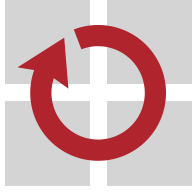
Übung zu Betriebssysteme

Threadumschaltung

7. & 10. Dezember 2017

Andreas Ziegler
Bernhard Heinloth

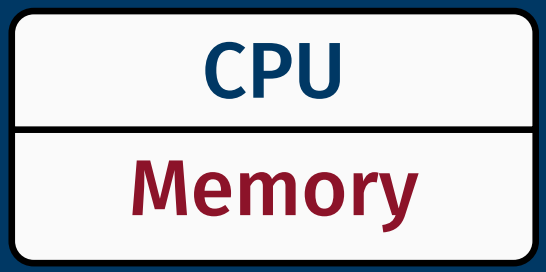
Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg



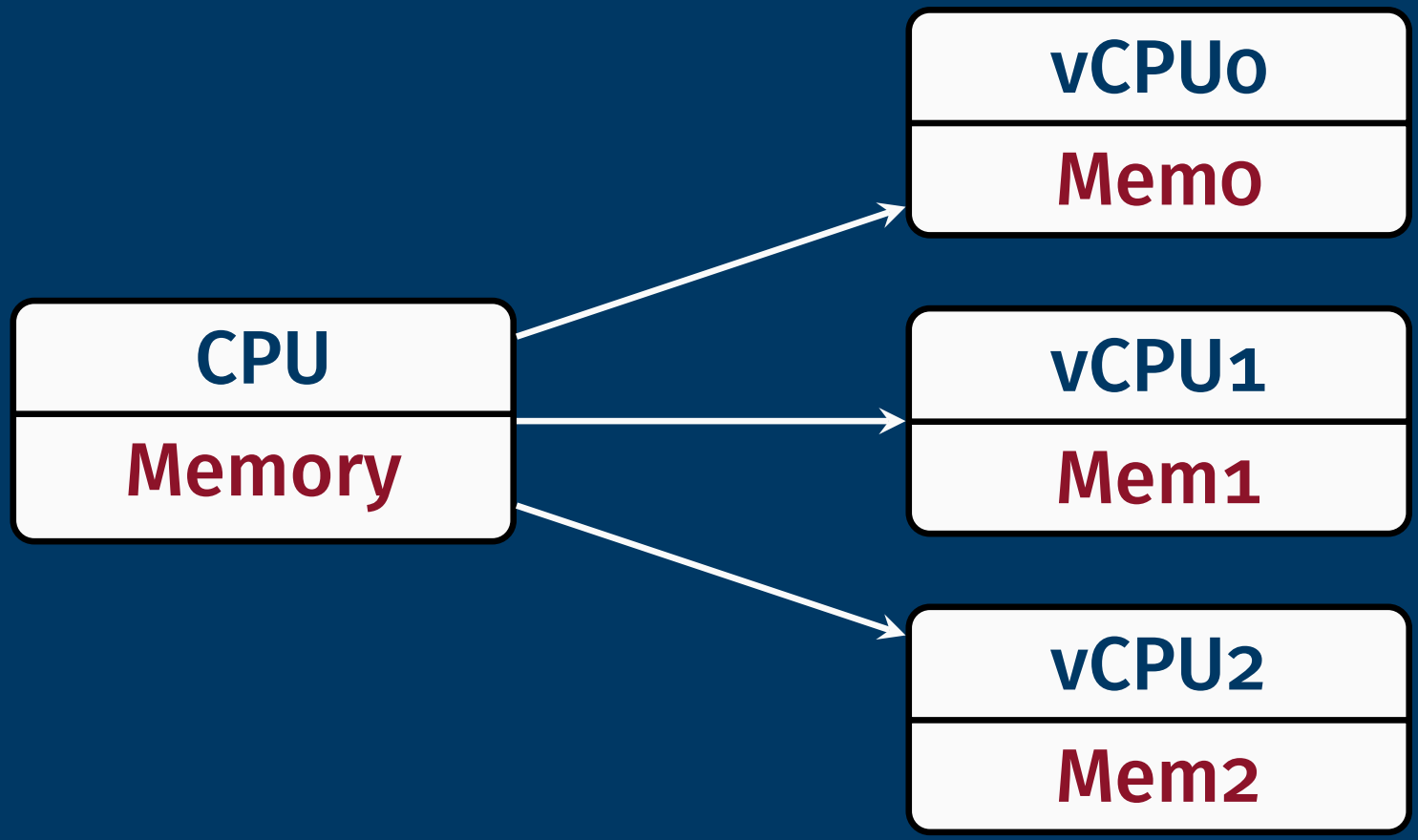
Lehrstuhl für Verteilte Systeme
und Betriebssysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT



tatsächliche
Hardware



tatsächliche Hardware → illusionierte Hardware

- Speicher virtualisieren
- CPU virtualiseren

- Speicher virtualisieren ist einfach:

```
#define MEMSIZE 100
char Mem0[MEMSIZE];
char Mem1[MEMSIZE];
char Mem2[MEMSIZE];
```

- CPU virtualiseren

- Speicher virtualisieren ist einfach:

```
#define MEMSIZE 100
char Mem0[MEMSIZE];
char Mem1[MEMSIZE];
char Mem2[MEMSIZE];
```

- CPU virtualiseren
  - nicht teilbar im Ort

- Speicher virtualisieren ist einfach:

```
#define MEMSIZE 100
char Mem0[MEMSIZE];
char Mem1[MEMSIZE];
char Mem2[MEMSIZE];
```

- CPU virtualiseren
  - nicht teilbar im Ort
  - aber teilbar in der Zeit

Koroutinen

Motivation: mehr Aktivitätsträger als CPUs

```
void foo(){
  int f = 42;

  while (f--){
    kout << "foo"
         << f
         << endl;

  }

}
```

```
void bar(){
  int b = 23;

  while (b--){
    kout << "bar"
         << b
         << endl;

  }

}
```

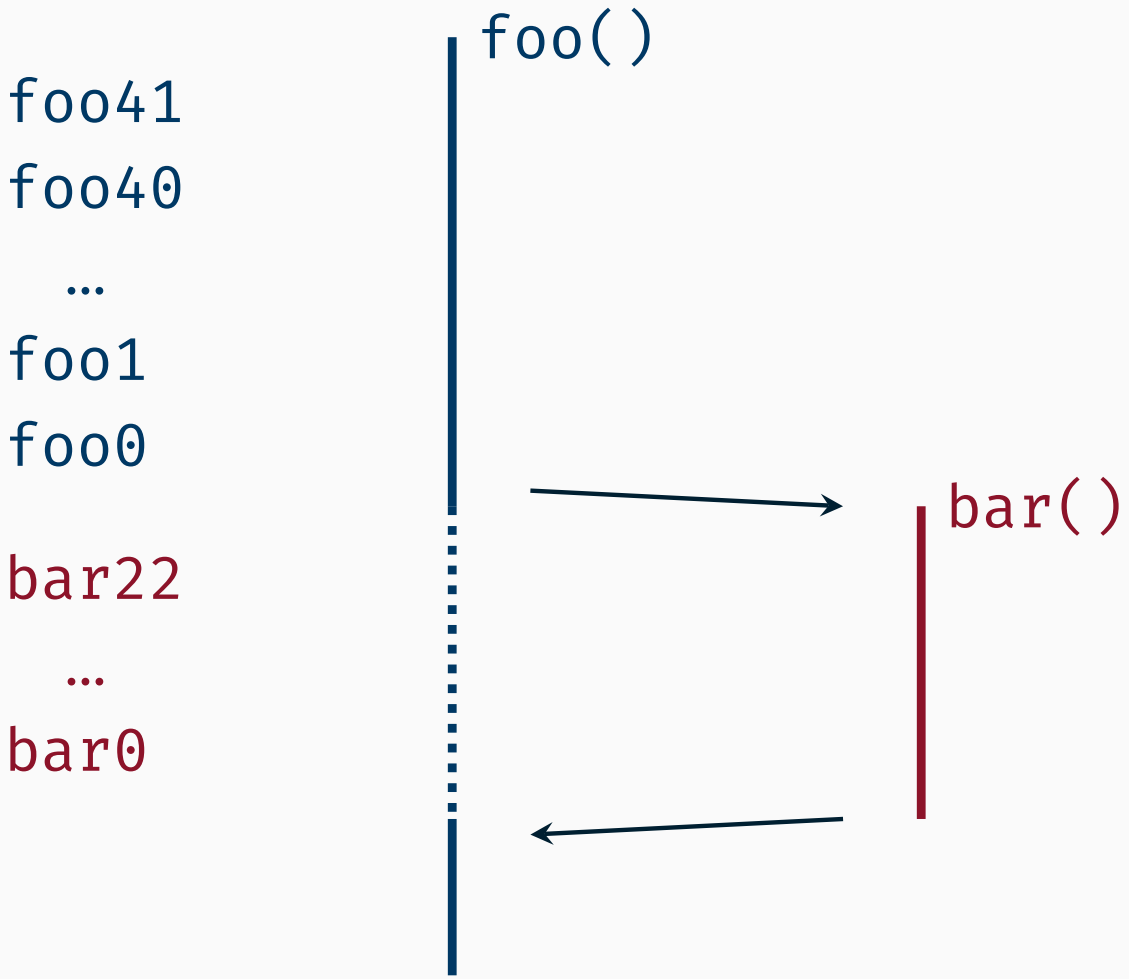
Einseitiger Aufruf

Einseitiger Aufruf

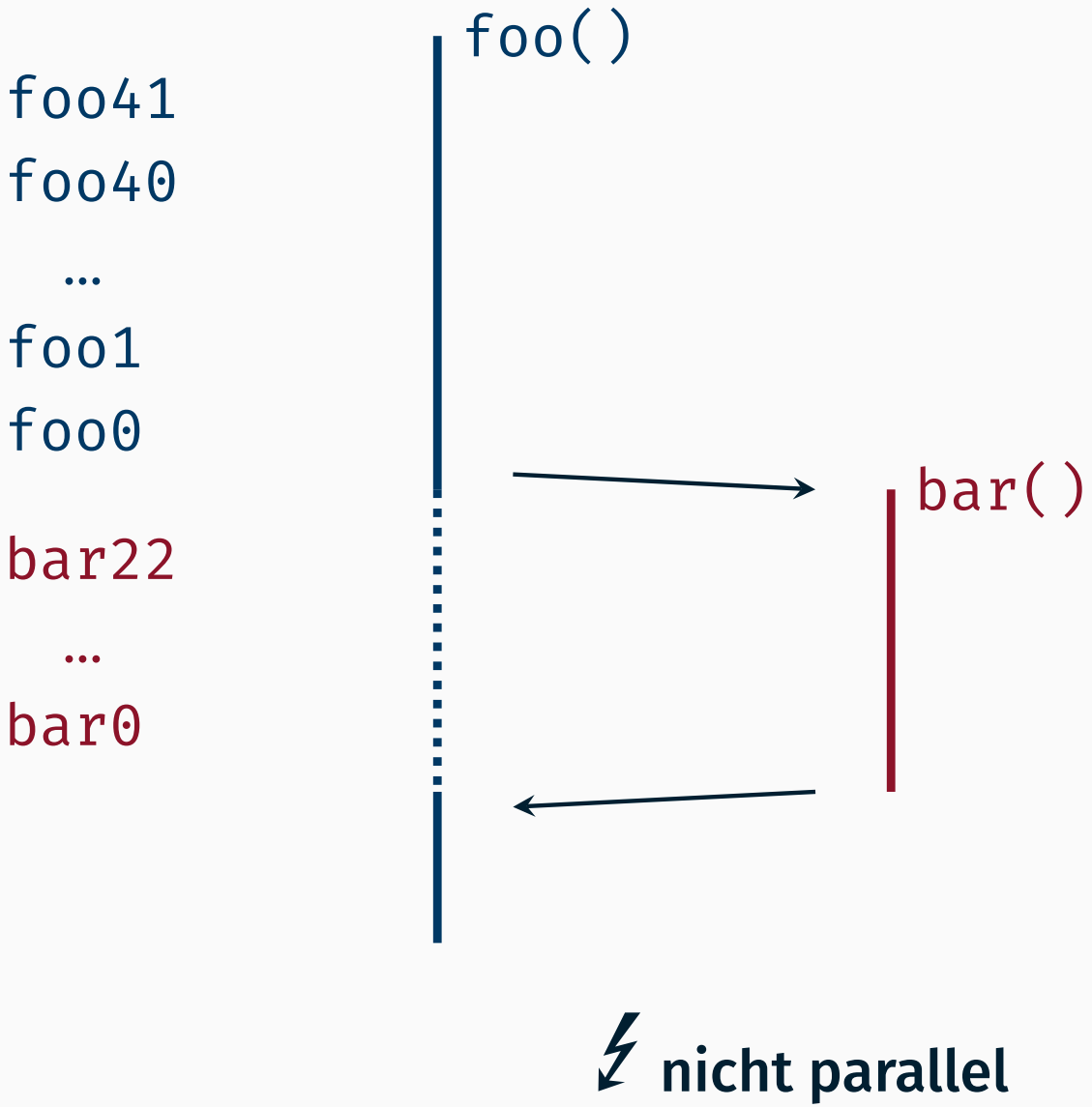
```
void foo(){
  int f = 42;

  while (f--){
    kout << "foo"
         << f
         << endl;

  }
  bar();
}
```

```
void bar(){
  int b = 23;

  while (b--){
    kout << "bar"
         << b
         << endl;

  }

}
```

Einseitiger Aufruf



Einseitiger Aufruf



nicht parallel

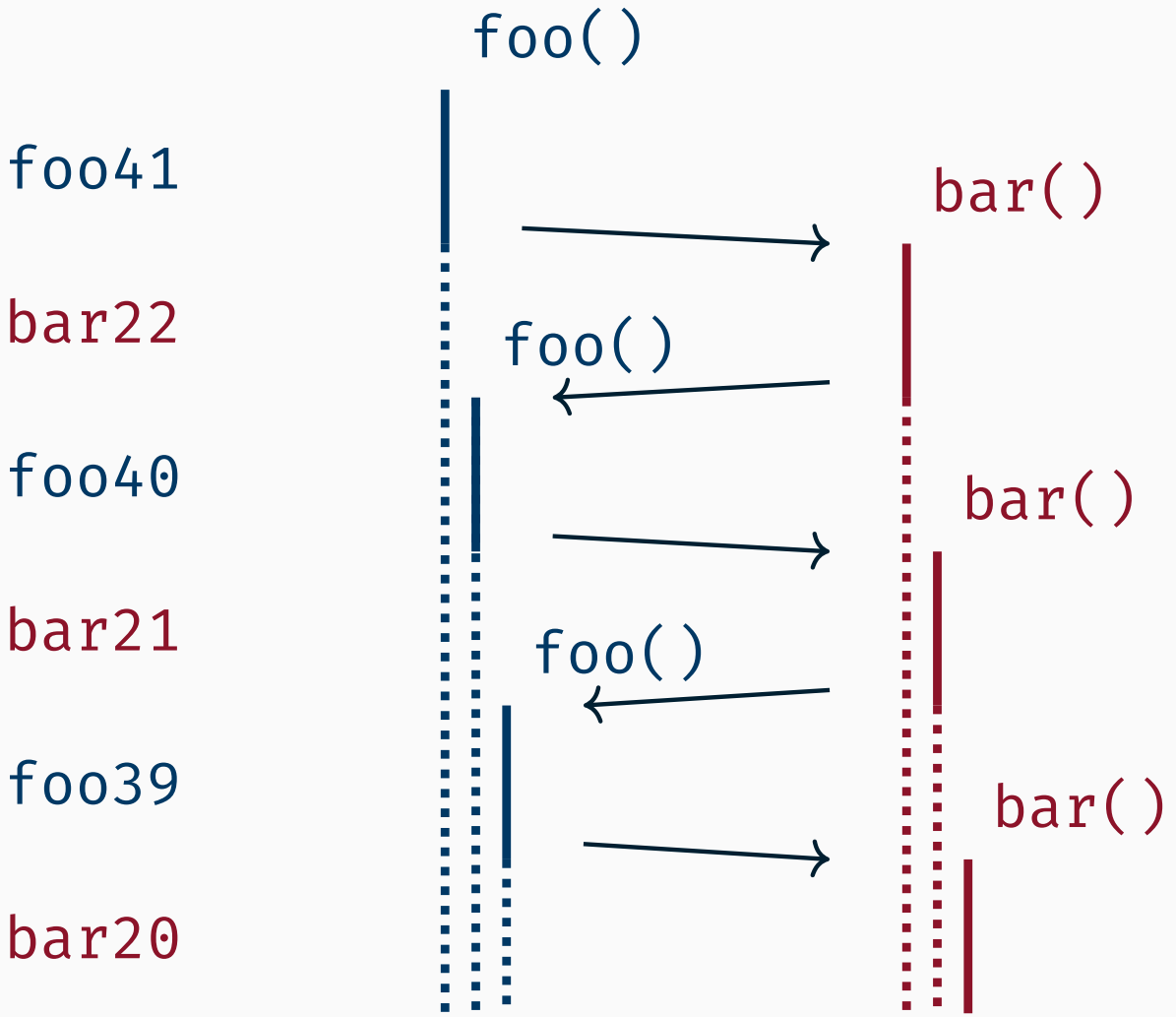
Gegenseitiger Aufruf

Gegenseitiger Aufruf

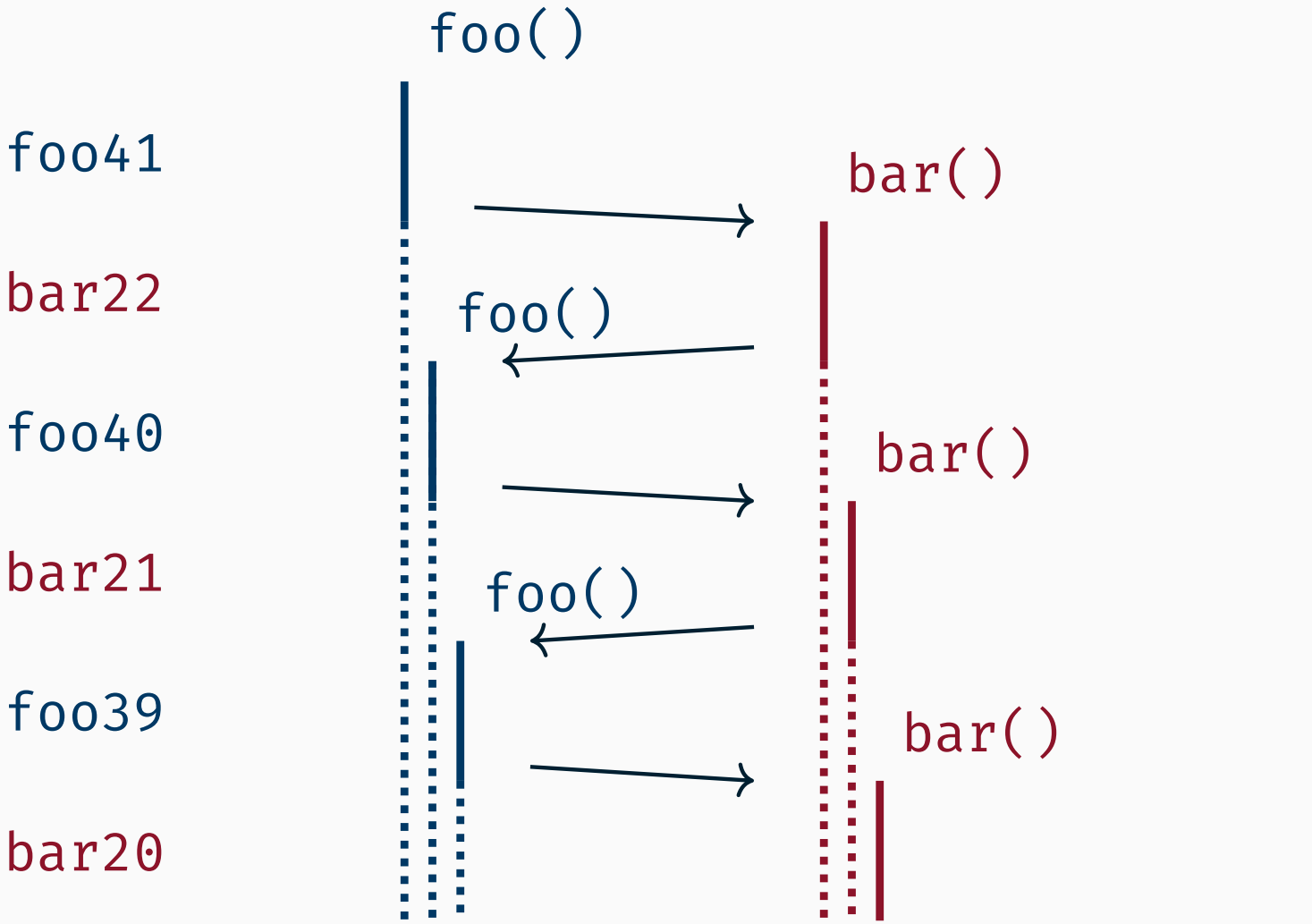
```
void foo(){
  static int f = 42;

  while (f--){
    kout << "foo"
         << f
         << endl;
    bar();
  }

}
```

```
void bar(){
  static int b = 23;

  while (b--){
    kout << "bar"
         << b
         << endl;
    foo();
  }

}
```

