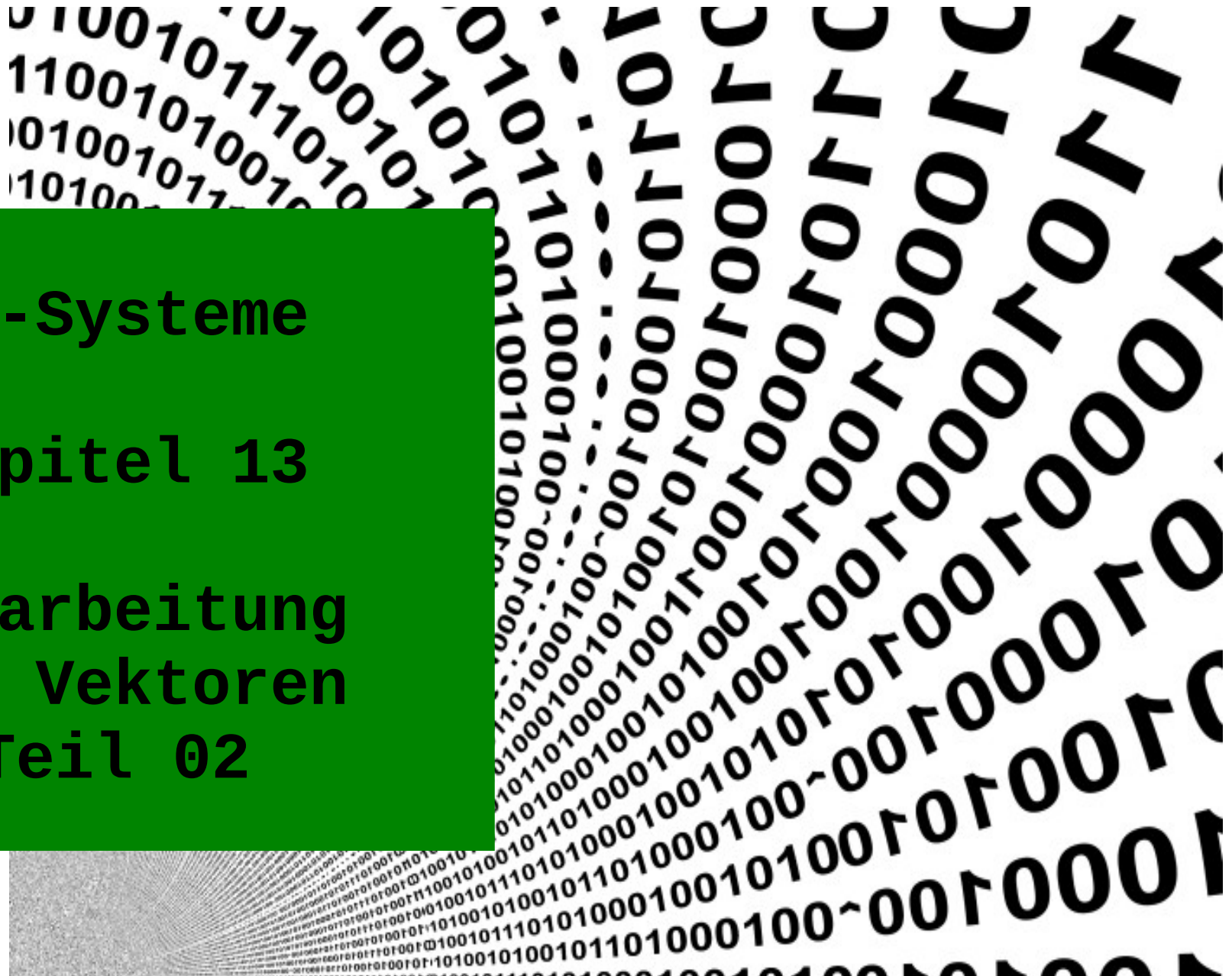


DS-Systeme

Kapitel 13

**Verarbeitung
von Vektoren
Teil 02**



Inhaltsverzeichnis

Thema	Seite
Vector Processing - Extrahieren und Einfügen	3
- Beispiel für vinsertps	6
Vector Processing - Arithmetik	7
- Addieren	8
- Beispiel	9
- Subtrahieren	10
- Aufgabe	11
- Multiplizieren	12
- Dividieren	13
Vector Processing - Arithmetik Spezial	
- Horizontales Addieren	14
- Aufgabe	15
- Horizontales Subtrahieren	16
- Minimum / Maximum	18
- Radizieren	19
- logische Operationen	20
- Typumwandlungen single <==> double	21
- Beispiel	22
- Typumwandlungen Gleitkommazahlen <==> Ganzzahlen	23
- Beispiel	24
Vector Processing - Vergleiche	25
- Aufgabe	30

Vector Processing – Extrahieren und Einfügen

Allgemein

extract- und **insert-Befehle** werden verwendet, um Teile eines **packed data vectors** zu extrahieren bzw. einzufügen.

