The background of the slide features a dense pattern of binary code (0s and 1s) in a light gray color, arranged in a way that creates a sense of depth and perspective. In the bottom-left corner, there is a small, circular fingerprint-like pattern, also in a light gray color, which appears to be a stylized representation of a fingerprint or a similar biometric identifier.

# DS-Systeme

## Kapitel 12

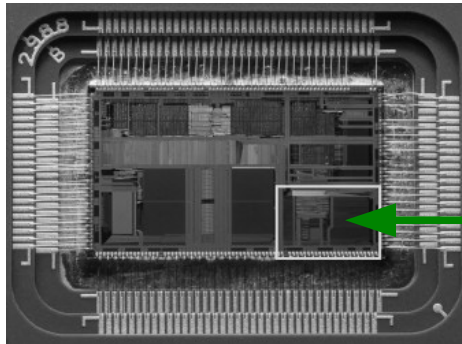
### Floating Point

Inhaltsverzeichnis

Thema	Seite
Floating Point - Historie	3
Floating Point - Register	4
Skalare mov-Befehle	6
Typumwandlungen	
- Gleitkommazahlen ==> Ganzzahlen	8
- Ganzzahlen ==> Gleitkommazahlen	9
- Beispiel	10
Funktionen	12
Arithmetik	14
- Aufgabe	17
Konstanten - Definition und Zugriff	19
Vergleiche	21
- Aufgabe	23

## Floating Point - Historie

- Historisch griff die **x86-Architektur** für Gleitkomma-Berechnungen um 1980 auf den x87-Coprozessor zurück (seit i486 als **FPU (Floating Point Unit)** in die CPU integriert). **SISD**-Befehle (**S**ingle **I**nstruction **S**ingle **D**ata).



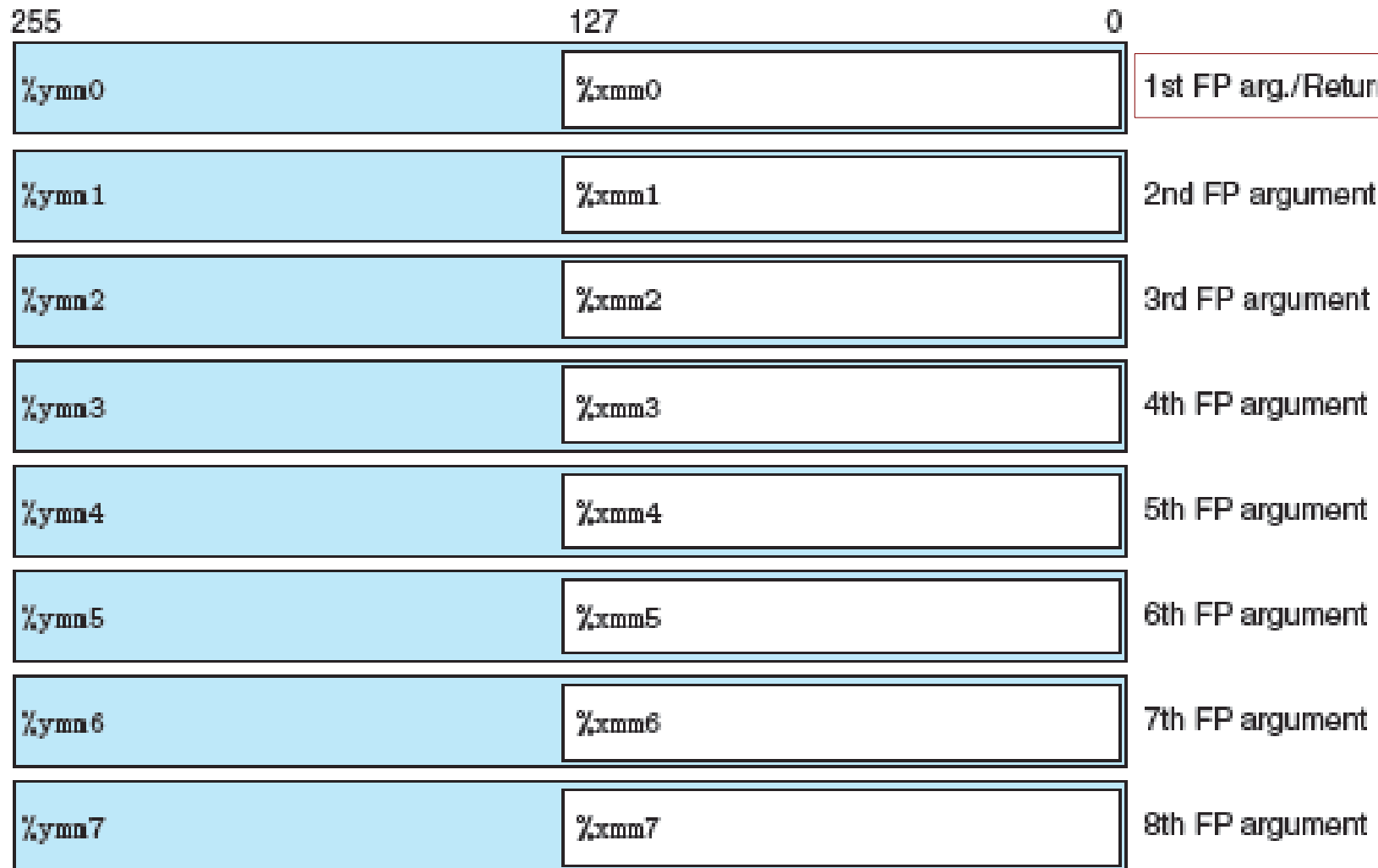
Größenverhältnis  
**FPU : CPU**  
im i486-Prozessor

