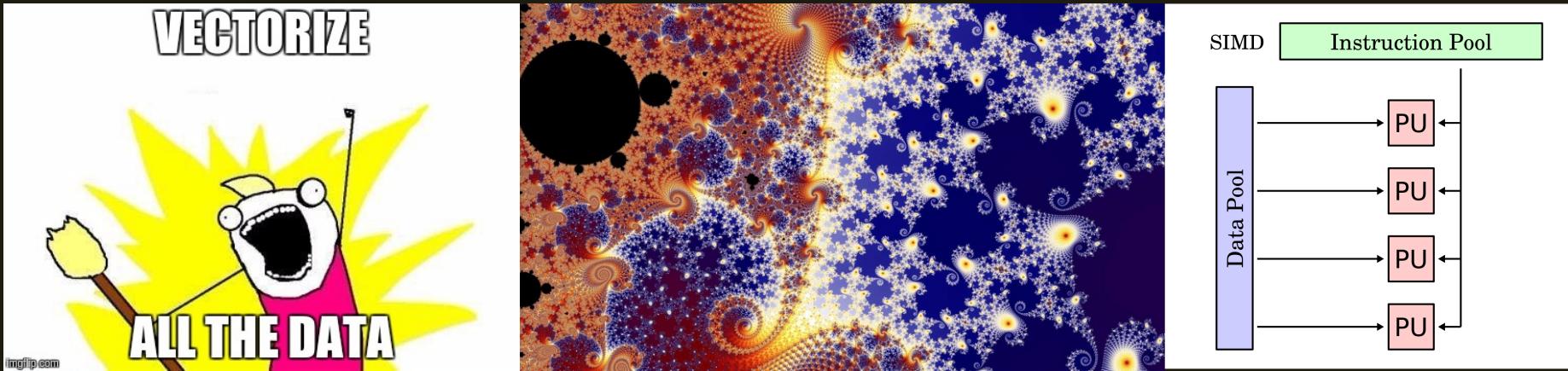


# Multimedia SS 2020



Ben Lorenz / Jonas Treumer

# Beispiel

- „Naive“ Bitmap-Abdunklung
  - Z.B. jede RGB-Komponente durch 2 teilen

```
void manipulate(bitmap_pixel_rgb_t* pixels, int count)
{
    for (int i = 0; i < count; i++)
    {
        bitmap_pixel_rgb_t* pixel = &pixels[i];

        pixel->r /= 2;
        pixel->g /= 2;
        pixel->b /= 2;
    }
}
```

# Beispiel

- „Naive“ Bitmap-Abdunklung
  - Workload simulieren und Zeit messen

```
//Manipulate bitmap pixels:  
clock_t t = clock();  
  
for (int i = 0; i < 4; i++)  
{  
    manipulate(pixels, width * height);  
}  
  
t = clock() - t;
```



# Beispiel

- „Naive“ Bitmap-Abdunklung
  - Workload simulieren und Zeit messen

```
//Manipulate bitmap pixels:  
clock_t t = clock();  
  
for (int i = 0; i < 4; i++)  
{  
    manipulate(pixels, width * height);  
}  
  
t = clock() - t;
```

~28ms

(clang -O3 -fno-tree-vectorize -fno-unroll-loops)

# Beispiel

- „Naive“ Bitmap-Abdunklung

- Disassemblieren

Beginn der Schleife



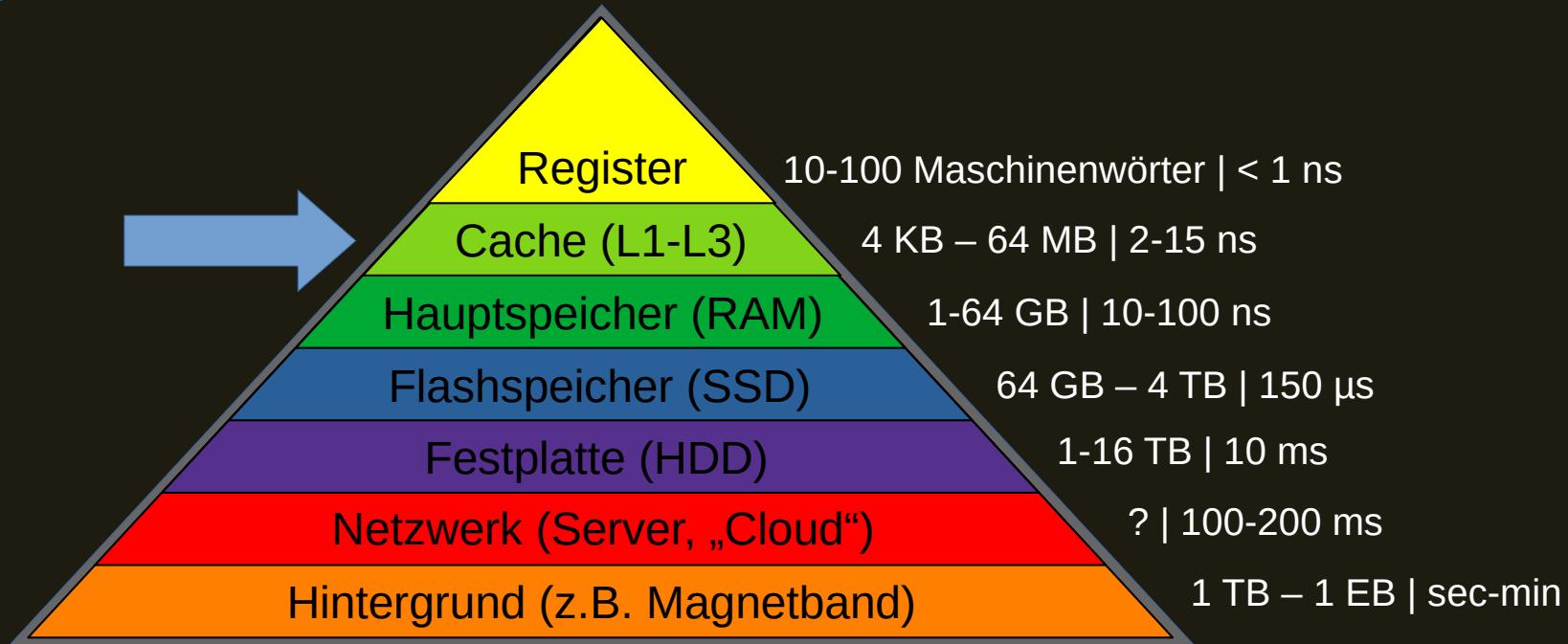
```
0x00000000000003c30 <+0>: test    esi,esi
0x00000000000003c32 <+2>: jle     0x3c54 <manipulate+36>
0x00000000000003c34 <+4>: mov     eax,esi
0x00000000000003c36 <+6>: xor     ecx,ecx
0x00000000000003c38 <+8>: nop
0x00000000000003c40 <+16>: shr    BYTE PTR [rdi+rcx*4],1
0x00000000000003c43 <+19>: shr    BYTE PTR [rdi+rcx*4+0x1],1
0x00000000000003c47 <+23>: shr    BYTE PTR [rdi+rcx*4+0x2],1
0x00000000000003c4b <+27>: add    rcx,0x1
0x00000000000003c4f <+31>: cmp    rax,rcx
0x00000000000003c52 <+34>: jne    0x3c40 <manipulate+16>
0x00000000000003c54 <+36>: ret
```

