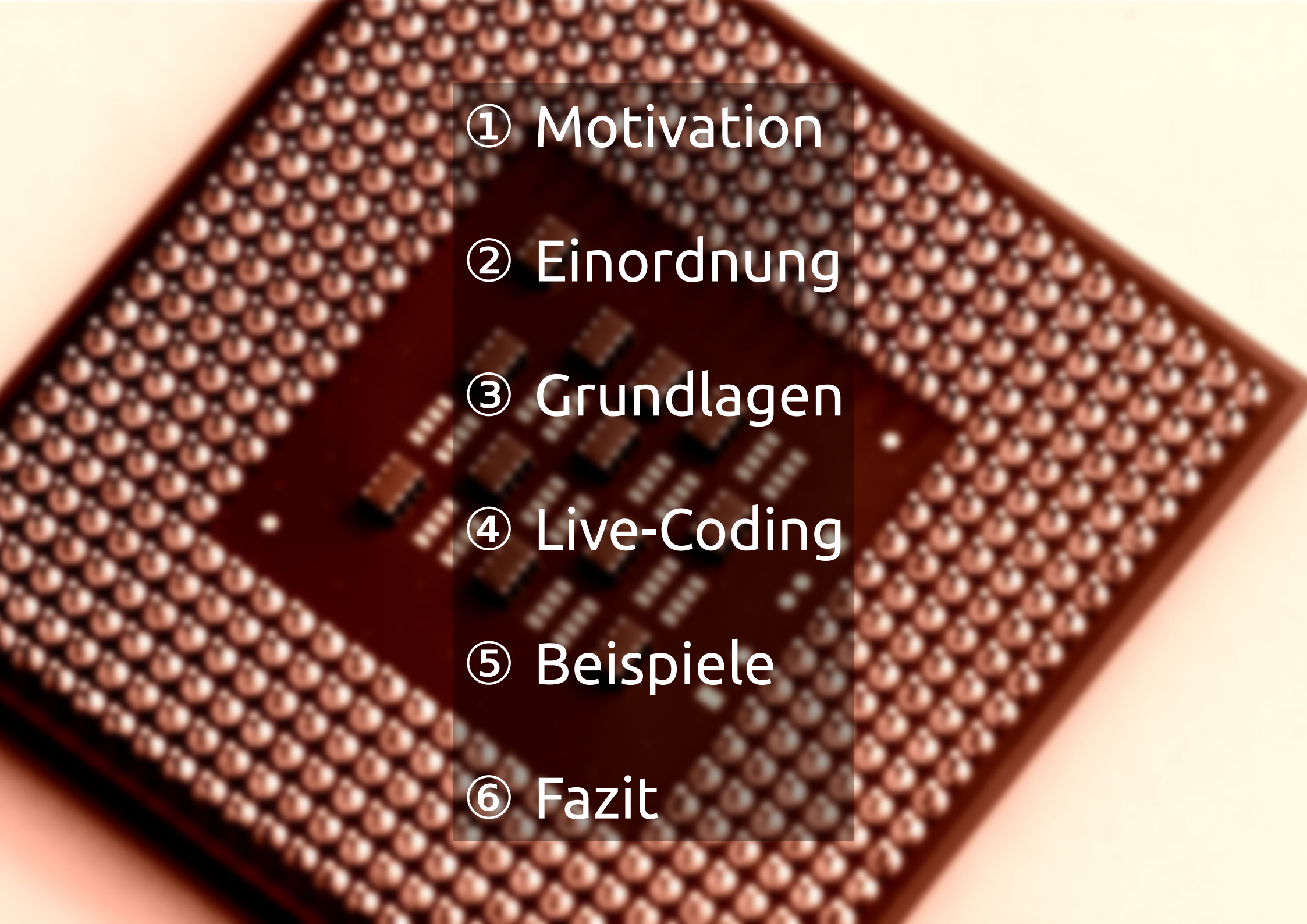

Wir schlagen den C++ Compiler
oder
manchmal darf es etwas
(x86) Assembler sein


Was Euch erwartet



① Motivation

② Einordnung

③ Grundlagen

④ Live-Coding

⑤ Beispiele

⑥ Fazit

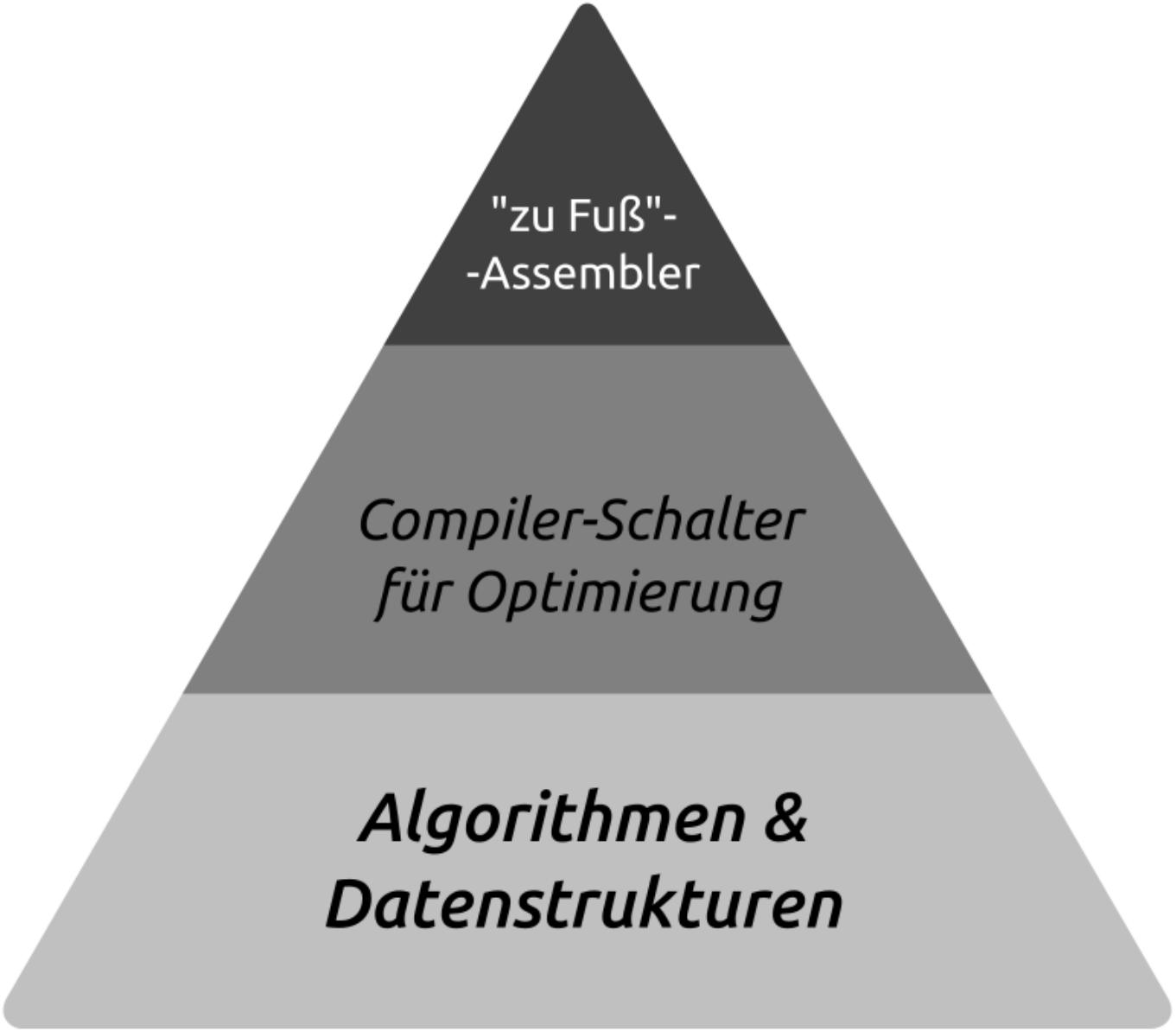
wieso stehe ich hier, wieso solltet Ihr u.U. zum Assembler greifen
wo passt dieser Vortrag rein in das große Feld der Optimierung
Grundlagen Compiler-Frontend, CPU, Register, Assember, Aufrufkonventionen
Beispiele unterschiedlichen Umfangs, Live-Coding (zum Ende hin)
vorläufiges Fazit/Zwischenbilanz, da noch nicht alle Beispiele fertig sind


Motivation

Assembler-Knowhow aus MC68000/Amiga-Zeiten wiederbeleben/übertragen
"bang per byte": ls -l hello-world.cpp vs. 4K Scene-Demos
"Trau' ich der Ausgabe vom Compiler?"
oft bleibt ein Großteil der Transistoren ungenutzt liegen
das GHz-Rennen ist schon lange vorbei
"nahe am Metall" läßt einen C++ im Alltag besser schätzen
Intrinsics - "nett gedacht, schlecht gemacht"
* sind genauso wenig CPU-Architektur-abhängig wie Assembler
* Lawine von Unterstrichen machen C++-Code nicht schöner
u.a. Wegbereiter für:
* Treiber- oder Systemprogrammierung
* Exploit-Analyse

