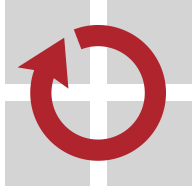
Übung zu Betriebssysteme

Interruptbehandlung

09. & 12. November 2018

Andreas Ziegler
Bernhard Heinloth

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

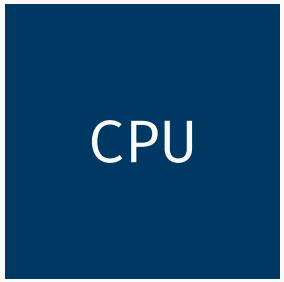


Lehrstuhl für Verteilte Systeme
und Betriebssysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

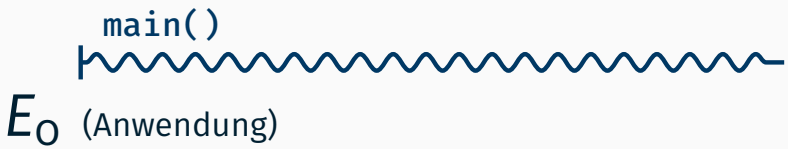
TECHNISCHE FAKULTÄT

Interrupts und Traps


CPU

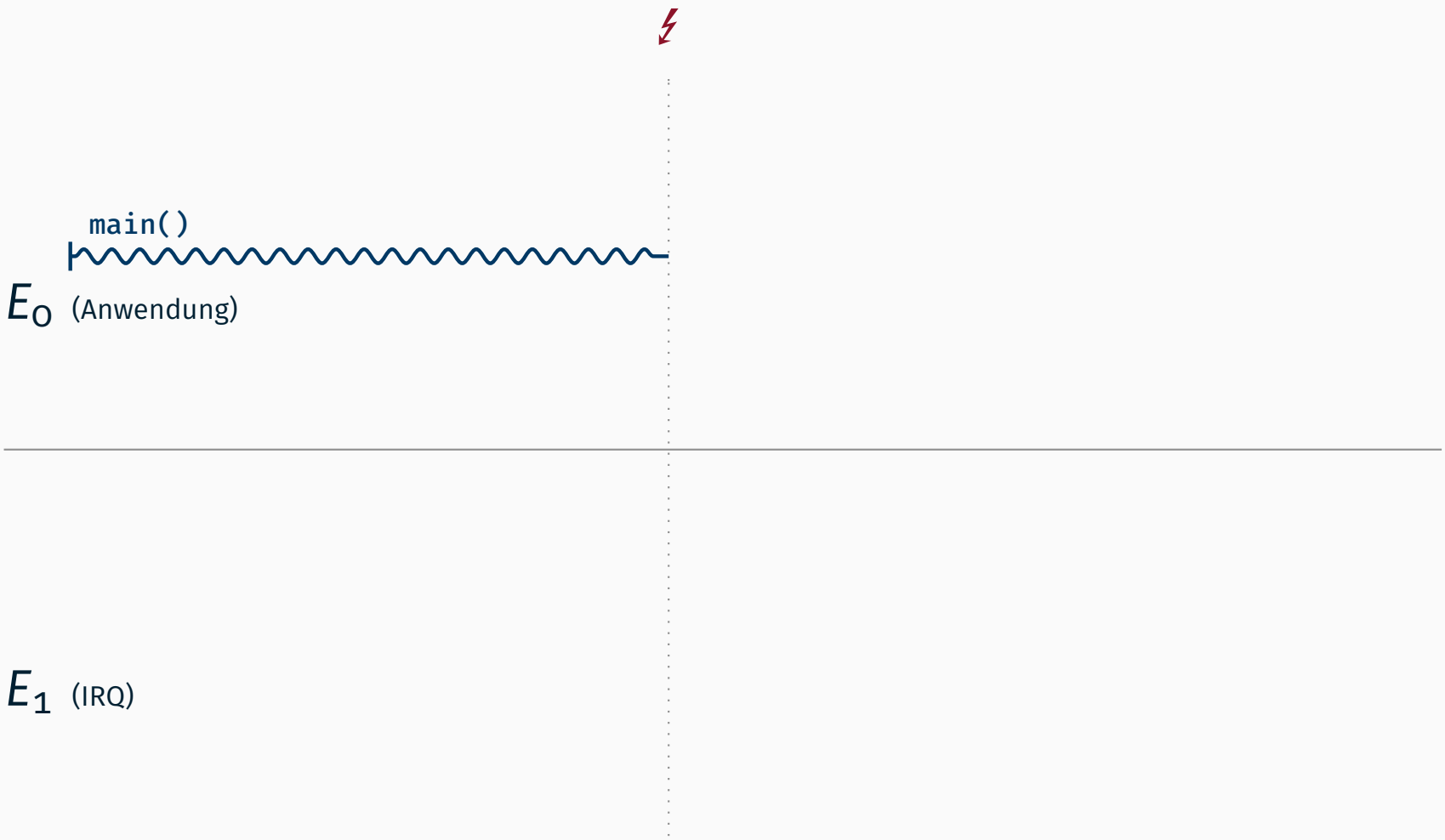

CPU

Unterbrechung (Anwendungssicht)