Beispiel: **vextractf128 \$1, ymm3, xmm0**

Ausgangsposition



vextractf128 \$1, ymm3, xmm0



Vector Processing – Extrahieren und Einfügen

Instruction	Src3	Src2	Src1	Dest	Description // <availability>
vextractf128		I8	X256	X128 / M128	Extract 128 bits of packed floating-point values from src1 and store results in dest -I8: i8[0]: 0 -> extract lower 128bit //AVX2
vinserf128	I8	X128 / M128	X256	X256	Insert 128 bits of packed floating-point values from src2 and the remaining values from src1 into dest -I8: i8[0]: 0 -> insert lower 128bit //AVX2

Operanden:

- X###** - xmm- bzw. ymm-Register-Operand mit einer Bitbreite von ###
M### - Memory-Operand mit einer Bitbreite von ###

Maximal ein Operand kann ein Memory-Operand sein.

I8: I8[0]: 0 -> extract lower 128 Bit	insert in lower 128 Bit of dest
1 -> extract upper 128 Bit	insert in upper 128 Bit of dest

- I8 - (= Immediate bestehend aus 8 Bit)
 I8[0] - (= Bit an Position 0 wird gesetzt)

Vector Processing – Extrahieren und Einfügen

Instruction	Src3	Src2	Src1	Dest	Description //<availability>
vextractps		I8	X128	X32/ M32	Extract one single-precision floating-point value from src1 at the offset specified by I8 and store the result in dest. Zero extend the results in 64-bit register if applicable -I8: i8[1:0] defines the nth float value to extract //AVX (SSE4.1)
vinsertps	I8	X128 /M32	X128	X128	-Insert a single-precision floating-point value selected by I8 from src2 and merge with values in src1 at the specified destination element specified by imm8 and write out the result and zero out destination elements in dest as indicated in I8 -I8: --i8[7:6] defines the nth float value in the src2 --i8[5:4] defines the nth float value in the dest --i8[3:0]: A 1 zeros the nth float value in the dest //AVX (SSE4.1)

- X32 bedeutet hier 32 Bit in einem GPR (general purpose register)

Hinweis:

Operandenbeschreibung wie auf vorheriger Folie

Vector Processing – Extrahieren und EinfügenBeispiel für **vinsertps** mit

.float 1.3, 0.0, 0.0, 0.0	in Adresse buffer
.float 2.1, 2.2, 2.3, 2.4	in xmm3
Ergebnis	in xmm0

vinsertps \$53, buffer, %xmm3, %xmm0**Position:** 7:6|5:4| 3:0

I8 = **53** (Basis 10) = **00|11|0101** (Basis 2)
 0| 3| 5 (Basis 10)