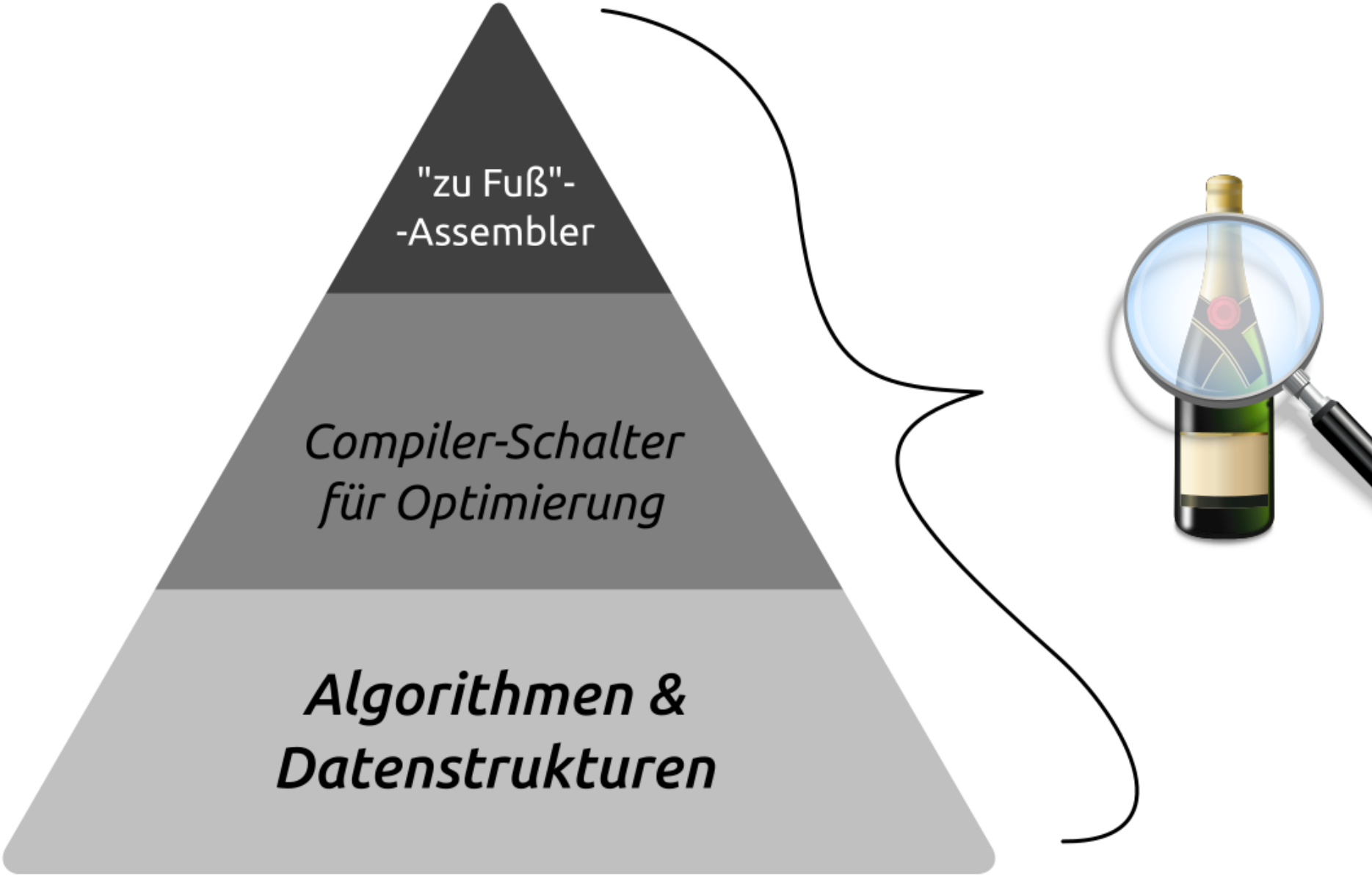

Einordnung

"Optimierungspyramide"



- ... in das große Gebiet der Optimierung
  * Algorithmen (Makro-Optimierung)
  * Datenstrukturen (Makro-Optimierung)

"Optimierungspyramide"

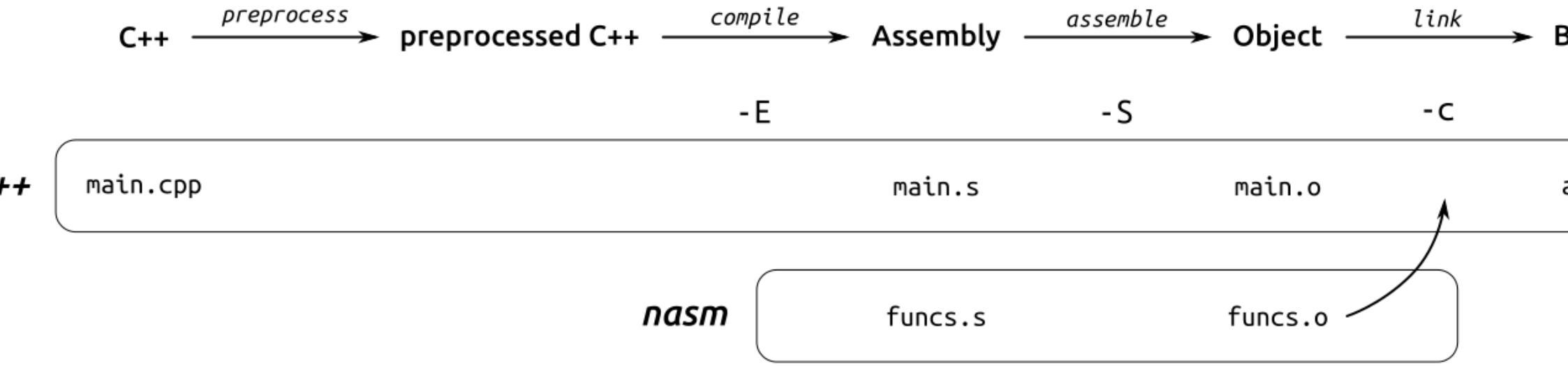


  * Messen, messen, messen (Flaschenhälse identifizieren)
  * ... um Flaschenhälse zu beseitigen
  * optional (für die Harten): CPU-Zyklen zählen und wegfeilen
  * optional (für die Steinbeisser): ELF-, PE/COFF-, Mach-O Format wegbürsten
- Optimierung im kleinen
- "die 20% des Projekts, die 80% der Zeit verbrennen"
- sehr Unix-oid (Linux, OSX,clang,gcc)
- Windows und MS' Werkzeuge kommen etwas kurz


Grundlagen

- RISC vs. CISC (bzw. intel vs. ARM)
- Register Modell x86 (64 bit)
- Aufrufkonentionen (calling conventions)
* GNU/Unix vs. MS/Windows
* MS' shadow-space ... WTF?!
- Compiler-Schalter (gcc, clang)
* -O0, -O1, -O2, -O3, -Os -Ofast
* stack- vs. register-passing
- Intrinsics vs Assembly
- SIMD
- objdump, hexdump