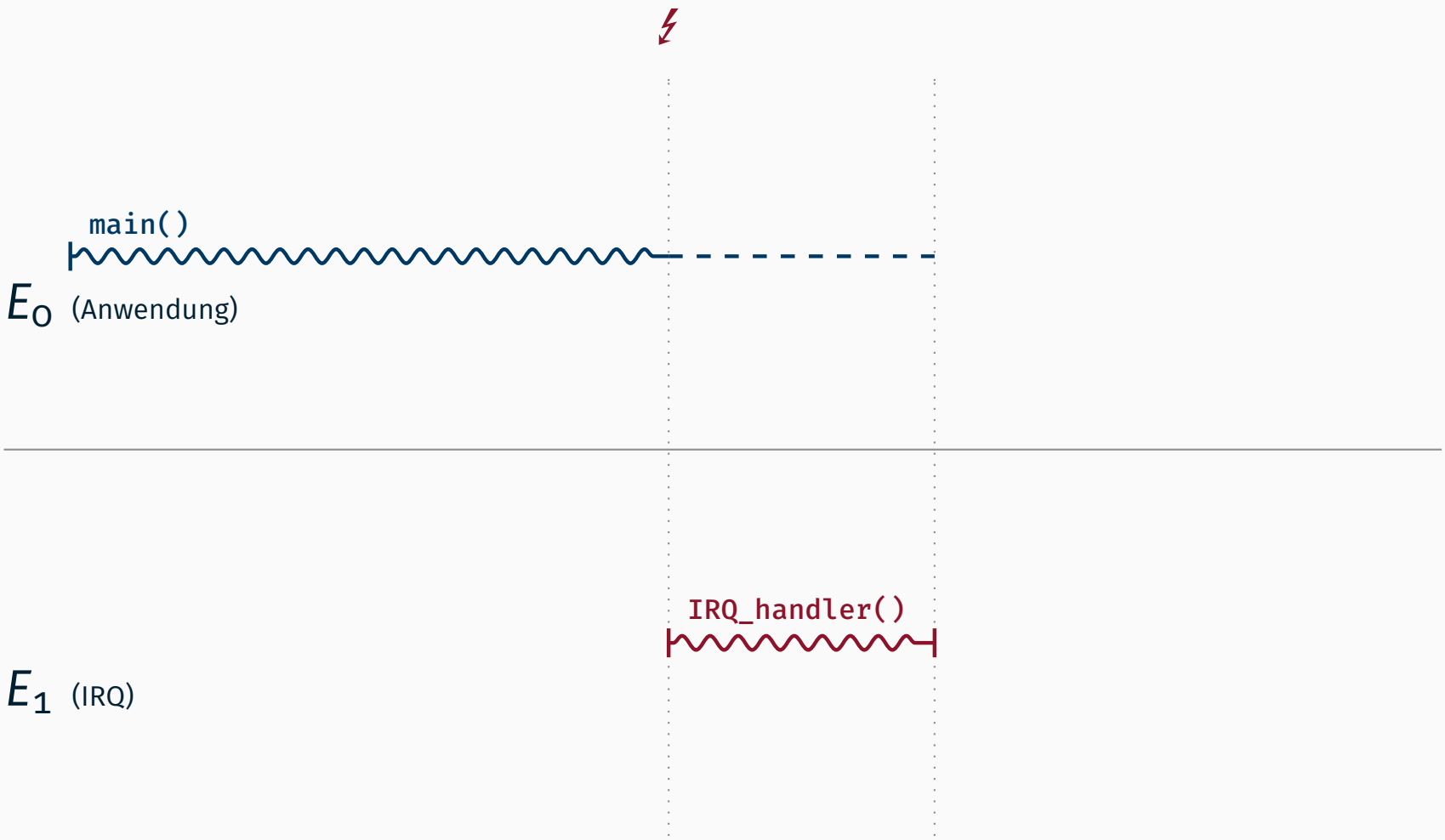


E_1 (IRQ)

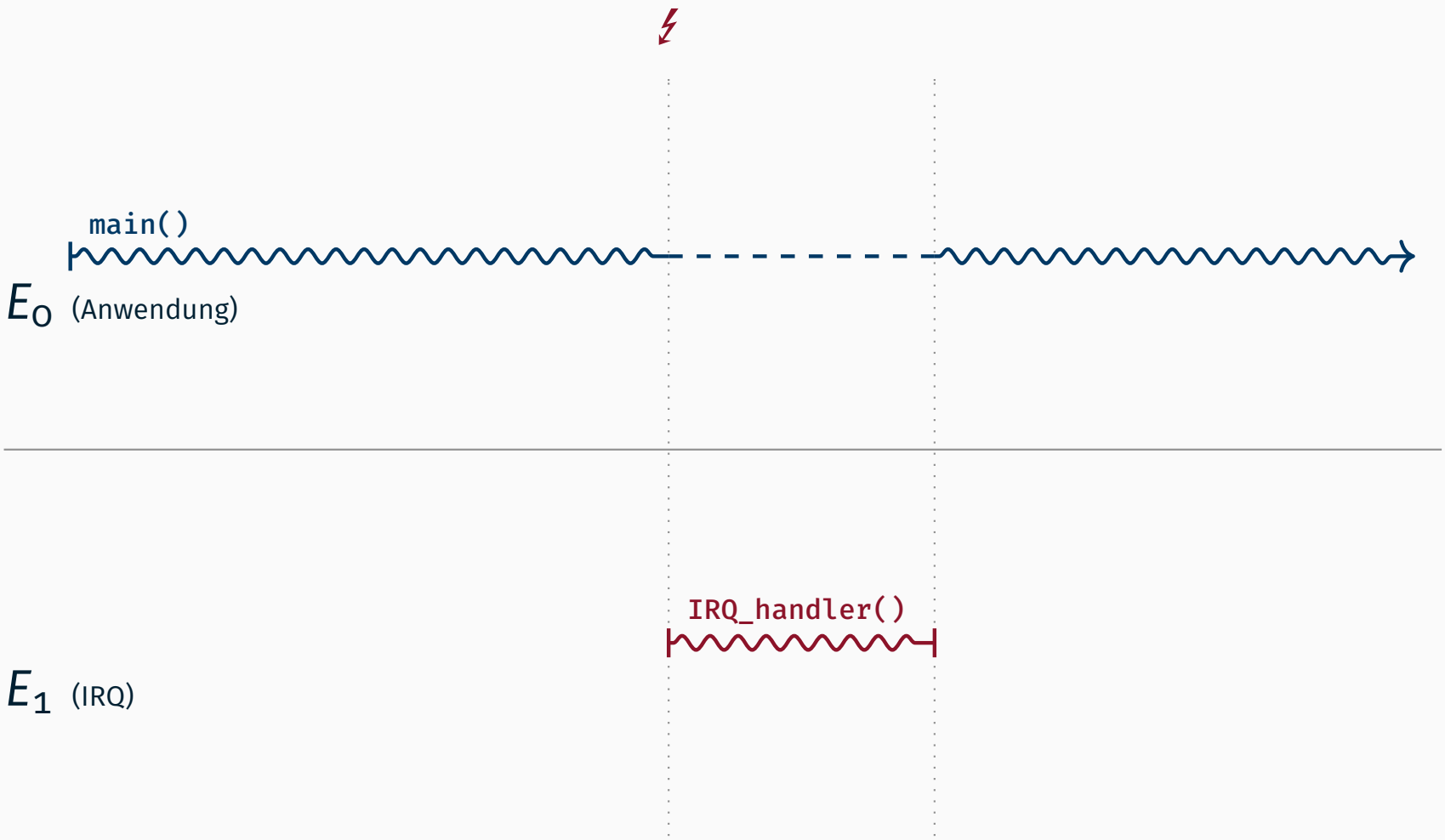
Unterbrechung (Anwendungssicht)