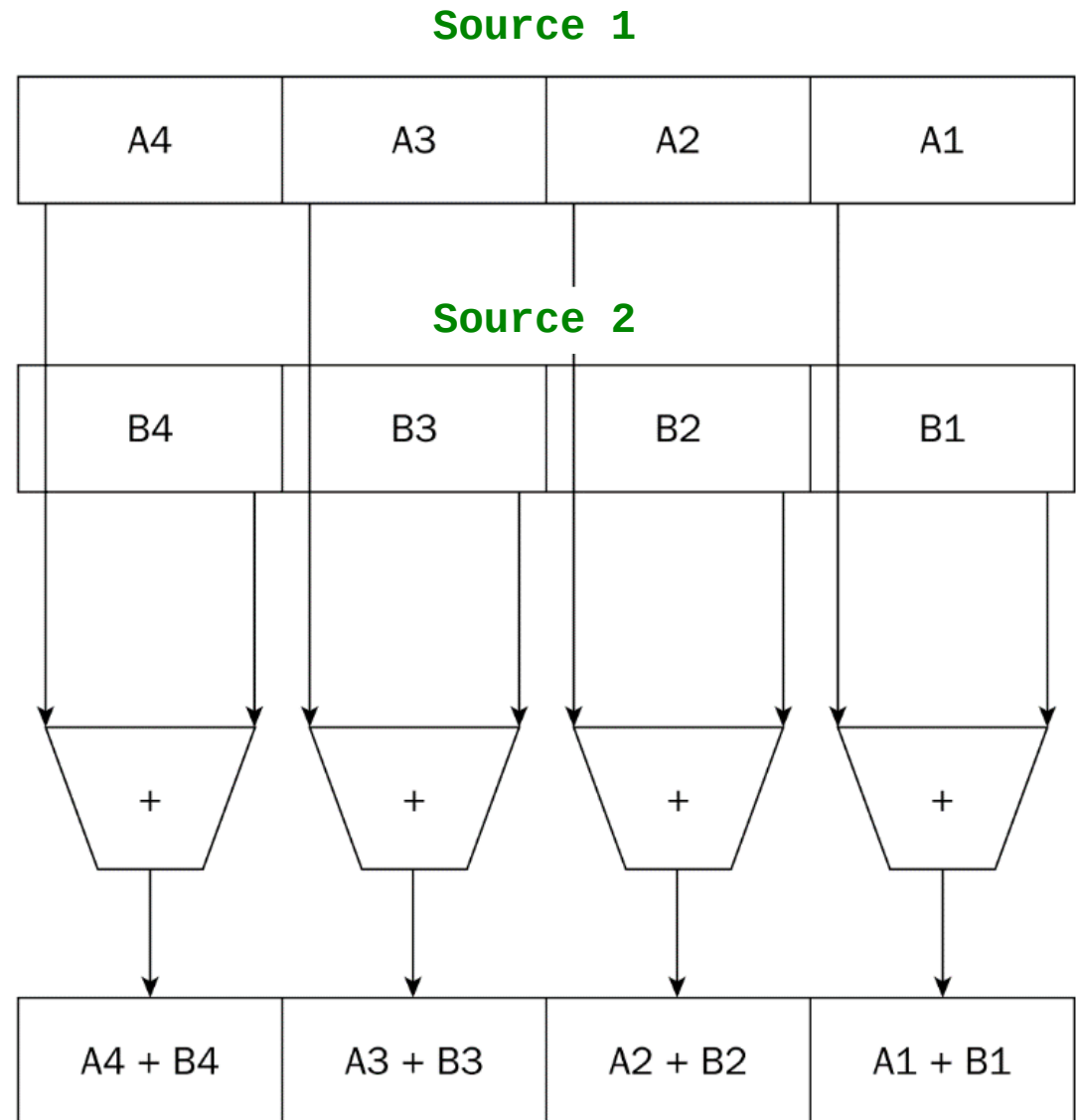
mit

Pos. 7:6: float-Wert an Position 0 in **buffer**Pos. 5:4: wird an Position 3 in **xmm0** abgelegt
(Rest von **xmm0** kommt aus **xmm3**)Pos. 3:0: Positionen 0 und 2 in **xmm0** werden durch 0.0 ersetzt
(sind in **0101** die Positionen, an denen die 1sen stehen)
(und in **xmm3** die Werte 2.1 und 2.3)Ergebnis in **xmm0**: **0.0, 2.2, 0.0, 1.3**Aufgabe:Wie würde das Ergebnis
mit I8 = 0xC6 aussehen?

Vector Processing - ArithmeticPrinzip der Vektorarithmetik am
Beispiel Addition

Der Source 2-Operand kann Memory
oder XMM- bzw. YMM-Register
sein.

Destination muss XMM- bzw. YMM-
Register sein.



Vector Processing - Arithmetik

Instruction	Source 2	Source 1	Desti- nation	Description //<availability>
vaddps	X/M	X	X	Add packed single-precision floating-point values from src2 to src1 and store result in dest.//AVX (SSE)
vaddpd	X/M	X	X	Add packed double-precision floating-point values from src2 to src1 and store result in dest.//AVX (SSE2)

Hinweis:

Befehl:

Suffix s = single precision
d = double precision

Operanden:

X = xmm-Register

M = 128 Bit- bzw. 256 Bit-Memory

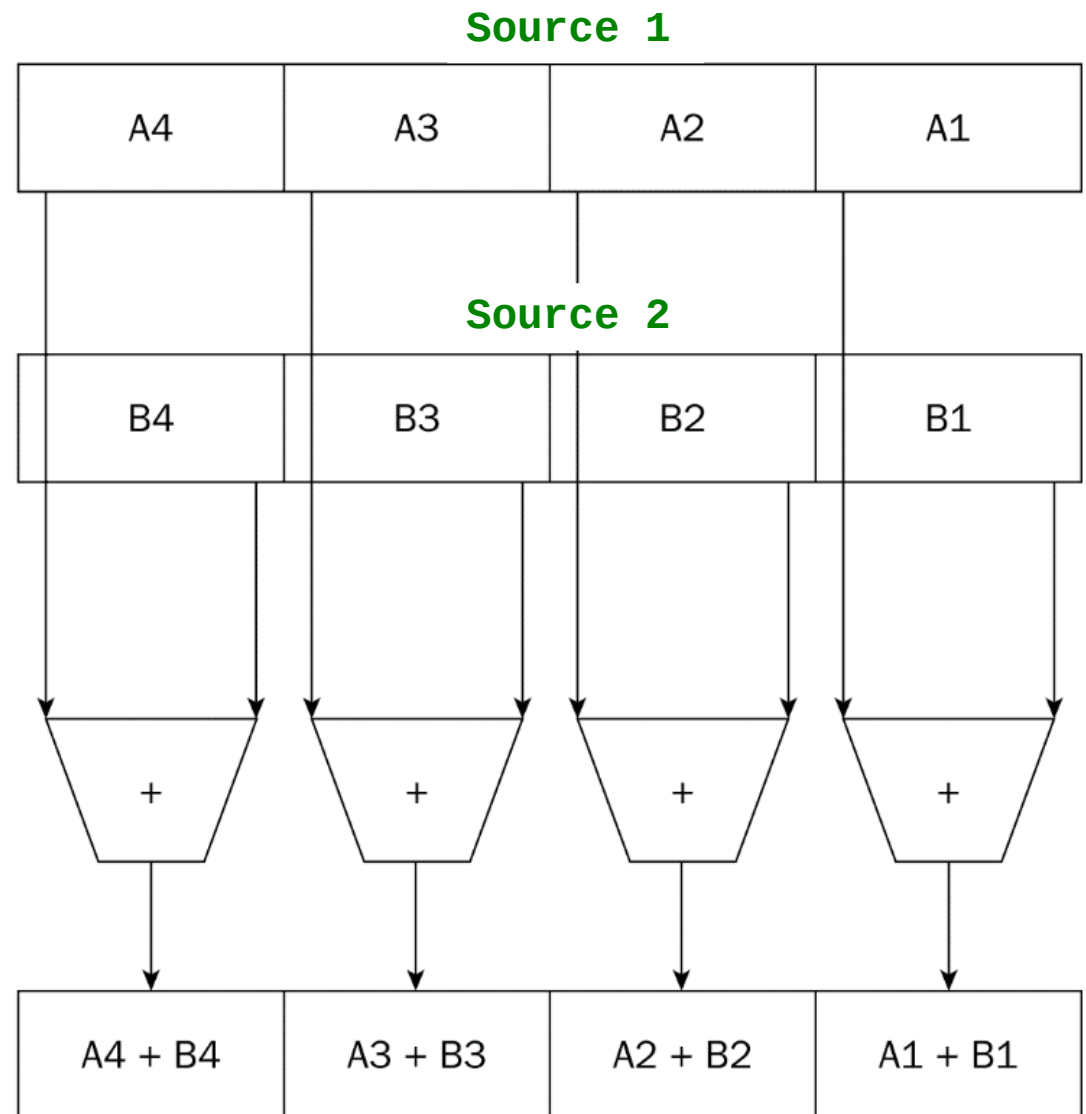
Maximal src2-Operand kann Memory-Operand sein.