Gegenseitiger Aufruf

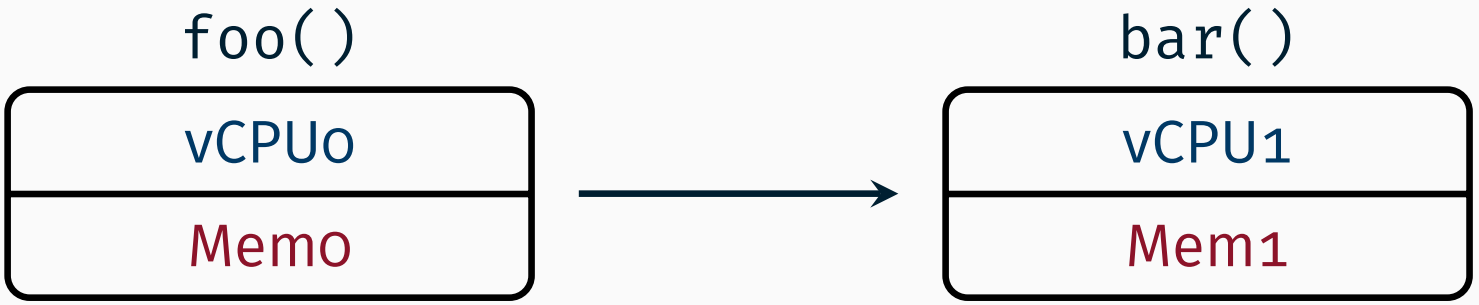


Gegenseitiger Aufruf

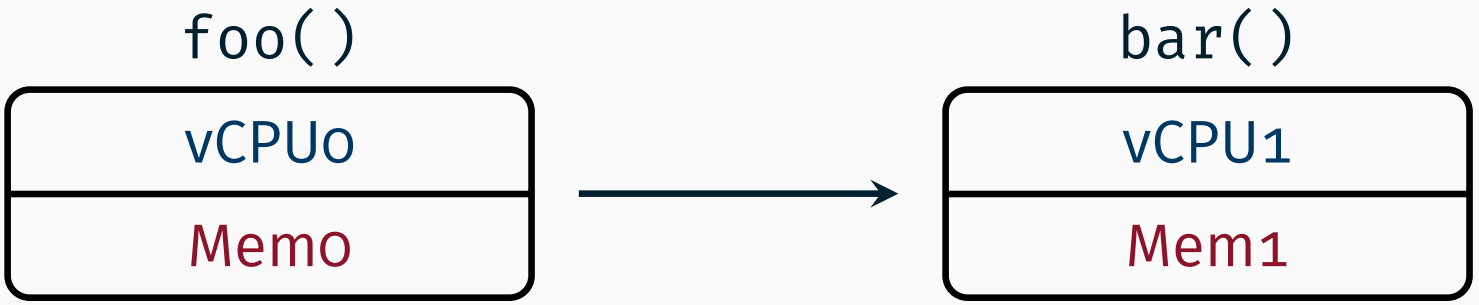


⚡ **hoher Speicherverbrauch & Endlosrekursion**

Umschaltung

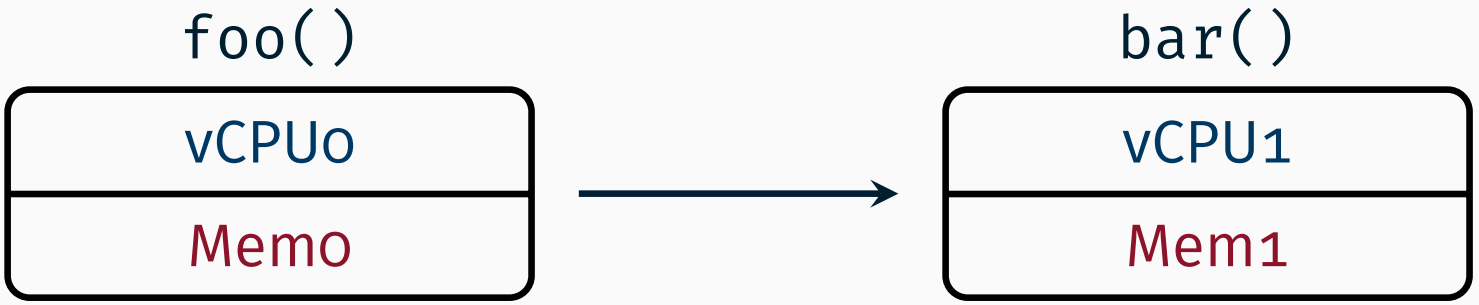


Umschaltung



Kontrollflusszustand sichern & laden

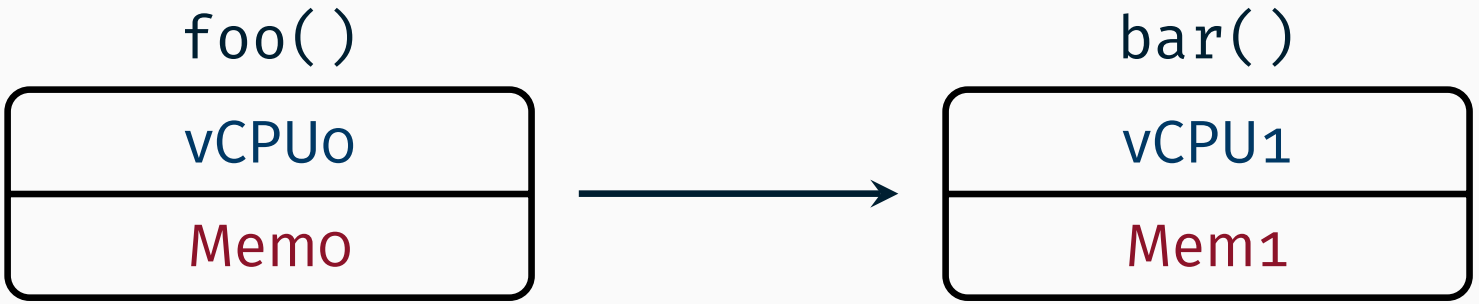
Umschaltung



Kontrollflusszustand sichern & laden

- Stack
- Register

Umschaltung

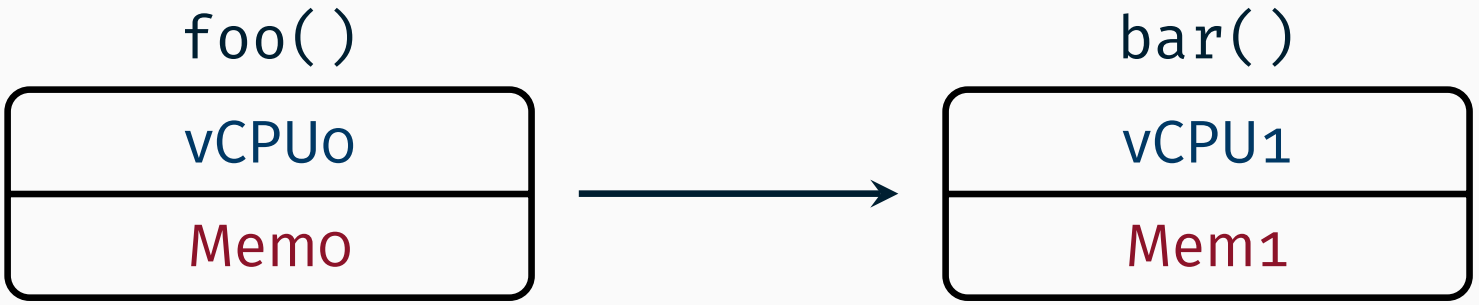


Kontrollflusszustand sichern & laden

- Stack
- Register

} `toc` (Thread of control)

Umschaltung



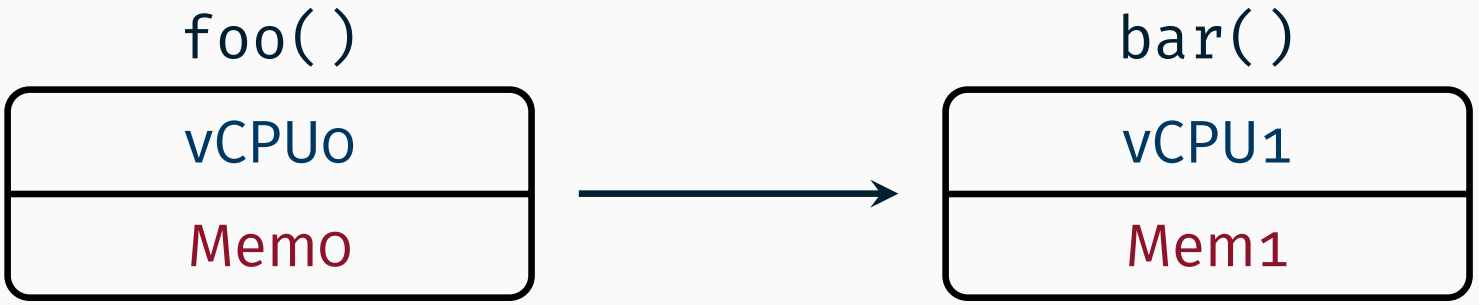
Kontrollflusszustand sichern & laden

- Stack
- Register

} `toc` (Thread of control)

- Umschaltefunktion

Umschaltung



Kontrollflusszustand sichern & laden

- Stack
- Register

} `toc` (Thread of control)

- Umschaltefunktion

```
toc_switch(toc *now, toc *then);
```

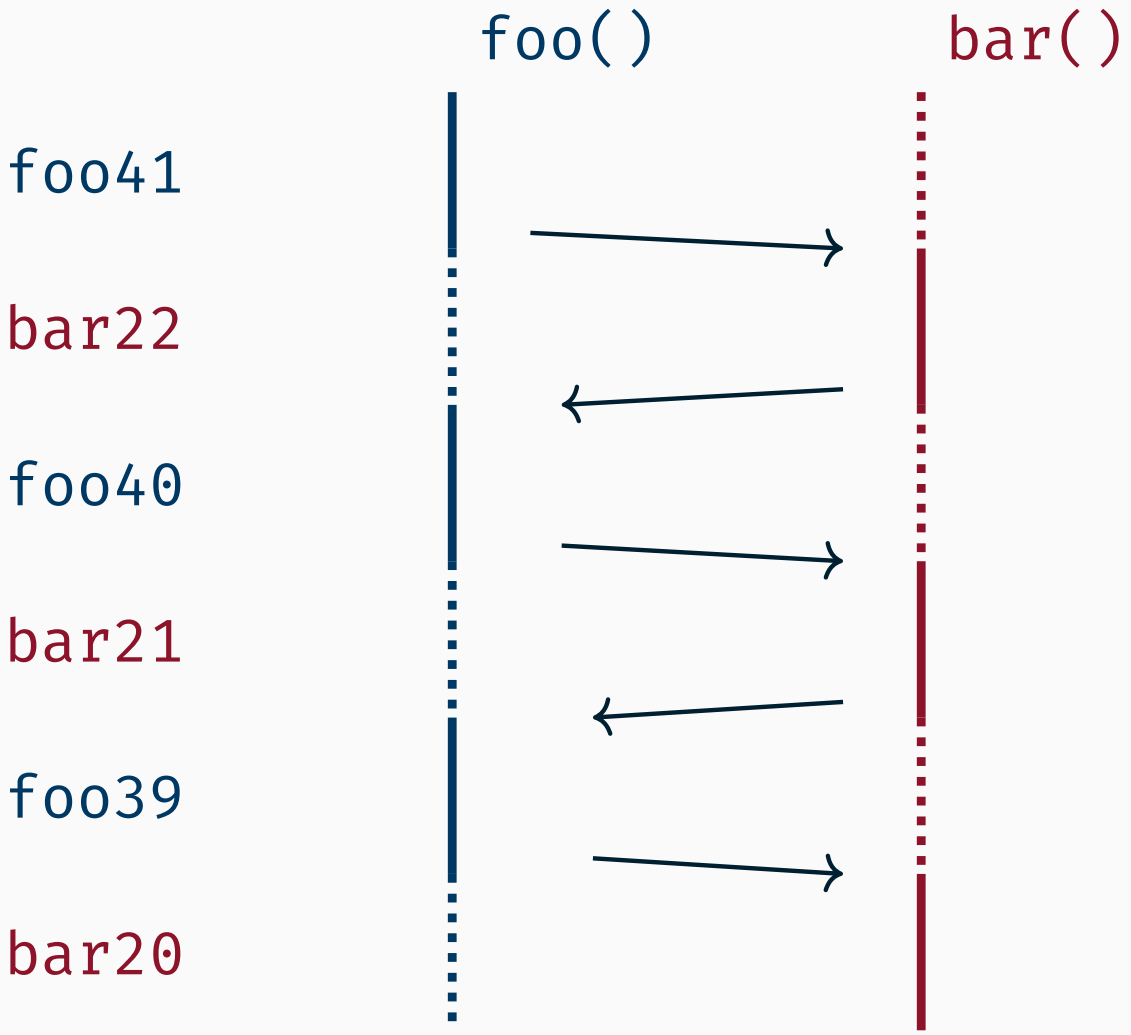
Wechselt vom aktuellen Kontext `now` nach `then`

Umschaltung

```
void foo(){
  int f = 42;

  while (f--){
    kout << "foo"
         << f
         << endl;
    toc_switch(
        &toc_foo,
        &toc_bar
      );
  }

}
```

```
void bar(){
  int b = 23;

  while (b--){
    kout << "bar"
         << b
         << endl;
    toc_switch(
        &toc_bar,
        &toc_foo
      );
  }

}
```

Umschaltung



Kontextsicherung

Kontextsicherung durch den Übersetzer

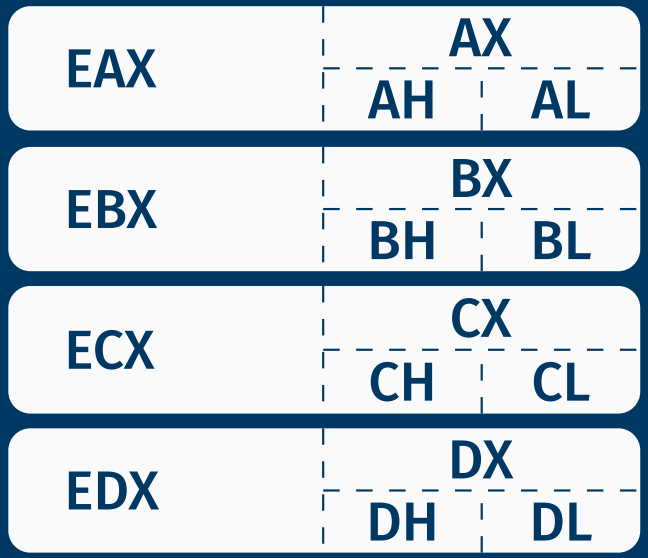
Kontextsicherung durch den Übersetzer

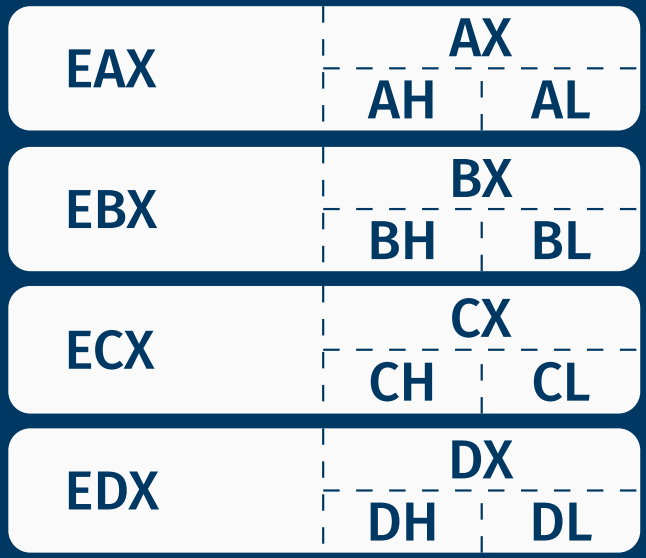
```
void foo(){
  ...

  // flüchtige Register
  // sichern

  bar();

  // flüchtige Register
  // wiederherstellen

  ...
}
```

Kontextsicherung durch den Übersetzer

```
void foo(){
  ...

  // flüchtige Register
  // sichern

  bar();

  // flüchtige Register
  // wiederherstellen

  ...
}
```

```
void bar(){
  // nicht-flüchtige
  // Register
  // sichern

  ...

  // nicht-flüchtige
  // Register
  // wiederherstellen

  return;
}
```