} count == 0 → return  
eax = count, i = 0  
Instruction padding!  
(r, g, b) >= 1  
i++; i < count  
return

~28ms

(clang -O3 -fno-tree-vectorize -fno-unroll-loops)

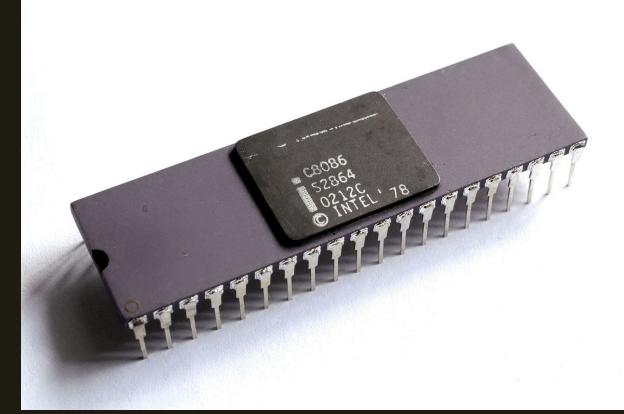
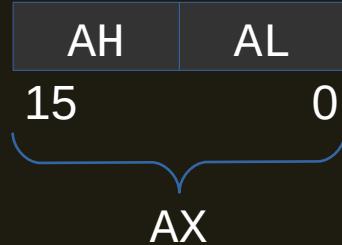
# Speicherhierarchie



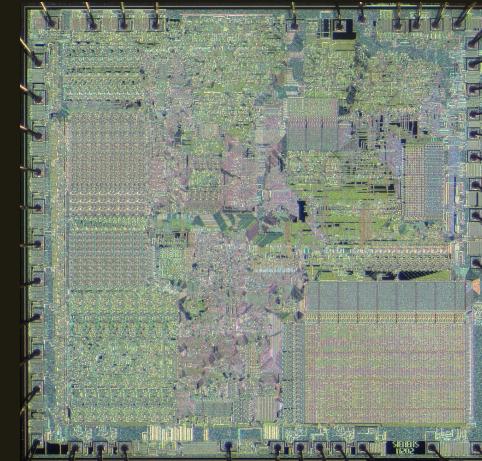
→ Alle Komponenten eines Pixels in ein Register bündeln?

# Prozessorregister

- 8086: 16-Bit-Prozessor (1978)
  - Befehlszeiger (IP bzw. PC)
  - Stapelzeiger (SP)
  - General-Purpose-Register (AX, BX, CX, ...)



[https://en.wikipedia.org/wiki/Intel\\_8086#/media/  
File:Intel\\_C8086.jpg](https://en.wikipedia.org/wiki/Intel_8086#/media/File:Intel_C8086.jpg)



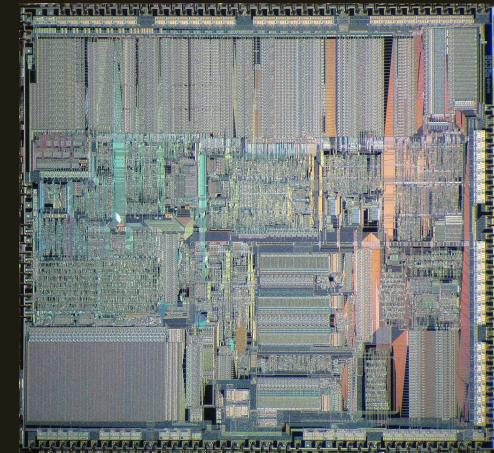
[https://de.wikipedia.org/wiki/Intel\\_8086#/media/Datei:Siemens\\_SAB\\_8086\\_die.JPG](https://de.wikipedia.org/wiki/Intel_8086#/media/Datei:Siemens_SAB_8086_die.JPG) 7

# Prozessorregister

- **80386: 32-Bit-Prozessor (1985)**
  - Abwärtskompatibel zur 16-Bit-Architektur
  - Erweiterung der Register auf 32 Bit
  - AX → EAX, SP → ESP, ...



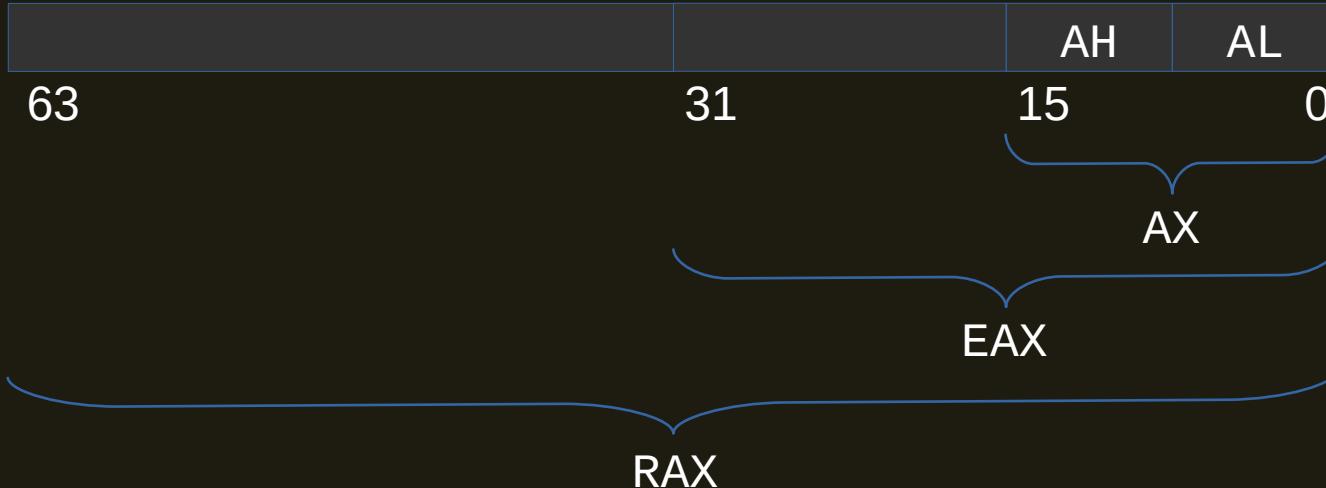
[https://en.wikipedia.org/wiki/Intel\\_80386#/media/File:KL\\_Intel\\_i386DX.jpg](https://en.wikipedia.org/wiki/Intel_80386#/media/File:KL_Intel_i386DX.jpg)



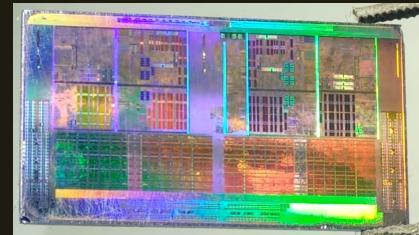
[https://de.wikipedia.org/wiki/Intel\\_80386#/media/Datei:Intel\\_80386\\_DX\\_Die.JPG](https://de.wikipedia.org/wiki/Intel_80386#/media/Datei:Intel_80386_DX_Die.JPG)

# Prozessorregister

- **Athlon 64:** 64-Bit-Prozessor (2003)
  - Abwärtskompatibel zur 16- und 32-Bit-Architektur
  - Erweiterung der Register auf 64 Bit
  - EAX → RAX, ESP → RSP, ...