Vector Processing - ArithmetikBeispiel:

```
.section .bss
.align 16
.lcomm tdataarr, 16

.section .data
.align 16
farr1:
.float 1.2, 2.3, -3.4, 5.6
.align 16
farr2:
.float 2.2, 3.3, 4.4, -6.6

.section .text
func:
vmovaps farr1, %xmm1
vaddps farr2, %xmm1, %xmm0
vmovaps %xmm0, tdataarr
ret
```



Vector Processing - Arithmetik

Instruction	Source 2	Source 1	Destination	Description // <availability>
vsubps	X/M	X	X	Subtract packed single-precision floating-point values in src2 from src1 and stores result in dest. //AVX (SSE)
vsubpd	X/M	X	X	Subtract packed double-precision floating-point values in src2 from src1 and stores result in dest. //AVX (SSE2)

Hinweis:

Subtrahend

Minuend

Befehl:

Suffix s = single precision
d = double precision

Operanden:

X = xmm-Register

M = 128 Bit- bzw. 256 Bit-Memory

Maximal src2-Operand kann Memory-Operand sein.

Resultat = src1 - src2

Vector Processing - Arithmetik - Aufgabe

```
.section .bss
.align 32
.lcomm tdataarr, 32

.section .data
.align 32
farr1:
    .float 1.2, 2.3, -3.4, 5.6, 6.7, 7.8, 8.9, 9.10
.align 32
farr2:
    .float 2.2, 3.3, 4.4, -6.6, 7.7, 8.8, 9.9, 10.10
```

```
.section .text
func:
    # TODO calc farr1-farr2 with vector instructions and save the result in
    tdataarr

    # HINT use leaq to generate pointers and hold them in %rdi, %rsi and
    %rcx and then use the memory access operator

    # HINT2 use the memory access operator with the appropriate offset to
    get the second bunch
```

Vector Processing - Arithmetik - Multiplikation

Instruction	Source 2	Source 1	Desti- nation	Description //<availability>
vmulps	X/M	X	X	Multiply packed single-precision floating-point values in src2 with src1 and store result in dest. //AVX (SSE)
vmulpd	X/M	X	X	Multiply packed double-precision floating-point values in src2 with src1 and store result in dest. //AVX (SSE2)

Hinweis:

Befehl:

Suffix s = single precision
d = double precision

Operanden:

X = xmm-Register

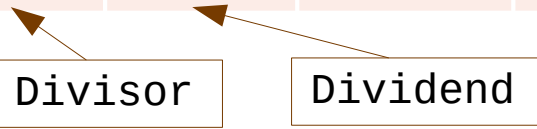
M = 128 Bit- bzw. 256 Bit-Memory

Maximal src2-Operand kann Memory-Operand sein.