EAX
AX
AH
AL
EBX
BX
BH
BL
ECX
CX
CH
CL
EDX
DX
DH
DL

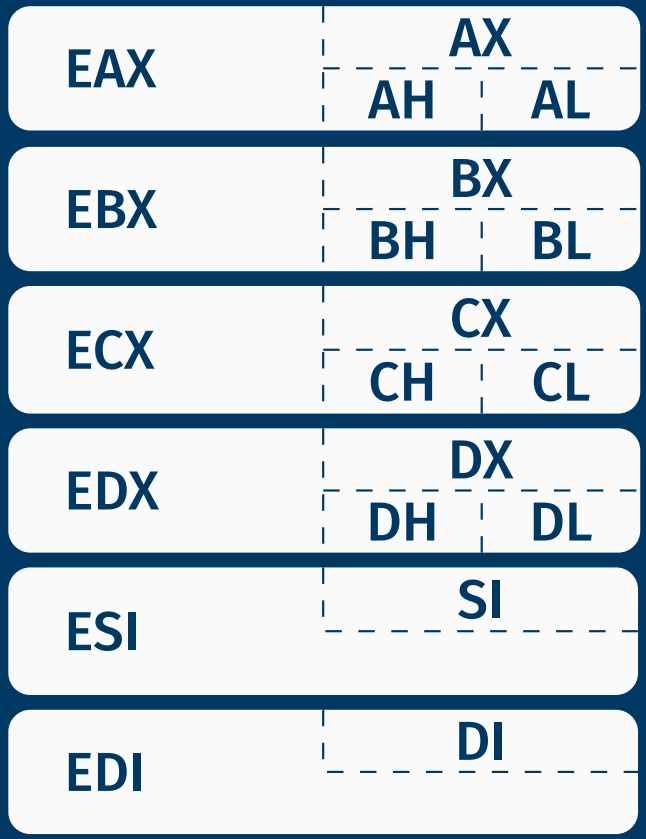


ursprünglich Akkumulatorregister

ursprünglich Basisregister

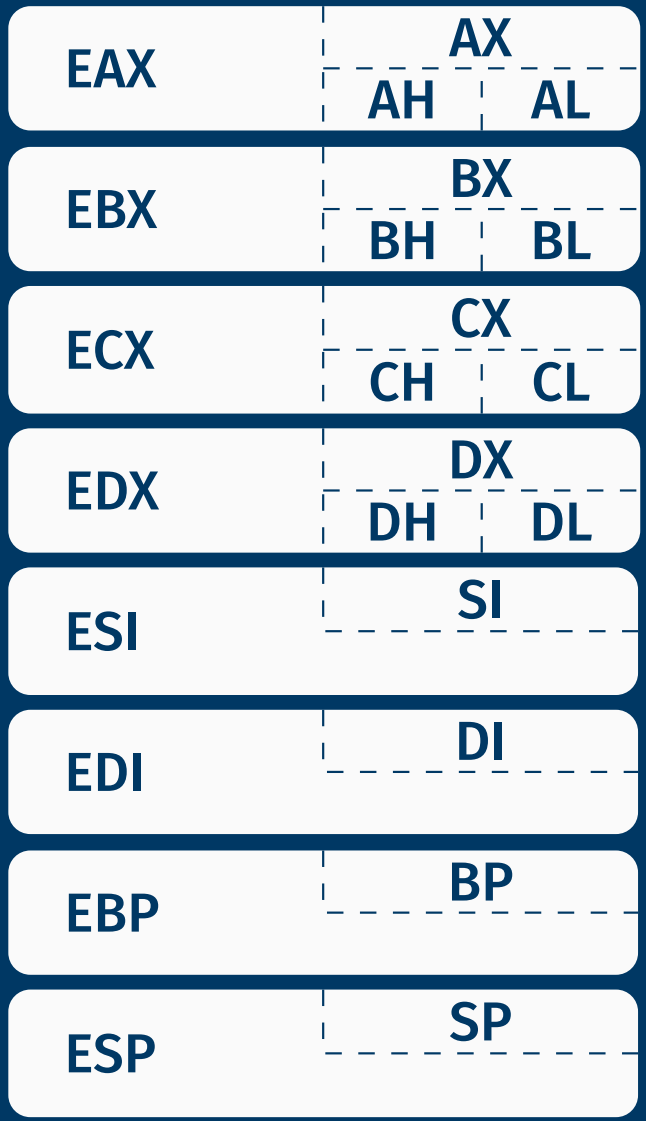
ursprünglich Zählerregister

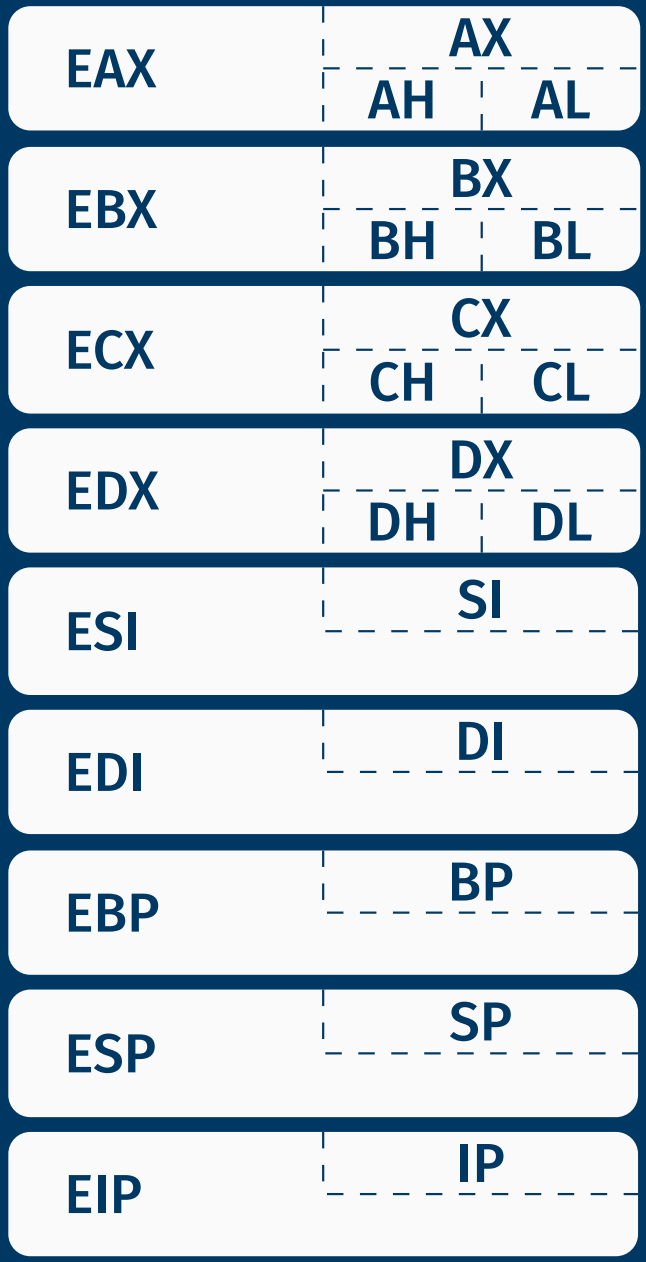
Daten-/Allzweckregister



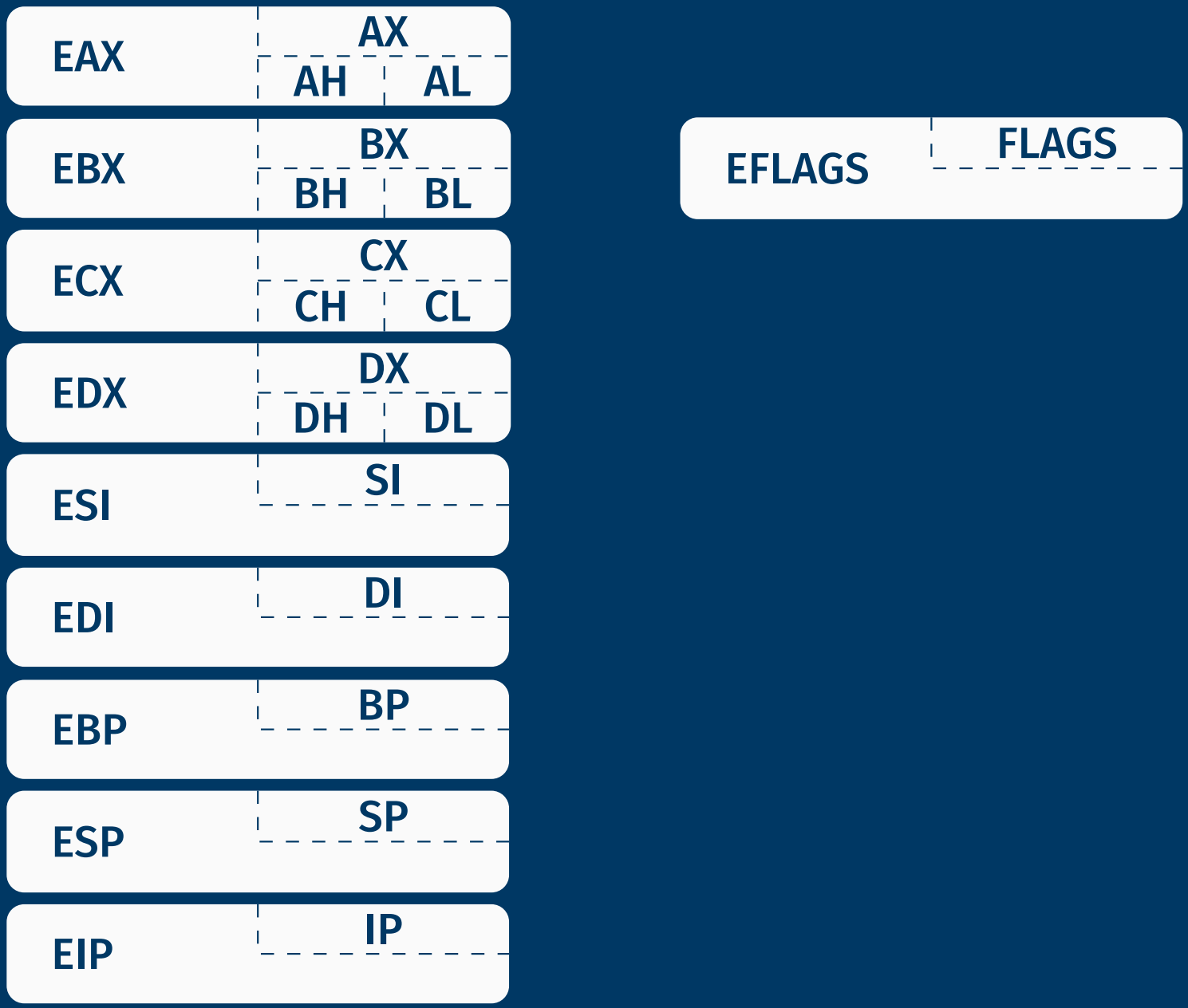
ursprünglich Zeichenkettenquellindex

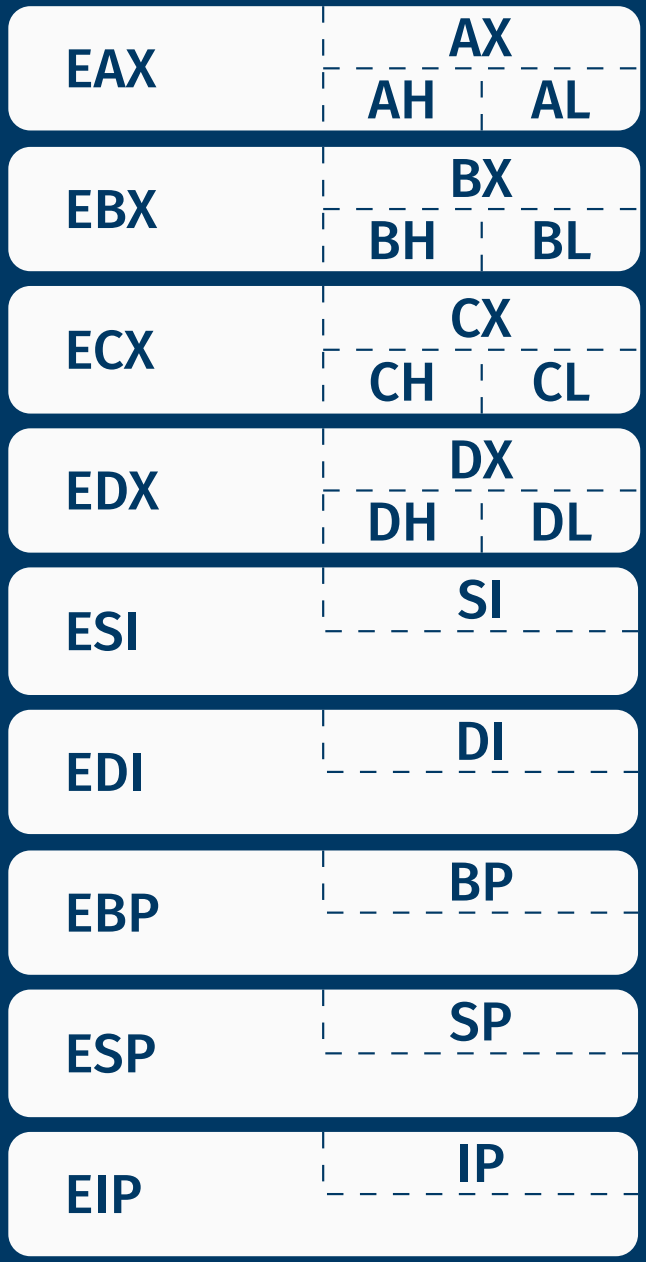
ursprünglich Zeichenkettenzielindex


EAX
AX
AH
AL
EBX
BX
BH
BL
ECX
CX
CH
CL
EDX
DX
DH
DL
ESI
SI
EDI
DI
EBP
BP
Basiszeiger für Stapelrahmen
ESP
SP
Stapelzeiger



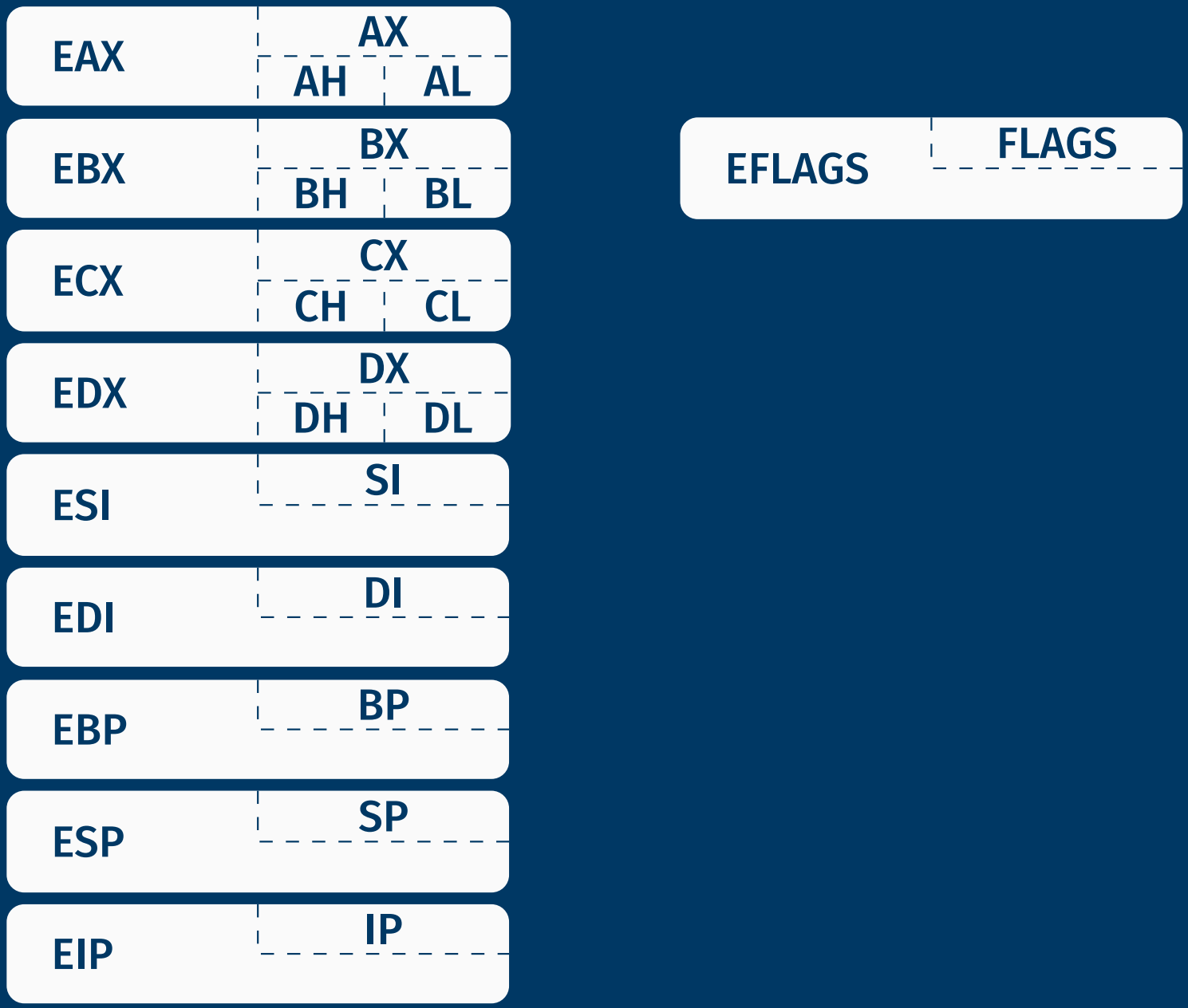
Befehls-/Instruktionszeiger


EAX
AX
AH
AL
EBX
BX
BH
BL
ECX
CX
CH
CL
EDX
DX
DH
DL
ESI
SI
EDI
DI
EBP
BP
ESP
SP
EIP
IP
EFLAGS
FLAGS

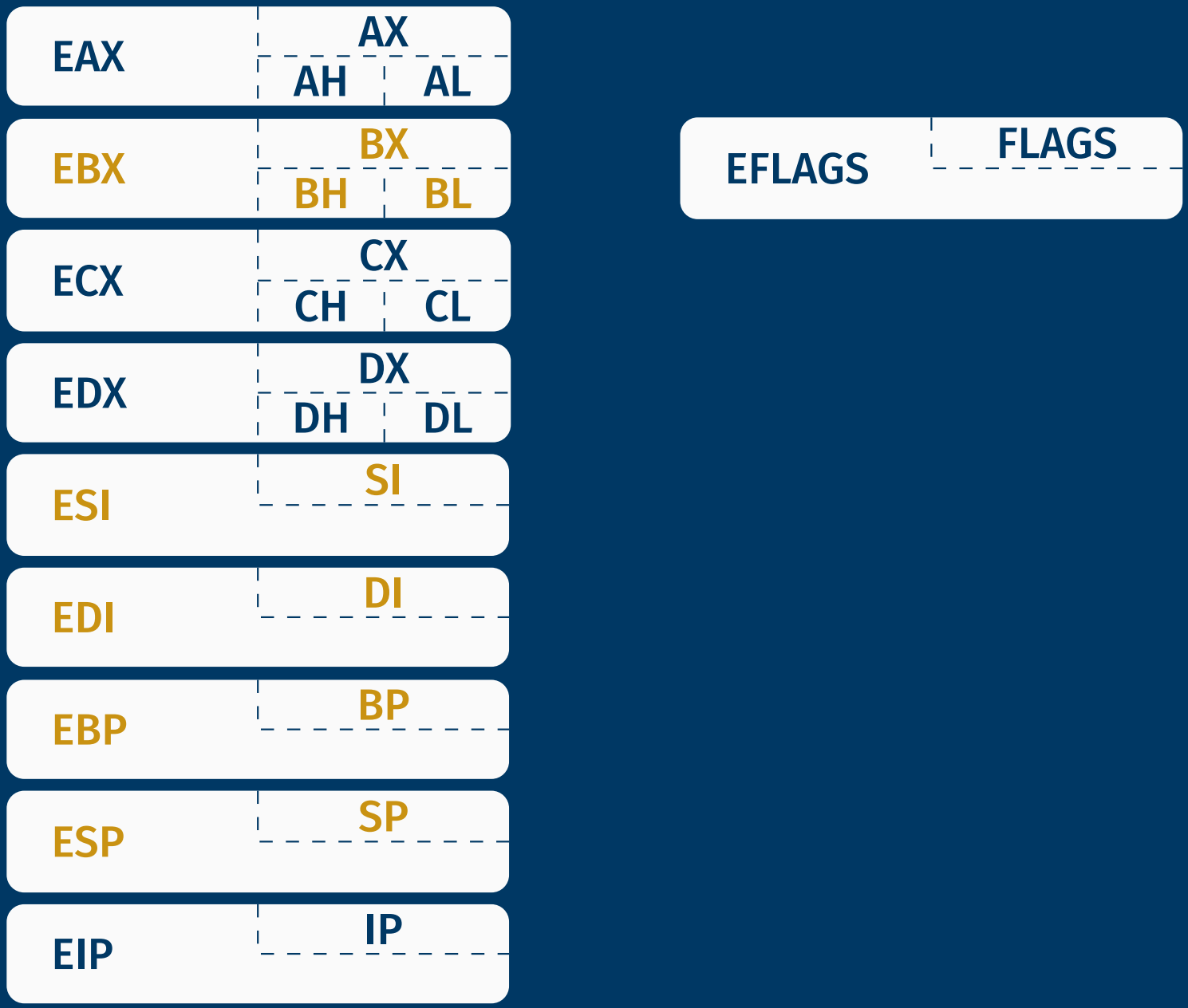

EAX
AX
AH
AL
EBX
BX
BH
BL
ECX
CX
CH
CL
EDX
DX
DH
DL
ESI
SI
EDI
DI
EBP
BP
ESP
SP
EIP
IP


EFLAGS
FLAGS
CS
DS
SS
ES
FS
GS

+ FPU, MMX, SSE, ...