[https://en.wikipedia.org/wiki/Athlon\\_64#/media/File:AMD\\_Athlon\\_64\\_3200+\\_ADA3200AE\\_P5AP.jpg](https://en.wikipedia.org/wiki/Athlon_64#/media/File:AMD_Athlon_64_3200+_ADA3200AE_P5AP.jpg)



<https://www.youtube.com/watch?v=Vg8q8EOi9Hw>

# Beispiel

- **Idee zur Beschleunigung**
  - Alle Komponenten in einen 32-Bit-Integer bündeln, dann shiften
  - Besser `memcpy()` als Pointer-Casting: *Strict Aliasing Rule* (→ Info)

```
void manipulate_u32(bitmap_pixel_rgb_t* pixels, int count)
{
    for (int i = 0; i < count; i++)
    {
        uint32_t pix;

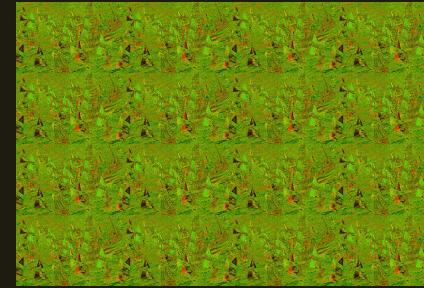
        memcpy(&pix, &pixels[i], sizeof(pix));
        pix >>= 1;
        memcpy(&pixels[i], &pix, sizeof(pix));
    }
}
```

Funktioniert das?

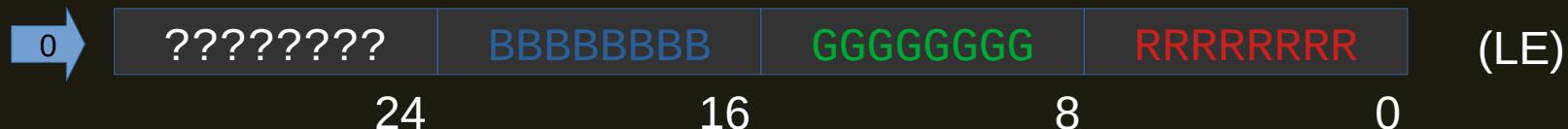


# Beispiel

- **Idee zur Beschleunigung**
  - Alle Komponenten in einen 32-Bit-Integer bündeln, dann shiften
  - Besser *memcpy()* als Pointer-Casting: *Strict Aliasing Rule* (→ Info)

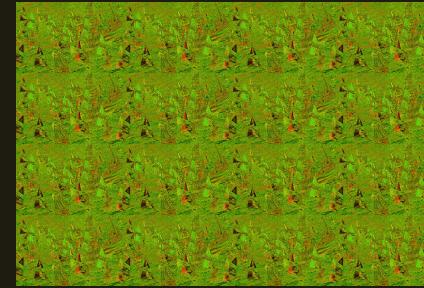


- Warum nicht?

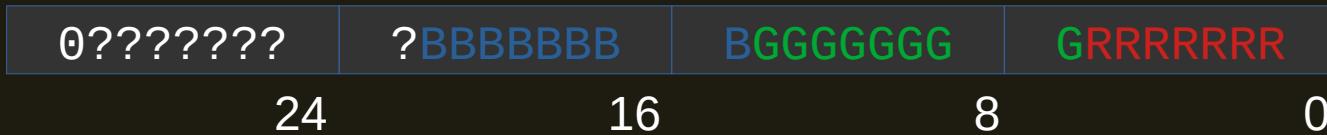


# Beispiel

- **Idee zur Beschleunigung**
  - Alle Komponenten in einen 32-Bit-Integer bündeln, dann shiften
  - Besser *memcpy()* als Pointer-Casting: *Strict Aliasing Rule* (→ Info)

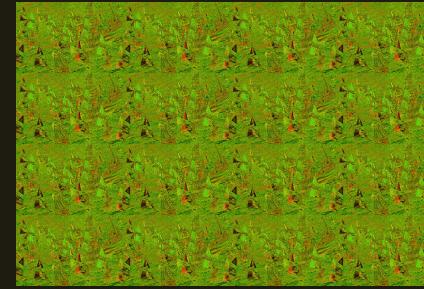


- Warum nicht?



# Beispiel

- **Idee zur Beschleunigung**
  - Alle Komponenten in einen 32-Bit-Integer bündeln, dann shiften
  - Besser *memcpy()* als Pointer-Casting: *Strict Aliasing Rule* (→ Info)

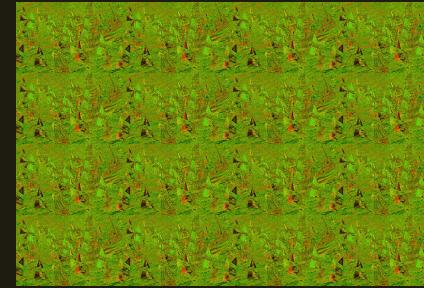


- Warum nicht?

|          |           |          |          |          |
|----------|-----------|----------|----------|----------|
| 0??????? | ?BBBBBBB  | BGGGGGGG | GRRRRRRR | (LE)     |
| &        | ????????? | 01111111 | 01111111 | 01111111 |
|          | 24        | 16       | 8        | 0        |

# Beispiel

- **Idee zur Beschleunigung**
  - Alle Komponenten in einen 32-Bit-Integer bündeln, dann shiften
  - Besser *memcpy()* als Pointer-Casting: *Strict Aliasing Rule* (→ Info)



- Warum nicht?

|            |          |          |          |      |
|------------|----------|----------|----------|------|
| ?????????? | 0BBBBBBB | 0GGGGGGG | 0RRRRRRR | (LE) |
| 24         | 16       | 8        | 0        |      |

# Beispiel

- **Idee zur Beschleunigung**
  - Alle Komponenten in einen 32-Bit-Integer bündeln, dann shiften
  - Besser `memcpy()` als Pointer-Casting: *Strict Aliasing Rule* (→ Kommentare)

```
void manipulate_u32(bitmap_pixel_rgb_t* pixels, int count)
{
    for (int i = 0; i < count; i++)
    {
        uint32_t pix;

        memcpy(&pix, &pixels[i], sizeof(pix));
        pix = (pix >> 1) & UINT32_C(0x7F7F7F7F);
        memcpy(&pixels[i], &pix, sizeof(pix));
    }
}
```

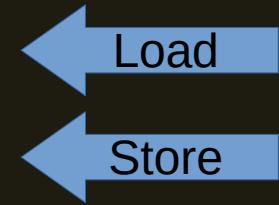
Maske

~28ms ~15ms  
(clang -O3 -fno-tree-vectorize -fno-unroll-loops)