Resultat = src1 * src2

Vector Processing - Arithmetik - Division

Instruction	Source 2	Source 1	Destination	Description // <availability>
vdivps	X/M	X	X	Divide packed single-precision floating-point values in src1 by packed single-precision floating-point values in src2 //AVX (SSE)
vdivpd	X/M	X	X	Divide packed double-precision floating-point values in src1 by packed double-precision floating-point values in src2 //AVX (SSE2)

Hinweis:

 Divisor

Dividend

Befehl:

Suffix s = single precision
 d = double precision

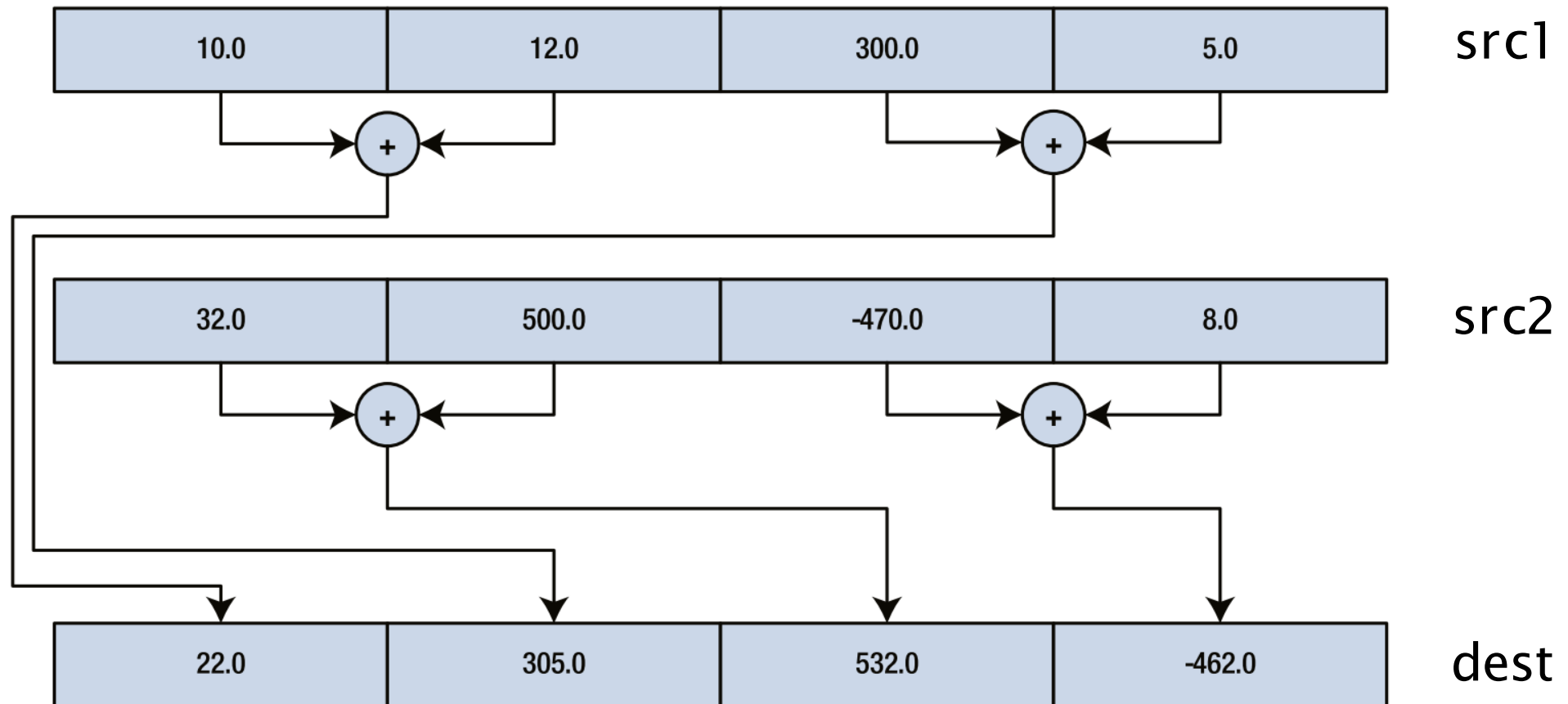
Operanden:

X = xmm-Register

M = 128 Bit- bzw. 256 Bit-Memory

Maximal src2-Operand kann Memory-Operand sein.

Resultat = src1 / src2

Vector Processing - Arithmetik - Spezial - horizontales AddierenAllgemeines Prinzip: horizontales Addieren

hier: Grafik 128 Bit-Register

Vector Processing - Arithmetik - Spezial - horizontales Addieren

Instruction	Source 2	Source 1	Destination	Description // <availability>
vhaddps	X/M	X	X	Horizontal add packed single-precision floating-point values from src1 and src2 in dest.//AVX (SSE3)
vhaddpd	X/M	X	X	Horizontal add packed double-precision floating-point values from src1 and src2 in dest.//AVX (SSE3)

horizontal

Hinweis:

Befehl:

Suffix s = single precision
d = double precision

Operanden:

X = xmm-Register

M = 128 Bit- bzw. 256 Bit-Memory

Maximal src2-Operand kann Memory-Operand sein.