Vom Quellcode zum Programm

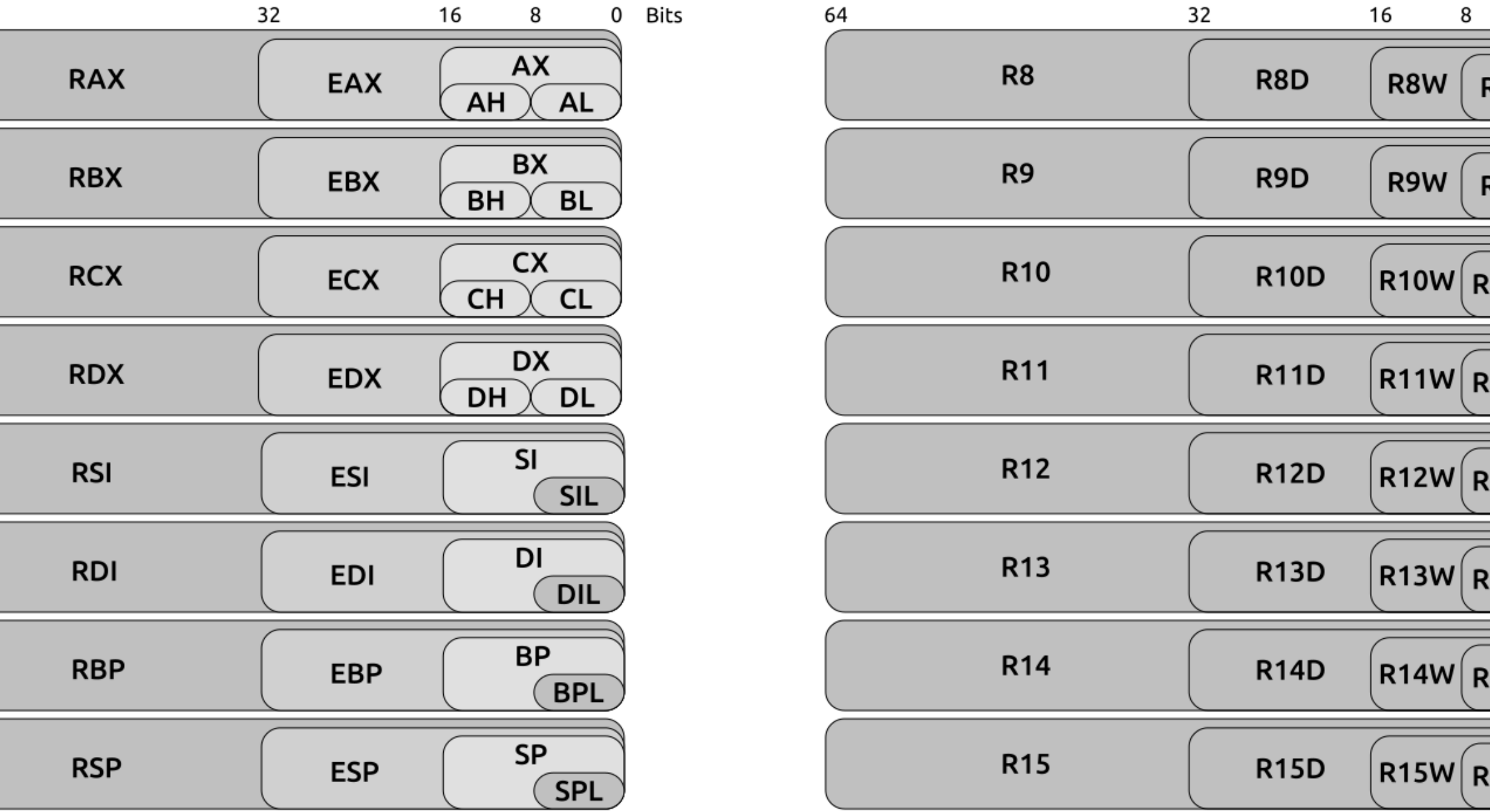


Compilerschalter

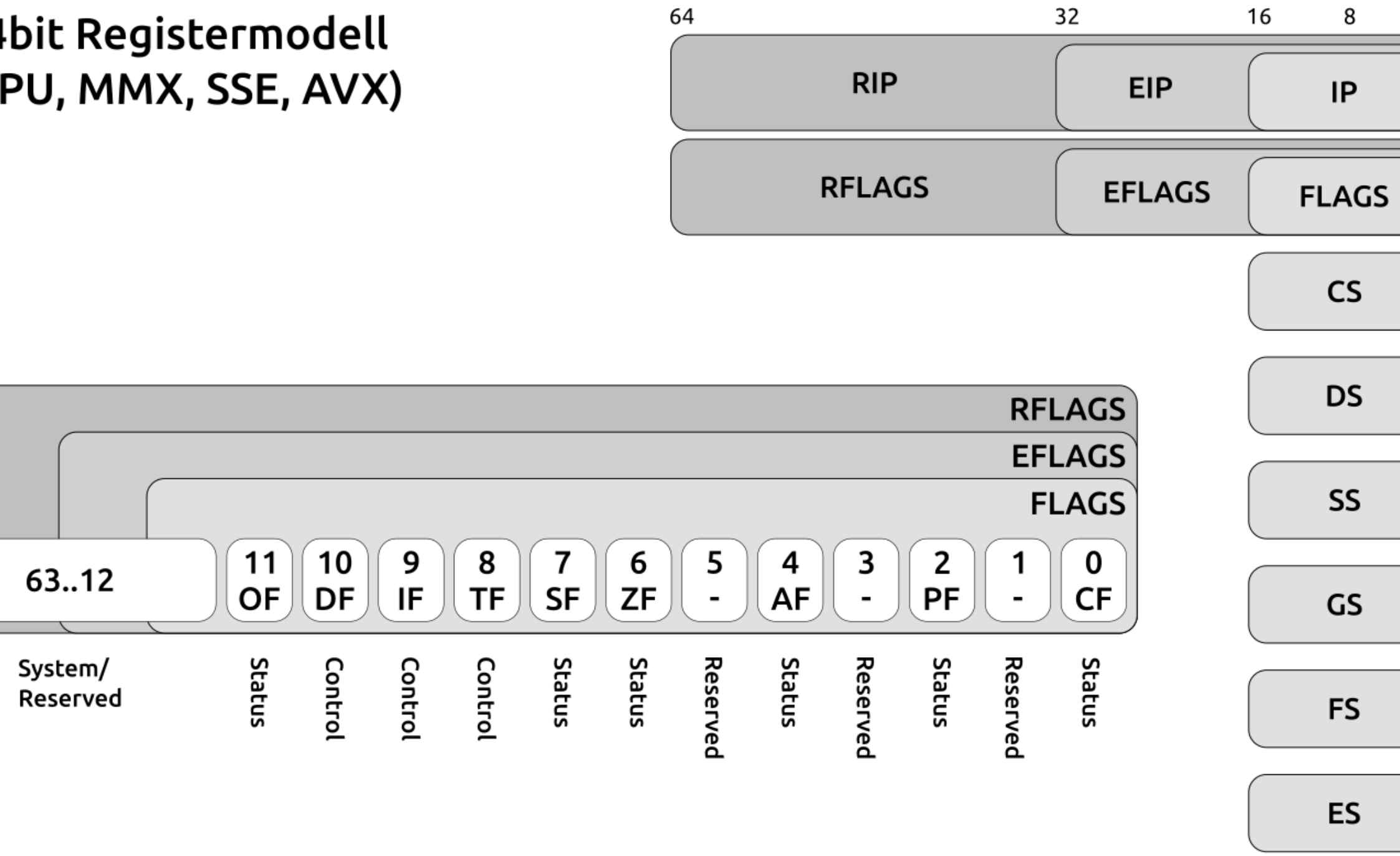
gcc (6.3)	clang++ (3.9.1)
-Og	-O0
-O0	
-O1	-O1
-O2	-O2
-O3	-O3/-O4
-Ofast	-Ofast
-Os	-Os
	-Oz

Ringmodell des x86

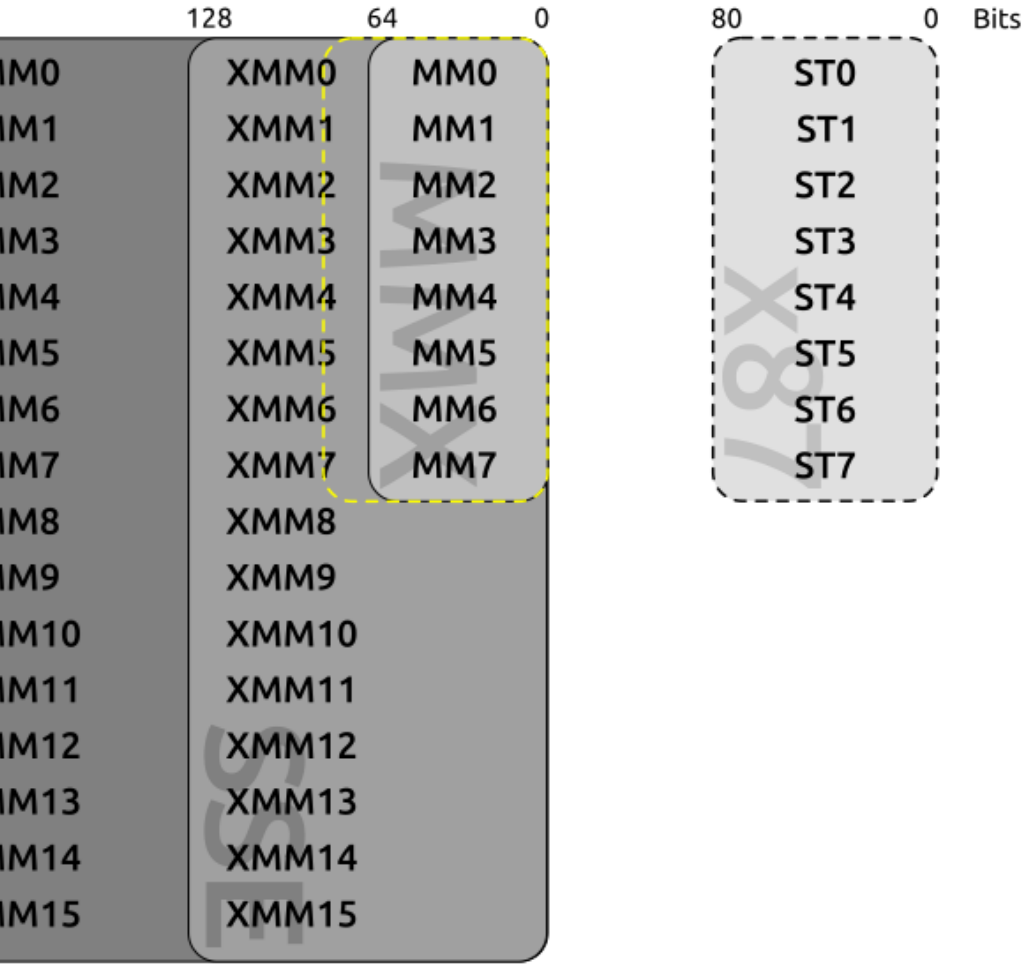
4bit Registermodell
PU, MMX, SSE, AVX)