Flüchtige versus nicht-flüchtige Register



Flüchtige versus nicht-flüchtige Register

Struktur zur Kontextsicherung

```
struct toc {
  void *ebx;
  void *esi;
  void *edi;
  void *ebp;
  void *esp;
};
```

Struktur zur Kontextsicherung

```
struct toc {
  void *ebx;
  void *esi;
  void *edi;
  void *ebp;
  void *esp;
};
```

Kontext sichern nach

```
struct toc * toc_obj;
```

Struktur zur Kontextsicherung

```
struct toc {
  void *ebx;
  void *esi;
  void *edi;
  void *ebp;
  void *esp;
};
```

Kontext sichern nach

```
struct toc * toc_obj;
```

in Assembler

```
mov eax, toc_obj
mov [eax +  0], ebx
mov [eax +  4], esi
mov [eax +  8], edi
mov [eax + 12], ebp
mov [eax + 16], esp
```

Struktur zur Kontextsicherung

```
struct toc {
  void *ebx;
  void *esi;
  void *edi;
  void *ebp;
  void *esp;
};
```

Kontext sichern nach

```
struct toc * toc_obj;
```

in Assembler

```
mov eax, toc_obj
mov [eax + ebx_offset], ebx
mov [eax + esi_offset], esi
mov [eax + edi_offset], edi
mov [eax + ebp_offset], ebp
mov [eax + esp_offset], esp
```

Die Abstände sind auch in der vorgegebenen Datei `toc.inc` *definiert*

Syntaxunterschiede bei x86-Assembler

Intel:

```
mov ebx, 0x17
```

(Ziel, Quelle)

`objdump -M intel`

AT&T:

```
mov $0x17, %ebx
```

(Quelle, Ziel)

Standard bei `objdump`

Kontextwechsel

Ziel: Instruktionszeiger `eip` ändern

Ziel: Instruktionszeiger `eip` ändern

Problem: Register `eip` kann nicht direkt beschrieben werden

Ziel: Instruktionszeiger `eip` ändern

Problem: Register `eip` kann nicht direkt beschrieben werden

Lösung: Änderung über `ret` und Adresse auf Stack (wie bei der Rückkehr nach einer Funktion)

Übersetung eines Funktionsaufrufs

```
int i = func(23, 42);
```

Übersetung eines Funktionsaufrufs

```
int i = func(23, 42);
```

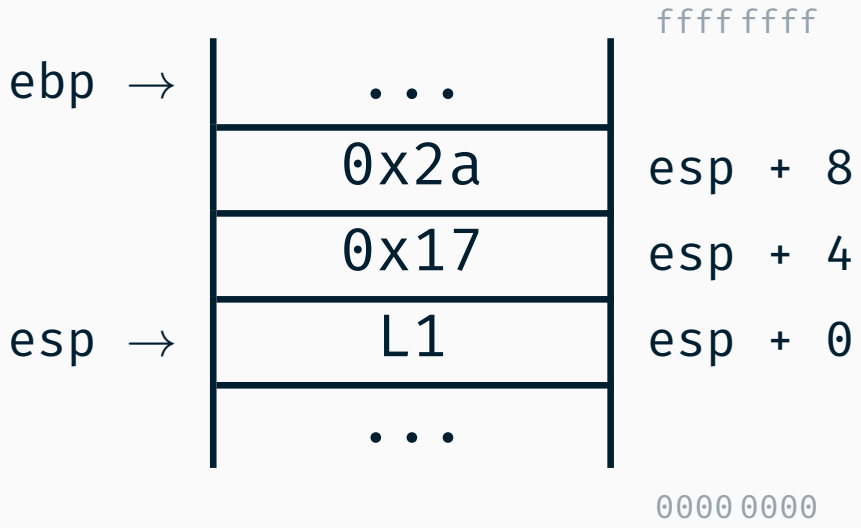
```
; 2. Parameter auf Stack
    push 0x2a

; 1. Parameter auf Stack
    push 0x17

; Funktionsaufruf
    call func
; pushed implizit die
; Rücksprungadresse
L1:
```

Übersetung eines Funktionsaufrufs

```
int i = func(23, 42);
```

```
; 2. Parameter auf Stack
    push 0x2a

; 1. Parameter auf Stack
    push 0x17

; Funktionsaufruf
    call func
; pushed implizit die
; Rücksprungadresse
L1:
```

Stack beim Funktionsaufruf:

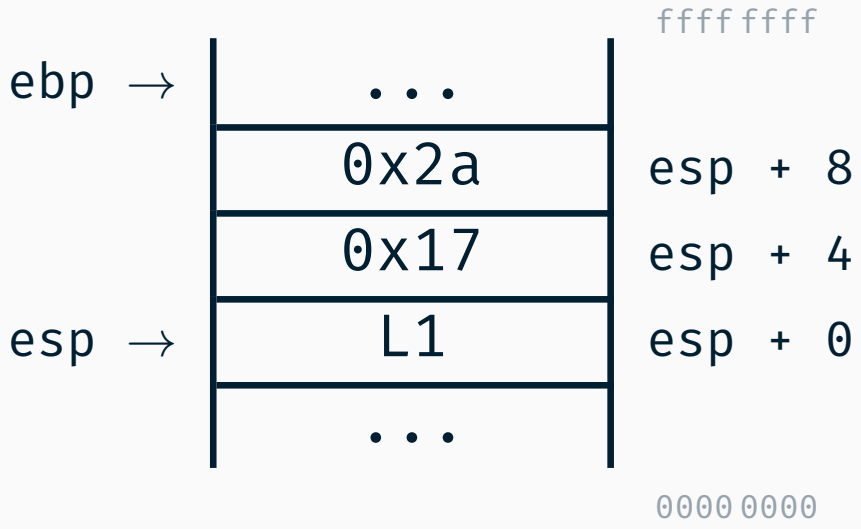


Übersetung eines Funktionsaufrufs

```
int i = func(23, 42);
```

```
; 2. Parameter auf Stack
    push 0x2a

; 1. Parameter auf Stack
    push 0x17

; Funktionsaufruf
    call func
; pushed implizit die
; Rücksprungadresse
L1:

; SP wiederherstellen
    add esp, 8

; Rückgabe sichern
    mov ebx, eax
```

Stack beim Funktionsaufruf:



Zur Erinnerung – für den Stack auf x86 gilt

- Der Stack „wächst“ von *oben* (hohe Adresse) nach *unten* (niedrige Adresse).
- Der Stackzeiger (`esp`) zeigt auf das zuletzt hinzugefügte Datum

  `push X` verringert zuerst den Wert von `esp` um 4 Byte und legt dann X an die resultierende Adresse

  `pop X` liest den Inhalt des Speichers, auf das `esp` zeigt, in X und erhöht anschließend den Wert von ESP um 4 Byte.

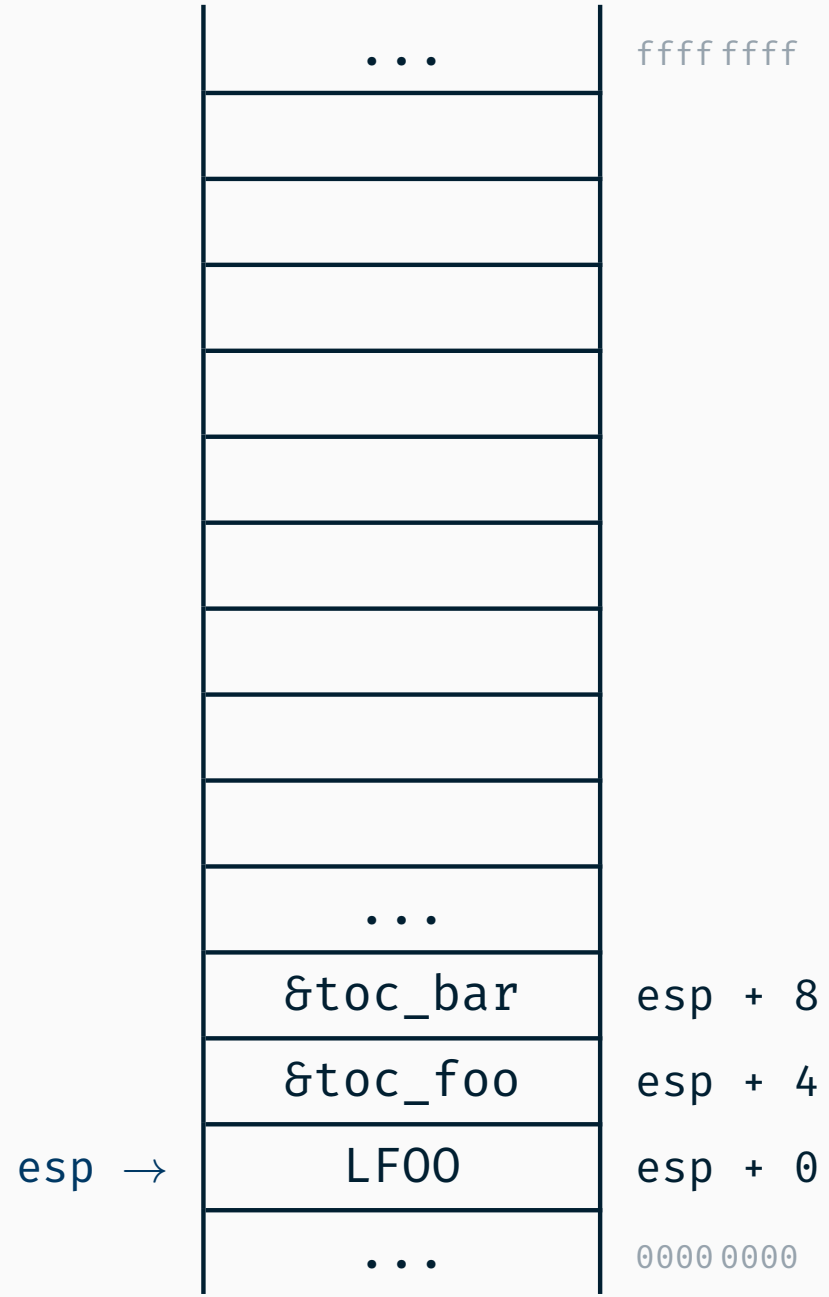
- Adressen sollten an der 4 Byte-Grenze ausgerichtet sein (sonst droht Performanzverlust)

Kontextwechsel

```
void foo(){
  int f = 42;

  while (f--){
    kout << "foo"
         << f
         << endl;
    toc_switch(
        &toc_foo,
        &toc_bar
      );
LFOO:
  }
}
```

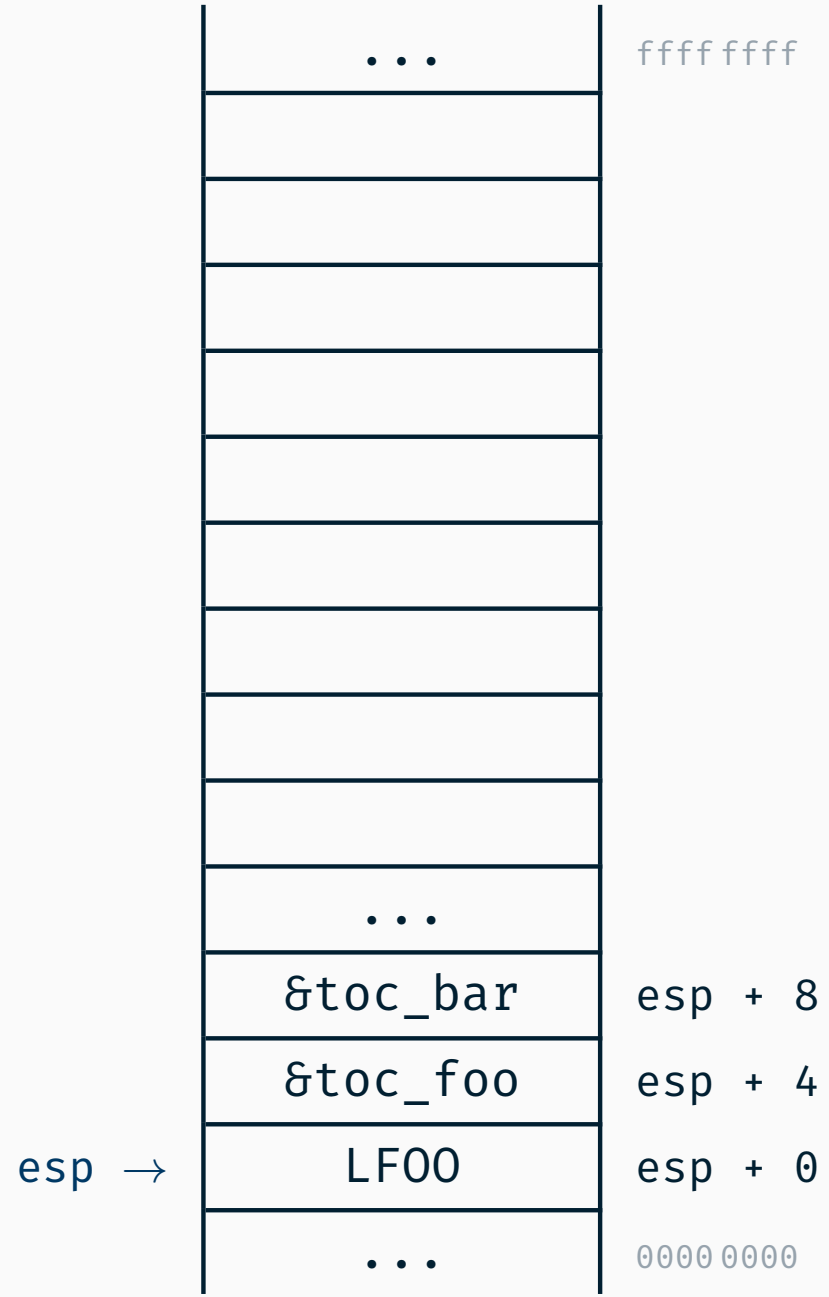
Kontextwechsel

```
void foo(){
  int f = 42;

  while (f--){
    kout << "foo"
         << f
         << endl;
    toc_switch(
        &toc_foo,
        &toc_bar
      );
LFOO:
  }
}
```



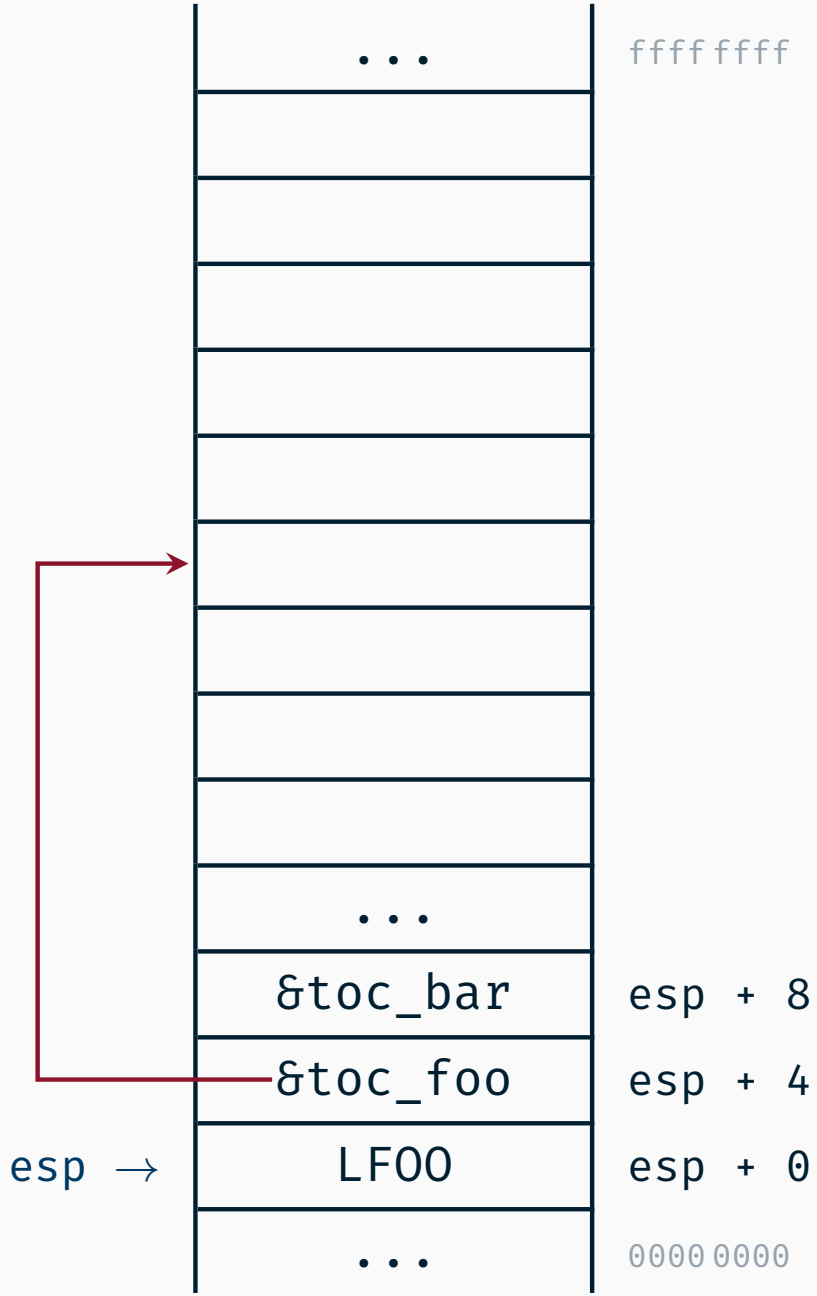
Kontextwechsel

```
; in Assembler
toc_switch:
```