Unterbrechung (Anwendungssicht)



Unterbrechung (Anwendungssicht)

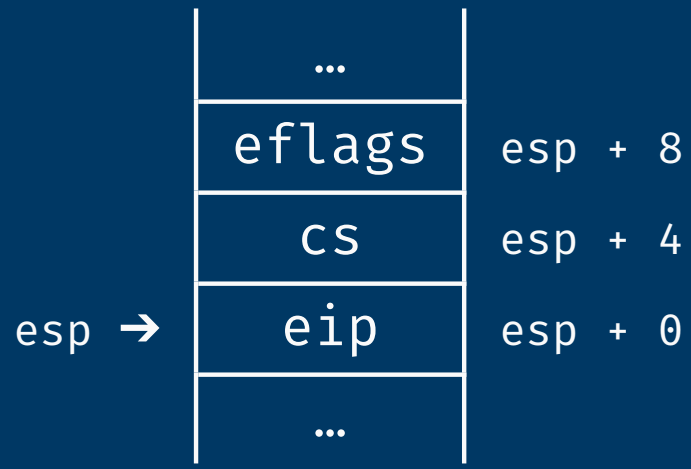


Minimaler zu sichernder Zustand?

Minimaler zu sichernder Zustand?

- CPU sichert automatisch

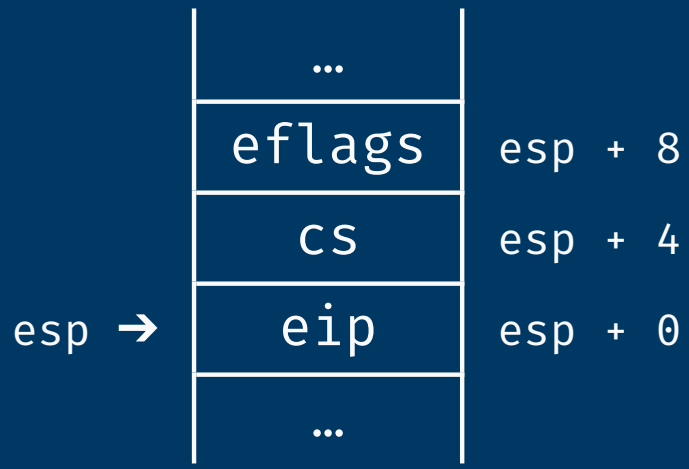
eflags Condition Codes
cs Aktuelles Code Segment
eip Programmzeiger/Rücksprungadresse



Minimaler zu sichernder Zustand?

- CPU sichert automatisch

  eflags Condition Codes
  cs Aktuelles Code Segment
  eip Programmzeiger/Rücksprungadresse



- Wiederherstellung des ursprünglichen Prozessorzustandes durch Befehl `iret`

Unterbrechungsbehandlung

```
;; Assembler

irq_entry:
  ;; Behandle IRQ


  iret
```

Unterbrechungsbehandlung

unter Verwendung einer Hochsprache

```
;; Assembler

irq_entry:
  ;; Behandle IRQ
  ;; in Hochsprache
  call guardian
  iret
```

Unterbrechungsbehandlung

unter Verwendung einer Hochsprache

```
;; Assembler

irq_entry:
  ;; Behandle IRQ
  ;; in Hochsprache
  call guardian
  iret
```

```
// C++


void guardian()
{
    // Magie.
}
```

Unterbrechungsbehandlung

unter Verwendung einer Hochsprache

```
;; Assembler

irq_entry:
  ;; Behandle IRQ
  ;; in Hochsprache
  call guardian
  iret
```

```
// C++


void guardian()
{
    // Magie.
}
```

```
heinloth:~/oostubs$ make
LD          build/system
./build/_startup.o: In Funktion »irq_entry«:
./boot/startup.asm:(.text+0x6f): Warnung: undefinierter
    Verweis auf »guardian«
```

Unterbrechungsbehandlung

unter Verwendung einer Hochsprache

```
;; Assembler

irq_entry:
  ;; Behandle IRQ
  ;; in Hochsprache
  call guardian
  iret
```

```
// C++


void guardian()
{
    // Magie.
}
```

```
heinloth:~/oostubs$ objdump -d build/guardian.o
Disassembly of section .text:

00000000 <_Z8guardianv>:
   0:  f3 c3                       repz ret
```

Unterbrechungsbehandlung

unter Verwendung einer Hochsprache

```
;; Assembler

irq_entry:
  ;; Behandle IRQ
  ;; in Hochsprache
  call guardian
  iret
```

```
// C++ (mit C Linkage)

extern "C"
void guardian()
{
    // Magie.
}
```

Unterbrechungsbehandlung

unter Verwendung einer Hochsprache

```
;; Assembler

irq_entry:
  ;; Behandle IRQ
  ;; in Hochsprache
  call guardian
  iret
```

```
// C++ (mit C Linkage)

extern "C"
void guardian()
{
    // Magie.
}
```

```
01 heinloth:~/oostubs$ objdump -d build/guardian.o
02 Disassembly of section .text:
03
04 00000000 <guardian>:
05    0:  f3 c3                       repz ret
```

Was ist mit den restlichen Registern?