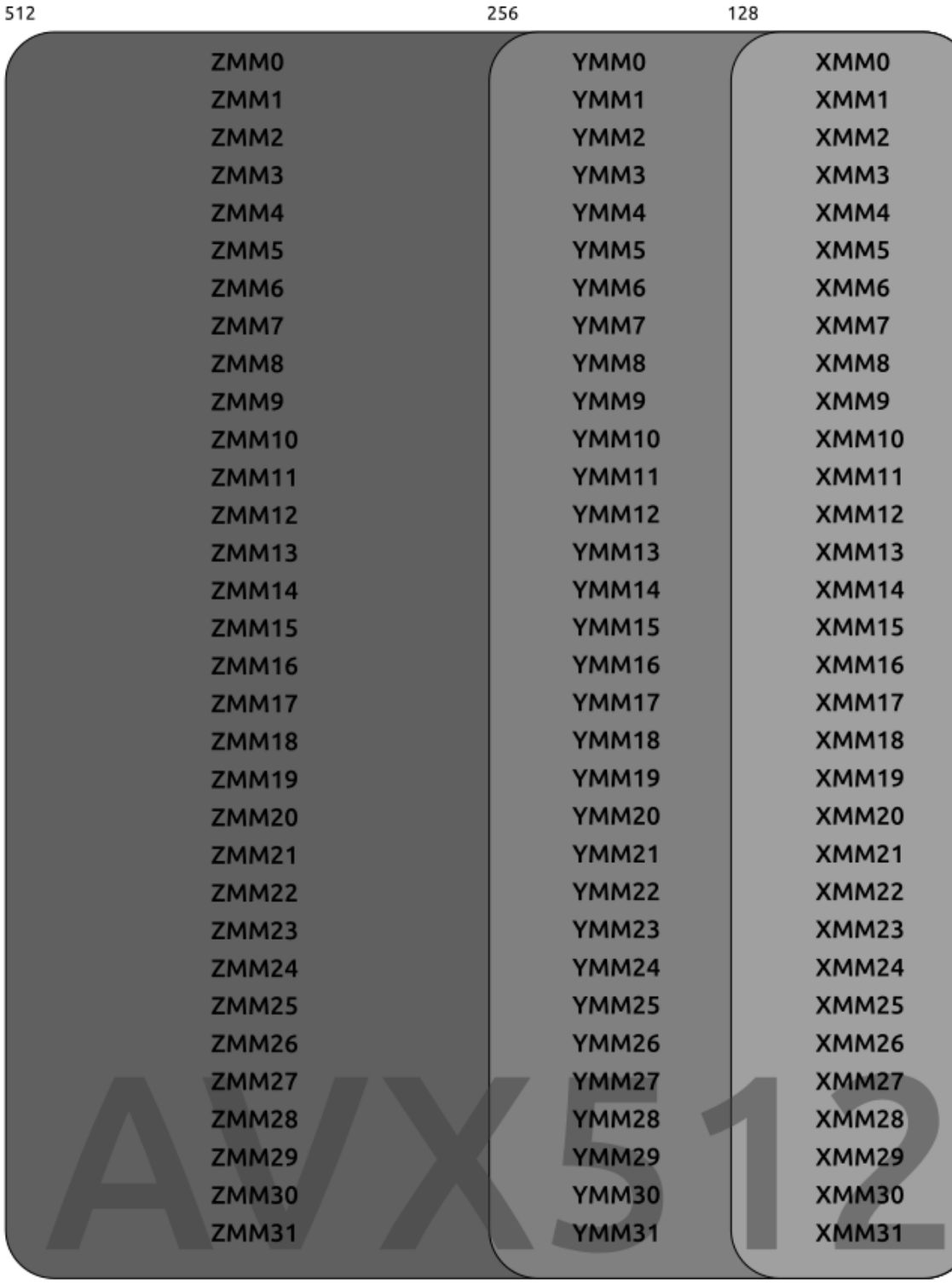


4bit Registermodell
PU, MMX, SSE, AVX)



4bit Registermodell
(CPU, MMX, SSE, AVX)





AMD64-Aufrufkonventionen - Unix (Linux, OSX, Solaris...)

Wenn Parameter Integer oder Zeiger sind...

... hat der **Caller** vor dem Aufruf des Callee:

rax, rdi, rsi, rdx, rcx, r8, r9, r10, r11

... zu sichern und Argumente in:

rdi, rsi, rdx, rcx, r8, r9

... zu übergeben. Das Ergebnis vom Callee holt er in:

rax

... ab.

... wird erwartet, dass der **Callee**:

rbp, rsp, rbx, r12, r13, r14, r15

... sichert. Aber er darf über:

rax, rdi, rsi, rdx, rcx, r8, r9, r10, r11

... frei verfügen und liefert das Ergebnis in:

rax

... ab.

Wenn Parameter Float oder Double sind...

... hat der **Caller** vor dem Aufruf des Callee:

xmm0..xmm7

... zu sichern und Argumente in:

xmm0..xmm7

... zu übergeben. Das Ergebnis vom Callee holt er in:

xmm0

... ab.

... wird erwartet, dass der **Callee**:

xmm8..xmm15

... sichert. Aber er darf über:

xmm0..xmm7

... frei verfügen und liefert das Ergebnis in:

xmm0

... ab.

AMD64-Aufrufkonventionen - Microsoft/Windows

Wenn Parameter Integer oder Zeiger sind...

... hat der **Caller** vor dem Aufruf des Callee:

rax, rcx, rdx, r8, r9, r10, r11

... zu sichern und Argumente in:

rcx, rdx, r8, r9

... zu übergeben. Das Ergebnis vom Callee holt er in:

rax

... ab. Zusätzlich **müssen** 32 Bytes "shadow space"

auf dem Stack alloziert werden!

... wird erwartet, dass der **Callee**:

rbx, rbp, rdi, rsi, rsp, r12, r13, r14, r15

... sichert. Aber er darf über:

rax, rcx, rdx, r8, r9, r10, r11

... frei verfügen und liefert das Ergebnis in:

rax

... ab.

Wenn Parameter Float oder Double sind...

... hat der **Caller** vor dem Aufruf des Callee:

xmm0..xmm3

... zu sichern und Argumente in:

xmm0..xmm3

... zu übergeben. Das Ergebnis vom Callee holt er in:

xmm0

... ab.