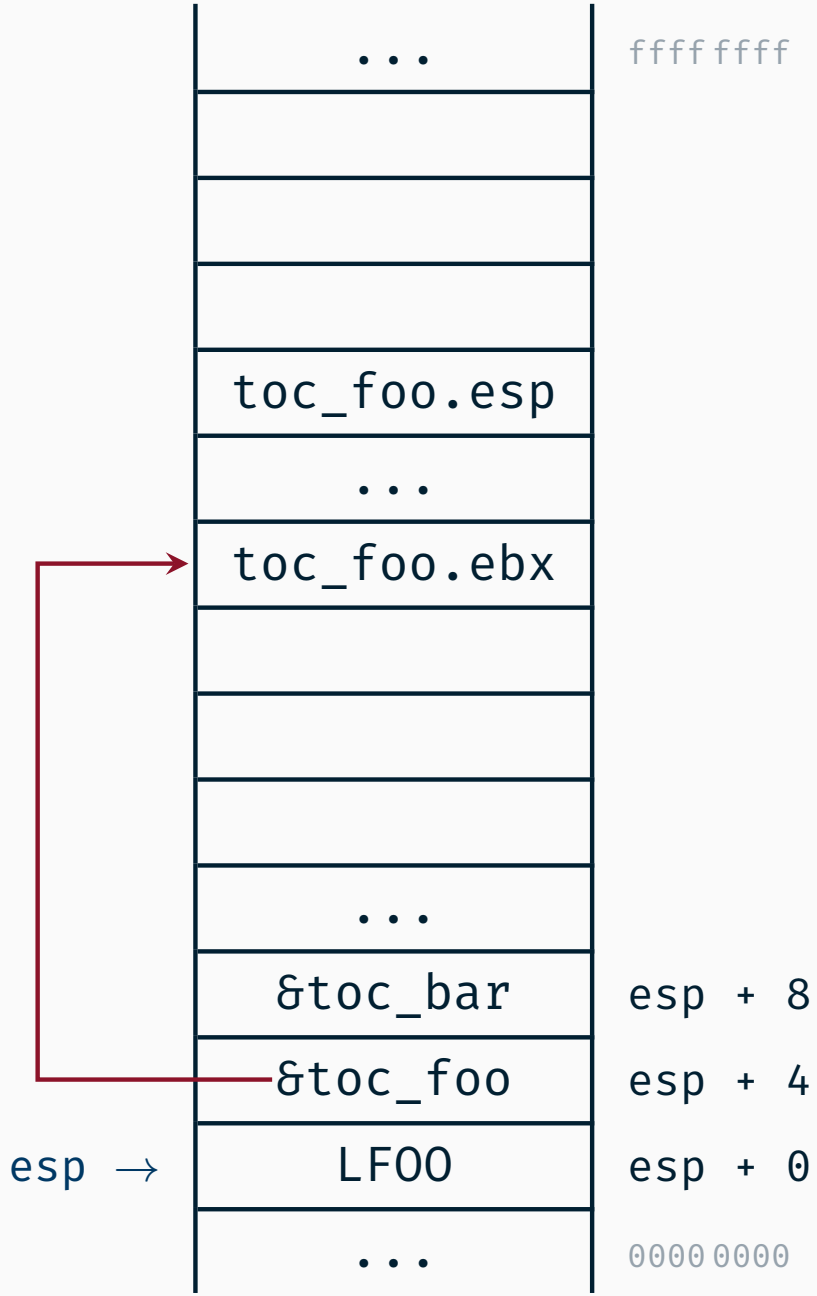
Kontextwechsel

```
; in Assembler
toc_switch:
```



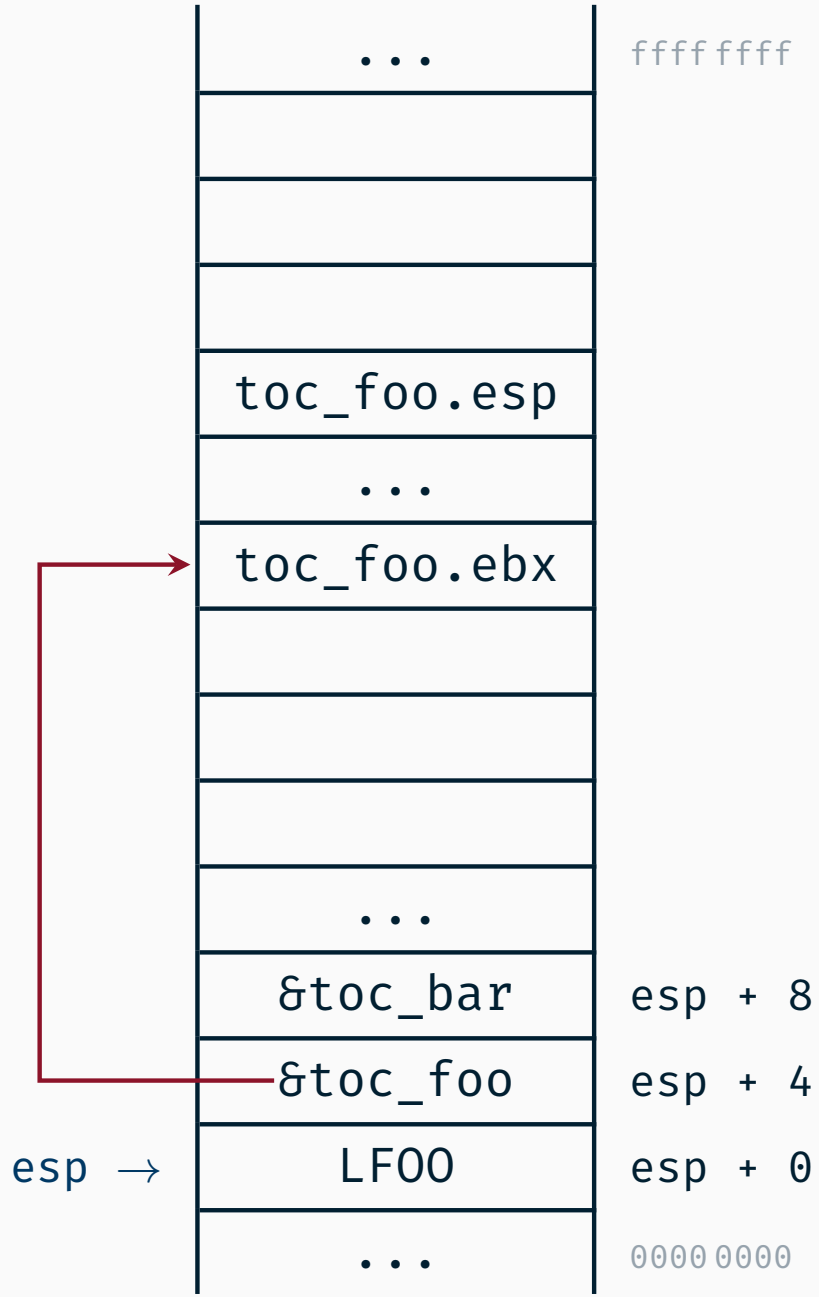
Kontextwechsel

```
; in Assembler
toc_switch:
```



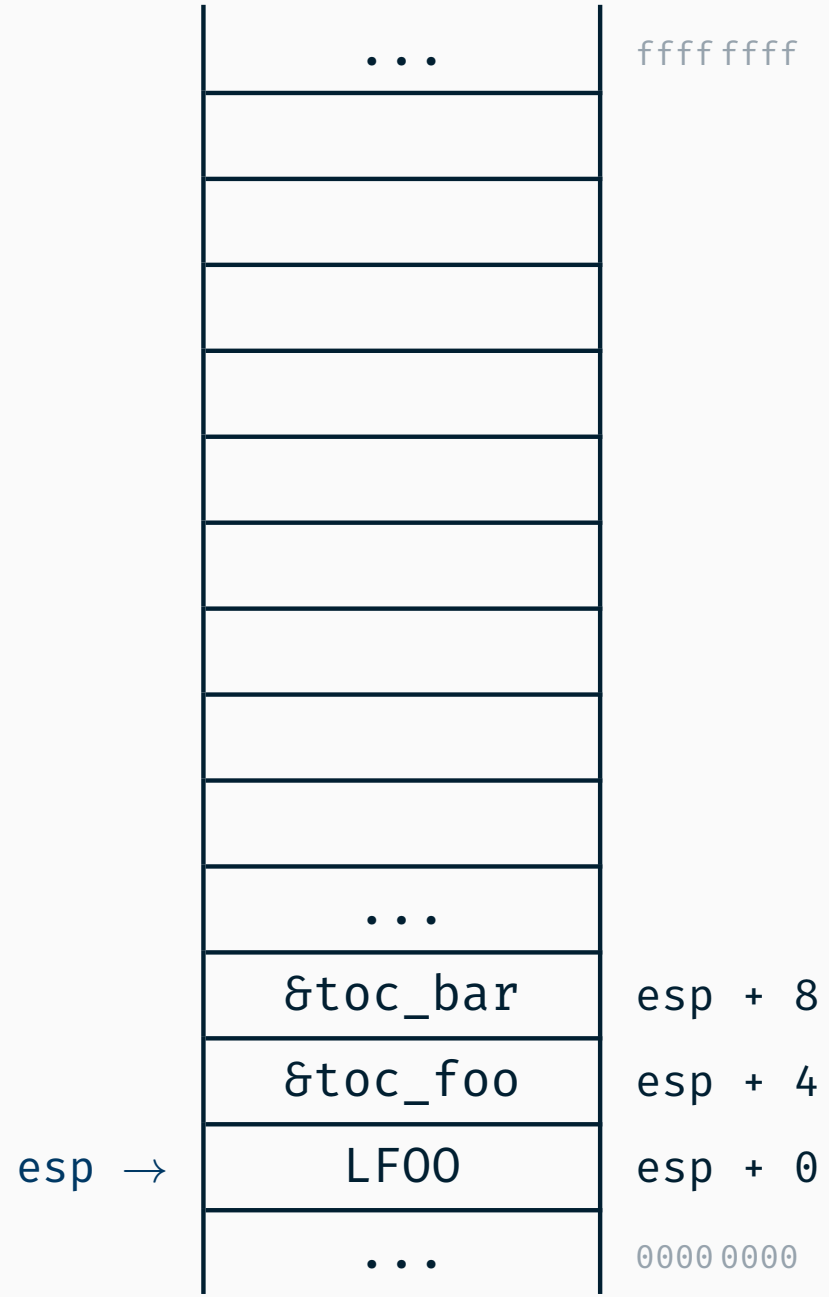
Kontextwechsel

```
; in Assembler
toc_switch:

; vorherige Register sichern
  mov eax, [esp + 4]
  mov [eax + ebx_offset], ebx
  ...
  mov [eax + esp_offset], esp
```



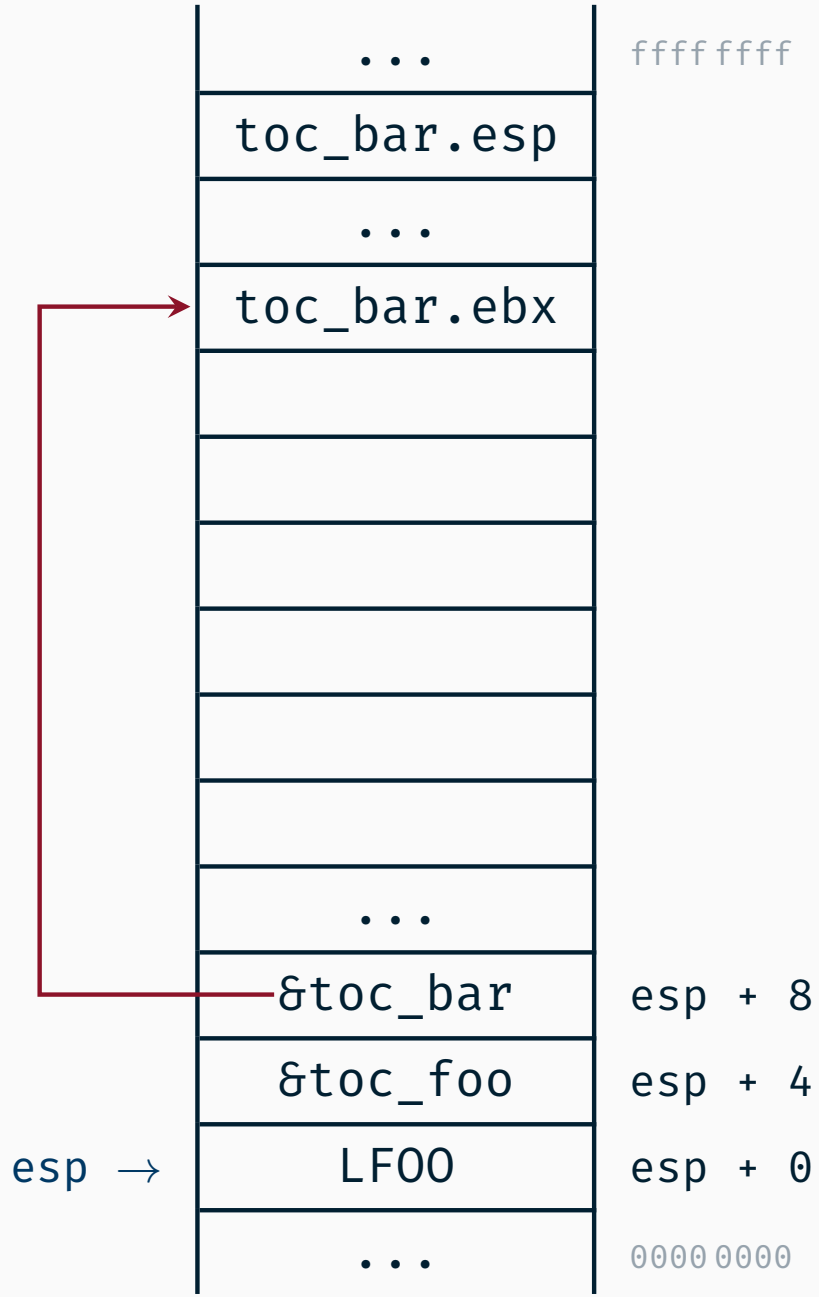
Kontextwechsel

```
; in Assembler
toc_switch:

; vorherige Register sichern
  mov eax, [esp + 4]
  mov [eax + ebx_offset], ebx
  ...
  mov [eax + esp_offset], esp
```



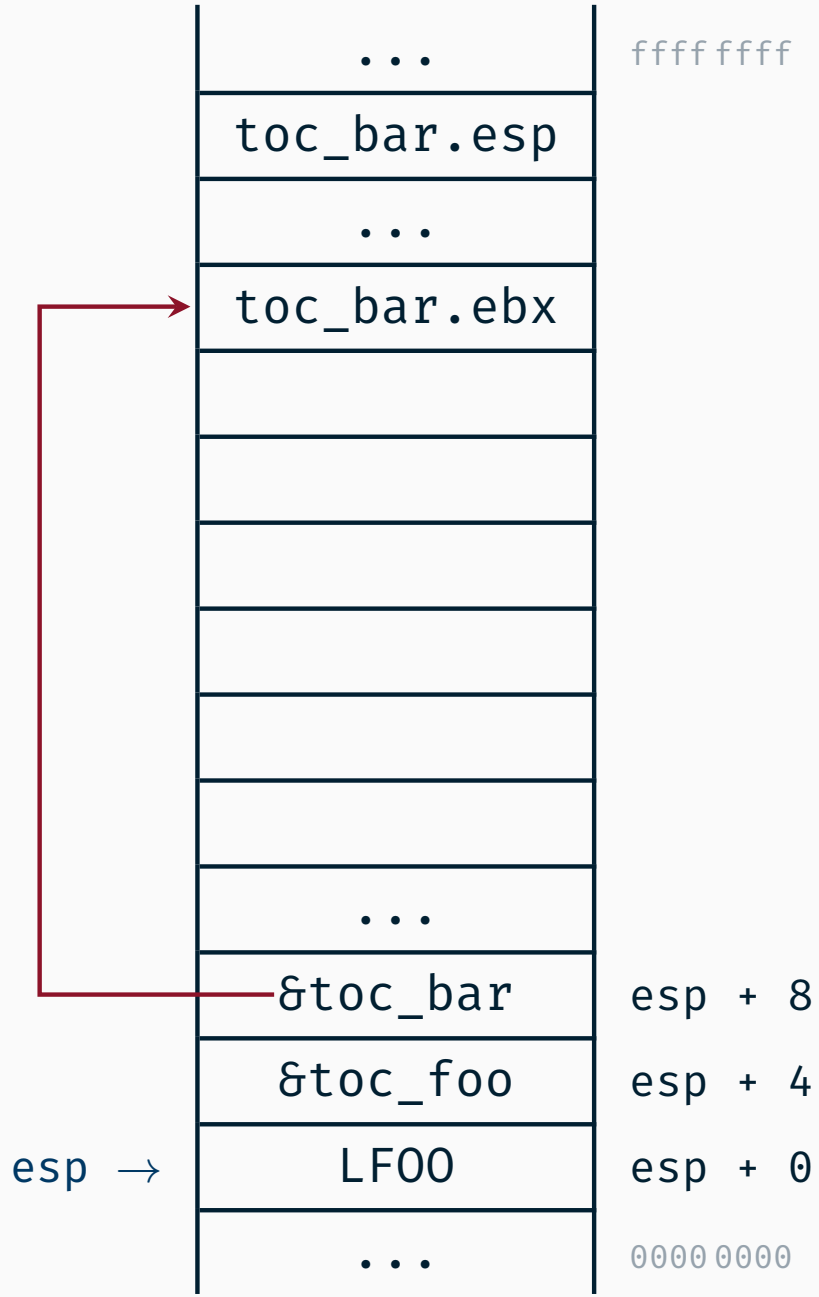
Kontextwechsel

```
; in Assembler
toc_switch:

; vorherige Register sichern
  mov eax, [esp + 4]
  mov [eax + ebx_offset], ebx
  ...
  mov [eax + esp_offset], esp
```