# Beispiel

- **Idee zur Beschleunigung**
  - Alle Komponenten in einen 32-Bit-Integer bündeln, dann shiften
  - Besser *memcpy()* als Pointer-Casting: *Strict Aliasing Rule* (→ Kommentare)

```
0x000000000000003c60 <+0>:    test    esi,esi
0x000000000000003c62 <+2>:    jle     0x3c87 <manipulate_u32+39>
0x000000000000003c64 <+4>:    mov     eax,esi
0x000000000000003c66 <+6>:    xor     ecx,ecx
0x000000000000003c68 <+8>:    nop     DWORD PTR [rax+rax*1+0x0]
0x000000000000003c70 <+16>:   mov     edx,DWORD PTR [rdi+rcx*4]
0x000000000000003c73 <+19>:   shr     edx,1
0x000000000000003c75 <+21>:   and     edx,0x7f7f7f7f
0x000000000000003c7b <+27>:   mov     DWORD PTR [rdi+rcx*4],edx
0x000000000000003c7e <+30>:   add     rcx,0x1
0x000000000000003c82 <+34>:   cmp     rax,rcx
0x000000000000003c85 <+37>:   jne     0x3c70 <manipulate_u32+16>
0x000000000000003c87 <+39>:   ret
```



~28ms ~15ms  
(clang -O3 -fno-tree-vectorize -fno-unroll-loops)

# Beispiel

- Weiter beschleunigen?
  - Zwei Pixel auf einmal in einen 64-Bit-Integer bündeln, dann shiften
  - Überhang für  $(count \% 2) \neq 0$  beachten!

```
void manipulate_u64(bitmap_pixel_rgb_t* pixels, int count)
{
    int vec_count = (count / 2) * 2;
    bitmap_pixel_rgb_t* overhang = &pixels[vec_count];

    for (; pixels != overhang; pixels += 2)
    {
        uint64_t pix;

        memcpy(&pix, pixels, sizeof(pix));
        pix = (pix >> 1) & UINT64_C(0x7F7F7F7F7F7F7F7F);
        memcpy(pixels, &pix, sizeof(pix));
    }

    manipulate_u32(overhang, count - vec_count);
}
```

~28ms ~15ms ~10ms

(clang -O3 -fno-tree-vectorize -fno-unroll-loops)

# Beispiel

```
0x00000000000003c90 <+0>:    mov    r9d,esi
0x00000000000003c93 <+3>:    shr    r9d,0x1f
0x00000000000003c97 <+7>:    add    r9d,esi
0x00000000000003c9a <+10>:   and    r9d,0xfffffff
0x00000000000003c9e <+14>:   movsxd r8,r9d
0x00000000000003ca1 <+17>:   lea    eax,[rsi+0x1]
0x00000000000003ca4 <+20>:   cmp    eax,0x3
0x00000000000003ca7 <+23>:   jb    0x3cd7 <manipulate_u64+71>
0x00000000000003ca9 <+25>:   lea    rdx,[r8*4+0x0]
0x00000000000003cb1 <+33>:   xor    ecx,ecx
0x00000000000003cb3 <+35>:   movabs r10,0x7f7f7f7f7f7f7f7f
0x00000000000003cbd <+45>:   nop    DWORD PTR [rax]
0x00000000000003cc0 <+48>:   mov    rax,QWORD PTR [rdi+rcx*1]
0x00000000000003cc4 <+52>:   shr    rax,1
0x00000000000003cc7 <+55>:   and    rax,r10
0x00000000000003cca <+58>:   mov    QWORD PTR [rdi+rcx*1],rax
0x00000000000003cce <+62>:   add    rcx,0x8
0x00000000000003cd2 <+66>:   cmp    rdx,rcx
0x00000000000003cd5 <+69>:   jne    0x3cc0 <manipulate_u64+48>
0x00000000000003cd7 <+71>:   sub    esi,r9d
0x00000000000003cda <+74>:   jle    0x3d07 <manipulate_u64+119>
```

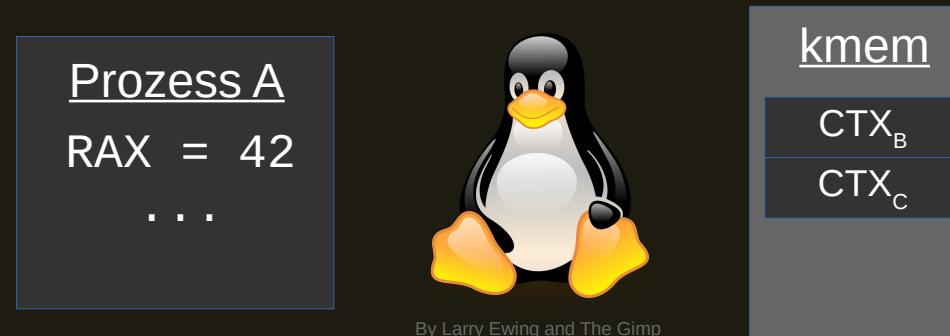
Load

Store

... abgeschnitten

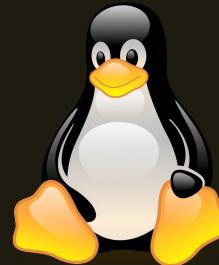
# SIMD

- **Weiter beschleunigen?**
  - Noch breitere Register
  - Spezielle Vektorinstruktionen (z.B. Einsparung des Maskierens im Beispiel)  
→ *SIMD (Single Instruction, Multiple Data)* bzw. *Instruction Level Parallelism*
- **Neue Register**
  - Kontextwechsel: Neue Register brauchen Unterstützung im Betriebssystem

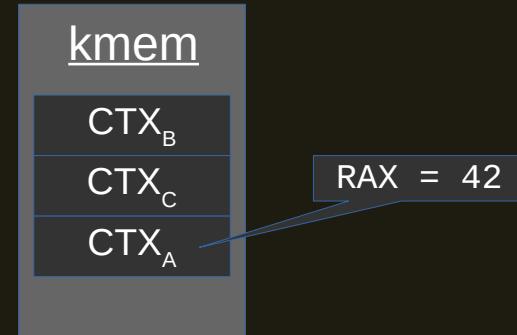


# SIMD

- **Weiter beschleunigen?**
  - Noch breitere Register
  - Spezielle Vektorinstruktionen (z.B. Einsparung des Maskierens im Beispiel)
    - SIMD (*Single Instruction, Multiple Data*) bzw. *Instruction Level Parallelism*
- **Neue Register**
  - Kontextwechsel: Neue Register brauchen Unterstützung im Betriebssystem

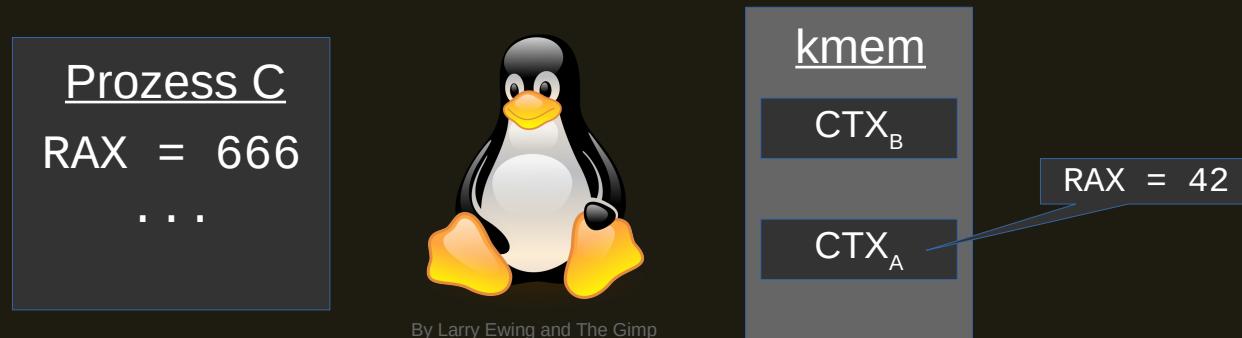


By Larry Ewing and The Gimp



# SIMD

- **Weiter beschleunigen?**
  - Noch breitere Register
  - Spezielle Vektorinstruktionen (z.B. Einsparung des Maskierens im Beispiel)
    - SIMD (*Single Instruction, Multiple Data*) bzw. *Instruction Level Parallelism*
- **Neue Register**
  - Kontextwechsel: Neue Register brauchen Unterstützung im Betriebssystem