Aufgabe:

```
ymm1 = 1.1 1.2 ... 1.7 1.8
ymm2 = 2.1 2.2 ... 2.7 2.8
```

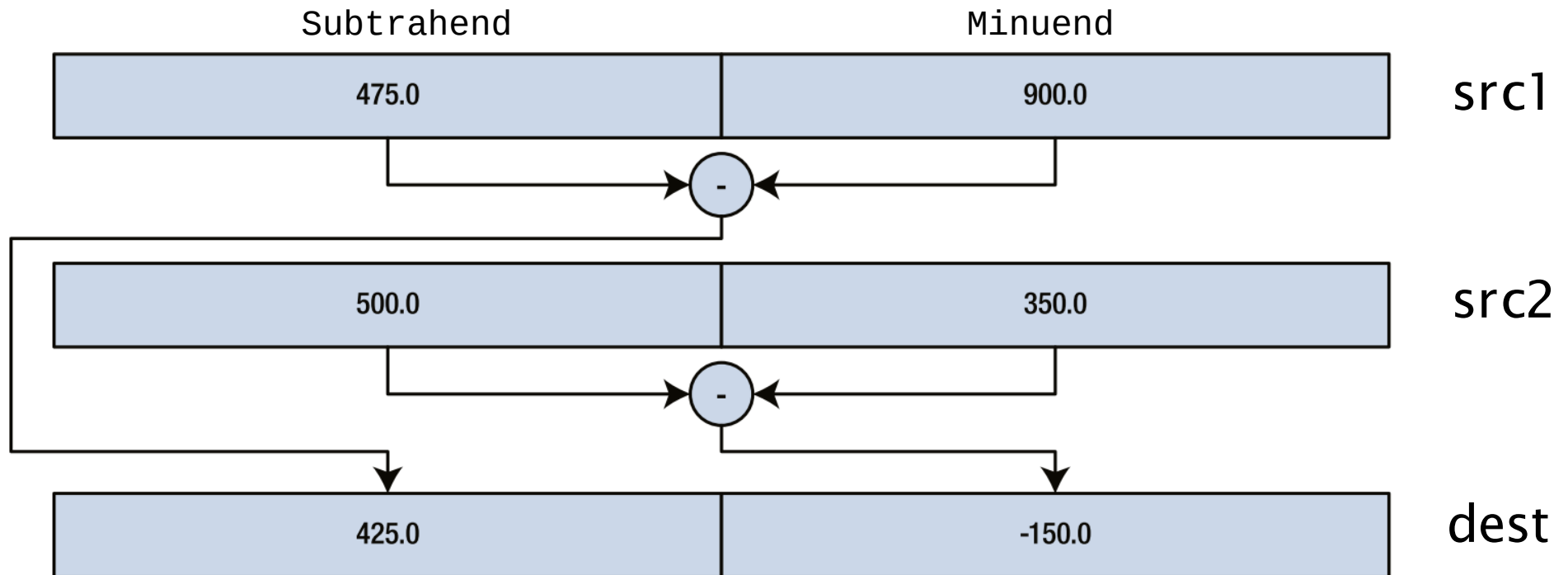
Was steht nach dem Befehl

```
vhaddps %ymm2, %ymm1, %ymm0
```

in ymm0?

Hinweis:

Funktioniert ähnlich wie bei **vshufps**.

Vector Processing - Arithmetik - Spezial - horizontales SubtrahierenAllgemeines Prinzip: horizontales Subtrahieren

hier: Grafik 128 Bit-Register

Vector Processing - Arithmetik - Spezial - horizontales Subtrahieren

Instruction	Source 2	Source 1	Desti- nation	Description //<availability>
vhsubps	X/M	X	X	Horizontal subtract packed single-precision floating-point values from src1 and src2 //AVX (SSE3)
vhsubpd	X/M	X	X	Horizontal subtract packed double-precision floating-point values from src1 and src2 //AVX (SSE3)



horizontal

Hinweis:

Befehl:

Suffix s = single precision
d = double precision

Operanden:

X = xmm-Register

M = 128 Bit- bzw. 256 Bit-Memory

Maximal src2-Operand kann Memory-Operand sein.

Vector Processing - Arithmetik - Spezial - Minimum / Maximum

Instruction	Source 2	Source 1	Destination	Description //<availability>
vminps	X/M	X	X	Return the minimum single-precision floating-point values between src1 and src2 in dest. //AVX (SSE)
vminpd	X/M	X	X	Return the minimum double-precision floating-point values between src1 and src2 in dest. //AVX (SSE2)
vmaxps	X/M	X	X	Return the maximum single-precision floating-point values between src1 and src2 in dest. //AVX (SSE)
vmaxpd	X/M	X	X	Return the maximum double-precision floating-point values between src1 and src2 in dest. //AVX (SSE2)

Hinweis:

Befehl:

Suffix s = single precision
d = double precision

Hier werden die jeweiligen Wertepaare aus den beiden Registern, die an den gleichen Positionen stehen, verglichen, ausgewertet und die Einzelergebnisse ins Zielregister übertragen.

Operanden:

X = xmm-Register

M = 128 Bit- bzw. 256 Bit-Memory

Maximal src2-Operand kann Memory-Operand sein.