- Die **x86-64 Architektur** greift für Gleitkomma-Berechnungen standardmäßig auf die neueren **SIMD**-Befehle zurück (**S**ingle **I**nstruction **M**ultiple **D**ata).
- **SIMD** entwickelte sich von **MMX (Multimedia Extensions)** über **SSE (Streaming SIMD Extensions)** zu **AVX (Advanced Vector Extensions)** [und AVX-512].
- Die Registernamen und -größen entwickelten sich von **MM (64 Bit)** zu **XMM (128 Bit)**, **YMM (256 Bit)** [und **ZMM (512 Bit)**].
- **MMX** arbeitete noch mit Integer, ab **SSE** wird mit Floating Point gearbeitet (auch skalare Werte).
- Im Folgenden wird AVX fokussiert.

## Floating Point - Register

AVX-512 entsprechend mit den Registern **zmmX** (512 Bit).

16 Register mit 128 bzw. 256 Bit Breite:



Register **xmm0** hat zwei Bedeutungen: 1.) Erstes Floating Point-Argument  
2.) Register für die Rückgabe eines Floating Point-Wertes.

Floating Point - Register

AVX-512 entsprechend mit den Registern **zmmX** (512 Bit).

16 Register mit 128 bzw. 256 Bit Breite:

<code>%ymm8</code>	<code>%xmm8</code>	Caller saved
<code>%ymm9</code>	<code>%xmm9</code>	Caller saved
<code>%ymm10</code>	<code>%xmm10</code>	Caller saved
<code>%ymm11</code>	<code>%xmm11</code>	Caller saved
<code>%ymm12</code>	<code>%xmm12</code>	Caller saved
<code>%ymm13</code>	<code>%ymm13</code>	Caller saved
<code>%ymm14</code>	<code>%xmm14</code>	Caller saved
<code>%ymm15</code>	<code>%xmm15</code>	Caller saved

## Skalare mov-Befehle

Instruction	Source	Destination	Description
<code>vmovss</code>	$M_{32}$	$X$	Move single precision
<code>vmovss</code>	$X$	$M_{32}$	Move single precision
<code>vmovsd</code>	$M_{64}$	$X$	Move double precision
<code>vmovsd</code>	$X$	$M_{64}$	Move double precision
<code>vmovaps</code>	$X$	$X$	Move aligned, packed single precision
<code>vmovapd</code>	$X$	$X$	Move aligned, packed double precision

Auch hier gilt, dass in einem **mov**-Befehl nur ein Speicherort ( $M_{xx}$ ) vorkommen darf.