Was ist mit den restlichen Registern?

- Müssen durch den Interrupthandler selbst gesichert werden.
  - entweder im Assembler-Teil oder
  - der Compiler generiert bereits entsprechend Code

Was ist mit den restlichen Registern?

- Müssen durch den Interrupthandler selbst gesichert werden.
  - entweder im Assembler-Teil oder
  - der Compiler generiert bereits entsprechend Code
- Kontextsicherung beim Aufruf von Funktionen

Was ist mit den restlichen Registern?

- Müssen durch den Interrupthandler selbst gesichert werden.
  - entweder im Assembler-Teil oder
  - der Compiler generiert bereits entsprechend Code
- Kontextsicherung beim Aufruf von Funktionen
  1. Aufrufende Funktion sichert alle Register, die sie braucht

Was ist mit den restlichen Registern?

- Müssen durch den Interrupthandler selbst gesichert werden.
  - entweder im Assembler-Teil oder
  - der Compiler generiert bereits entsprechend Code
- Kontextsicherung beim Aufruf von Funktionen
  1. Aufrufende Funktion sichert alle Register, die sie braucht
  2. Aufgerufene Funktion sichert alle Register, die sie verändert

Was ist mit den restlichen Registern?

- Müssen durch den Interrupthandler selbst gesichert werden.
  - entweder im Assembler-Teil oder
  - der Compiler generiert bereits entsprechend Code
- Kontextsicherung beim Aufruf von Funktionen
  1. Aufrufende Funktion sichert alle Register, die sie braucht
  2. Aufgerufene Funktion sichert alle Register, die sie verändert
  3. Ein Teil der Register wird vom Aufrufer, ein anderer Teil vom Aufgerufenen gesichert

Was ist mit den restlichen Registern?

- Müssen durch den Interrupthandler selbst gesichert werden.
  - entweder im Assembler-Teil oder
  - der Compiler generiert bereits entsprechend Code
- Kontextsicherung beim Aufruf von Funktionen
  1. Aufrufende Funktion sichert alle Register, die sie braucht
  2. Aufgerufene Funktion sichert alle Register, die sie verändert
  3. Ein Teil der Register wird vom Aufrufer, ein anderer Teil vom Aufgerufenen gesichert
- In der Praxis wird Variante 3 verwendet
  - Aufteilung ist grundsätzlich compilerspezifisch
  - CPU-Hersteller definiert jedoch Konventionen, damit Interoperabilität auf Binärcodeebene sichergestellt ist

Aufteilung der Register in zwei Gruppen

Aufteilung der Register in zwei Gruppen

- Flüchtige Register (*scratch registers*)
  - Compiler geht davon aus, dass Unterprogramm den Inhalt verändert
  - Aufrufer muss Inhalt gegebenenfalls sichern
  - Beim x86 sind `eax`, `ecx`, `edx` und `eflags` als flüchtig definiert

Aufteilung der Register in zwei Gruppen

- Flüchtige Register (*scratch registers*)
  - Compiler geht davon aus, dass Unterprogramm den Inhalt verändert
  - Aufrufer muss Inhalt gegebenenfalls sichern
  - Beim x86 sind **`eax`**, **`ecx`**, **`edx`** und **`eflags`** als flüchtig definiert
- Nicht-flüchtige Register (*non-scratch registers*)
  - Compiler geht davon aus, dass der Inhalt durch Unterprogramm nicht verändert wird
  - Aufgerufene Funktion muss Inhalt gegebenenfalls sichern
  - Beim x86 sind alle sonstigen Register als nicht-flüchtig definiert: **`ebx`**, **`esi`**, **`edi`**, **`ebp`** und **`esp`**

Aufteilung der Register in zwei Gruppen

- Flüchtige Register (*scratch registers*)
  - Compiler geht davon aus, dass Unterprogramm den Inhalt verändert
  - Aufrufer muss Inhalt gegebenenfalls sichern
  - Beim x86 sind **`eax`**, **`ecx`**, **`edx`** und **`eflags`** als flüchtig definiert
- Nicht-flüchtige Register (*non-scratch registers*)
  - Compiler geht davon aus, dass der Inhalt durch Unterprogramm nicht verändert wird
  - Aufgerufene Funktion muss Inhalt gegebenenfalls sichern
  - Beim x86 sind alle sonstigen Register als nicht-flüchtig definiert: **`ebx`**, **`esi`**, **`edi`**, **`ebp`** und **`esp`**

Interrupt-Handler müssen auch flüchtige Register sichern!

Unterbrechungsbehandlung (Kontextsicherung)

```
;; Assembler

irq_entry:

  call guardian

  iret
```

Unterbrechungsbehandlung (Kontextsicherung)

```
;; Assembler

irq_entry:
  ;; Kontext sichern
  push edx
  push ecx
  push eax


  call guardian


  pop eax
  pop ecx
  pop edx
  iret
```

Unterbrechungsbehandlung (Kontext)

```
;; Assembler

irq_entry:
  ;; Kontext sichern
  push edx
  push ecx
  push eax


  call guardian


  pop eax
  pop ecx
  pop edx
  iret
```

```
// C++


extern "C"
void guardian()
{
    // Magie.
}
```

Unterbrechungsbehandlung (Kontext)

```
;; Assembler

irq_entry:
  ;; Kontext sichern
  push edx
  push ecx
  push eax


  call guardian


  pop eax
  pop ecx
  pop edx
  iret
```

```
// C++

struct irq_context {









} __attribute__((packed));

extern "C"
void guardian(irq_context* c)
{
    // Magie.
}
```

