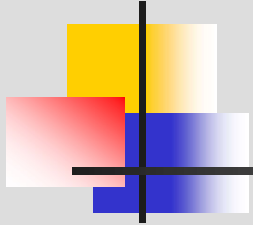




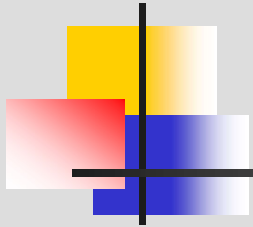
Arithmetic ve Logical Operations

- ALU (Arithmetic Logical Unit): CPU nun Aritmetik ve logic islemlerinin yapildigi kismına denir.
 - Temel iki operation
 - Addition (Toplama)
 - Negation (NOT islemi)
- Islemler sayıların temsil edilme şekline göre değişiklik gösterirler.
- Hangi sayı temsil şekli işlemleri daha çok kolaylaştırıyorsa o bilgisayar tasarımı kullanılmaktadır.



Overflow

- Bilgisayar mimarisi sabit uzunluktaki veriler üzerinde işlem yaparlar.
 - 32-bit / 64-bit işlemci
- Overflow
 - Eger islemin sonucu o işlem için ayrılan alana sığmazsa buna overflow denir
 - Overflow detection (tesbiti) önemli
 - Aksi halde yanlış sonuçlar kullanılarak işlemler yapılır.

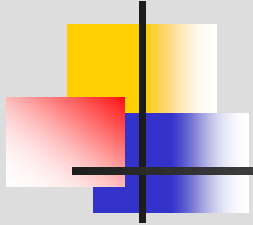


Boolean Operations

- Bir boolean variable iki degerden birini alabilir.
 - False (0)
 - True (1)

input	zero	one	invert	same
0	0	1	1	0
1	0	1	0	1

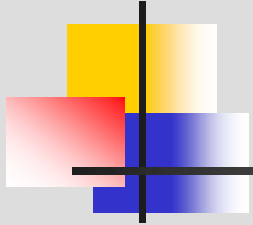
Unary Boolean Operations



Boolean Operations

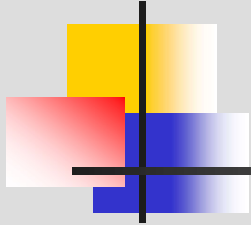
- Binary boolean operations
 - Iki variable uzerinde islem yapilir

a	b	and	or	nand	nor	xor	xnor
0	0	0	0	1	1	0	1
0	1	0	1	1	0	1	0
1	0	0	1	1	0	1	0
1	1	1	1	0	0	0	1



not x, y
and x, y, z
or x, y, z
nand x, y, z
nor x, y, z
xor x, y, z
xnor x, y, z

SAL daki Logical Operations



Maskeleme İşlemi

- Boolean işlemlerinde birden fazla değişken aynı bellek gözüne yazıldığında kullanılırlar.
- Değişkenlerin bellek gözünden çıkarılması işlemine masking denir.

cell:	.word	0x43686172
mask1:	.word	0xff000000
mask2:	.word	0x00ff0000
mask3:	.word	0x0000ff00
mask4:	.word	0x000000ff

and result, cell, mask1

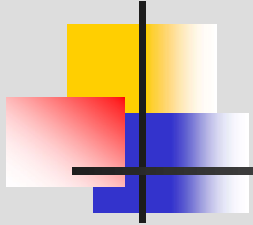
Shift Operations (Oteleme Islemleri)

➤ Logical Shift

➤ Logical right shift:

- Bitler 1 pozisyon saga otelenir
- En sagdaki bit (LSB) atilir
- En soldaki (MSB) bite 0 atanir.