... wird erwartet, dass der **Callee**:

xmm4..xmm15

... sichert. Aber er darf über:

xmm0..xmm3

... frei verfügen und liefert das Ergebnis in:

xmm0

... ab.

```
l = 0x2A;
ox = 0xAFFE;
x = 0xDEADBEEF;
dx = 0xCAFE4711DEED0815;

);

;
```

x86 Assembler (intel-Syntax, nasm):

```
mov al, 2Ah
mov bx, 0AFFEh
mov ecx, 0DEADBEEFh
mov rdx, 0CAFE4711DEED0815h

call func

ret
```

x86 Assembler (AT&T-Syntax, ga

```
movb $0x2A, %al
movw $0xAFFE, %bx
movl $0xDEADBEEF, %ecx
movq $0xCAFE4711DEED081

callq func

retq
```

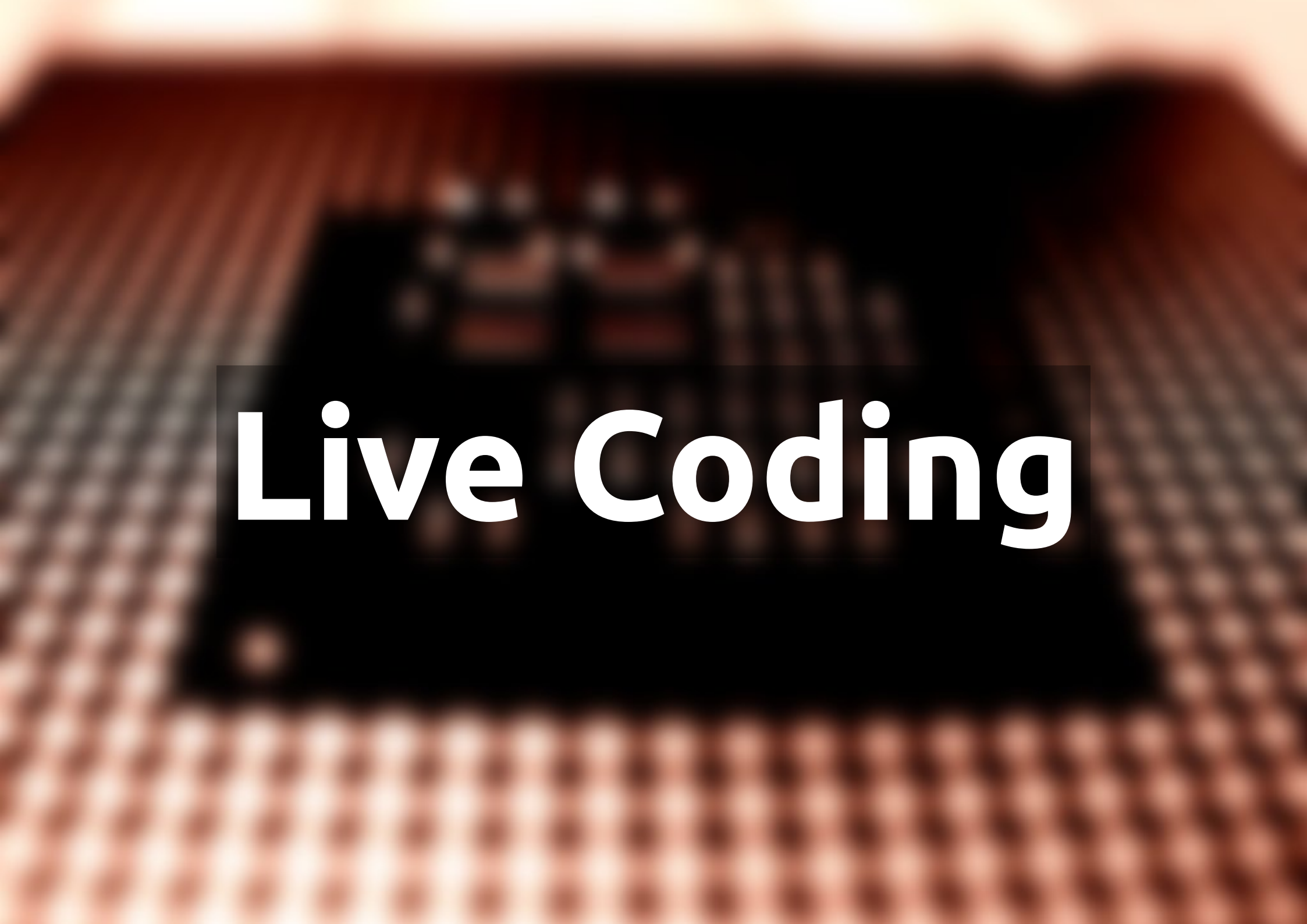
```
ed sum;
nsigned i = 0; i <= 42; ++i) {
m += i;
```

x86 Assembler (intel-Syntax, nasm):

```
sum: resd 1
xor eax, eax
mov ecx, 02Ah
myForLoop:
    add eax, ecx
loop myForLoop
mov [sum], eax
```

x86 Assembler (AT&T-Synt

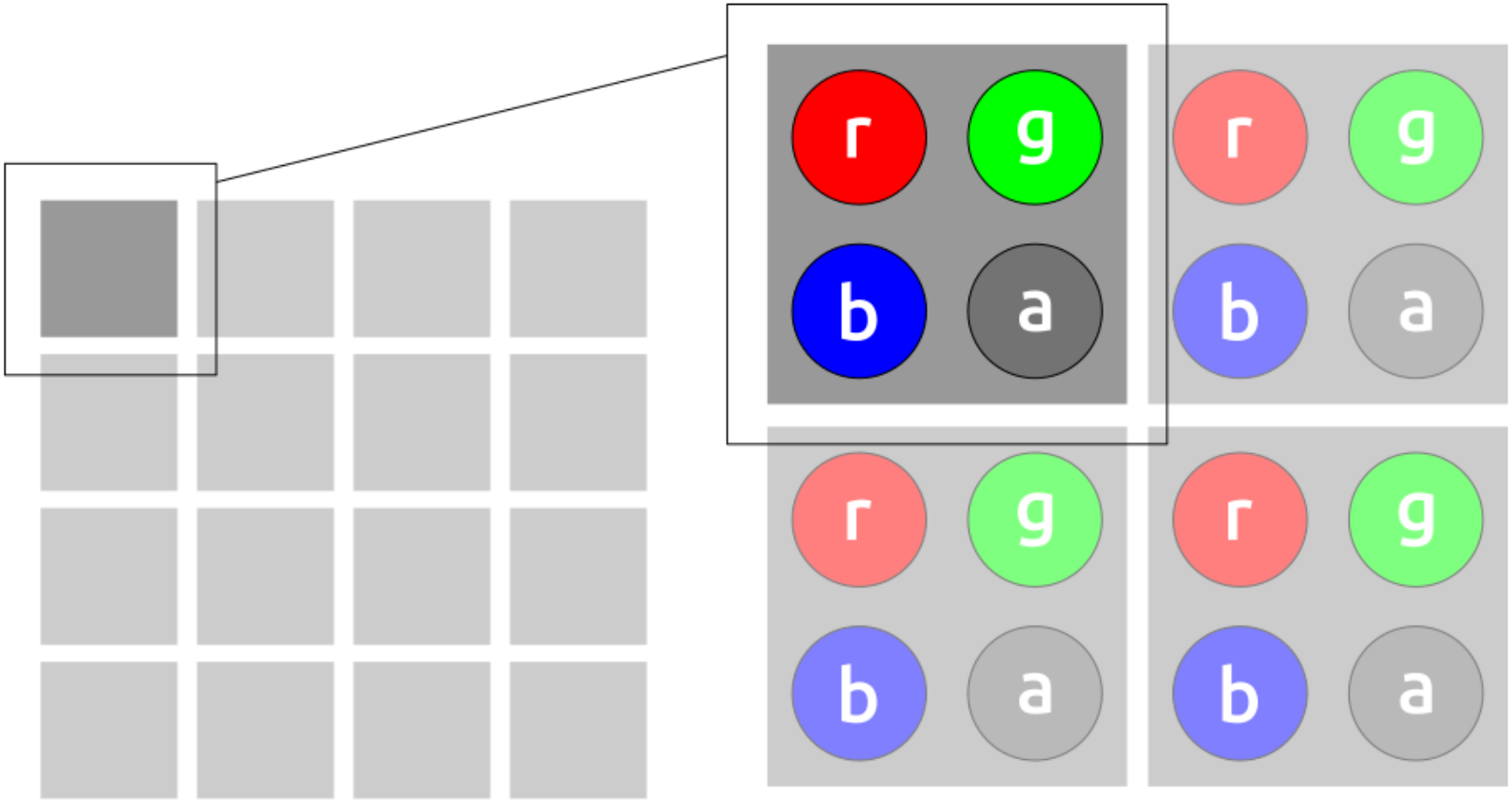
```
sum: .size sum, 4
xorl %eax, %eax
movl $0x2A, %ecx
myForLoop:
    addl %ecx, %eax
loop MyForLoop
movl %eax, sum
```