### Bedeutung der Befehle:

z.B. **vmovss**:

**v** - Vektor-Befehle (AVX)

**mov** - move

**s** - skalar (Befehl für einzelnen FP-Wert)

**s/d** - single/double precision (einfache/doppelte Genauigkeit)

$M_{32}$  - Speicherort für 32 Bit-Wert

**X** - stellvertretend für ein **xmm**-Register


Vektororientierte Befehle (also nicht skalare) benötigen **.align 16**.

### Hinweis:

Die alten SSE-Befehle (ohne führendes **v**) nicht mit diesen neuen AVX-Befehlen mischen.

Skalare mov-BefehleBeispiel:C/C++-Code:

```
float float_mov(float v1,
               float *src,
               float *dst)
{
    float v2 = *src;
    *dst = v1;
    return v2;
}
```




Zeiger werden nach wie vor in General Purpose-Registern (Allzweck-Registern) abgelegt (hier **rdi** und **rsi**).

Assembler-Code:

```
.globl float_mov
.type float_mov, @function
# float float_mov(float v1, float *src, float *dst)
# v1 in %xmm0, src in rdi, dst in rsi

float_mov:
    # Copy v1 (prevent overwrite)
    vmovss %xmm0, (%rsi)
    vmovss (%rsi), %xmm1
    # Read v2 from src
    vmovss (%rdi), %xmm0
    # Write v1 to dst
    vmovss %xmm1, (%rsi)
    # Return v2 in %xmm0
    ret
```



Befehl **vmovss** muss immer aus einer Speicheradresse und einem Register bestehen.

## Typumwandlungen (Ganzzahlen ==> Gleitkommazahlen)

Die Destination muss ein xmm-Register sein.

**Der dritte Parameter ist normalerweise der gleiche wie der zweite.**

Umwandlungen von Ganzzahlen (int) in Gleitkommazahlen (floating point)  
(Befehle mit drei Operanden):

Instruction	Source 1	Source 2	Destination	Description
<code>vcvtsi2ss</code>	$M_{32}/R_{32}$	X	X	Convert integer to single precision
<code>vcvtsi2sd</code>	$M_{32}/R_{32}$	X	X	Convert integer to double precision
<code>vcvtsi2ssq</code>	$M/R_{64}$	X	X	Convert quad word integer to single precision
<code>vcvtsi2sdq</code>	$M/R_{64}$	X	X	Convert quad word integer to double precision

Befehlsaufbau:

`instr src1, src2,  
dest`

in der Vorlesung  
"Vektoren" wird

`instr src3, src2,  
src1, dest`

verwendet.

cvt = convert

Endung q:  
quad word = long



Typumwandlungen (Gleitkommazahlen ==> Ganzzahlen)

Umwandlungen von Gleitkommazahlen (floating point) in Ganzzahlen (int)  
(Befehle mit zwei Operanden):

Instruction	Source	Destination	Description
<code>vcvttss2si</code>	$X/M_{32}$	$R_{32}$	Convert with truncation single precision to integer
<code>vcvttsd2si</code>	$X/M_{64}$	$R_{32}$	Convert with truncation double precision to integer
<code>vcvttss2siq</code>	$X/M_{32}$	$R_{64}$	Convert with truncation single precision to quad word integer
<code>vcvttsd2siq</code>	$X/M_{64}$	$R_{64}$	Convert with truncation double precision to quad word integer

englisch:  
t = truncate

deutsch:  
abschneiden

TypumwandlungenBeispiel:C/C++-Code:

```
double fcvt(int i, float *fp,
double *dp, long *lp)
{
    float  f = *fp;
    double d = *dp;
    long   l = *lp;

    *lp = (long)    d;
    *fp = (float)   i;
    *dp = (double)  l;

    return (double) f;
}
```

In diesem Fall (wenn es nur um den niederwertigsten float-Wert in `xmm0` geht) ist der erste Befehl redundant.