Vector Processing - Arithmetik - Spezial - Radizieren

Instruction	Source 2	Source 1	Desti- nation	Description // <availability>
vsqrtps		X/M	X	Computes Square Roots of the packed single-precision floating-point values in src1 and stores the result in dest.//AVX (SSE)
vsqrtpd		X/M	X	Computes Square Roots of the packed double-precision floating-point values in src1 and stores the result in dest.//AVX (SSE2)

Hinweis:

Befehl:

Suffix s = single precision

d = double precision

Operanden:

X = xmm-Register

M = 128 Bit- bzw. 256 Bit-Memory

Maximal src2-Operand kann Memory-Operand sein.

Vector Processing - Arithmetik - Spezial - logische Operationen

Instruction	Source 2	Source 1	Destination	Description //<availability>
vandps	X/M	X	X	Return the bitwise logical AND of packed values in src1 and src2 //AVX (SSE)
vandpd	X/M	X	X	Return the bitwise logical AND of packed values in src1 and src2 //AVX (SSE2)
vandnps	X/M	X	X	Return the bitwise logical NAND of packed values in src1 and src2 //AVX (SSE)
vandnpd	X/M	X	X	Return the bitwise logical NAND of packed values in src1 and src2 //AVX (SSE2)
vorps	X/M	X	X	Return the bitwise logical OR of packed values in src1 and src2 //AVX (SSE)
vorpd	X/M	X	X	Return the bitwise logical OR of packed values in src1 and src2 //AVX (SSE2)
vxorps	X/M	X	X	Return the bitwise logical XOR of packed values in src1 and src2 //AVX (SSE)
vxorpd	X/M	X	X	Return the bitwise logical XOR of packed values in src1 and src2 //AVX (SSE2)

Hinweise wie vorherige Seite.

vxorps %xmm0, %xmm0, %xmm0
setzt Register %xmm0 auf 0

Vector Processing - Floating Point - Umwandlung - SPFP <- -> DPFP

Instruction	Source 2	Source 1	Destination	Description // <availability>
vcvtps2pd		X/M	X	Convert two packed single-precision floating-point values in src1 to two packed double-precision floating-point values in dest //AVX (SSE2)
vcvtpd2ps		X/M	X	Convert two packed double-precision floating-point values in src1 to two single-precision floating-point values in dest //AVX (SSE2)

vcvtps2pd

bei Ziel 128 Bit: es werden die beiden unteren float nach double konvertiert, die oberen beiden werden verworfen bzw. überschrieben

bei Ziel 256 Bit: es werden (die unteren) 4 float in 4 double konvertiert

vcvtpd2ps

bei Ziel 128 Bit: die beiden double werden in 2 float konvertiert und an den beiden unteren Plätze abgelegt (obere 64 Bit werden auf 0 gesetzt)

bei Ziel 256 Bit: die vier double werden in 4 float konvertiert und an den vier unteren Plätze abgelegt (obere 128 Bit werden auf 0 gesetzt)

Vector Processing - Floating Point - Umwandlung - SPFP <- -> DPFP

Beispiel: Umwandlung SPFP -> DPFP

```
.section .bss
.align 16
.lcomm tdataarr, 16

.section .data
.align 16
farr1:
    .float 1.2, 2.3
.align 16
farr2:
    .float 2.2, 3.3
.section .text
func:
    vmovaps farr1, %xmm0
    vmovaps farr2, %xmm1
    #convert to double
    vcvtps2pd %xmm0, %xmm0
    #convert to double
    vcvtps2pd %xmm1, %xmm1
    vaddpd %xmm1, %xmm0, %xmm0
    vmovapd %xmm0, tdataarr
ret
```

Ergebnis in tdataarr:

(double) 3.4, (double) 5.6

Vector Processing - Umwandlung - FP <--> Integer

Instruction	Source 2	Source 1	Destination	Description //<availability>
vcvtps2dq		X/M	X	Convert packed single-precision floating-point values from src1 to packed signed doubleword values in dest. //AVX (SSE2)
vcvtpd2dq		X/M	X	Convert packed double-precision floating-point values from src1 to packed signed doubleword values in dest. //AVX (SSE2)
vcvtdq2ps		X/M	X	Convert packed signed doubleword integers from src1 to packed single-precision floating-point values in dest. //AVX (SSE2)
vcvtdq2pd		X/M	X	Convert packed signed doubleword integers from src1 to packed double-precision floating-point values in dest. //AVX (SSE2)

Hinweis:

Befehl:

Suffix s = single precision
d = double precision

Operanden:

X = xmm-Register
M = 128 Bit-Memory

Umwandlungen:

float ==> int
double ==> int
int ==> float
int ==> double

'q' bedeutet hier
nicht "quad"

Maximal src1-Operand kann
Memory-Operand sein.