Live Coding

Beispiele

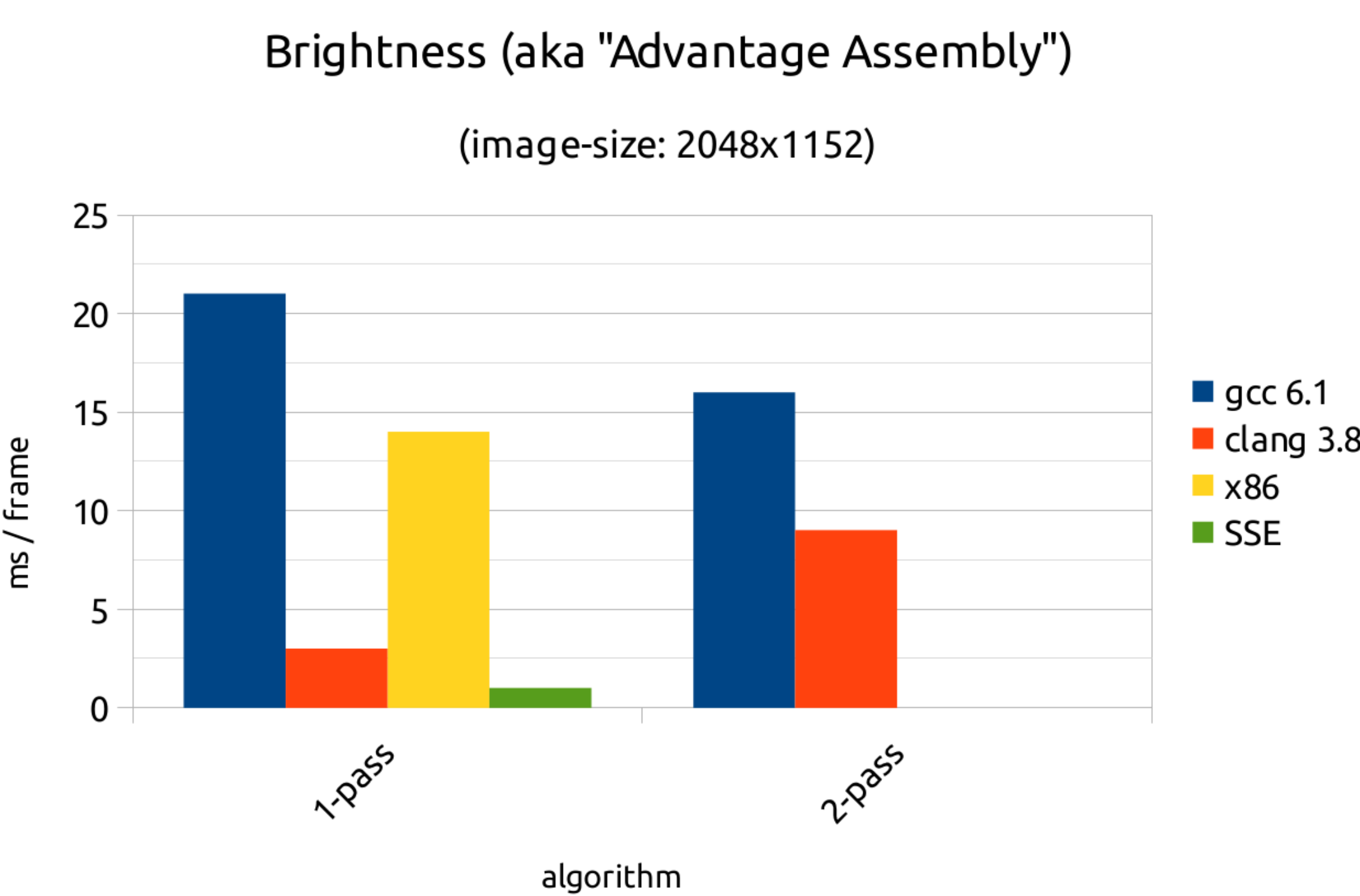
- brightness
- box-blur
- reaction-diffusion
- Laufzeit Vergleich/Diagram
 * brightness
   * Algoritmus erkläre
   * naive, 2-way, simple x86 asm, sse3, mt sse3
 * box-blur
   * Algoritmus erkläre (1-pass, 2-pass, sliding-window trick)
   * naive, 2-way, avx
 * reaction-diffusion naive, sse3
   * gleichung, Algorighmus erklären
   * naive, SSE für Gradient-Matrix

Brightness



$$P\,(\mathrm{r}, \mathrm{g}, \mathrm{b}, \mathrm{a}) \Rightarrow P'\,(\mathrm{r+v}, \mathrm{g+v}, \mathrm{b+v}, \mathrm{a+v})$$


Brightness Code

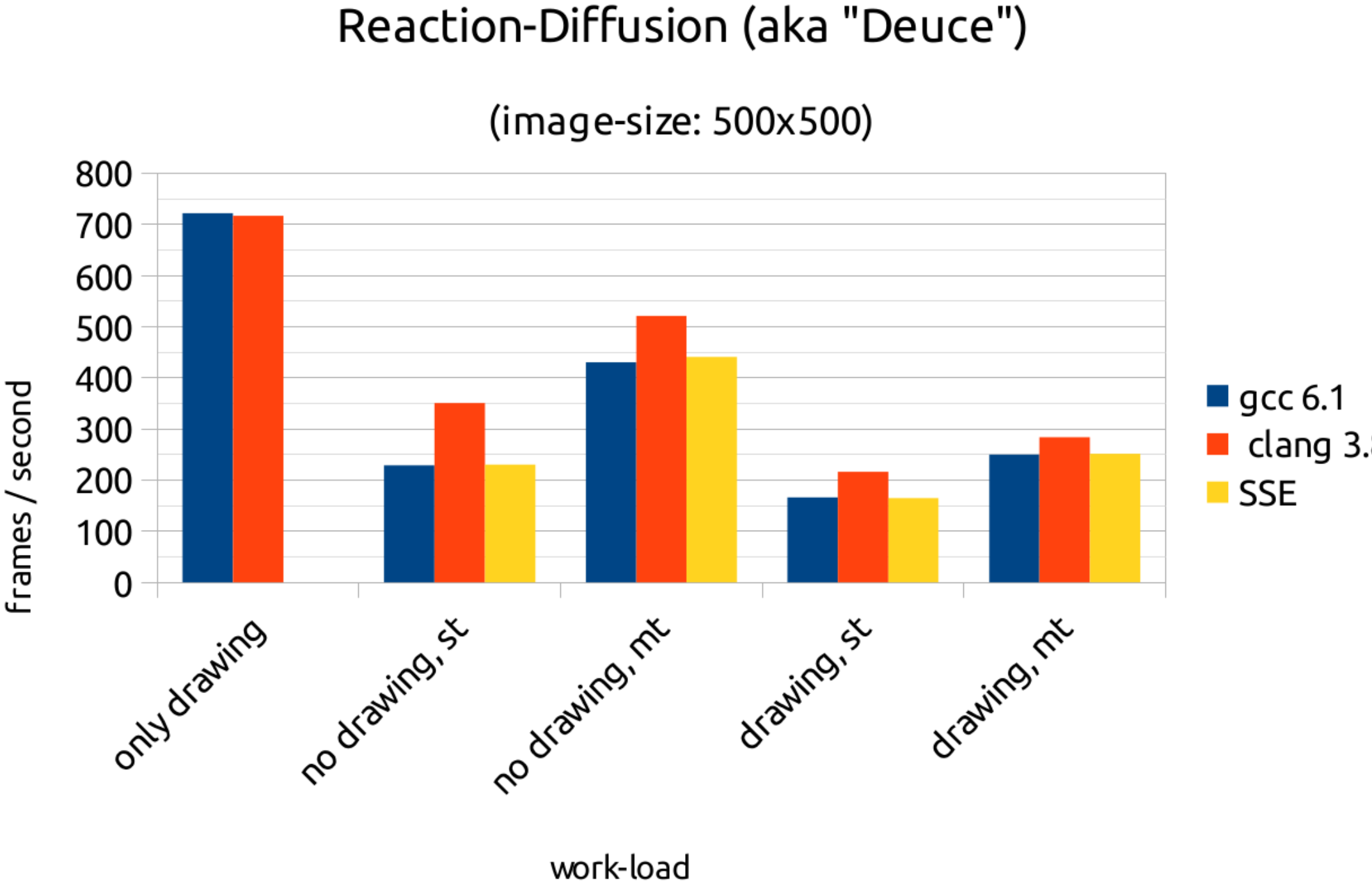

Brightness (aka "Advantage Assembly")
(image-size: 2048x1152)
25
20
15
10
5
0
ms / frame
1-pass
2-pass
algorithm
gcc 6.1
clang 3.8
x86
SSE