Unterbrechungsbehandlung (Kontext)

```
;; Assembler

irq_entry:
  ;; Kontext sichern
  push edx
  push ecx
  push eax


  call guardian


  pop eax
  pop ecx
  pop edx
  iret
```

```
// C++

struct irq_context {
    uint32_t eax;
    uint32_t ecx;
    uint32_t edx;



} __attribute__((packed));

extern "C"
void guardian(irq_context* c)
{
    // Magie.
}
```

Unterbrechungsbehandlung (Kontext)

```assembler
;; Assembler

irq_entry:
  ;; Kontext sichern
  push edx
  push ecx
  push eax
  ;; Pointer auf Stack
  push esp
  call guardian
  ;; Stack aufräumen
  add esp, 4
  pop eax
  pop ecx
  pop edx
  iret
```

```cpp
// C++

struct irq_context {
    uint32_t eax;
    uint32_t ecx;
    uint32_t edx;



} __attribute__((packed));

extern "C"
void guardian(irq_context* c)
{
    // Magie.
}
```

Unterbrechungsbehandlung (Kontext)

```
;; Assembler

irq_entry:
  ;; Kontext sichern
  push edx
  push ecx
  push eax
  ;; Pointer auf Stack
  push esp
  call guardian
  ;; Stack aufräumen
  add esp, 4
  pop eax
  pop ecx
  pop edx
  iret
```

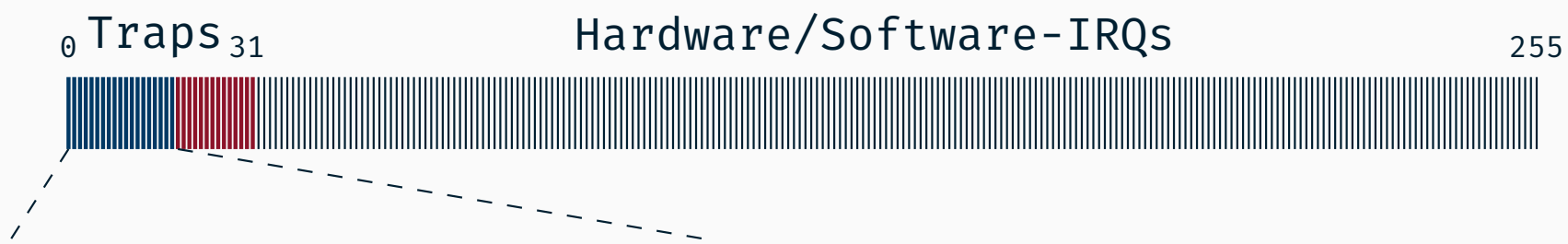
```
// C++

struct irq_context {
    uint32_t eax;
    uint32_t ecx;
    uint32_t edx;
    uint32_t eip;
    uint32_t cs;
    uint32_t eflags;
} __attribute__((packed));

extern "C"
void guardian(irq_context* c)
{
    // Magie.
}
```


CPU


0-255
CPU

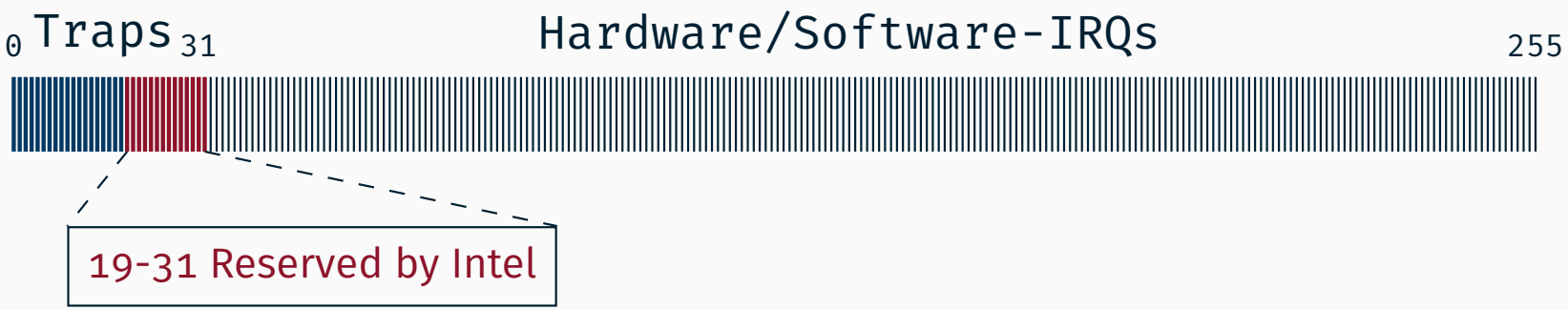
Interruptvektoren (x86)

Interruptvektoren (x86)

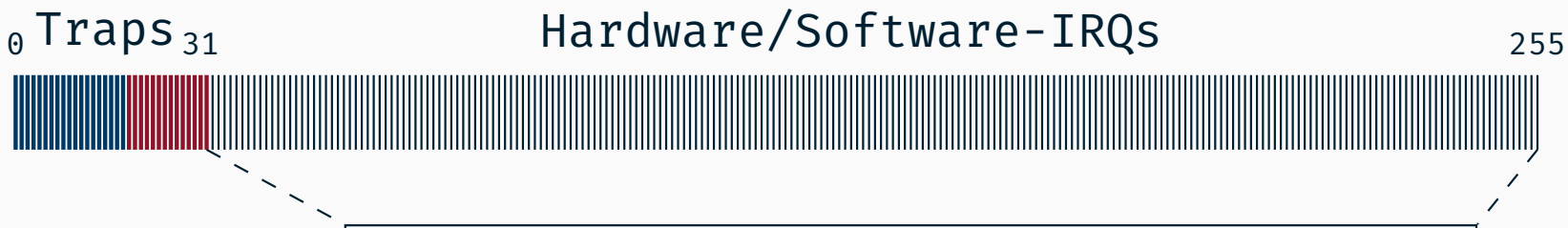


0	**Division-by-Zero**
1	Debug Exception
2	Non-Maskable Interrupt(NMI)
3	**Breakpoint (INT 3)**
4	Overflow Exception
5	Bound Exception
6	**Invalid Opcode**
7	FPU not Available
8	Double Fault
9	Coprocessor Segment Overrun
10	Invalid TSS
11	Segment not Present
12	Stack Exception
13	**General Protection Fault**
14	**Page Fault**
15	*Reserved*
16	Floating-Point Error
17	Alignment Check
18	Machine Check