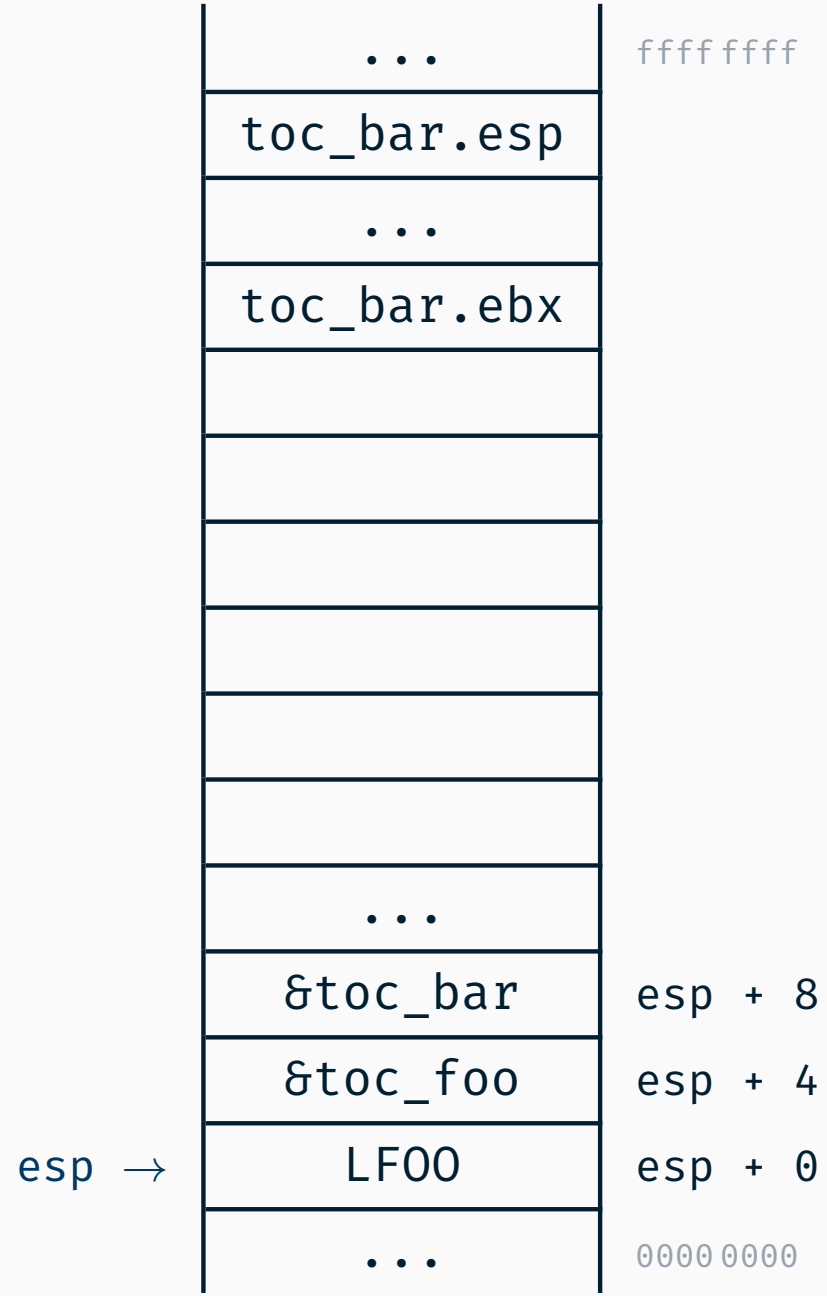
Kontextwechsel

```
; in Assembler
toc_switch:

; vorherige Register sichern
  mov eax, [esp + 4]
  mov [eax + ebx_offset], ebx
  ...
  mov [eax + esp_offset], esp

; neue Register laden
  mov eax, [esp + 8]
  mov ebx, [eax + ebx_offset]
  ...
  mov esp, [eax + esp_offset]
```



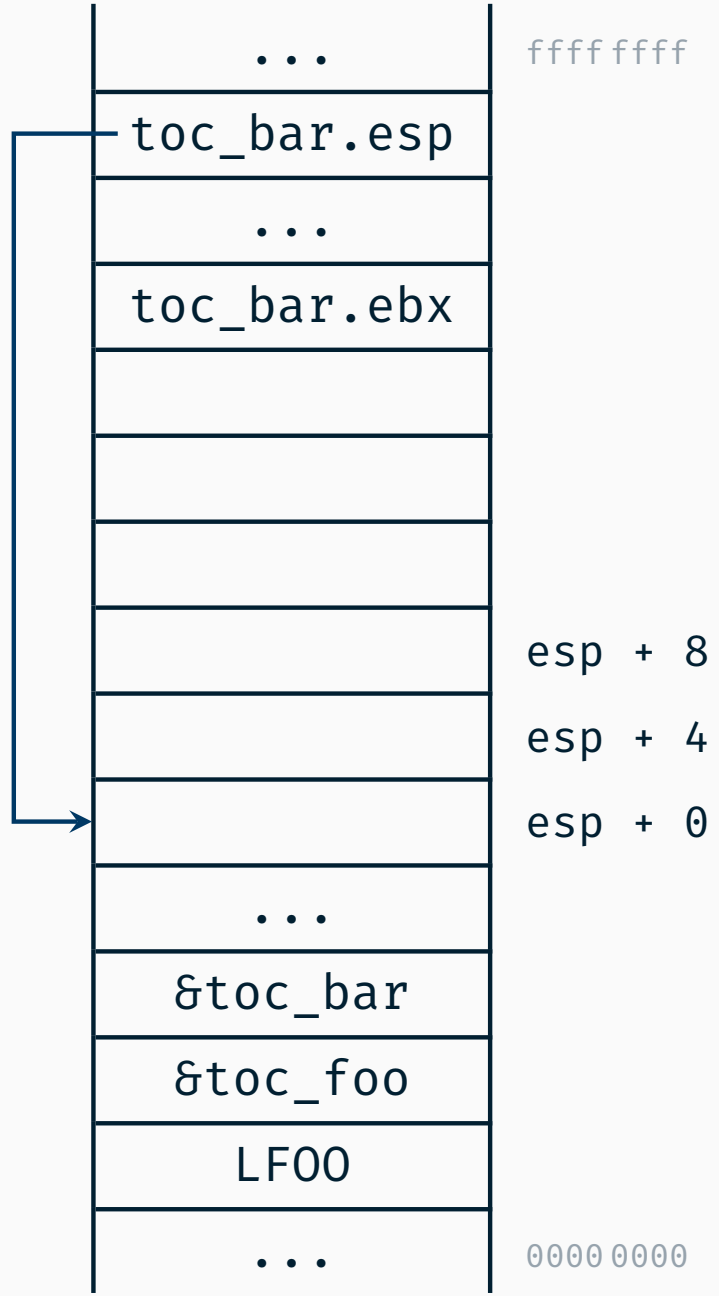
Kontextwechsel

```
; in Assembler
toc_switch:

; vorherige Register sichern
  mov eax, [esp + 4]
  mov [eax + ebx_offset], ebx
  ...
  mov [eax + esp_offset], esp

; neue Register laden
  mov eax, [esp + 8]
  mov ebx, [eax + ebx_offset]
  ...
  mov esp, [eax + esp_offset]
```



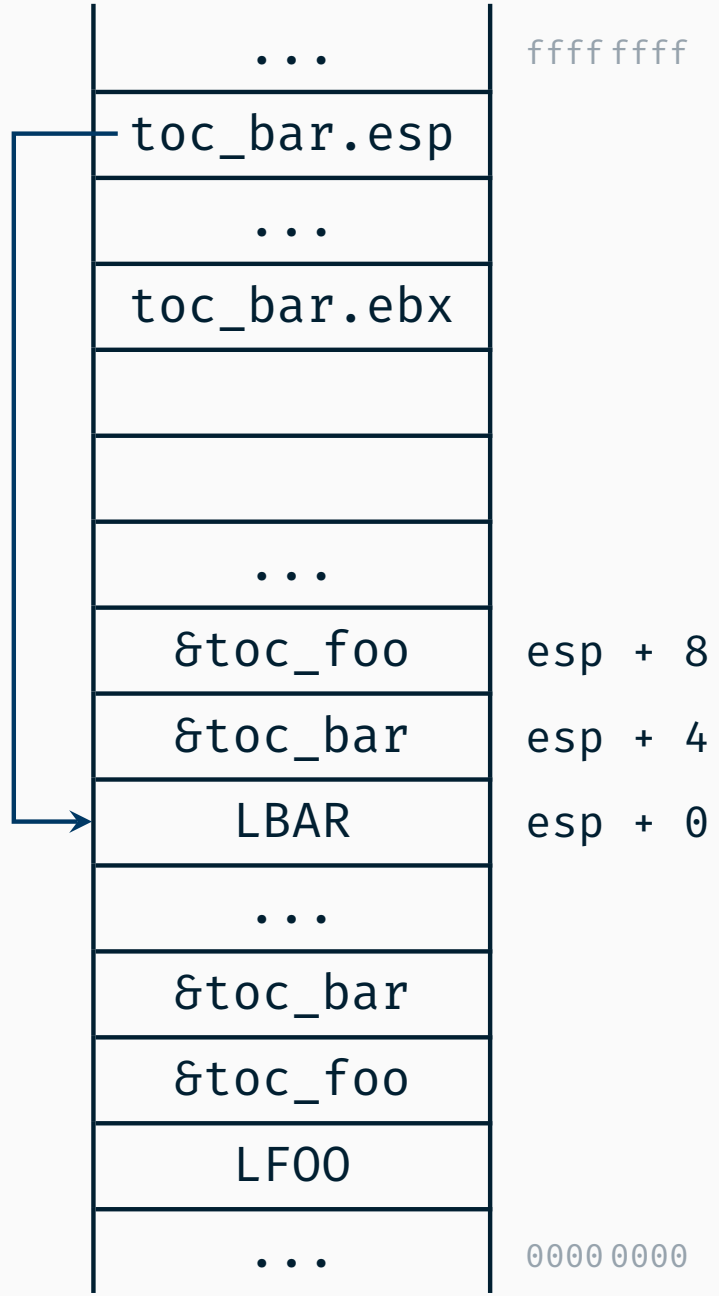
Kontextwechsel

```
; in Assembler
toc_switch:

; vorherige Register sichern
  mov eax, [esp + 4]
  mov [eax + ebx_offset], ebx
  ...
  mov [eax + esp_offset], esp

; neue Register laden
  mov eax, [esp + 8]
  mov ebx, [eax + ebx_offset]
  ...
  mov esp, [eax + esp_offset]
```



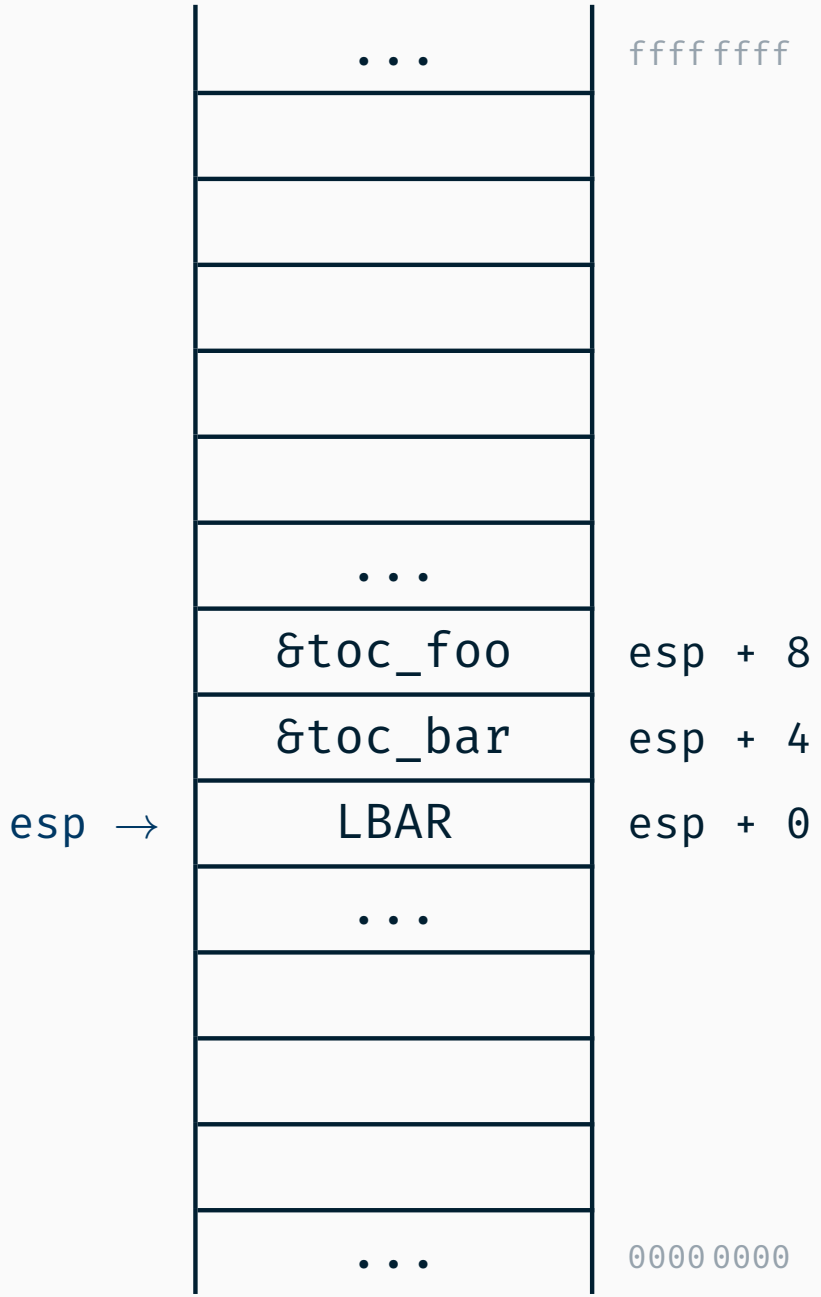
Kontextwechsel

```
; in Assembler
toc_switch:

; vorherige Register sichern
  mov eax, [esp + 4]
  mov [eax + ebx_offset], ebx
  ...
  mov [eax + esp_offset], esp

; neue Register laden
  mov eax, [esp + 8]
  mov ebx, [eax + ebx_offset]
  ...
  mov esp, [eax + esp_offset]
```



Kontextwechsel

```
; in Assembler
toc_switch:

; vorherige Register sichern
  mov eax, [esp + 4]
  mov [eax + ebx_offset], ebx
  ...
  mov [eax + esp_offset], esp

; neue Register laden
  mov eax, [esp + 8]
  mov ebx, [eax + ebx_offset]
  ...
  mov esp, [eax + esp_offset]

; "Zurückkehren"
  ret
```



Koroutine (erstmalig) starten

```
void coroutine(void *arg){ ... }
```

Koroutine (erstmalig) starten

```
void coroutine(void *arg){ ... }
```

- Wie rufe ich die aller erste Koroutine auf?

Koroutine (erstmalig) starten

```
void coroutine(void *arg){ ... }
```

- Wie rufe ich die aller erste Koroutine auf?
  - Kann ich mit leeren Registern starten?

Koroutine (erstmalig) starten

```
void coroutine(void *arg){ ... }
```

- Wie rufe ich die aller erste Koroutine auf?
  - Kann ich mit leeren Registern starten? Nicht ganz.
  - Einsprungsfunktion

`toc_go(toc *regs);`

Lädt den Kontext `regs` (quasi 2. Hälfte von `toc_switch`, ohne Sichern des aktuellen Kontexts)

Koroutine (erstmalig) starten

```
void coroutine(void *arg){ ... }
```

- Wie rufe ich die aller erste Koroutine auf?
  - Kann ich mit leeren Registern starten? Nicht ganz.
  - Einsprungsfunktion

```
toc_go(toc *regs);
```

Lädt den Kontext `regs` (quasi 2. Hälfte von `toc_switch`, ohne Sichern des aktuellen Kontexts)

- Was wird für jede Koroutine gebraucht?

Koroutine (erstmalig) starten

```
void coroutine(void *arg){ ... }
```

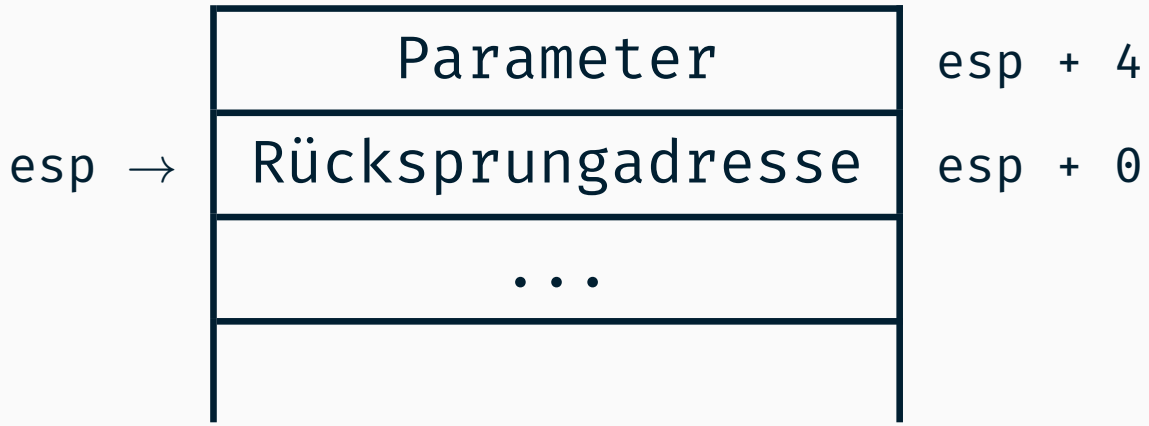
- Wie rufe ich die aller erste Koroutine auf?
  - Kann ich mit leeren Registern starten? Nicht ganz.
  - Einsprungsfunktion

```
toc_go(toc *regs);
```

Lädt den Kontext `regs` (quasi 2. Hälfte von `toc_switch`, ohne Sichern des aktuellen Kontexts)

- Was wird für jede Koroutine gebraucht?
  - Kontextstruktur `toc`
  - eigenen, präparierter Stack

Stack aufsetzen



Stack aufsetzen

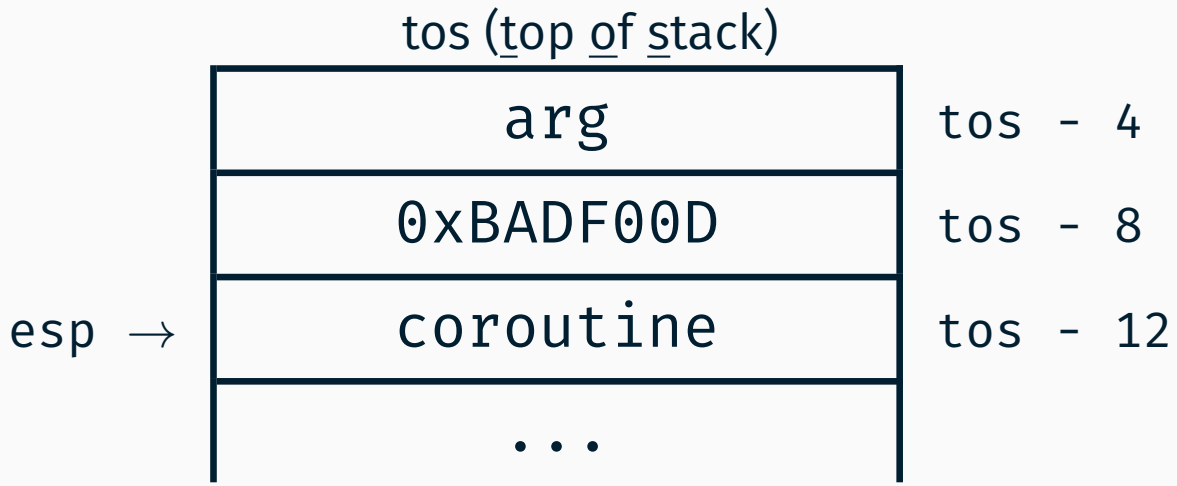
```
void coroutine(void *arg){ ... }
```

Stack aufsetzen

```
void coroutine(void *arg){ ... }
```

Stack aufsetzen

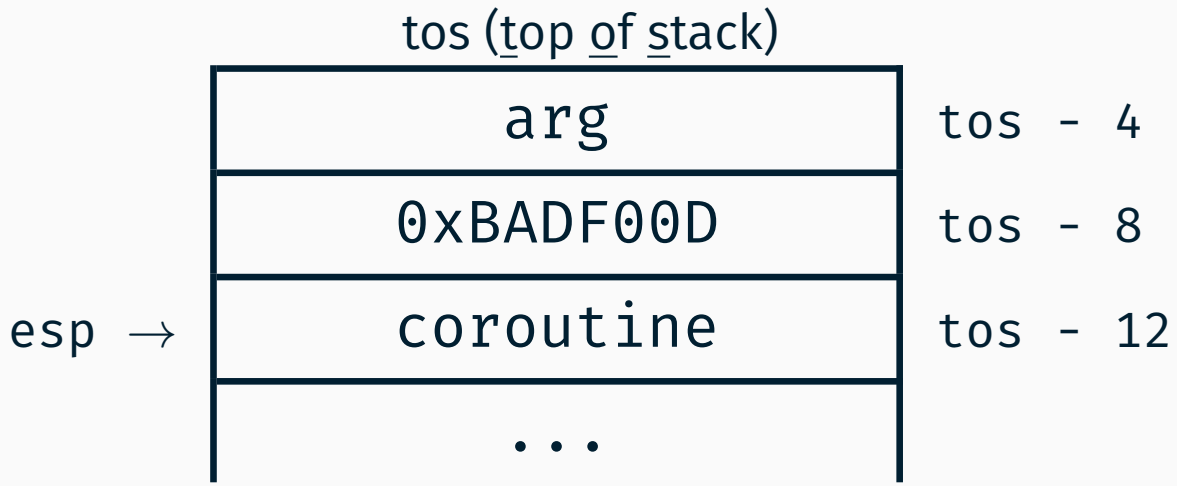
```
void coroutine(void *arg){ ... }
```



was passiert nun bei `toc_go`/`toc_switch`?