Reaction-Diffusion nach Gray-Scott Modell

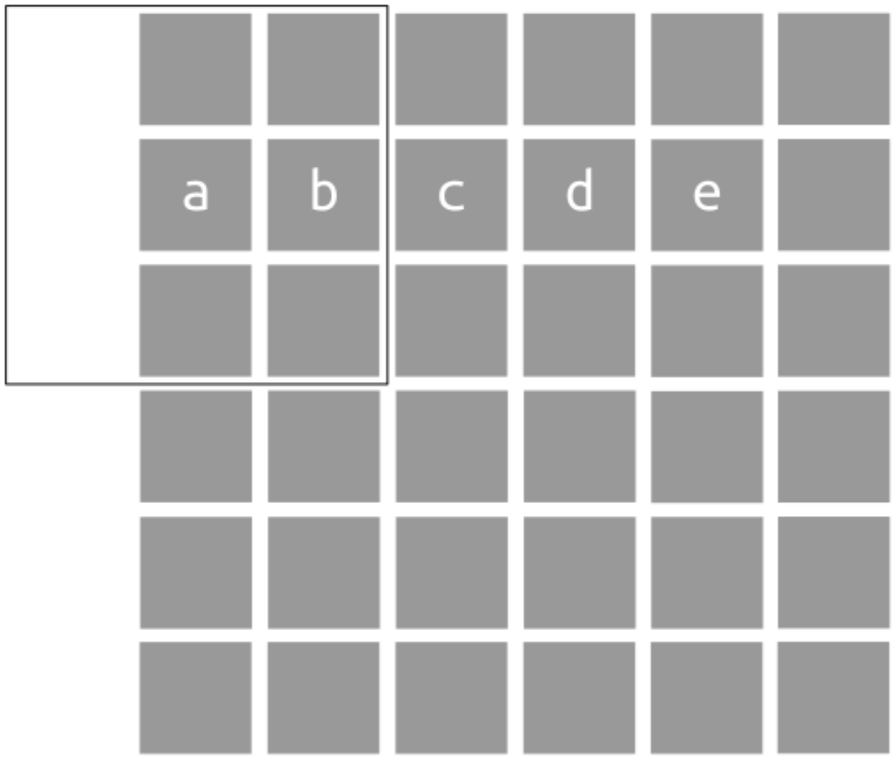
$$a' = a\left(r_a \nabla^2 a - ab^2 + f(1-a)\right)\Delta t$$

$$b' = b\left(r_b \nabla^2 b + ab^2 + (f+k)b\right)\Delta t$$

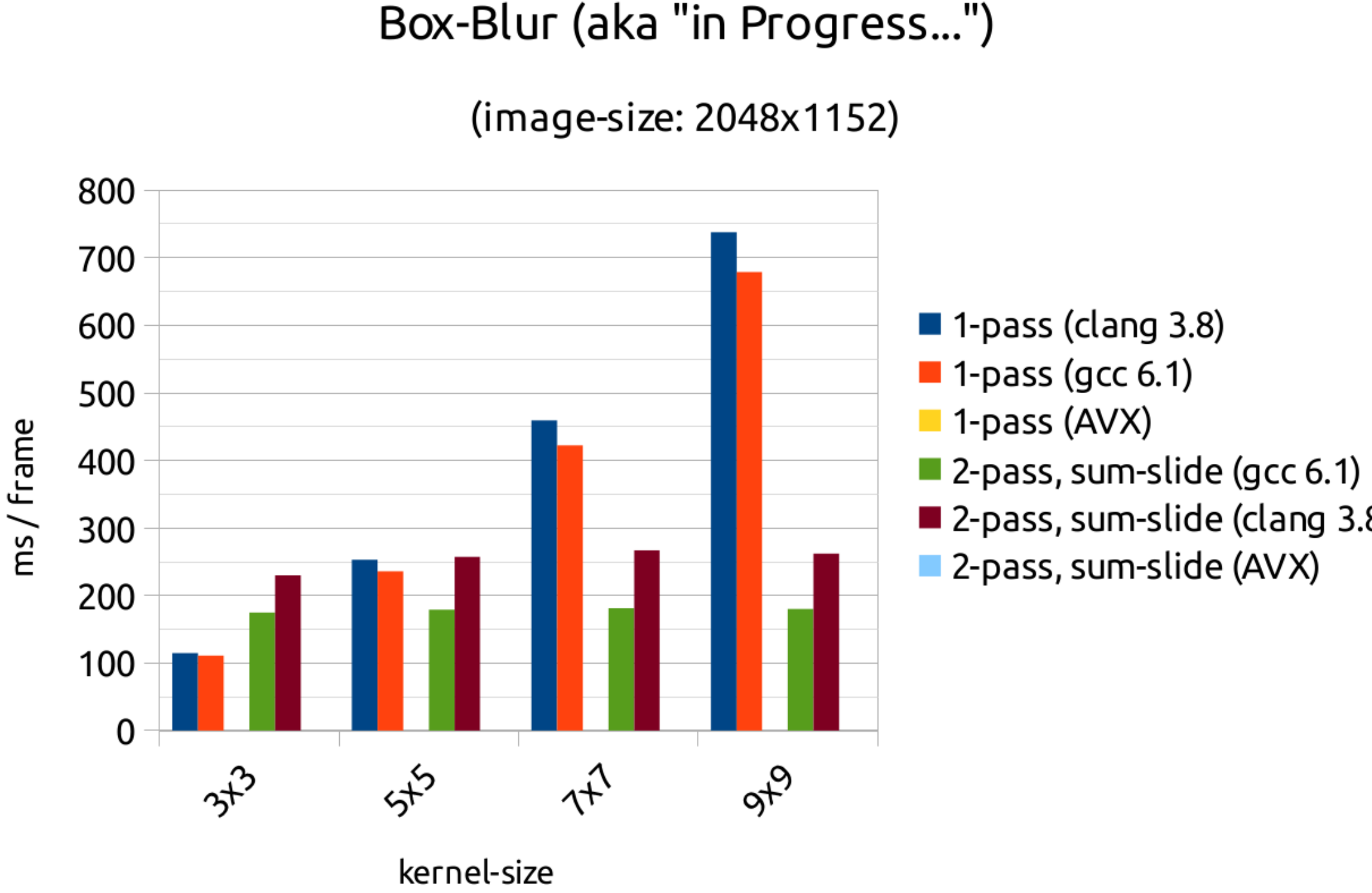
$$\nabla^2 a: \begin{pmatrix} 0.05 & 0.2 & 0.05 \\ 0.2 & -1.0 & 0.2 \\ 0.05 & 0.2 & 0.05 \end{pmatrix} \begin{pmatrix} a_{i-1,j-1} & a_{i,j-1} & a_{i+1,j-1} \\ a_{i-1,j} & a_{i,j} & a_{i+1,j} \\ a_{i-1,j+1} & a_{i,j+1} & a_{i+1,j+1} \end{pmatrix}$$


Reaction-Diffusion (aka "Deuce")
(image-size: 500x500)
800
700
600
500
400
300
200
100
0
frames / second
only drawing
no drawing, st
no drawing, mt
drawing, st
drawing, mt
work-load
gcc 6.1
clang 3.
SSE

Box-Blur

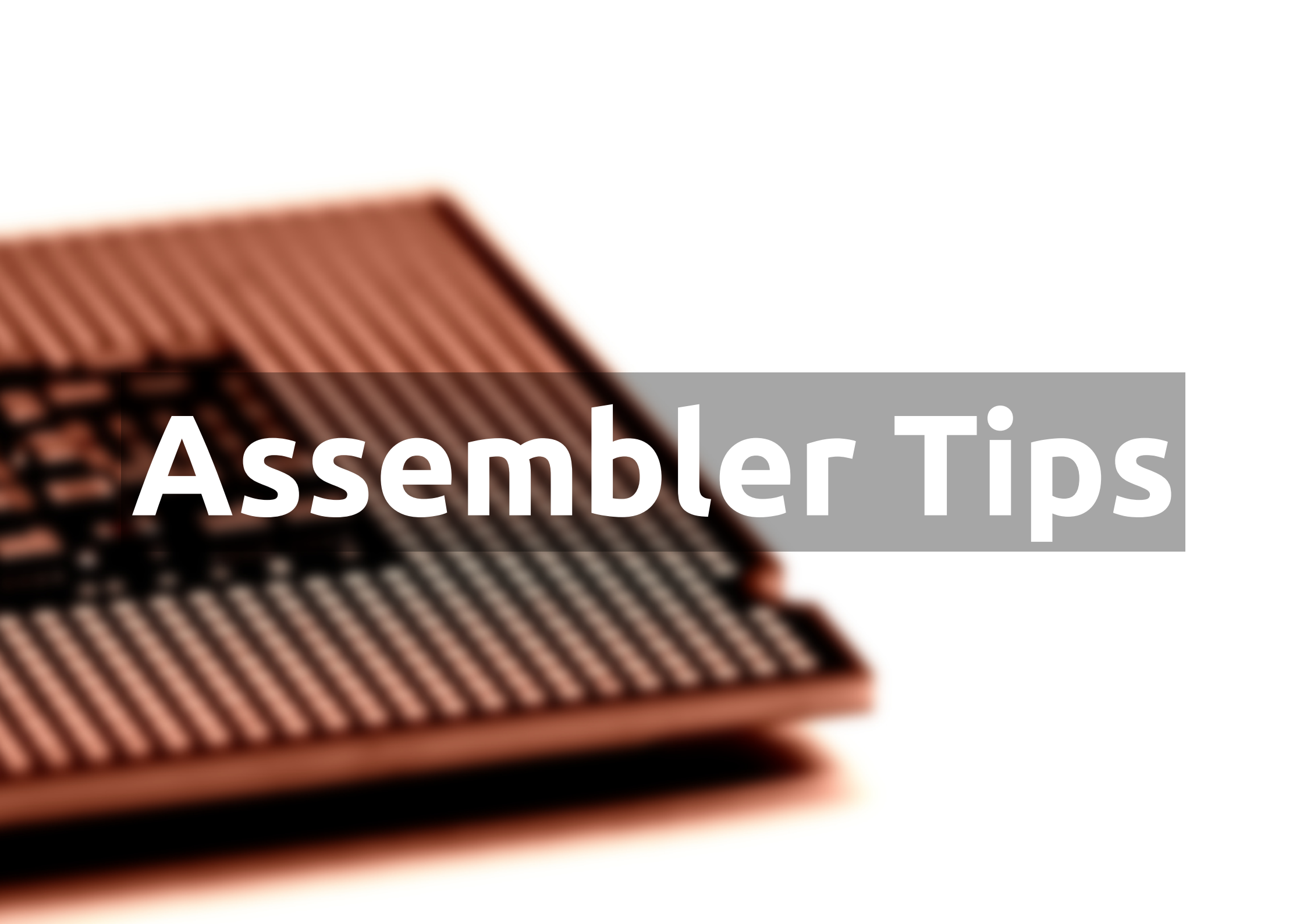



```
   a b c d e
(0+a+b)/3
  (a+b+c)/3     = (0+a+b-0+c)/3 = (0+a+b)/3+(c-0)/3
    (b+c+d)/3   = (a+b+c-a+d)/3 = (a+b+c)/3+(d-a)/3
      (c+d+e)/3 = (b+c+d-b+e)/3 = (b+c+d)/3+(e-b)/3
```