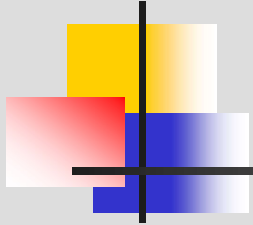
0	1	1	0	0	1	1	1
0	0	1	1	0	0	1	1
0	0	0	1	1	0	0	1



Shift Operations

- Logical Left Shift
 - Bitler bir pozisyon sola otelenirler
 - En soldaki bit (MSB) atilir
 - En sagdaki bit (LSB) 0 olur

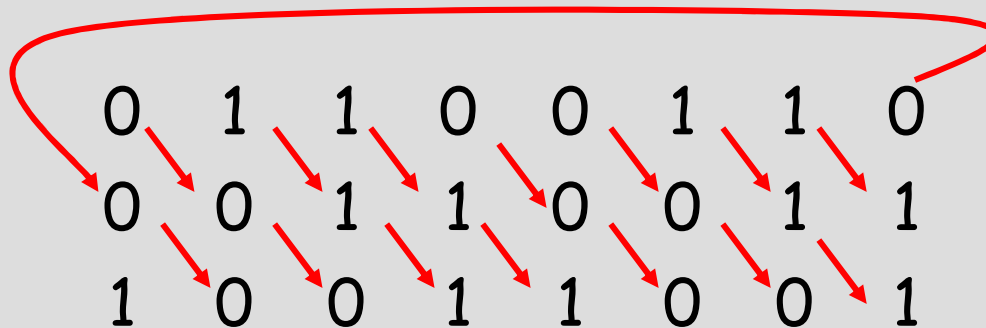
0	1	1	0	0	1	1	1
1	1	0	0	1	1	1	0
1	0	0	1	1	1	0	0

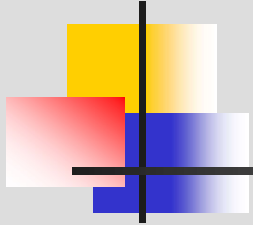


Rotate

➤ Rotate Right

- Bitler saga dogru bir pozisyon kaydirilir
- LSB, MSB olur

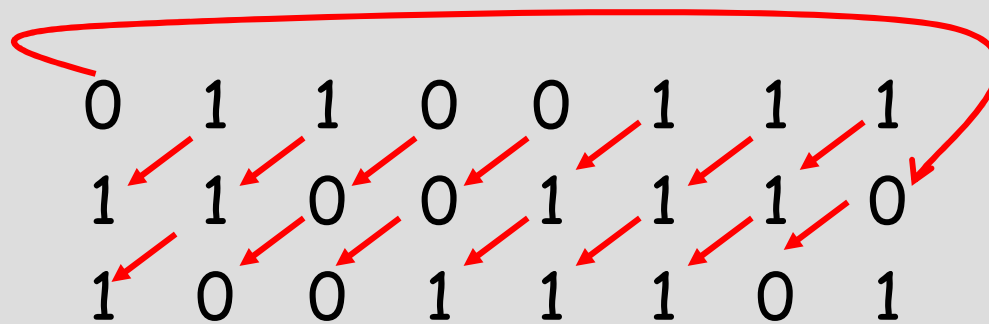


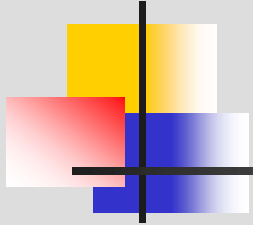


Rotate

➤ Rotate Left

- Bitler bir pozisyon sola otelenir
- MSB, LSB olur

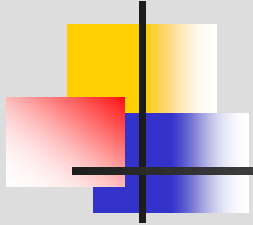




Arithmetic Shift

- Arithmetic left shift
- Logical left shift le ayni
 - Bitler bir pozisyon sola otelenirler
 - En soldaki bit (MSB) atilir
 - En sagdaki bit (LSB) 0 olur

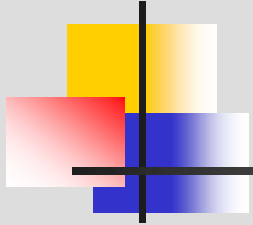
0	1	1	0	0	1	1	1
1	1	0	0	1	1	1	0
1	0	0	1	1	1	0	0



Arithmetic Left Shift

- Sayının arithmetic 1 bit otelenmesi o sayının 2 ile carpimi anlamına gelir.

