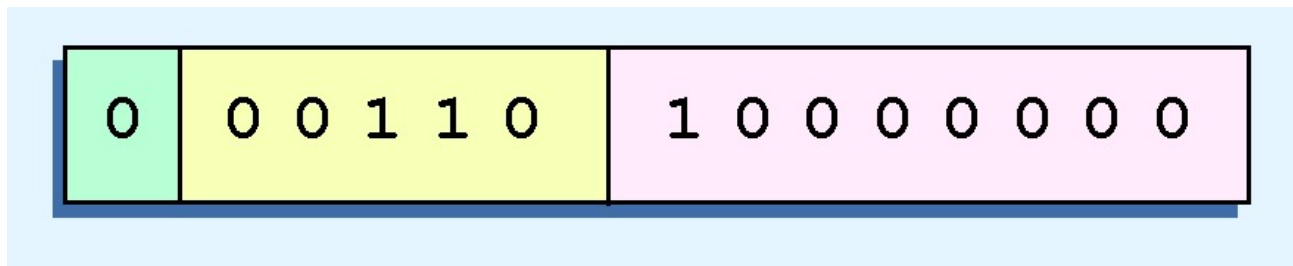
Floating-Point Representation



- The significand is always preceded by an implied binary point.
- Thus, the significand always contains a fractional binary value.
- The exponent indicates the power of 2 by which the significand is multiplied.

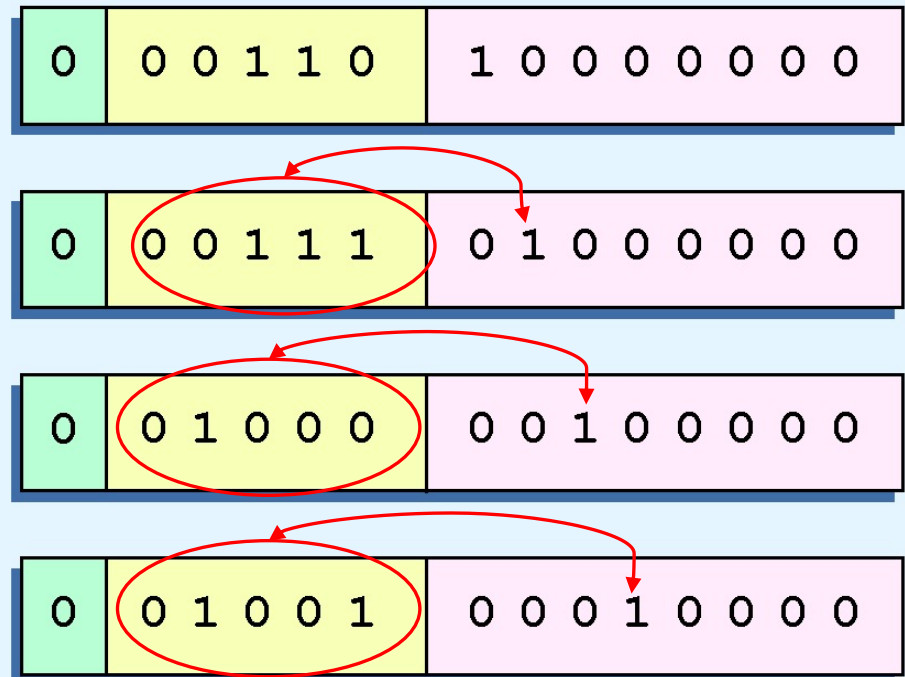
Floating-Point Representation

- Example:
 - Express 32_{10} in the simplified 14-bit floating-point model.
- We know that 32 is $2^5 \rightarrow (32 = 2^5)$ So in (binary) scientific notation $32_{10} \Rightarrow (1.0 \times 2^5)_2 = (0.1 \times 2^6)_2$
 - In a moment, we'll explain why we prefer the second notation versus the first.
- Using this information, we put 110 ($= 6_{10}$) in the exponent field and 1 in the significand as shown.