Stack aufsetzen

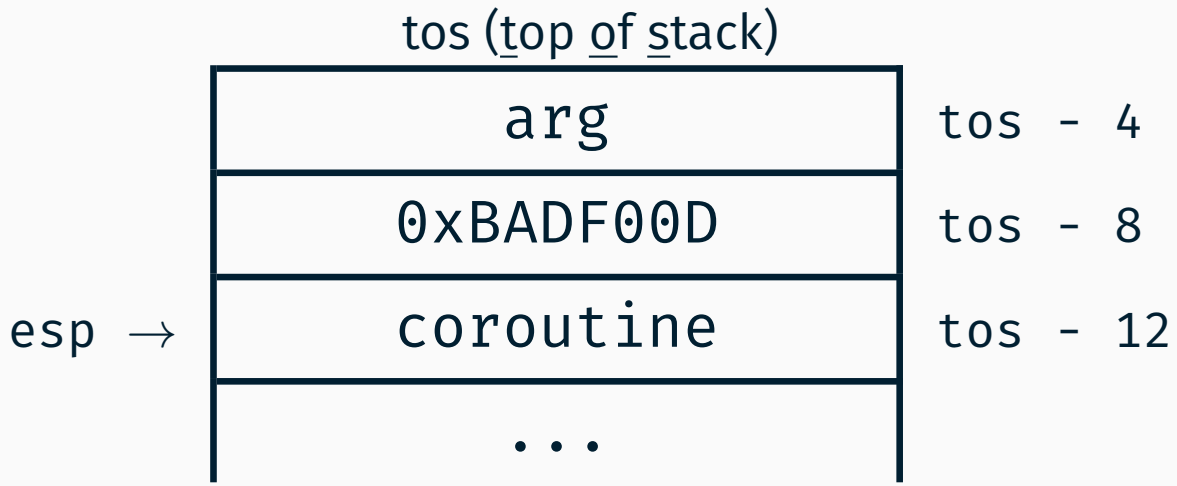
```
void coroutine(void *arg){ ... }
```



was passiert wenn `coroutine` abgearbeitet ist?

Stack aufsetzen

```
void coroutine(void *arg){ ... }
```

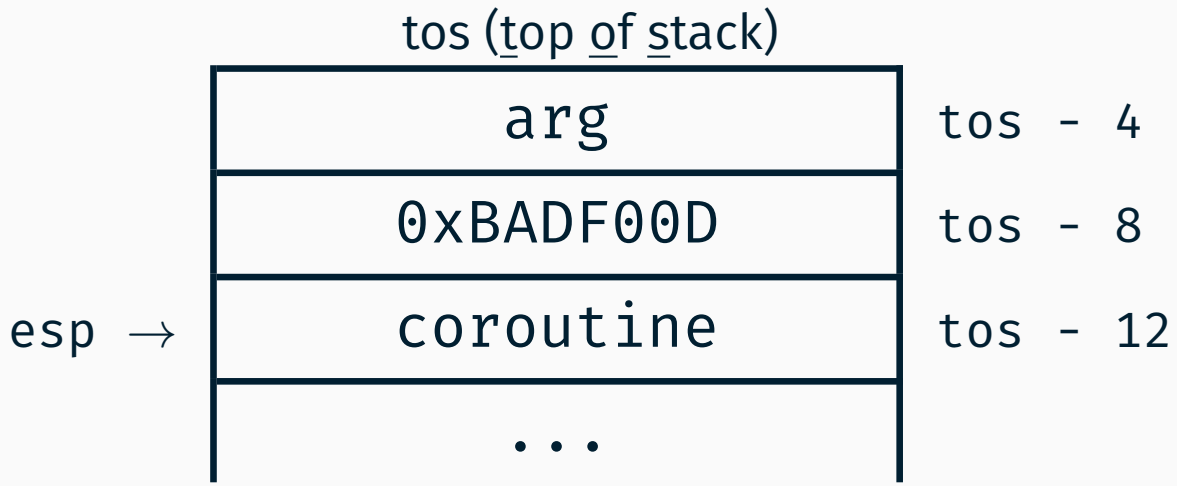



```
void toc_settle(struct toc *regs, void *tos,
        void (*coroutine)(void*), void* arg);
```

Präperiert den Stack für den ersten Einsprung

Stack aufsetzen

```
void coroutine(void *arg){ ... }
```



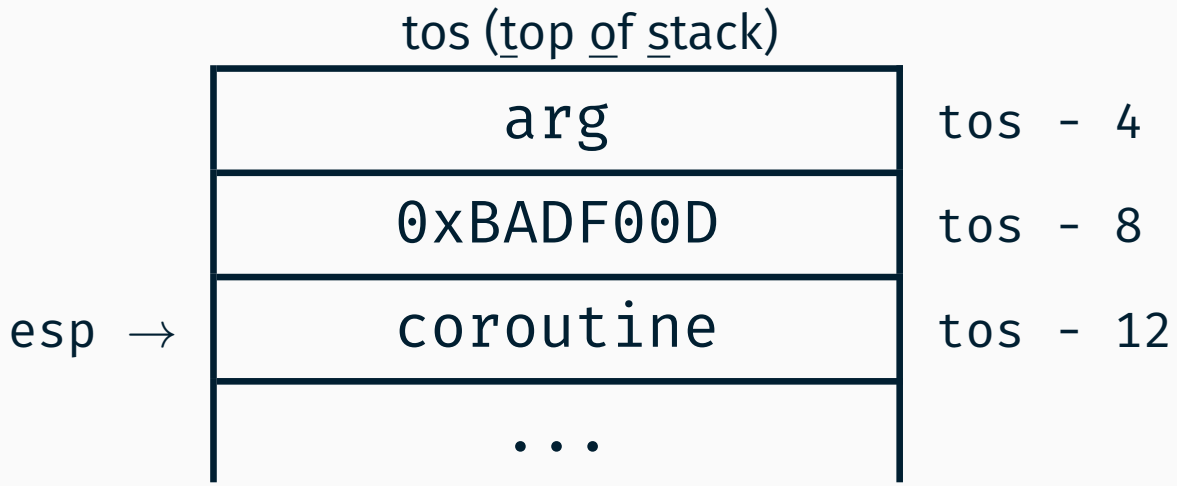

```
void toc_settle(struct toc *regs, void *tos,
        void (*coroutine)(void*), void* arg);
```

Präperiert den Stack für den ersten Einsprung

- in C[++] (statt Assembler)

Stack aufsetzen

```
void coroutine(void *arg){ ... }
```



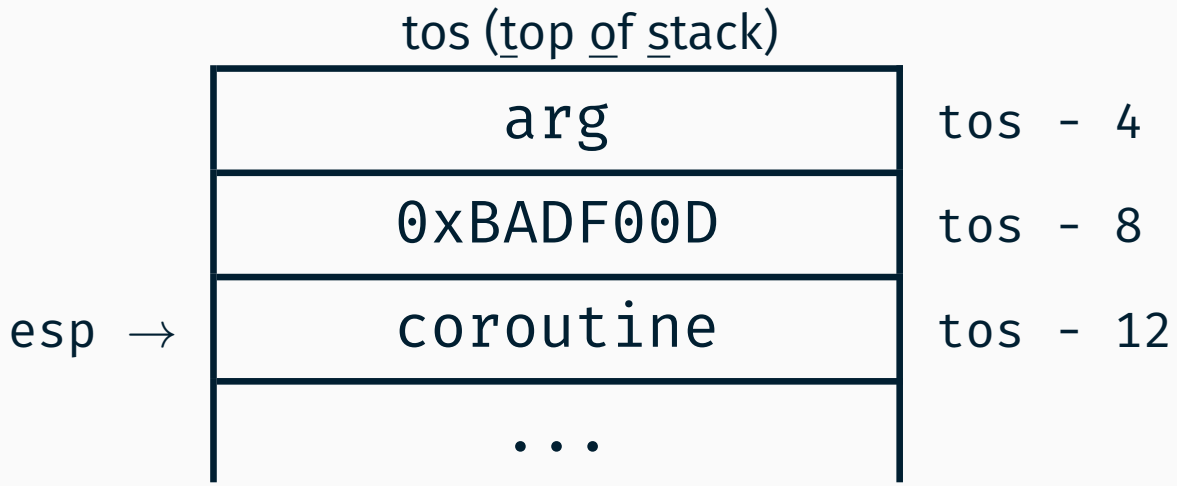

```
void toc_settle(struct toc *regs, void *tos,
        void (*coroutine)(void*), void* arg);
```

Präperiert den Stack für den ersten Einsprung

- in C[++] (statt Assembler)
- statisch übergebenen Speicher `tos` als Stack aufsetzen

Stack aufsetzen

```
void coroutine(void *arg){ ... }
```



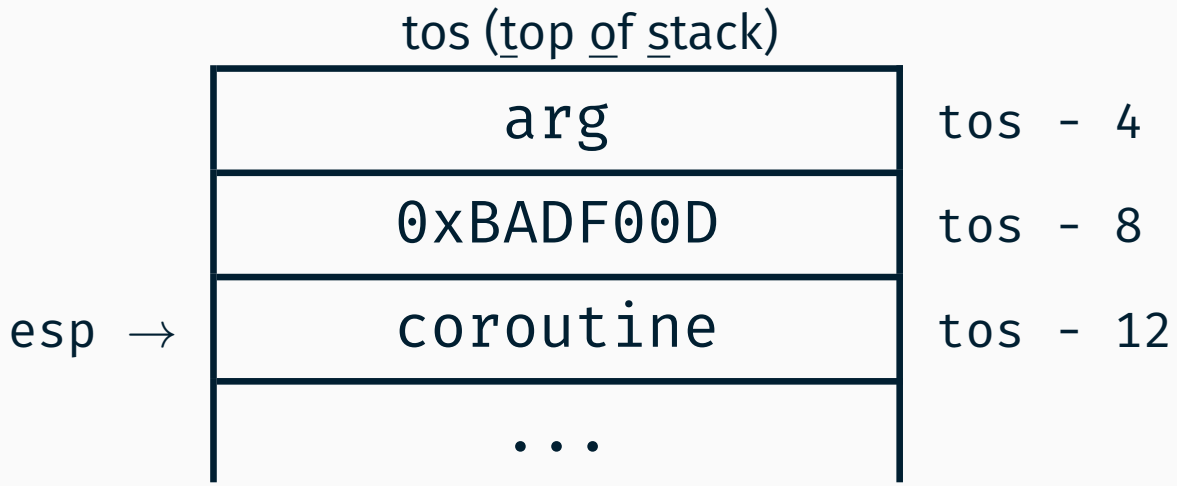

```
void toc_settle(struct toc *regs, void *tos,
        void (*coroutine)(void*), void* arg);
```

Präperiert den Stack für den ersten Einsprung

- in C[++] (statt Assembler)
- statisch übergebenen Speicher `tos` als Stack aufsetzen
- Pointerarithmetik ist hilfreich

Stack aufsetzen

```
void coroutine(void *arg){ ... }
```




```
void toc_settle(struct toc *regs, void *tos,
        void (*coroutine)(void*), void* arg);
```

Präperiert den Stack für den ersten Einsprung

- in C[++] (statt Assembler)
- statisch übergebenen Speicher `tos` als Stack aufsetzen
- Pointerarithmetik ist hilfreich
- `esp` in Kontextstruktur `regs` entsprechend setzen

Umsetzung in OOStuBS/MPStuBS

Threads

```
class Thread {
    struct toc regs;
  public:
    Thread(void* tos);
    virtual void action() = 0;
}
```

Threads

```
class Thread {
    struct toc regs;
  public:
    Thread(void* tos);
    virtual void action() = 0;
}
```

Adresse einer virtuellen Member-Funktion nicht (ohne weiteres) ermittelbar

Threads

```
class Thread {
    struct toc regs;
  public:
    Thread(void* tos);
    virtual void action() = 0;
}
```

Adresse einer virtuellen Member-Funktion nicht (ohne weiteres) ermittelbar

```
void kickoff (Thread* t){
  t->action();
}
```

Scheduler

Scheduler verwaltet Koroutinen zentral

```
class Scheduler {
    Queue<Thread> routines;
  public:
    void ready(Thread *);
    void schedule();
    void resume();
};
```

Scheduler (Beispiel)

```
// Thread AppFoo
void AppFoo::action(){
  while (1){
    kout << "foo" << endl;
    scheduler.resume();
  }
}
```

```
// Thread AppBar
void AppBar::action(){
  while (1){
    kout << "bar" << endl;
    scheduler.resume();
  }
}
```

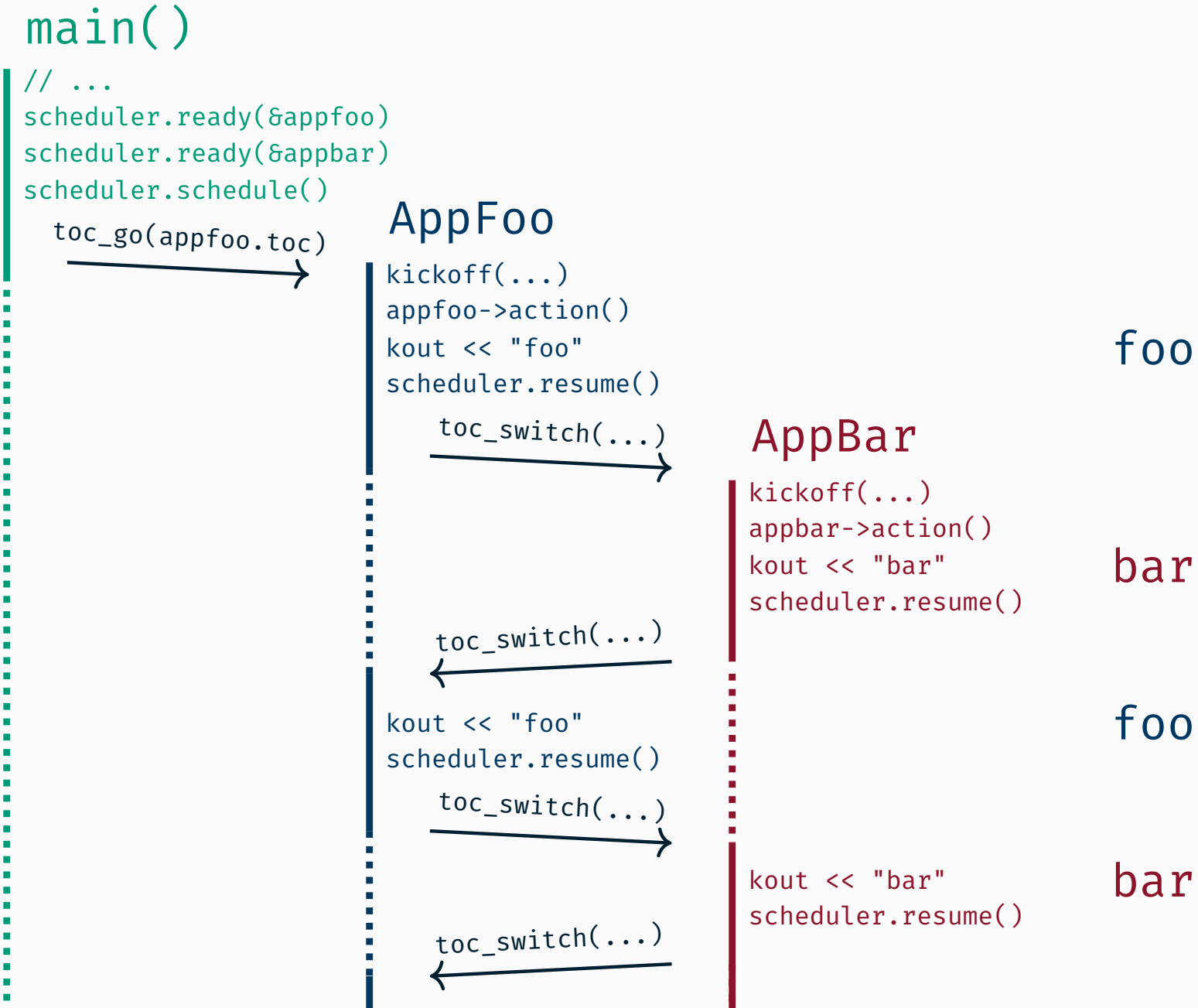
Scheduler (Beispiel)

```
// Thread AppFoo
void AppFoo::action(){
  while (1){
    kout << "foo" << endl;
    scheduler.resume();
  }
}
```

```
// Thread AppBar
void AppBar::action(){
  while (1){
    kout << "bar" << endl;
    scheduler.resume();
  }
}
```

```
// main.cc
AppFoo appfoo;
AppBar appbar;
int main (){
  // ...
  scheduler.ready(&appfoo);
  scheduler.ready(&appbar);
  scheduler.schedule();
}
```

Scheduler (Beispiel)



Scheduler

Scheduler verwaltet Koroutinen zentral

```
class Scheduler {
    Queue<Thread> routines;
  public:
    void ready(Thread *);
    void schedule();
    void resume();
};
```