1	0	0	1	1	1	0	1	-99
0	0	1	1	1	0	1	0	58
0	1	1	1	0	1	0	0	116



Arithmetic Right Shift

- Logical right shift gibi. Tek farkı sign bit extended (MSF sayinin sign bitiyle ayni)
- Bir sayinin 1 bit arithmetic saga otelenmesi o sayinin 2 ile bolunmesi anlamina gelir.

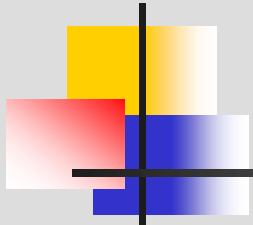
0	1	1	0	0	1	1	1	103
0	0	1	1	0	0	1	1	51
0	0	0	1	1	0	0	1	25

Addition (Toplama) / Subtraction (Cikarma)

Unsigned Integers:

$$\begin{array}{r}
 1 \\
 010010 \text{ (} 18_{10} \text{)} \\
 + 011000 \text{ (} 24_{10} \text{)} \\
 \hline
 101010 \text{ (} 42_{10} \text{)}
 \end{array}$$

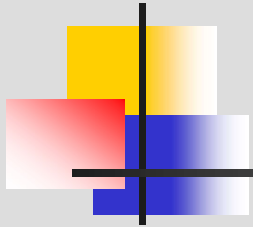
c_i	x_i	y_i	z_i	c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



Unsigned integers

a	b	a-b
0	0	0
0	1	borrow
1	0	1
1	1	0
10	1	1

$$\begin{array}{r} 01000 \\ - 00110 \\ \hline 00010 \end{array}$$



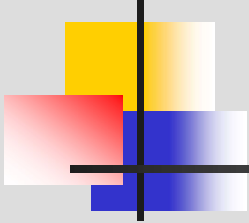
Sign magnitude

$$\begin{array}{r} 1\ 00100 \\ +\ 1\ 00101 \\ \hline 1\ 01001 \end{array}$$

$$\begin{array}{r} 0\ 00110 \\ +\ 1\ 10010 \\ \hline \end{array}$$

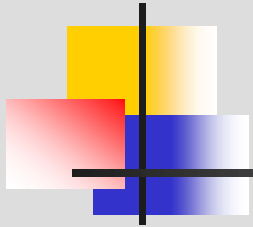


$$\begin{array}{r} 10010 \\ -\ 00110 \\ \hline 1\ 01100 \end{array}$$

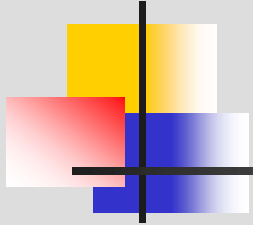


Two's Complement Addition/Subtraction

- Isaret bitlerine bakilmaksizin addition icin ayni algoritma uygulanir.
- Substraction additive inverse alinip addition algoritmasi kullanilmak suretiyle gerceklestirilir.
- Unsigned number larin toplami icin kullanan ayni devre two's complementi icin toplama ve cikarmada kullanilabilir



$$\begin{array}{r} 0010 \ 0010 \\ + 1001 \ 1000 \\ \hline 1011 \ 1010 \end{array} \quad \begin{array}{l} 34_{10} \\ -104_{10} \\ -70 \end{array}$$



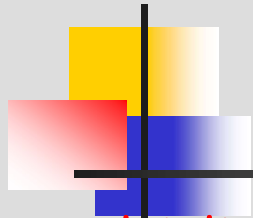
Overflow

$$\begin{array}{r} \overset{1}{\text{---}} \quad \overset{1}{\text{---}} \\ 1111 \ 1000 \quad -8_{10} \\ + 1111 \ 1000 \quad -8_{10} \\ \hline 1111 \ 0000 \quad -16_{10} \end{array}$$

dogru sonuc
no overflow

$$\begin{array}{r} \overset{0}{\text{---}} \quad \overset{1}{\text{---}} \\ 0111 \ 1110 \quad 126_{10} \\ + 0110 \ 0000 \quad 96_{10} \\ \hline 1101 \ 1110 \quad -34_{10} \end{array}$$