Assembler-Code:

```
# double fcvt(int i, float *fp,
                double *dp, long *lp)
# i in edi, fp in rsi,
  dp in rdx, lp in rcx

fcvt:
    vmovss (%rsi), %xmm0
    movq (%rcx), %rax
    vcvtsd2siq (%rdx), %r8
    movq %r8, (%rcx)
    vcvtsi2ss %edi, %xmm1, %xmm1
    vmovss %xmm1, (%rsi)
    vcvtsi2sdq %rax, %xmm1, %xmm1
    vmovsd %xmm1, (%rdx)

# The following two instructions
  convert float f to double
    vunpcklps %xmm0, %xmm0, %xmm0
    vcvtps2pd %xmm0, %xmm0

    ret
```

Typumwandlungen

```
double fcvf(int i, float *fp,
double *dp, long *lp)
{
    float  f = *fp;
    double d = *dp;
    long   l = *lp;

    *lp = (long)    d;
    *fp = (float)   i;
    *dp = (double)  l;

    return (double) f;
}
```

Assembler-Code kommentiert:

```
# double fcvf(int i, float *fp, double *dp, long *lp)
# i in edi, fp in rsi, dp in rdx, lp in rcx

fcvf:
    vmovss (%rsi), %xmm0           # Get f = *fp
    movq (%rcx), %rax              # Get l = *lp

    # Get d = *dp and conv. to long
    vcvttss2siq (%rdx), %r8
    movq %r8, (%rcx)               # Store at lp
    vcvtsi2ss %edi, %xmm1, %xmm1   # Convert i to float
    vmovss %xmm1, (%rsi)           # Store at fp
    # Convert l to double
    vcvtsi2sdq %rax, %xmm1, %xmm1
    vmovsd %xmm1, (%rdx)           # Store at dp

    # The following two instructions
    # convert float f to double
    vunpcklps %xmm0, %xmm0, %xmm0
    vcvtps2pd %xmm0, %xmm0
    ret
```

## Funktionen

- Wie Gleitkommawerte an Funktionen übergeben werden ist wieder in den **Calling Conventions** geregelt (Hier x86-64 Linux 64 Bit (System V AMD64 ABI))
  - Alle XMM-Register sind **Caller saved** (Der **Callee** darf alle überschreiben).
  - Bis zu acht Funktionsparameter in den XMM-Registern `%xmm0` - `%xmm7` sind möglich.
  - Die Rückgabe eines Gleitkommawertes erfolgt im Register `%xmm0`.

## Funktionen

### Beispiele:

**double f1(int x, double y, long z);**  
x in %edi, y in %xmm0, und z in %rsi

**double f2(double y, int x, long z);**  
y in %xmm0, x in %edi, und z in %rsi

**double f1(float x, double \*y, long \*z);**  
x in %xmm0, y in %rdi, und z in %rsi

Die Ganzzahl-Parameter werden wie gehabt in den Registern **rdi**, **rsi** u.s.w. platziert.

Die Gleitkomma-Parameter werden in den Registern **xmm0**, **xmm1**, u.s.w. abgelegt.

Das geschieht immer – auch wenn **xmm0** für die Rückgabe genutzt wird.

## Arithmetik

Beachten Sie, dass die Operanden hier immer in der Reihenfolge der  $S_2$ ,  $S_1$  aufgeführt sind.

Single	Double	Effect	Description
<code>vaddss</code>	<code>vaddsd</code>	$D \leftarrow S_2 + S_1$	Floating-point add
<code>vsubss</code>	<code>vsubsd</code>	$D \leftarrow S_2 - S_1$	Floating-point subtract
<code>vmulss</code>	<code>vmulsd</code>	$D \leftarrow S_2 \times S_1$	Floating-point multiply
<code>vdivss</code>	<code>vdivsd</code>	$D \leftarrow S_2 / S_1$	Floating-point divide
<code>vmaxss</code>	<code>vmaxsd</code>	$D \leftarrow \max(S_2, S_1)$	Floating-point maximum
<code>vminss</code>	<code>vminsd</code>	$D \leftarrow \min(S_2, S_1)$	Floating-point minimum
<code>sqrtps</code>	<code>sqrtsd</code>	$D \leftarrow \sqrt{S_1}$	Floating-point square root

Die Befehle haben ein oder zwei Source- und einen Destination-Operanden.

Der erste Operand kann ein **xmm-Register** oder **Memory** sein, der zweite und der Destination-Operand müssen immer **xmm-Register** sein.