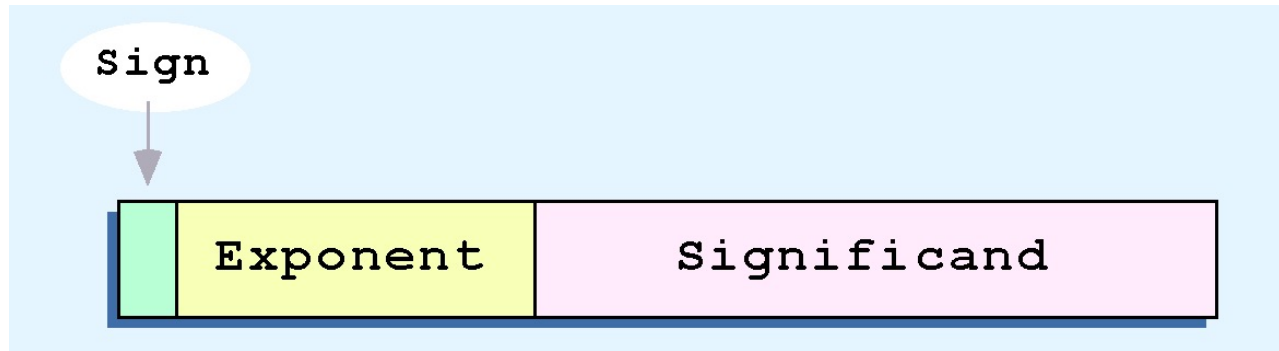


Floating-Point Representation

- The illustrations shown at the right are *all* equivalent representations for 32 using our simplified model.
- Not only do these synonymous representations waste space, but they can also cause confusion.



Floating-Point Representation



- Another problem with our system is that we have made no allowances for negative exponents. We have no way to express $0.5 (=2^{-1})$! (Notice that there is no sign in the exponent field.)

**All of these problems can be fixed
with no changes to our basic model.**

Floating-Point Representation

- To resolve the problem of synonymous forms, we establish a rule that the first digit of the significand must be 1, with no ones to the left of the radix point.
- This process, called *normalization*, results in a unique pattern for each floating-point number.
 - In our simple model, all significands must have the form 0.1xxxxxxxxx
 - For example, $4.5_{10} = 100.1_2 \times 2^0 = 1.001 \times 2^2 = \text{0.1001} \times 2^3$.
The last expression is correctly normalized.

$0.5 = 1/2 = 2^{-1}$

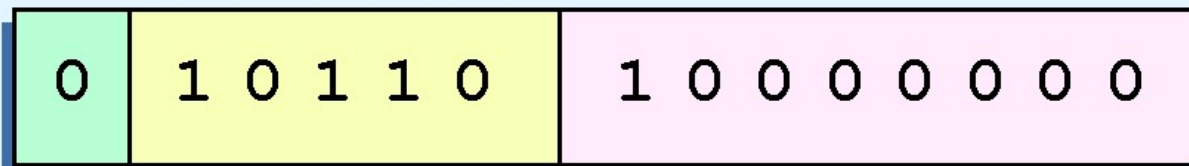
$1 = 2^0$

Floating-Point Representation

- To provide for **negative exponents**, we will use a *biased exponent*.
- A bias is a number that is approximately midway in the range of values expressible by the exponent. We **subtract the bias** from the **value in the exponent** to **determine its true value**.
 - *In our case, we have a 5-bit exponent. We will use 16 for our bias. This is called excess-16 representation.*
- In our model, exponent values less than 16 are negative, representing fractional numbers.

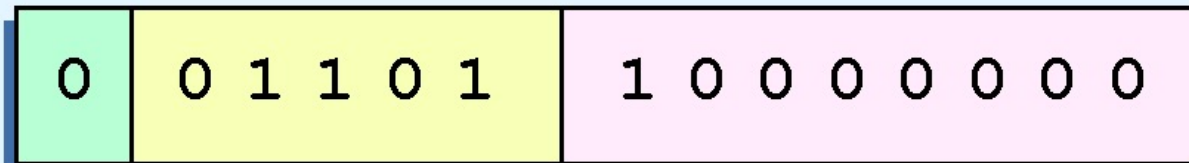
Floating-Point Representation

- Example:
 - Express 32_{10} in the revised 14-bit floating-point model.
- We know that $32 = 1.0 \times 2^5 = 0.1 \times 2^6$.
- To use our excess 16 biased exponent, we add 16 to 6 $\rightarrow (16+6) = 22$, giving 22_{10} ($=10110_2$).
- So we have:



Floating-Point Representation

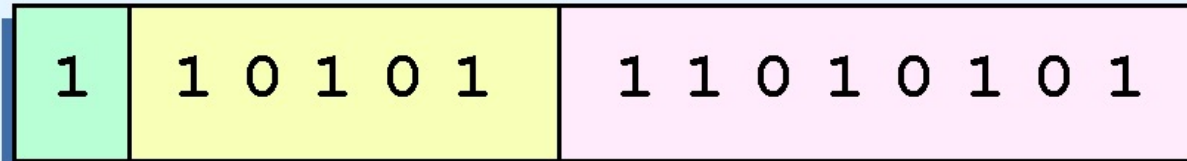
- Example:
 - Express 0.0625_{10} in the revised 14-bit floating-point model.
- We know that 0.0625 is $1/2^4 = 2^{-4}$. So in (binary) scientific notation $0.0625 = 1.0 \times 2^{-4} = 0.1 \times 2^{-3}$.
- To use our excess 16 biased exponent, we add (16) to $(-3) \rightarrow (16+(-3))=13$, giving $13_{10} (=01101_2)$.



Floating-Point Representation

- Example:
 - Express -26.625_{10} in the revised 14-bit floating-point model.
- We find $26.625_{10} = 11010.101_2$. Normalizing, we have: $26.625_{10} = 0.11010101_2 \times 2^5$.
- To use our excess 16 biased exponent, we add 16 to 5, giving 21_{10} ($=10101_2$). We also need a 1 in the sign bit.

$$0.625 = (0.5 + 0.125)_{10} = 2^{-1} + 2^{-3} = 1/2 + 1/3 = 0.101_2$$



Floating-Point Representation

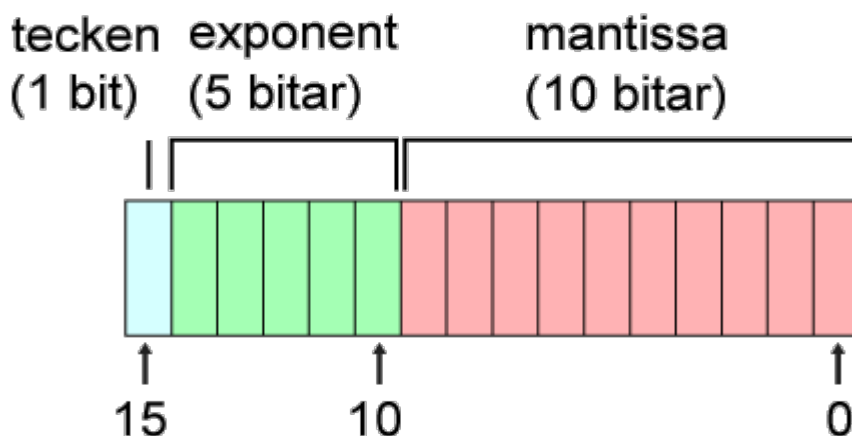
- The IEEE has established a standard for floating-point numbers
- The IEEE-754 *single precision* floating point standard uses an 8-bit exponent (with a bias of 127) and a 23-bit significand.
- The IEEE-754 *double precision* standard uses an 11-bit exponent (with a bias of 1023) and a 52-bit significand.

Floating-Point Representation

- In both the IEEE single-precision and double-precision floating-point standard, the significant has an implied 1 to the LEFT of the radix point.
 - The format for a significand using the IEEE format is: 1.xxx...
 - For example, $4.5 = .1001 \times 2^3$ in IEEE format is $4.5 = 1.001 \times 2^2$. The 1 is implied, which means it does not need to be listed in the significand (the significand would include only 001).