Interruptvektoren (x86)



Interruptvektoren (x86)



32-255 Einträge für IRQs

- Softwareauslösung mit `int <vec#>`
- Hardwareauslösung durch externe Geräte

Interruptvektoren (x86)

Kann durch Prozessorbefehle maskiert werden

- `cli` (Clear Interrupt Flag) Interruptleitung sperren
- `sti` (Set Interrupt Flag) Interruptleitung freigeben

Unterbrechungsbehandlung (für Vektor 6)

```
;; Assembler

irq_entry:
  ;; Kontext sichern
  push edx
  push ecx
  push eax
  ;; Pointer auf Stack
  push esp


  call guardian
  ;; Stack aufräumen
  add esp, 4
  pop eax
  pop ecx
  pop edx
  iret
```

```
// C++

extern "C"
void guardian(

    irq_context* c
  )
{
    // Magie:









}
```

Unterbrechungsbehandlung (für Vektor 6)

```
;; Assembler

irq_entry:
  ;; Kontext sichern
  push edx
  push ecx
  push eax
  ;; Pointer auf Stack
  push esp


  call guardian
  ;; Stack aufräumen
  add esp, 4
  pop eax
  pop ecx
  pop edx
  iret
```

```
// C++

extern "C"
void guardian(

    irq_context* c
  )
{
    // Magie:
    switch (      ){
        case KBD:
            kbd.magic();
            break;
        case TMR:
            tmr.magic();
            break;
    }
}
```

Unterbrechungsbehandlung (für Vektor 6)

```
;; Assembler

irq_entry:
  ;; Kontext sichern
  push edx
  push ecx
  push eax
  ;; Pointer auf Stack
  push esp


  call guardian
  ;; Stack aufräumen
  add esp, 4
  pop eax
  pop ecx
  pop edx
  iret
```

```
// C++

extern "C"
void guardian(

    irq_context* c
  )
{
    // Magie:
    switch (vector){
        case KBD:
            kbd.magic();
            break;
        case TMR:
            tmr.magic();
            break;
    }
}
```

Unterbrechungsbehandlung (für Vektor 6)

```
;; Assembler

irq_entry:
  ;; Kontext sichern
  push edx
  push ecx
  push eax
  ;; Pointer auf Stack
  push esp


  call guardian
  ;; Stack aufräumen
  add esp, 4
  pop eax
  pop ecx
  pop edx
  iret
```

```
// C++

extern "C"
void guardian(
    uint32_t vector,
    irq_context* c
  )
{
    // Magie:
    switch (vector){
        case KBD:
            kbd.magic();
            break;
        case TMR:
            tmr.magic();
            break;
    }
}
```

Unterbrechungsbehandlung (für Vektor 6)

```
;; Assembler

irq_entry_6:
  ;; Kontext sichern
  push edx
  push ecx
  push eax
  ;; Pointer auf Stack
  push esp


  call guardian
  ;; Stack aufräumen
  add esp, 4
  pop eax
  pop ecx
  pop edx
  iret
```

```
// C++

extern "C"
void guardian(
    uint32_t vector,
    irq_context* c
  )
{
    // Magie:
    switch (vector){
        case KBD:
            kbd.magic();
            break;
        case TMR:
            tmr.magic();
            break;
    }
}
```

Unterbrechungsbehandlung (für Vektor 6)

```
;; Assembler

irq_entry_6:
  ;; Kontext sichern
  push edx
  push ecx
  push eax
  ;; Pointer auf Stack
  push esp
  ;; Interruptnummer
  push 6
  call guardian
  ;; Stack aufräumen
  add esp, 4
  pop eax
  pop ecx
  pop edx
  iret
```

```
// C++

extern "C"
void guardian(
    uint32_t vector,
    irq_context* c
  )
{
    // Magie:
    switch (vector){
        case KBD:
            kbd.magic();
            break;
        case TMR:
            tmr.magic();
            break;
    }
}
```

Unterbrechungsbehandlung (für Vektor 6)

```
;; Assembler

irq_entry_6:
  ;; Kontext sichern
  push edx
  push ecx
  push eax
  ;; Pointer auf Stack
  push esp
  ;; Interruptnummer
  push 6
  call guardian
  ;; Stack aufräumen
  add esp, 4 * 2
  pop eax
  pop ecx
  pop edx
  iret
```

```
// C++

extern "C"
void guardian(
    uint32_t vector,
    irq_context* c
  )
{
    // Magie:
    switch (vector){
        case KBD:
            kbd.magic();
            break;
        case TMR:
            tmr.magic();
            break;
    }
}
```

Unterbrechungsbehandlung (Binden)

```
;; Assembler

irq_entry_6:
  ;; Kontext sichern
  push edx
  push ecx
  push eax
  ;; Pointer auf Stack
  push esp
  ;; Interruptnummer
  push 6
  call guardian
  ;; Stack aufräumen
  add esp, 4 * 2
  pop eax
  pop ecx
  pop edx
  iret
```

```
01 heinloth:~/oostubs$ make
02 DEP     dep/guardian.d
03 ASM     build/_startup.o
04 CXX     build/guardian.o
05 LD      build/system
```