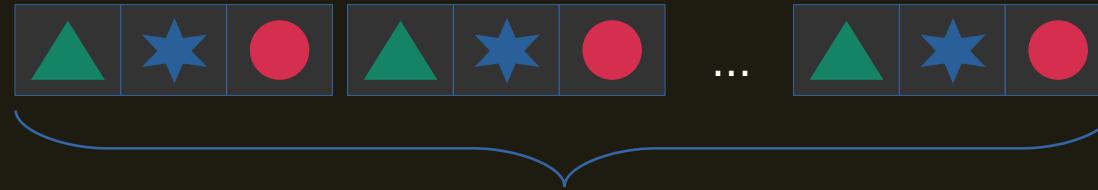


- Neue Instruktionen

- Gegeben: Ein *struct*-Datentyp mit  $m$  heterogenen Komponenten ...



- ... und ein Array bestehend aus  $n$  solcher *structs*:



**Array of Structures (AoS) mit  $n$  Elementen**

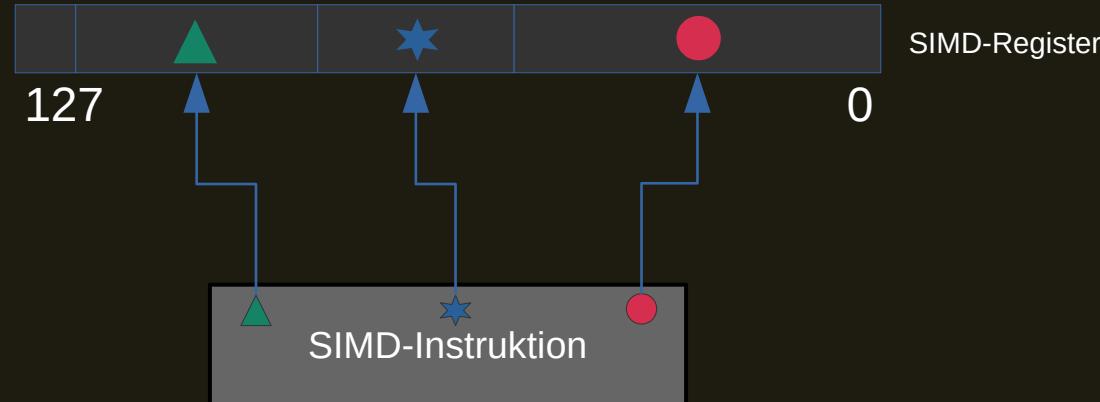
- Neue Instruktionen

- Auf jeder *struct*-Instanz müssen heterogene Operationen ausgeführt werden.
- Naiver Ansatz:

```
for (int i = 0; i < n; i++)  
{  
    ▲- Operation(a[i].▲)  
    ★- Operation(a[i].★)  
    ●- Operation(a[i].●)  
}
```

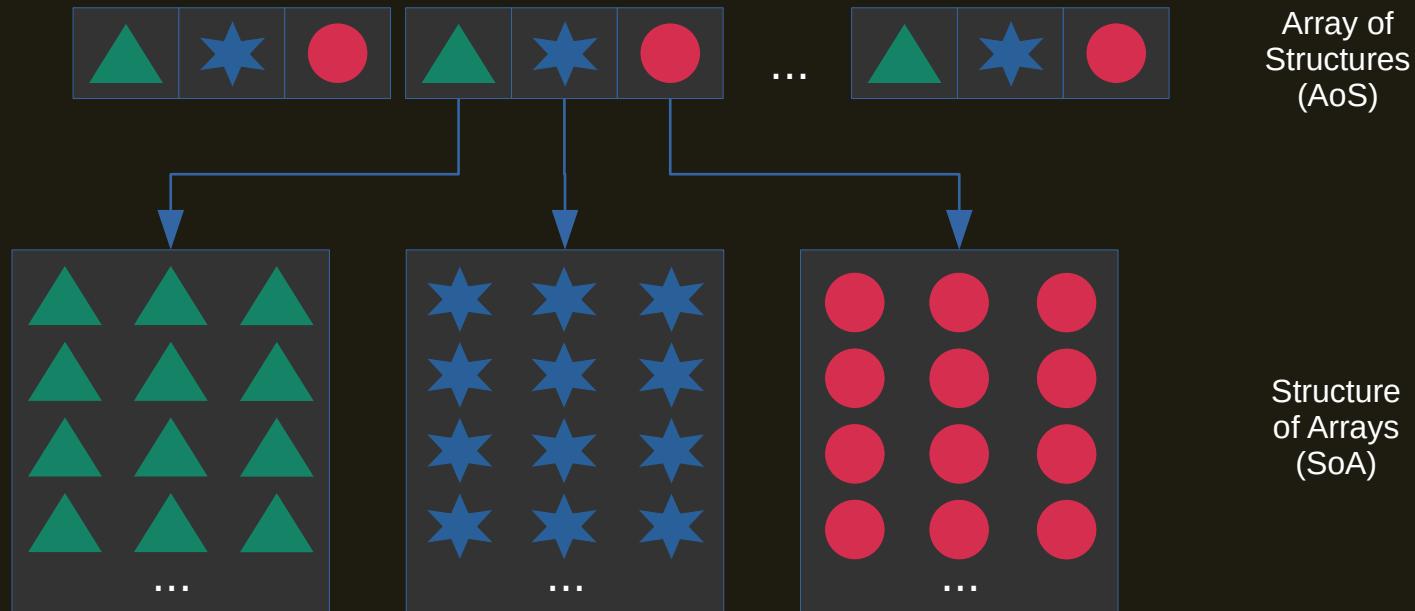
# SIMD

- **Neue Instruktionen**
  - Auf jeder *struct*-Instanz müssen heterogene Operationen ausgeführt werden.
  - SIMD-Ansatz:



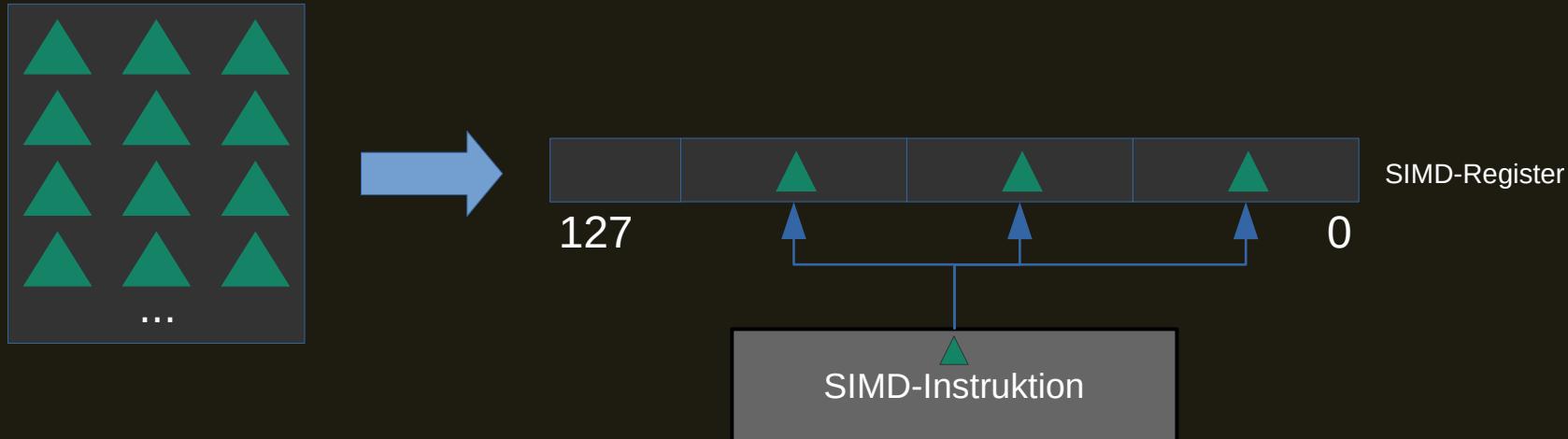
→ kombinatorische Explosion des Instruktionssatzes

- Neue Instruktionen
  - Besser: Zerlegung in homogene Arrays ...



# SIMD

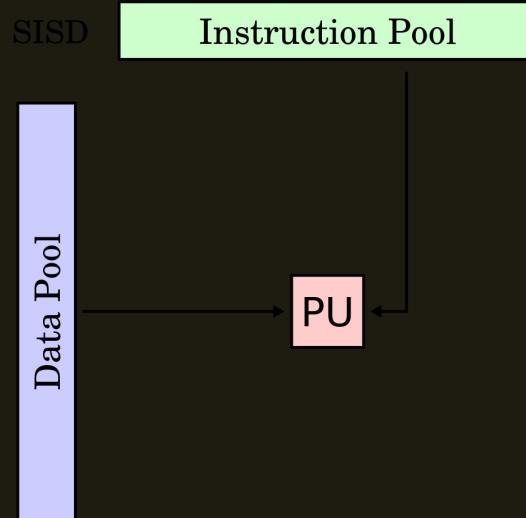
- Neue Instruktionen
  - ... und separate Verarbeitung.



→ nur drei neue Instruktionen, aber Overhead für Transformationen:  
AoS → SoA → AoS

# Flynn-Taxonomie

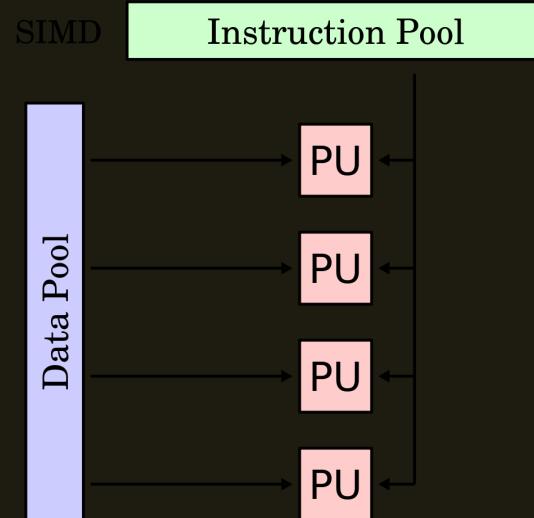
- **Single Instruction, Single Data**
  - Keine Parallelverarbeitung
  - Eine Instruktion operiert auf einem Datum.



<https://upload.wikimedia.org/wikipedia/commons/thumb/a/ae/SISD.svg/1024px-SISD.svg.png>

# Flynn-Taxonomie

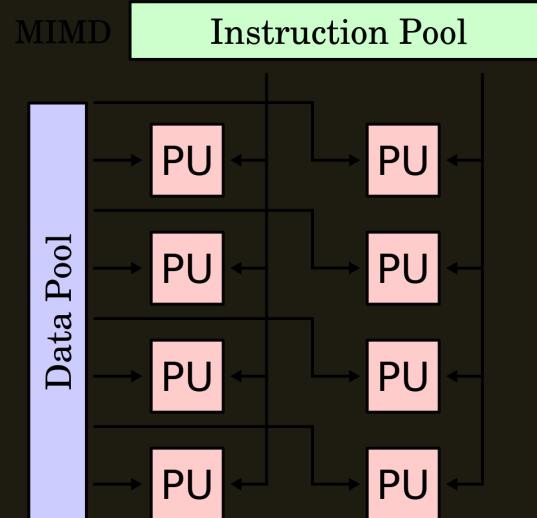
- Single Instruction, Multiple Data
  - „Unser Fall“: Vektorrechner
  - Eine Instruktion operiert auf mehreren Daten.



<https://upload.wikimedia.org/wikipedia/commons/thumb/a/ae/SIMD.svg/1024px-SIMD.svg.png>

# Flynn-Taxonomie

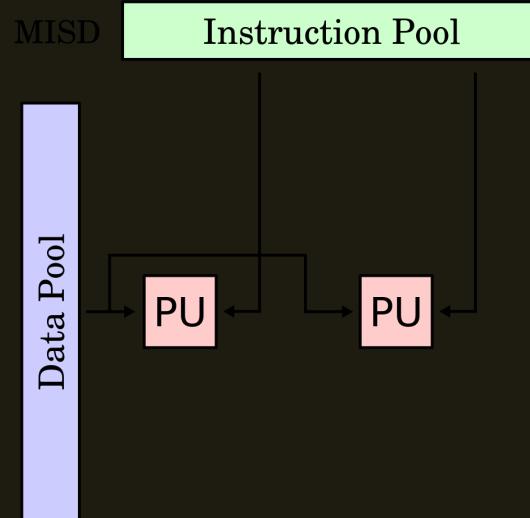
- **Multiple Instructions, Multiple Data**
  - Vektorrechner mit mehreren Kernen
  - Mehrere Instruktionen operieren auf mehreren Daten.