Exempel Flyttal - (IEEE 754)

- Flyttal är ett sätt att representera rationella tal.
- IEEE 754 finns i flera varianter beroende på antal bitar.
- Vi använder (t.ex) ett 16 bitars flyttal som:



För att räkna ut värdet v för ett flyttal kan vi använda formeln

$$v = (-1)^{\text{teckenbiten}} \cdot 2^{\text{exponent}-15} \cdot (1, \text{mantissa}_2)$$

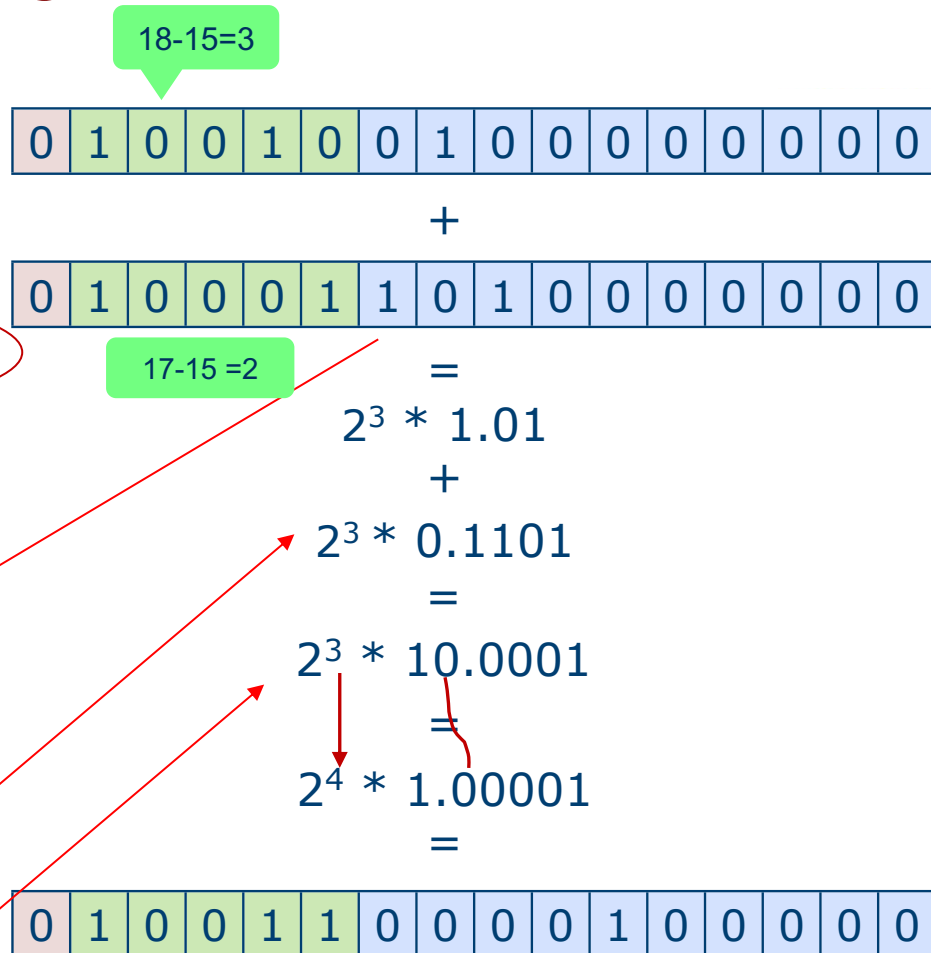
- Exponenten - 15 kommer från att vi har 5 bitar för exponenten: $2^5/2 - 1 = 32/2 - 1 = 16 - 1 = 15$.

På det viset kan man representera tal som är både större och mindre än 1, eftersom om exponentbitarna ≥ 15 får man en positivt exponent, vilket ger ett flyttal > 0 .

Om man istället har ett värde i exponentbitarna < 15 får man en negativ exponent, vilket kommer att flytta decimalkommat så att flyttalet < 0 .