Unterbrechungsbehandlung (Binden)

```
;; Assembler

irq_entry_6:
  ;; Kontext sichern
  push edx
  push ecx
  push eax
  ;; Pointer auf Stack
  push esp
  ;; Interruptnummer
  push 6
  call 0x1004f40 <guardian>
  ;; Stack aufräumen
  add esp, 4 * 2
  pop eax
  pop ecx
  pop edx
  iret
```

```
01 heinloth:~/oostubs$ make
02 DEP     dep/guardian.d
03 ASM     build/_startup.o
04 CXX     build/guardian.o
05 LD      build/system
```

Unterbrechungsbehandlung (Binden)

```
;; Assembler

irq_entry_6:
  ;; Kontext sichern
  push edx
  push ecx
  push eax
  ;; Pointer auf Stack
  push esp
  ;; Interruptnummer
  push 6
  call 0x1004f40 <guardian>
  ;; Stack aufräumen
  add esp, 4 * 2
  pop eax
  pop ecx
  pop edx
  iret
```

```
01 heinloth:~/oostubs$ make
02 DEP     dep/guardian.d
03 ASM     build/_startup.o
04 CXX     build/guardian.o
05 LD      build/system
```

Speicheradressen beim **Binden**:

100 4f40 <guardian>

Unterbrechungsbehandlung (Binden)

```
Maschinencode



52
51
50

54

6a 06
e8 3d 4e 00 00

83 c4 08
58
59
5a
cf
```

```
01 heinloth:~/oostubs$ make
02 DEP     dep/guardian.d
03 ASM     build/_startup.o
04 CXX     build/guardian.o
05 LD      build/system
```

Speicheradressen beim **Binden**:

```
1004f40 <guardian>
```

Unterbrechungsbehandlung (Binden)

```
Speicher  Maschinencode
addresse


10000f8:  52
10000f9:  51
10000fa:  50

10000fb:  54

10000fc:  6a 06
10000fe:  e8 3d 4e 00 00

1000103:  83 c4 08
1000106:  58
1000107:  59
1000108:  5a
1000109:  cf
```

```
01 heinloth:~/oostubs$ make
02 DEP     dep/guardian.d
03 ASM     build/_startup.o
04 CXX     build/guardian.o
05 LD      build/system
```

Speicheradressen beim **Binden**:

`100 4f40 <guardian>`

Unterbrechungsbehandlung (Binden)

```
Speicher  Maschinencode
addresse


10000f8:  52
10000f9:  51
10000fa:  50

10000fb:  54

10000fc:  6a 06
10000fe:  e8 3d 4e 00 00

1000103:  83 c4 08
1000106:  58
1000107:  59
1000108:  5a
1000109:  cf
```

```
01 heinloth:~/oostubs$ make
02 DEP     dep/guardian.d
03 ASM     build/_startup.o
04 CXX     build/guardian.o
05 LD      build/system
```

Speicheradressen beim **Binden**:

```
100 4f40 <guardian>

100 00f8 <irq_entry_6>
```

Unterbrechungsbehandlungen

```
%macro IRQ 1
align 8
irq_entry_%1:
  ;; Kontext sichern
  push edx
  push ecx
  push eax
  ;; Pointer auf Stack
  push esp
  ;; Interruptnummer
  push %1
  call guardian
  ;; Stack aufräumen
  add esp, 8
  pop eax
  pop ecx
  pop edx
  iret
%endmacro
```

```
01 heinloth:~/oostubs$ make
02 DEP     dep/guardian.d
03 ASM     build/_startup.o
04 CXX     build/guardian.o
05 LD      build/system
```

Speicheradressen beim **Binden**:

100 4f40 <guardian>

100 00f8 <irq_entry_6>

Unterbrechungsbehandlung

```
%macro IRQ 1
align 8
irq_entry_%1:
  ;; Kontext sichern
  push edx
  push ecx
  push eax
  ;; Pointer auf Stack
  push esp
  ;; Interruptnummer
  push %1
  call guardian
  ;; Stack aufräumen
  add esp, 8
  pop eax
  pop ecx
  pop edx
  iret
%endmacro
```

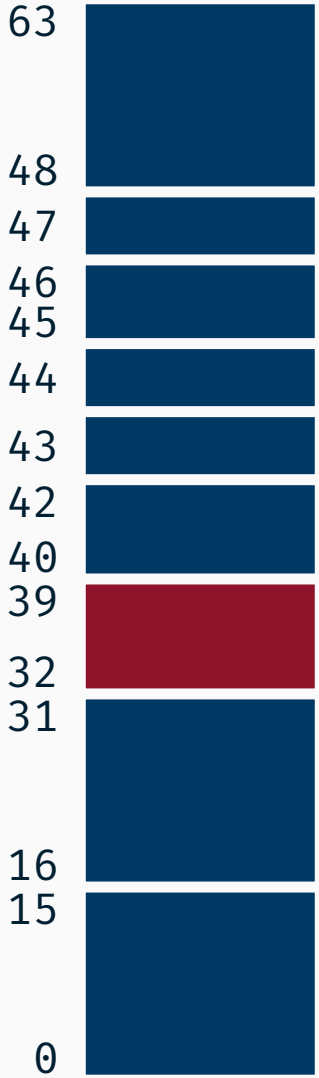
```
01 heinloth:~/oostubs$ make
02 DEP     dep/guardian.d
03 ASM     build/_startup.o
04 CXX     build/guardian.o
05 LD      build/system
```

Speicheradressen beim **Binden**:

```
100 4f40 <guardian>
         ...
100 00f8 <irq_entry_6>
100 00e0 <irq_entry_5>
100 00c8 <irq_entry_4>
100 00b0 <irq_entry_3>
100 0098 <irq_entry_2>
100 0080 <irq_entry_1>
100 0068 <irq_entry_0>
```

Woher weiß die CPU wo die entsprechende Unterbrechungsbehandlung liegt?