<https://upload.wikimedia.org/wikipedia/commons/thumb/a/ae/MIMD.svg/1024px-MIMD.svg.png>

# Flynn-Taxonomie

- **Multiple Instructions, Single Data**
  - Ungewöhnliche Architektur
  - Beispiel: Redundanz (eine Instruktion „rechnet“, eine andere „validiert“)

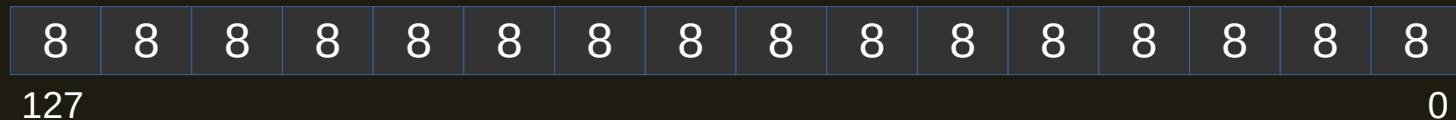


<https://upload.wikimedia.org/wikipedia/commons/thumb/a/ae/MISD.svg/1024px-MISD.svg.png>

# Implementierungen

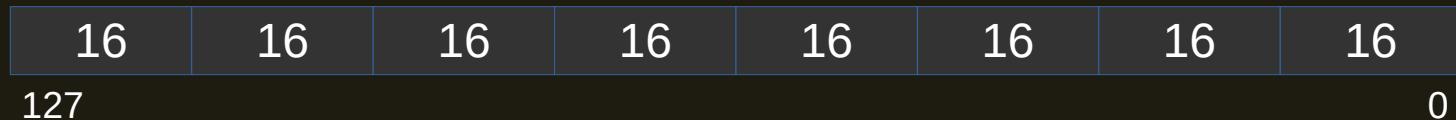
- **AltiVec**

- Motorola, IBM (*Vector Multimedia Extension*) und Apple (*Velocity Engine*)
- PowerPC-Architektur (vor Apples x86-Wechsel)
- Entwicklung: 1996-1998
- 128-Bit-Register (32 Stück)
- Drei-Register-Operationen
- Datentypen: Integer (8-32 Bit), Floating Point (Single Precision)



# Implementierungen

- **AltiVec**
  - Motorola, IBM (*Vector Multimedia Extension*) und Apple (*Velocity Engine*)
  - PowerPC-Architektur (vor Apples x86-Wechsel)
  - Entwicklung: 1996-1998
  - 128-Bit-Register (32 Stück)
  - Drei-Register-Operationen
  - Datentypen: Integer (8-32 Bit), Floating Point (Single Precision)



# Implementierungen

- **Altivec**

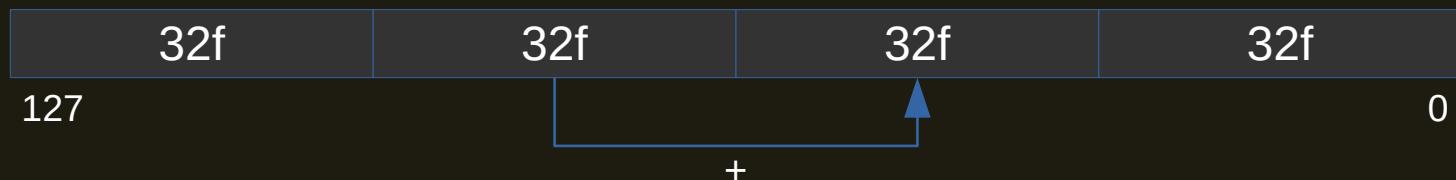
- Motorola, IBM (*Vector Multimedia Extension*) und Apple (*Velocity Engine*)
- PowerPC-Architektur (vor Apples x86-Wechsel)
- Entwicklung: 1996-1998
- 128-Bit-Register (32 Stück)
- Drei-Register-Operationen
- Datentypen: Integer (8-32 Bit), Floating Point (Single Precision)

|     |    |    |    |
|-----|----|----|----|
| 32  | 32 | 32 | 32 |
| 127 |    |    | 0  |

# Implementierungen

- **Altivec**

- Motorola, IBM (*Vector Multimedia Extension*) und Apple (*Velocity Engine*)
- PowerPC-Architektur (vor Apples x86-Wechsel)
- Entwicklung: 1996-1998
- 128-Bit-Register (32 Stück)
- Drei-Register-Operationen
- Datentypen: Integer (8-32 Bit), Floating Point (Single Precision)
- Unterstützung für *horizontale* Instruktionen



# Implementierungen

- **MMX**

- Ursprünglich bedeutungslose Abkürzung (*Multimedia Extension*, *Matrix Math Extension*)
- x86-Architektur
- 1997 erschienen
- 64-Bit-Register (8 Stück): MM0 ... MM7
- Datentypen: Integer (8-64 Bit), aber kein Floating Point
- Keine neue Registerhardware (Chipgröße, Kontextwechsel)
- Stattdessen: Doppelbelegung der x87-FPU-Register
- Sättigungsarithmetik: Clamping in Hardware

# Implementierungen



$(\text{RGB} + 16) \% 256$



# Implementierungen



sat(RGB + 16)



# Implementierungen

- **3DNow!**
  - 3D-Leistung damals primär durch die FPU begrenzt
  - x86-Architektur
  - 1998 erschienen
  - Erweiterung von MMX um Floating-Point-Daten (Single Precision)
  - Drittbelegung für x87-FPU-Register (zusätzlich zu MMX)
  - Horizontale Operationen

# Implementierungen

- SSE

- *Streaming SIMD Extensions*
- x86(-64)-Architektur
- 1999 erschienen
- 128-Bit-Register (8 Stück): XMM0 ... XMM7
- Ab x86-64: XMM8 ... XMM15
- Neue Hardware → Unterstützung in den Betriebssystemen erforderlich
- Ursprünglich: Floating-Point-Daten (Single Precision), aber kein Integer
- SSE-Register als Teil der Calling Convention für x86-64:

```
double test(double a, double b)
{
    return a + b;
}
```

|  |
|--|
| 0x00000000000001130 <+0>: addsd  xmm0,xmm1 |
| 0x00000000000001134 <+4>: ret              |

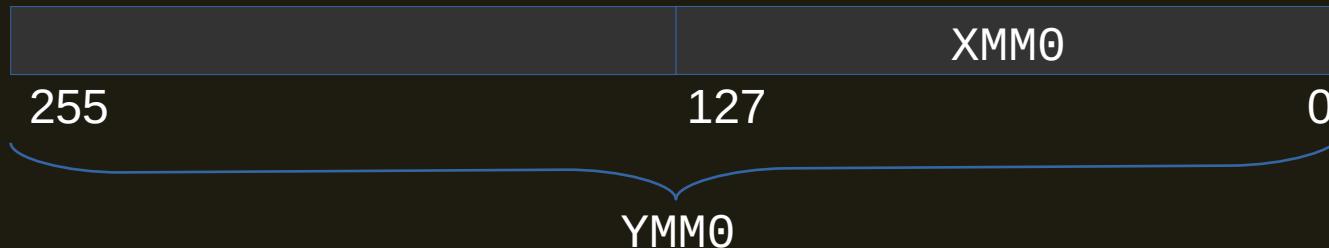
# Implementierungen

- SSE
  - SSE2: Integerdaten, Double Precision
  - SSE3: Horizontale Operationen
  - SSSE3: Ergänzungen für Integerdaten (Beträge, Skalarprodukte, weitere horiz. Operationen, Shuffling, ...)
  - SSE4: Weitere Ergänzungen (Minima / Maxima, Casts, CRC, Stringoperationen, ...)