Flyttalsaritmetik- Addition

- Kräver att båda flyttalen har samma exponent
 - Skifta upp det minst av de båda flyttalen
 - Utför beräkningen
 - Normera resultatet till ett nytt flyttal



$$2^2 * 1.101 = 2^3 * 0.1101$$

$$2^3 * 1.01 + 2^3 * 0.1101 =$$

$$\begin{array}{r} 1.0100 + \\ 0.1101 \\ \hline 10.0001 \end{array}$$

Flyttalsaritmetik- Multiplikation

- Kräver inte att båda flyttalen har samma exponent
 - Addera exponenterna
 - Multiplicera
 - Normera resultatet till ett nytt flyttal

0	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

*

0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

=

$$(2^2 * 1.1) * (2^{-1} * 1)$$

=

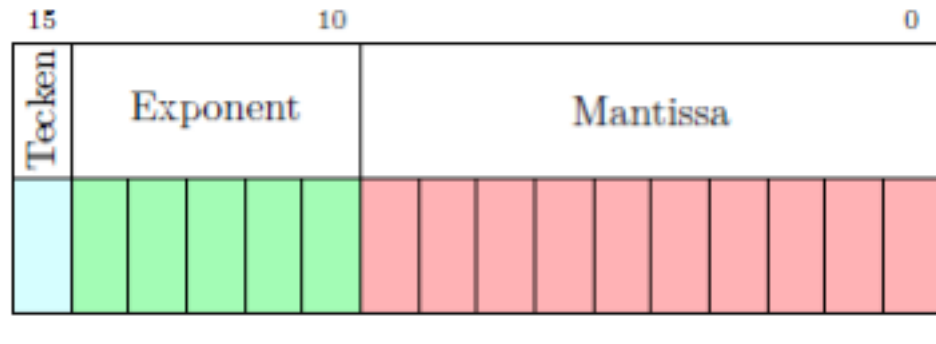
$$2^{2-1} * (1.1 * 1) = 2^1 * 1.1$$

=

0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Addition av Flyttal:

Flyttal är ett sätt att representera stora, små och rationella tal. I nedanstående uppgift används standarden IEEE 754 för 16-bitars flyttal. Se bilden nedan om flyttals binärt lagring.



Beräkna operation **A+B**. Svara med ett 16-bitars binär flyttal enligt bilden ovan och IEEE 754.

Flyttal A : 0 10110 110000000

Flyttal B : 0 10101 101000000

Formeln för att få ut det decimala värdet v ur ett flyttal:

$$v = (-1)^{\text{teckenbit}} \times 2^{\text{exponent}-15} \times (1, \text{mantissa})_2$$

- **Steg 1:** Flyttal A: Har exponentbitarna 10110_2 vilket ger oss 22_{10} och Flyttal B har exponentbitarna 10101_2 vilket ger oss 21_{10} . Enligt formeln tar vi då $22-15$ då blir 7 för A . $21-15$ bli 6 för B.
- **Steg 2:** Omvandla så att vi ser exponenterna: $2^{22-15} * 1,11 + 2^{21-15} * 1,101 = 2^7 * 1,11 + 2^6 * 1,101$
- **Steg 2:** Skifta upp det minsta talet så att båda talen har samma exponent (**normalisera**):

$$2^6 * 1,101 \rightarrow 2^7 * 0,1101$$

Nu har talen samma exponent så mantissorna adderas

- **Steg 3:** Addera mantissorna för de båda talen: $1,11_2 + 0,1101_2 = 10,1001_2$ (exponenten är 7) $\rightarrow 1,01001_2$ (exponenten bli 8)

Vi ska dra av 1 då mantissan blir ,01001₂

- **Steg 4:** Skapa ett flyttal enligt stegen ovan.

$$\text{Exponenten: } 15+8=23_{10} \rightarrow 10111_2$$

- **Resultatet är:** 0 10111 0100100000

Conclusion

- Data representation (Binary)
- Number system (Decimal, Binary , hexadecimal)
- Converting Between bases
- Signed Integer Representation (signed and unsigned)
 - one's complement
 - Two's complement
 - Shift & rotate
- Floating- Point Representation
 - Different standard
 - IEEE-754