yanlis sonuc
overflow

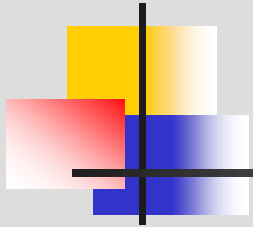


Multiplication

multiplicand $\rightarrow 1\ 1\ 0\ 1 = -3_{10}$

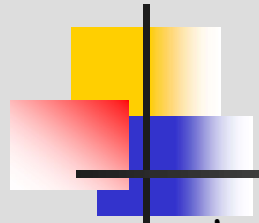
multiplier $\rightarrow 0\ 1\ 1\ 0 = 6_{10}$

$$\begin{array}{r} \times \\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 1 = -3_{10} \\ \times 0\ 0\ 0\ 0\ 0\ 1\ 1\ 0 = 6_{10} \\ \hline 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 1 \\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 1 \\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\ + 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\ \hline 1\ 1\ 1\ 0\ 1\ 1\ 1\ 0 = -18 \end{array}$$



$$\begin{array}{r} 11111000 = -8_{10} \\ \times 11111000 = -8_{10} \\ \hline 00000000 \\ 00000000 \\ 00000000 \\ 11111000 \\ 11111000 \\ 11111000 \\ 11111000 \\ + 11111000 \\ \hline 01000000 = 64 \end{array}$$

X ve Y nin carpim programi



```

        .data
X:      .word
Y:      .word
ls_sum  .word  0
test:   .word

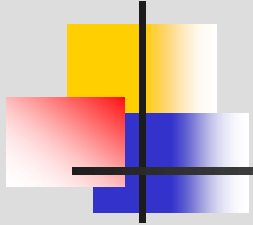
        ms_sum:      .word  0
        bitsum:      .word
        mask:        .word  0x1
    
```

.text

```

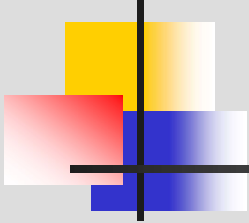
__start:and    test, X, mask        # strip off appropriate multiplier bit
           beqz   test, shift      # skip addition if multiplier is zero

           add    ms_sum, ms_sum, Y # add partial sum
shift:      and    bitsum, ms_sum, 1 # determine lsb of ms_sum
           or     ls_sum, ls_sum, bitsum # place lsb of ms_sum in lsb of ls_sum
           ror    ls_sum, ls_sum, 1  # shift ls_sum, moving new bit into msb
           sra    ms_sum, ms_sum, 1  # shift ms_sum, maintaining sign
           sll    mask, mask, 1      # update index
           bnez   mask, __start      # branch if not last iteration
           done
    
```



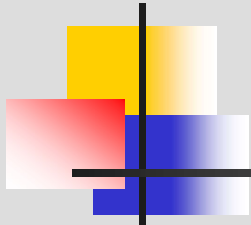
Floating Point (FP) Arithmetic

- Bilgisayar designında floating point sayılarınmin gosterimi onemli bir yer tutar.
- Duyarliligi yuksek olmasi istenilen islemlerde floating point islemlerinin hizli olmasi istenir
- Floating Point Operations Per Second (FLOPS)
 - Scientific bilgisayarların performans karsilastirimlarında kullanilir.
- Floating Point Operations Integer Operationlarından daha yavastir



Hardware versus Software Calculatiuons

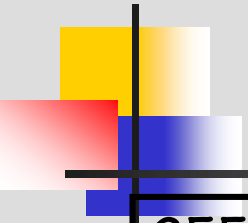
- Hesaplamalar nasıl yapılmalı
 - Hardware Implementation: Devreler (circuits)
FP işlemlerini yapar.
 - Hızlı
 - Pahalı
 - Software
 - Ucuz (devre acısından)
 - Yavaş (10 un bir kaç kuvveti kadar yavaş)



$$2.25 + 134.0625$$

0	1	0	0	0	(1)	0	0	1	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	1	1	0	(1)	0	0	0	0	1	1	0	0	0	0

	0	1	0	0	0	1	1	0	(0)	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0
+	0	1	0	0	0	0	1	1	0	(1)	0	0	0	0	0	1	1	0	0	0	0	0	1	0	0	0	0	0
<hr/>																												
	0	1	0	0	0	0	1	1	0	(1)	0	0	0	1	0	0	0	1	0	1	0	0	0	0	0	0	0	0