Einzigster Befehl in dieser Liste mit nur einem Source-Operanden.

Die Bezeichnung geht von folgender Annahme aus:

**Befehl**  $S_1$ ,  $S_2$ ,  $D$  # im Kapitel "Vektoren" wird **Befehl**  $S_3$ ,  $S_2$ ,  $S_1$ ,  $D$  verwendet

Das ist insbesondere relevant für **sub** und **div**.

Hinweis: **sqrtps** (SSE-Befehl) | **vsqrtps** (AVX-Befehl, ist anders aufgebaut)

ArithmetikBeispiel:C/C++-Code:

```
double funct(double a, float x,  
             double b, int i)  
{  
    return x * a - b / i;  
}
```

Assembler-Code:

```
# double funct(double a, float x, double b, int i)  
# a in %xmm0, x in %xmm1, b in %xmm2, i in edi  
funct:  
    # The following two instructions convert x to double  
    vunpcklps %xmm1, %xmm1, %xmm1  
    vcvtps2pd %xmm1, %xmm1  
    vmulsd %xmm0, %xmm1, %xmm0  
    vcvtsi2sd %edi, %xmm1, %xmm1  
    vdivsd %xmm1, %xmm2, %xmm2  
    vsubsd %xmm2, %xmm0, %xmm0  
    ret
```

ArithmetikC/C++-Code:

```
double funct(double a, float x,
            double b, int i)
{
    return x * a - b / i;
}
```

Assembler-Code kommentiert:

```
.text
.globl funct
.type funct, @function
# double funct(double a, float x, double b, int i)
# a in %xmm0, x in %xmm1, b in %xmm2, i in edi

funct:
    # The following two instructions convert x to double
    vunpcklps %xmm1, %xmm1, %xmm1
    vcvtps2pd %xmm1, %xmm1
    vmulsd %xmm0, %xmm1, %xmm0      # Multiply a by x
    vcvtsi2sd %edi, %xmm1, %xmm1    # Convert i to double
    vdivsd %xmm1, %xmm2, %xmm2      # Compute b / i
    vsubsd %xmm2, %xmm0, %xmm0      # Subtract from a * x
    ret                             # Return
```



Arithmetik - AufgabeC/C++-Code:

```
double funct2(double w, int x, float y, long z){  
    // Vervollständigen Sie den C-Code  
}
```

Assembler-Code:

```
#double funct2(double w, int x, float y, long z)  
#w in %xmm0, x in %edi, y in %xmm1, z in %rsi  
funct2:  
    vcvtsi2ss %edi, %xmm2, %xmm2  
    vmulss %xmm1, %xmm2, %xmm1  
    # vunpcklps %xmm1, %xmm1, %xmm1 #*  
    vcvtps2pd %xmm1, %xmm2          #* Convert to double  
    vcvtsi2sdq %rsi, %xmm1, %xmm1  
    vdivsd %xmm1, %xmm0, %xmm0  
    vsubsd %xmm0, %xmm2, %xmm0  
    ret
```

Arithmetik - AufgabeC/C++-Code:

```
#include <math.h> // sqrt

float min(float f1, float f2){
    return (f1 < f2) ? f1 : f2;
}

int main(){
    float result = min(6.0, 5.76);
    result = sqrt(result);
    return (int) (result * 10);
}
```

Aufgabe:

Schreiben Sie den zugehörigen Assemblercode.

Verwenden Sie dabei für alle mathematischen Operationen die Tabelle für arithmetische Floatingpoint-Befehle.

Assembler-Code:

```
.section .data
format: .asciz "Result = %d\n"

.align 16
arr:
.float 6.0, 5.76, 10.0

.section .text
.globl main
.type main, @function
main:
    pushq %rbp
    movq %rsp, %rbp

    # insert your code here

    # print result
    movq $format, %rdi
    movl %eax, %esi
    movq $0, %rax
    call printf

    # exit main
    movq $0, %rax
    popq %rbp
```

Konstanten - Definition und Zugriff

AVX-Gleitkomma-Befehle können keine Immediate-Werte nutzen.  
 Allokieren und initialisieren von Speicher für eine Konstante:

C/C++-Code:

```
double cel2fahr(double temp)
{
    return 1.8 * temp + 32.0;
}
```

Assembler-Code:

```
.section .rodata
    .align 8
.LC2:
    .double 1.8

    .align 8
.LC3:
    .double 32.0

.section .text
.globl cel2fahr
.type cel2fahr, @function

# double cel2fahr(double temp)
# temp in %xmm0

cel2fahr:
    vmulsd .LC2, %xmm0, %xmm0
    vaddsd .LC3, %xmm0, %xmm0
    ret
```

Konstanten - Definition und Zugriff

```

.section .rodata
# .align 8
#.LC2:
# .long 3435973837 # Low-order 4 bytes of 1.8
# .long 1073532108 # High-order 4 bytes of 1.8
# .align 8
#.LC3:
# .long 0 # Low-order 4 bytes of 32.0
# .long 1077936128 # High-order 4 bytes of 32.0
.align 8
.LC2:
.double 1.8
.align 8
.LC3:
.double 32.0

.text
.globl cel2fahr
.type cel2fahr, @function
# double cel2fahr(double temp)
# temp in %xmm0
cel2fahr:
    vmulsd .LC2, %xmm0, %xmm0 # Multiply by 1.8
    vaddsd .LC3, %xmm0, %xmm0 # Add 32.0
    ret

```

Assembler-Code kommentiert:C/C++-Code:

```

double cel2fahr(double temp)
{
    return 1.8 * temp + 32.0;
}

```

Vergleiche

Instruction		Based on	Description
<code>ucomiss</code>	$S_1, S_2$	$S_2 - S_1$	Compare single precision
<code>ucomisd</code>	$S_1, S_2$	$S_2 - S_1$	Compare double precision

`vucomiss` und  
`vucomisd`  
entsprechend

Hinweis:

- Der zweite Operand muss zwingend ein `xmm`-Register sein, der erste Operand kann ein `xmm`-Register oder Speicher sein.

Ganzzahlvergleich: `cmp` (compare)

Gleitkommavergleich: `ucomi` (unordered compare)

Vergleiche

Ordering $S_2:S_1$	CF	ZF	PF
Unordered	1	1	1
$S_2 < S_1$	1	0	0
$S_2 = S_1$	0	1	0
$S_2 > S_1$	0	0	0

- **ZF** (Zero-Flag), **CF** (Carry-Flag) und **PF** (Parity-Flag) werden gesetzt. Abfragen wie less (<), less or equal (<=), greater (>) und greater or equal (>=) werden hier nicht verwendet, weil sie alle das Overflow-Flag und / oder Sign-Flag nutzen würden, das hier aber nicht gesetzt wird. Stattdessen below (<), below or equal (<=), above (>) und above or equal (>=) (s. Kap.8 – Teil 1, S. 9/10)
- Das Parity-Flag wird dann gesetzt, wenn mindestens einer der Operanden **NaN** (Not a Number) ist (auch **unordered** genannt).

Vergleiche - AufgabeC/C++-Code:

```
double funct3(int *ap, double b, long c, float *dp){
    // Vervollständigen Sie die Funktion
}
```

Assembler-Code:

```
#double funct3(int *ap, double b, long c, float *dp)
#ap in %rdi, b in %xmm0, c in %rsi, dp in %rdx
funct3:
    vmovss (%rdx), %xmm1
    vcvtsi2sd (%rdi), %xmm2, %xmm2
    ucomisd %xmm2, %xmm0
    jbe .L8
    vcvtsi2ssq %rsi, %xmm0, %xmm0
    vmulss %xmm1, %xmm0, %xmm1
    # vunpcklps %xmm1, %xmm1, %xmm1  #*
    vcvtps2pd %xmm1, %xmm0           #*Convert to double
    ret
.L8:
    vaddss %xmm1, %xmm1, %xmm1
    vcvtsi2ssq %rsi, %xmm0, %xmm0
    vaddss %xmm1, %xmm0, %xmm0
    # vunpcklps %xmm0, %xmm0, %xmm0  #*
    vcvtps2pd %xmm0, %xmm0           #*Convert to double
    ret
```

Darf nicht **jle**  
sein, s. Erklärung  
vorherige Seite.