OOStuBS immer synchrone Aufrufe
$\rightarrow$ Konsistenz gesichert

Scheduler

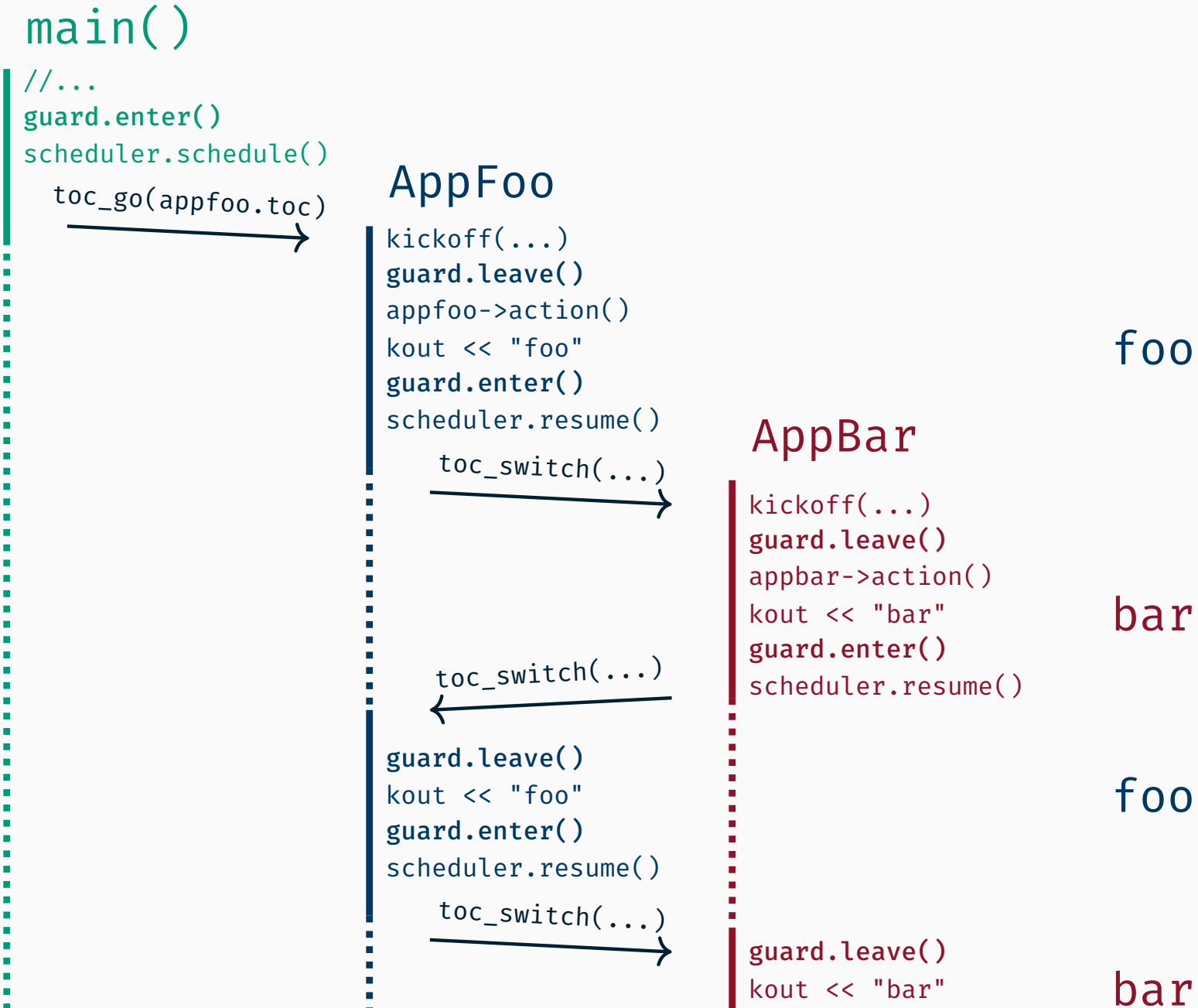
Scheduler verwaltet Koroutinen zentral

```
class Scheduler {
    Queue<Thread> routines;
  public:
    void ready(Thread *);
    void schedule();
    void resume();
};
```

OOStuBS immer synchrone Aufrufe
→ Konsistenz gesichert

MPStuBS Synchronisation notwendig

Scheduler (synchronisiert)

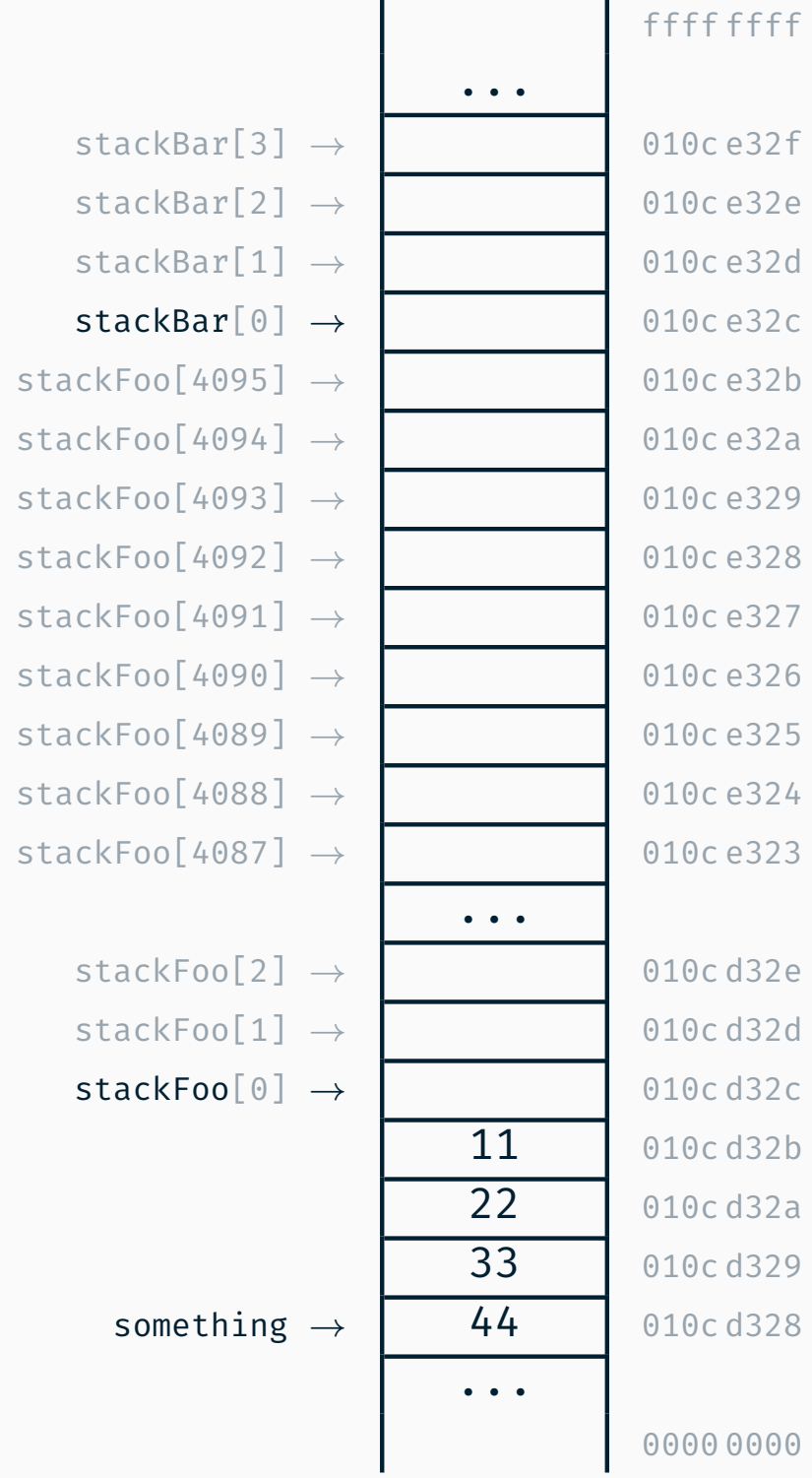


Fragen?

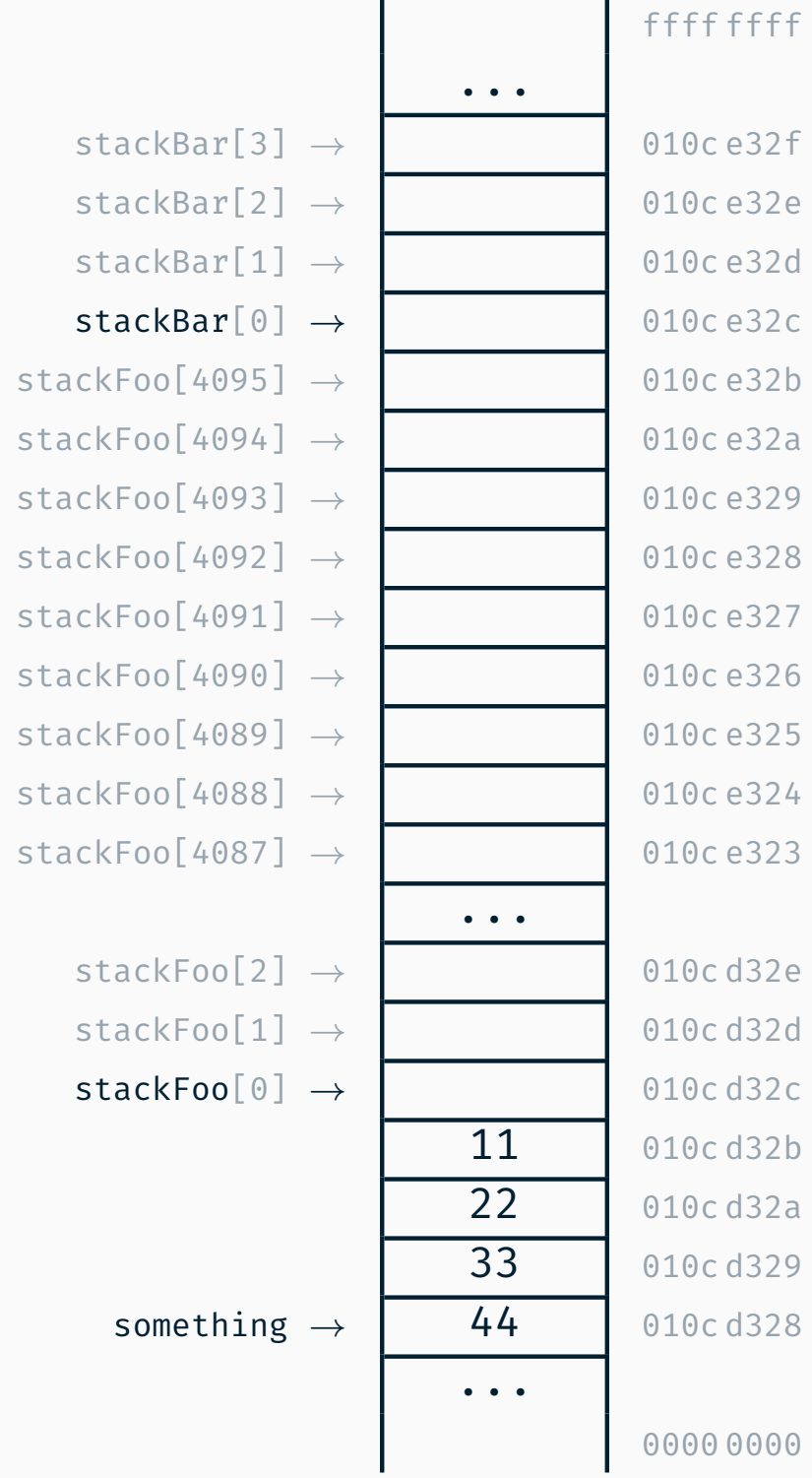
Nächste Woche (14. & 17. Dezember)
Abgabe von Aufgabe 3 im Huber-CIP

```
#define STACKSIZE 4096

int something=0x11223344;
char stackFoo[STACKSIZE];
char stackBar[STACKSIZE];
```

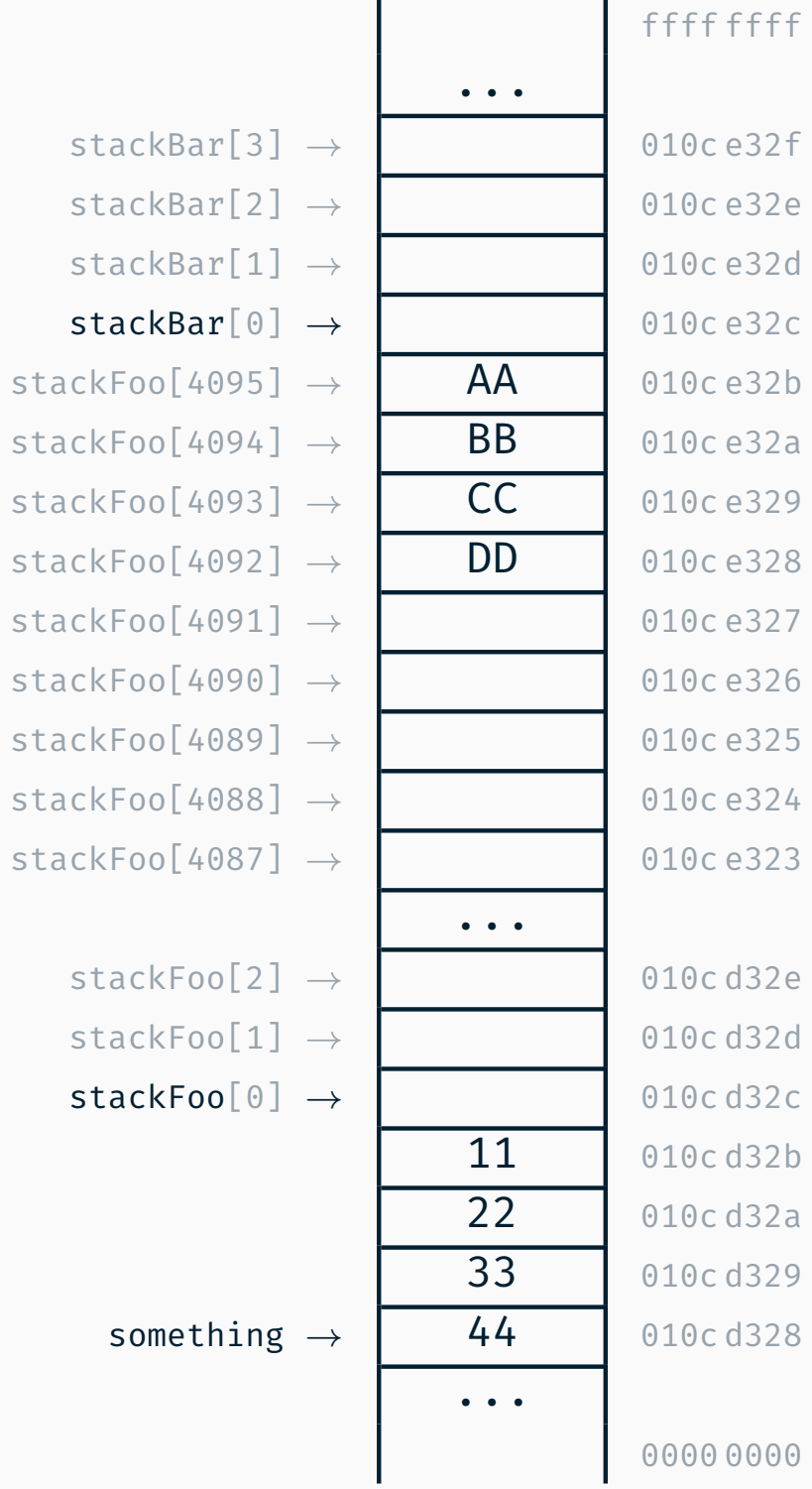
```
#define STACKSIZE 4096

int something=0x11223344;
char stackFoo[STACKSIZE];
char stackBar[STACKSIZE];
```



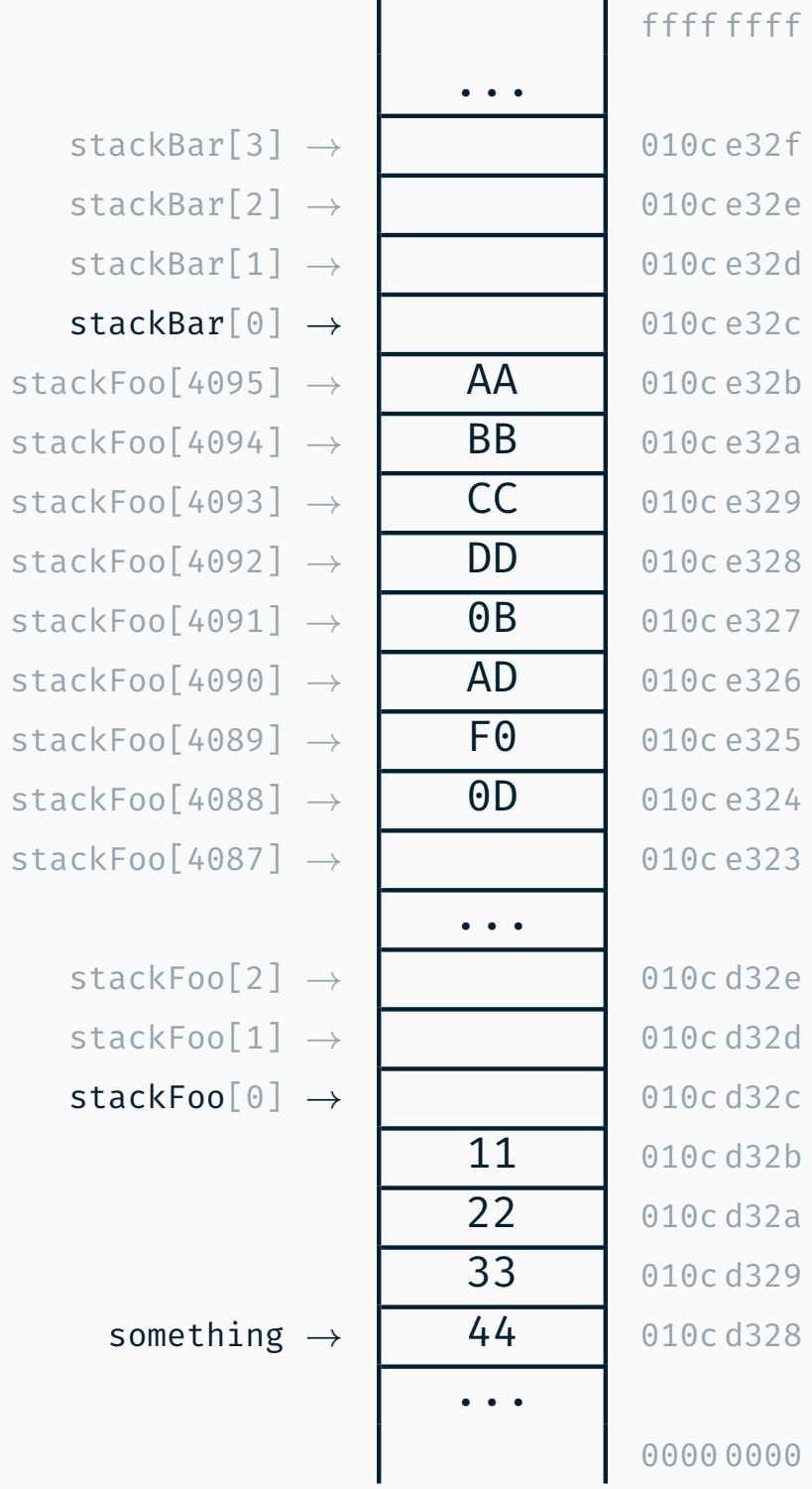
```
#define STACKSIZE 4096

int something=0x11223344;
char stackFoo[STACKSIZE];
char stackBar[STACKSIZE];

void* tos = &(stackFoo[4092]);
void** esp = (void**) tos;
```



```
#define STACKSIZE 4096

int something=0x11223344;
char stackFoo[STACKSIZE];
char stackBar[STACKSIZE];

void* tos = &(stackFoo[4092]);
void** esp = (void**) tos;

*esp = 0xAABBCCDD;
```



```
#define STACKSIZE 4096

int something=0x11223344;
char stackFoo[STACKSIZE];
char stackBar[STACKSIZE];

void* tos = &(stackFoo[4092]);
void** esp = (void**) tos;

*esp = 0xAABBCCDD;

esp--;
*esp = 0xBADF00D;
```



```
#define STACKSIZE 4096

int something=0x11223344;
char stackFoo[STACKSIZE];
char stackBar[STACKSIZE];

void* tos = &(stackFoo[4092]);
void** esp = (void**) tos;

*esp = 0xAABBCCDD;

esp--;
*esp = 0xBADF00D;

// ...

// Anschliessend:
// esp in toc schreiben
```