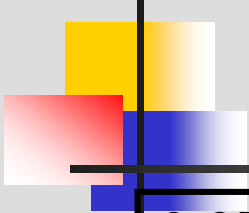


255.0625 + 134.0625

0	1000	0110	(1)	111	1111	0001	0000	0000	0000
0	1000	0110	(1)	000	0110	0001	0000	0000	0000

	0	1000	0110	(1)	111	1111	0001	0000	0000	0000
+	0	1000	0110	(1)	000	0110	0001	0000	0000	0000
<hr/>										
	0	1000	0110	(11)	000	0101	0010	0000	0000	0000

0	1000	0111	(1)	100	0010	1001	0000	0000	0000
---	------	------	-----	-----	------	------	------	------	------



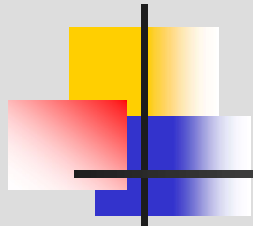
2.25 + -134.0625

0	1000	0110	(0)000	0010	0100	0000	0000	0000
1	1000	0110	(1)000	0110	0001	0000	0000	0000

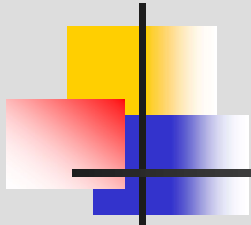
	0000	0000	0000	0010	0100	0000	0000	0000
+	1111	1111	0111	1001	1111	0000	0000	0000
<hr/>								
	1111	1111	0111	1100	0011	0000	0000	0000

(1)000	0011	1101	0000	0000	0000
--------	------	------	------	------	------

1	1000	0110	(1)000	0011	1101	0000	0000	0000
---	------	------	--------	------	------	------	------	------

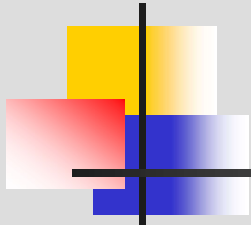


	.data	
FloatX:	.float	134.0625
FloatY	.float	2.25
Float_X_plus_Y:	.float	
X:	.word	
Y:	.word	
X_F:	.word	
X_E:	.word	
Y_F:	.word	
Y_E:	.word	
X_time_Y:	.word	
X_plus_Y:	.word	
X_plus_Y_F:	.word	
X_plus_Y_E:	.word	
X_plus_Y_S:	.word	
small_F:	.word	
diff:	.word	
F_mask:	.word	0x007ffffff
E_mask:	.word	0x7f800000
S_MASK:	.word	0x80000000
Hidden_one:	.word	0x00800000
zero:	.word	0
max_F:	.word	0x01000000



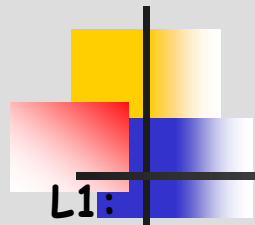
```
.text
# Extract E (exponent) and F (significand).
__start: move    X, FloatX
          and     X_F, X, F_Mask           # get X_F
          or      X_F, X_F, Hidden_one     # add hidden bit
          bgtz    X, DoX_E                 # skip if positive
          sub     X_F, zero, X_F           # convert to 2's comp.
DoX_E:    and     X_E, X, E_mask           # get X_E
          srl     X_E, X_E, 23             # align
          sub     X_E, X_E, 127            # convert to 2's comp.

          move    Y, FloatY
          and     Y_F, Y, F_Mask           # get Y_F
          or      Y_F, Y_F, Hidden_one     # add hidden bit
          bgtz    Y, DoY_E                 # skip if positive
          sub     Y_F, zero, Y_F           # convert to 2's comp.
DoY_E:    and     Y_E, Y, E_mask           # get Y_E
          srl     Y_E, Y_E, 23             # align
          sub     Y_E, Y_E, 127            # convert to 2's comp.
```