# Implementierungen

- AVX

- *Advanced Vector Extensions*
- x86(-64)-Architektur
- 2008 erschienen
- 256-Bit-Register (8 bzw. 16 Stück): YMM0 . . . YMM7 resp. YMM15
- Erweiterung der existierenden SSE-Register → neue Hardware → Betriebssysteme anpassen



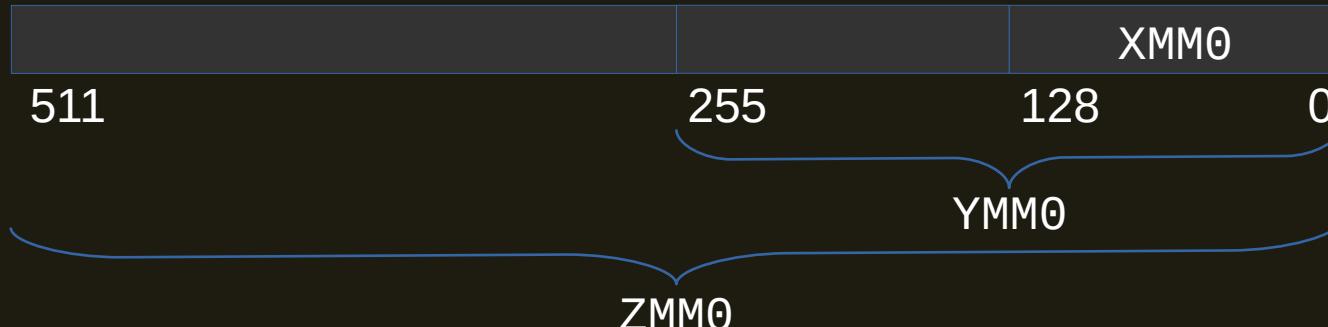
# Implementierungen

- AVX

- Drei-Operanden-Format (auch für SSE-Instruktionen):

$$a := b \text{ op } c \text{ statt } a := a \text{ op } b$$

- Ursprünglich: Floating-Point-Daten (Single und Double Precision), aber kein Integer
- **AVX2**: Erweiterung auf Integerdaten, Drei-Operanden-Format für weitere Instruktionen (z.B. bitweise Manipulationen), Gather / Scatter, Shifts, Fused Multiply-Accumulate
- **AVX-512**: Erweiterung der AVX-Register auf 512 Bit, 32 Stück  $\rightarrow$  ZMM0 ... ZMM31



# Implementierungen

- **NEON, Helium und SVE**
  - SIMD-Instruktionen für die ARM-Plattform
  - **NEON:**
    - Einführung mit ARMv7 (2004)
    - 128-Bit-Register (16 Stück, q0 ... q15)
    - Integer- und Floating-Point-Daten (Half / Single Precision; Double Precision skalar)
    - Später (AArch64, 2011): Erhöhung auf 32 Register, vektorisierte Double Precision
    - Drei-Operanden-Format
    - Horizontale Operationen
  - **Helium:** NEON-Erweiterung, > 150 neue Instruktionen (z.B. Gather / Scatter)
  - **Scalable Vector Extension:** 128-2048 Bit, dynamische Vektorlängen (HPC-Bereich)

# Implementierungen

| <u>Name</u> | <u>Seit</u> | <u>Breite</u>                       | <u>Register</u>       | <u>Datentypen</u>                             | <u>Header</u> |
|-------------|-------------|-------------------------------------|-----------------------|---|---------------|
| AltiVec     | 1996-98     | 128 Bit (32x)                       | separat               | Int (8-32), FP (Single)                       | altivec.h     |
| MMX         | 1997        | 64 Bit (8x)                         | x87-FPU-Zweitbelegung | Int (8-64)                                    | mmINTRIN.h    |
| 3DNow!      | 1998        |                                     | x87-FPU-Drittbelegung | Int (8-64), FP (Single)                       | -             |
| SSE         | 1999        | 128 Bit<br>(8x [x86], 16x [x86-64]) | separat (XMM*)        | FP (Single)                                   | xmmINTRIN.h   |
| SSE2        | 2000        |                                     |                       | Int (8-64),<br>FP (Single, Double)            | emmmINTRIN.h  |
| SSE3        | 2006        |                                     |                       |   | pmmINTRIN.h   |
| SSSE3       | 2006        |                                     |                       |   | tmmINTRIN.h   |
| SSE4        | 2006        |                                     |                       |   | nmmINTRIN.h   |
| AVX         | 2008        | 256 Bit<br>(8x [x86], 16x [x86-64]) | separat (XMM* → YMM*) | FP (Single, Double)                           | immmINTRIN.h  |
| AVX2        | 2013        |                                     |                       | Int (8-64),<br>FP (Single, Double)            |               |
| AVX-512     | 2013        | 512 Bit (32x)                       | separat (YMM* → ZMM*) | Int (8-64),<br>FP (Half, Single, Double [He]) | zmmINTRIN.h   |
| NEON        | 2004        | 128 Bit (16x, 32x [AArch64])        | separat (q*)          |   | arm_neon.h    |
| Helium      | 2019        | 128 Bit (8x)                        | separat (q*)          |   | arm_mve.h     |
| SVE         | 2016-20     | Variabel (128-2048 Bit)             | separat               | Int, FP                                       | arm_sve.h     |

# Anwendung

- Wie programmiere ich mit SIMD-Instruktionen?
  - 1) Den Compiler automatisch vektorisieren lassen (gut für einfache Fälle)
  - 2) Inline-Assembler schreiben
  - 3) Intrinsics verwenden: <https://software.intel.com/sites/landingpage/IntrinsicsGuide>

## Intrinsics Guide

### Technologies

- MMX
- SSE
- SSE2
- SSE3
- SSSE3
- SSE4.1
- SSE4.2
- AVX
- AVX2
- FMA
- AVX-512
- KNC
- SVML
- Other

The Intel Intrinsics Guide is an interactive reference tool for Intel intrinsic instructions, which are C style functions that provide access to many Intel instructions - including Intel® SSE, AVX, AVX-512, and more - without the need to write assembly code.

shift

?

|  |          |
|--|----------|
| <code>__m256i _mm256_alignr_epi8 (__m256i a, __m256i b, const int imm8)</code> | vpalignr |
| <code>__m256i _mm256_bslli_epi128 (__m256i a, const int imm8)</code>           | vpslldq  |
| <code>__m128i _mm_bslli_si128 (__m128i a, int imm8)</code>                     | pslldq   |
| <code>__m256i _mm256_bsrlri_epi128 (__m256i a, const int imm8)</code>          | vpsrldq  |
| <code>__m128i _mm_bsrlri_si128 (__m128i a, int imm8)</code>                    | psrldq   |
| <code>__m128i _mm_sll_epi16 (__m128i a, __m128i count)</code>                  | psllw    |
| <code>__m256i _mm256_sll_epi16 (__m256i a, __m128i count)</code>               | vpsllw   |
| <code>__m128i _mm_sll_epi32 (__m128i a, __m128i count)</code>                  | pslld    |
| <code>__m256i _mm256_sll_epi32 (__m256i a, __m128i count)</code>               | vpslld   |
| <code>__m128i _mm_sll_epi64 (__m128i a, __m128i count)</code>                  | psllq    |
| <code>__m256i _mm256_sll_epi64 (__m256i a, __m128i count)</code>               | vpsllq   |
| <code>__m128i _mm_slli_epi16 (__m128i a, int imm8)</code>                      | psllw    |