Box-Blur (aka "in Progress...")
(image-size: 2048x1152)
800
700
600
500
400
300
200
100
0
ms / frame
3x3
5x5
7x7
9x9
kernel-size
1-pass (clang 3.8)
1-pass (gcc 6.1)
1-pass (AVX)
2-pass, sum-slide (gcc 6.1)
2-pass, sum-slide (clang 3.8
2-pass, sum-slide (AVX)

Fazit (vorläufig)

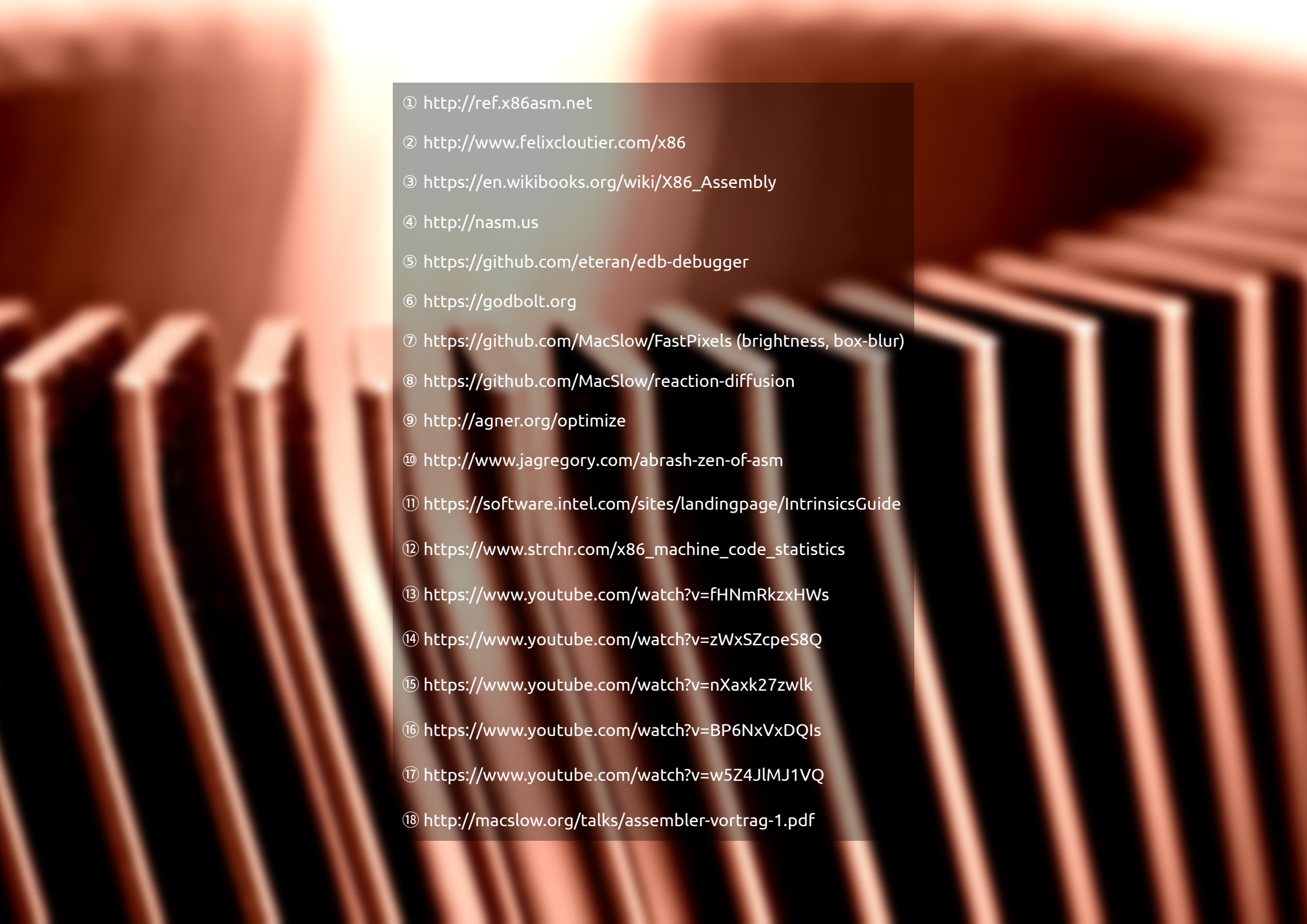
aktuelle Compiler sind wirklich gut
clang < gcc
man kann den u.U. Compiler schlagen
manuelle Anpassungen sind nötig: ILP, SIMD (SSE, AVX usw), cache-Bewusstsein
Gewinn intimer Hardwarekenntnisse
es ist Arbeit, aber nicht schwierig
absolute Kontrolle


Assembler Tips

div/idiv vermeiden (laaaaahm) und stattdessen mit Kehrwert multiplizieren
shr/sar & shl nutzen wenn möglich... für Integer-Div/Mul zur Basis 2
Sprünge vermeiden und lieber cmovcc, setcc etc nutzen
lea als billig-fma verwenden wenn möglich
die cvt**** vermeiden wenn möglich (langsam)
"dec <op> jnz label" ist schneller als "loop"
"xor <op>, <op>" ist schneller als "mov <op>, 0"
jl & jg sind vorzeichenbehaftet, jb & ja sind NICHT vorzeichenbehaftet
"mov eax, eax" nullt die oberen 32 Bits von rax (gilt für alle 64 Bit Regs)
jmp/jcc packen die Rücksprungadresse NICHT auf den Stack, call macht das schon
Berechnen ist meist schneller als (aus dem Speicher) holen
"weniger Stack ist besser als mehr Stack"
wenn aus dem Speichen/Heap lesen dann bitte aligned (unaligned ist sehr lahm, bei älteren CPUs)
gcc -O2 -Q --help=optimizers
alias opt=opt-3.8 ; llvm-as < /dev/null | opt -O2 -disable-output -debug-pass=Arguments


Gibt's Fragen?


Besten Dank für
Eure Aufmerksamkeit!



① http://ref.x86asm.net

② http://www.felixcloutier.com/x86

③ https://en.wikibooks.org/wiki/X86_Assembly

④ http://nasm.us

⑤ https://github.com/eteran/edb-debugger

⑥ https://godbolt.org

⑦ https://github.com/MacSlow/FastPixels (brightness, box-blur)

⑧ https://github.com/MacSlow/reaction-diffusion

⑨ http://agner.org/optimize

⑩ http://www.jagregory.com/abrash-zen-of-asm

⑪ https://software.intel.com/sites/landingpage/IntrinsicsGuide

⑫ https://www.strchr.com/x86_machine_code_statistics

⑬ https://www.youtube.com/watch?v=fHNmRkzxHWs

⑭ https://www.youtube.com/watch?v=zWxSZcpeS8Q

⑮ https://www.youtube.com/watch?v=nXaxk27zwlk

⑯ https://www.youtube.com/watch?v=BP6NxVxDQIs

⑰ https://www.youtube.com/watch?v=w5Z4JlMJ1VQ

⑱ http://macslow.org/talks/assembler-vortrag-1.pdf