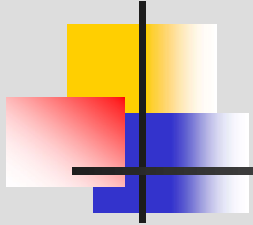


Determine which input is smaller

	sub	diff, Y_E, X_E	
	bltz	diff, X_bigger	
	move	X_plus_Y_E, Y_E	
	move	X_plus_Y_F, Y_F	
	move	small_F, X_F	
	b	LittleF	
X_bigger:	move	X_plus_Y_E, X_E	
	move	X_plus_Y_F, X_F	
	move	small_F, Y_F	
	sub	diff, zero, diff	
LittleF:	sra	small_F, small_F, diff	# denormalize little F



	add	X_plus_Y_F, small_F, X_plus_Y_F	# add Fs
	and	X_plus_Y_S, X_plus_Y_F, S_mask	
	beqz	X_plus_Y_F, Zero	
	bgez	X_plus_Y_F, L1	# skip if positive
	sub	X_plus_Y_F, zero, X_plus_Y_F	# convert to sign/mag
L1:	move	X_plus_Y_E, X_plus_Y_E	
	blt	X_plus_Y_F, max_F, NotTooBig	# skip if no overflow
	srl	X_plus_Y_F, X_plus_Y_F, 1	# divide F by 2
	add	X_plus_Y_E, X_plus_Y_E, 1	# adjust E
	b	normalized	
Zero:	move	Float_X_plus_Y, 0	
	b	Finished	
TooSmall:	sll	X_plus_Y_F, X_plus_Y_F, 1	# multiply F by 2
	sub	X_plus_Y_E, X_plus_Y_E, 1	# adjust E
NotTooBig:	blt	X_plus_Y_F, Hidden_one, TooSmall	# check if still too big
normalized:	sub	X_plus_Y_F, X_plus_Y_F, Hidden_one	# delete hidden one
	add	X_plus_Y_E, X_plus_Y_E, 127	# convert to bias-127
	sll	X_plus_Y_E, X_plus_Y_E, 23	# align properly
	or	X_plus_Y, X_plus_Y_E, X_plus_Y_F	# merge E, F
	or	X_plus_Y, X_plus_Y, X_plus_Y_S	# merge S
	move	Float_X_plus_Y, X_plus_Y	# move to floating point
Finished:	done		



Multiplication

- Floating Addition dan daha basit
- 4 adim
 - Mantissa lar uzerinde unsigned multiplication yap
 - Exponents lere ekle
 - Sonucu normalize hale getir
 - Sonucun isaret bitini belirle

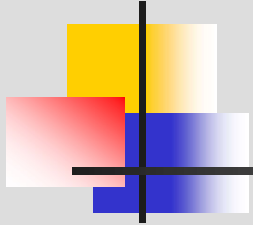
$$18.0 * 9.5$$

0 1 000 0011 (1) 001 0000 0000 0000 0000 0000
 0 1 000 0010 (1) 001 1 000 0000 0000 0000 0000

1 000 0011 4
 + 1 000 0010 3
 —————
 1 000 0110 7

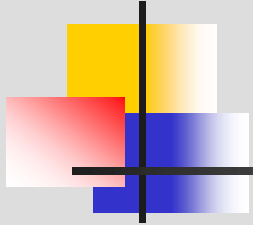
1 001 0000
 × 1 001 1 000
 —————
 0000 0000
 0 0000 000
 00 0000 00
 1 00 1 000 0
 1 001 0000
 0 0000 000
 00 0000 00
 + 1 00 1 000 0
 —————
 1 01 01 01 1 000 0000

0 1 000 0110 (1) 010 1 011 0000 0000 0000 0000



Division

- Multiplication a benzer
- 4 adim
 - Mantissalar uzerinde unsigned division yap
 - Divisorun exponentini dividend in exponentinden cikar
 - Sonucu normalize yap
 - Sonucun isaret bitini belirle.



Overflow ve Underflow

➤ Overflow

- Normalized sonucun exponenti (biased-127) kendine ayrılan yere sigmadiginda olur

➤ Underflow

- Sonucun temsil edilemeyecek kadar 0 ya yakın olmasıyla olur