Vector Processing - Umwandlung - FP <--> IntegerBeispiel:

```
.section .data
.align 16
farr1:
.float 11.1, 12.2, 13.5, 14.6
.align 16
darr1:
.double 111.1, 122.2
.align 16
int1:
.int 10, 20, 30, 40
```

```
.section .text
.globl main
.type main, @function
main:
    pushq %rbp
    # 4 floats -> 4 ints
    vmovaps farr1, %xmm1
    vcvtps2dq %xmm1, %xmm0

    # 2 doubles -> 2 ints
    vmovapd darr1, %xmm2
    vcvtpd2dq %xmm2, %xmm0

    # 4 ints -> 4 floats
    vmovaps int1, %xmm3
    vcvtdq2ps %xmm3, %xmm0

    # 4 ints -> 2 double
    # xmm0 completely 0
    vxorps %xmm0, %xmm0, %xmm0
    vcvtdq2pd %xmm3, %xmm0
    movq $0, %rax
    popq %rbp
    ret
```


Vector Processing - Vergleiche

Allgemein:

Die Vergleiche werden immer elementweise ausgeführt.

Wenn das Ergebnis **true** ist, ist der Wert **0xFFFFFFFF** ansonsten **0x00000000**

Beispiel:

```
vcmpsps $1, %xmm1, %xmm0, %xmm0 or vcmpltss %xmm1, %xmm0, %xmm0
```

1.2	2.3	-3.4	5.6	src1/xmm0
2.2	3.3	4.4	-6.6	src2/xmm1
0xFFFFFFFF	0xFFFFFFFF	0xFFFFFFFF	0x00000000	dest/xmm0

\$1 = less than

Vector Processing - Vergleiche

Beispiel: less than: src1 < src2

```
.section .bss
.align 16
.lcomm tdataarr, 16

.section .data
.align 16
farr1:
    .float 1.2, 2.3, -3.4, 5.6
.align 16
farr2:
    .float 2.2, 3.3, 4.4, -6.6

.section .text
func:
    vmovaps farr1, %xmm0
    vmovaps farr2, %xmm1
    # cmp: imm=$1=lt -> vcmpltss
    # cmp: ->src1 < src2
    vcmpps $1, %xmm1, %xmm0, %xmm0
    vmovaps %xmm0, tdataarr
    ret
```

Resultat:

```
0xFFFFFFFF, 0xFFFFFFFF,
0xFFFFFFFF, 0x00000000
```

Vector Processing - Vergleiche

Instruction	Src3	Src2	Src1	Dest	Description // <availability>
vcmpps	I8	X/M	X	X	Compare packed single-precision floating-point values in src2 and src1 using bits 4:0 of src3/I8 as a comparison predicate; if true, dest part contains all '1', if false -> all '0' //AVX (SSE)
vcmppd	I8	X/M	X	X	Compare packed double-precision floating-point values in src2 and src1 using bits 4:0 of src3/I8 as a comparison predicate; if true, dest part contains all '1', if false -> all '0' //AVX (SSE2)

Hinweis:

Befehl:

Suffix s = single precision
d = double precision

Operanden:

X = xmm- bzw ymm-Register

M = 128 Bit- bzw. 256 Bit-Memory

Maximal src2-Operand kann Memory-Operand
sein.

I8:

(siehe Tabelle nächste Seite)

Vector Processing - Vergleiche

PredOp	Predicate	Description	Pseudo-Instructions
0	EQ	Src1 == Src2	vcmpeq(s d)
1	LT	Src1 < Src2	vcmlts(s d)
2	LE	Src1 <= Src2	vcmls(s d)
3	UNORD	Src1 && Src2 are unordered	vcmpunords(s d)
4	NEQ	Src1 != Src2	vcmpneq(s d)
13	GE	Src1 >= Src2	vcmpges(s d)
14	GT	Src1 > Src2	vcmpgts(s d)
7	ORD	Src1 && Src2 are ordered	vcmpords(s d)

Hinweis:

Es gibt auch Pseudo-Instruktionen mit 3 Operanden (siehe vierte Spalte) anstelle der **vcmpss** / **vcmpsd** mit 4 Operanden inclusive I8-Operanden.

Vector Processing - Vergleiche

UNORD:

Vergleicht zwei Gleitkommawerte daraufhin, ob mindestens einer von ihnen **NaN** oder **INF** / **-INF** ist.

falls ja ==> true

falls nein ==> false

ORD:

Vergleicht zwei Gleitkommawerte daraufhin, ob mindestens einer von ihnen **NaN** oder **INF/-INF** ist.

falls nein ==> true

falls ja ==> false

Vector Processing - Arithmetik - Aufgabe

```
.section .bss
.align 16
.lcomm tdataarr, 16

.section .data
.align 16
farr1:
.float 1.2, 2.3, -3.4, 5.6, 6.7, 7.8, 8.9, 9.10 #sum=38.2
```

```
.section .text
func:
# TODO calc sum farr1 with vector instructions and save the result in
  tdataarr

# HINT use %rdi und %rsi to hold pointers generated with leaq and use the
  memory access operator

# HINT2 use vaddps and use 2x vhaddps for the final summary within a
  vector

# HINT3 use %xmm registers
```