Interrupt Deskriptor



Offset (high): *oberer* Teil der Einsprungsaddresse für die Interruptbehandlung (z.B. `irq_entry_6`)

Present: Eintrag aktiv (`1`) oder inaktiv (`0`)

Descriptor Privilege Level

Storage Segment: `0` für Interrupt und Traps

Mode: 16-bit (`0`) oder 32-bit (`1`)

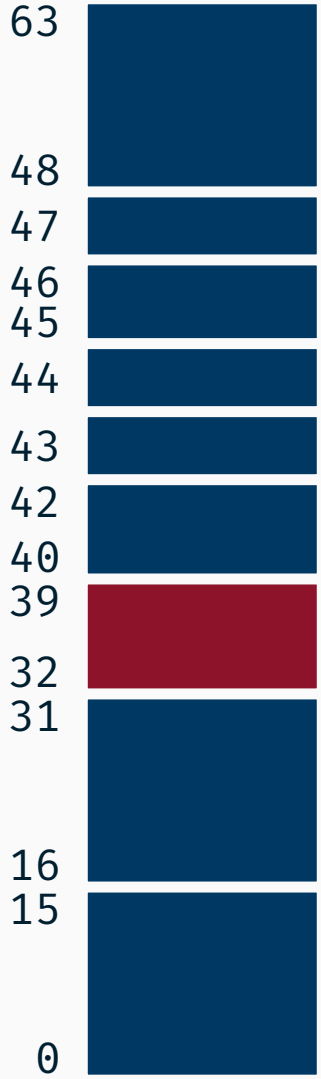
Type: Task (`5`), Interrupt (`6`) oder Trap (`7`)?

Unused – muss `0` sein

Selector: Codesegment, in das beim Interrupt gewechselt wird (i.d.R. Kernel-Codesegment)

Offset (low): *unterer* Teil der Einsprungsaddresse für die Interruptbehandlung

Interrupt Deskriptor (Beispiel)



Offset (high): *oberer* Teil der Einsprungsaddresse für die Interruptbehandlung (z.B. `irq_entry_6`)

Present: Eintrag aktiv (`1`) oder inaktiv (`0`)

Descriptor Privilege Level

Storage Segment: `0` für Interrupt und Traps

Mode: 16-bit (`0`) oder 32-bit (`1`)

Type: Task (`5`), Interrupt (`6`) oder Trap (`7`)?

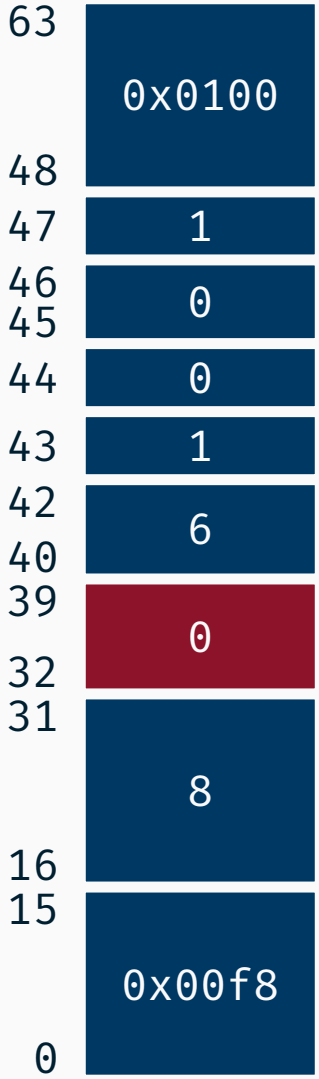
Unused – muss `0` sein

Selector: Codesegment, in das beim Interrupt gewechselt wird (i.d.R. Kernel-Codesegment)

Offset (low): *unterer* Teil der Einsprungsaddresse für die Interruptbehandlung

für **`100 00f8 <irq_entry_6>`**

Interrupt Deskriptor (Beispiel)



Offset (high): *oberer* Teil der Einsprungsaddresse für die Interruptbehandlung (z.B. `irq_entry_6`)

Present: Eintrag aktiv (`1`) oder inaktiv (`0`)

Descriptor Privilege Level

Storage Segment: `0` für Interrupt und Traps

Mode: 16-bit (`0`) oder 32-bit (`1`)

Type: Task (`5`), Interrupt (`6`) oder Trap (`7`)?

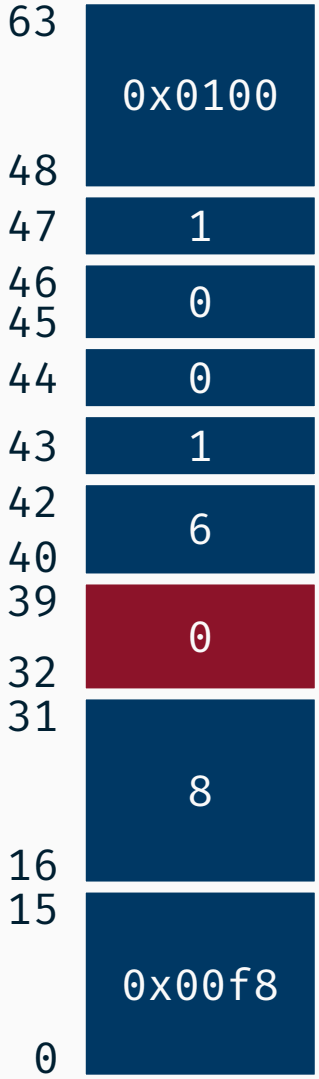
Unused – muss `0` sein

Selector: Codesegment, in das beim Interrupt gewechselt wird (i.d.R. Kernel-Codesegment)

Offset (low): *unterer* Teil der Einsprungsaddresse für die Interruptbehandlung

für **`100 00f8 <irq_entry_6>`**

Interrupt Deskriptor (Beispiel)



Offset (high): *oberer* Teil der Einsprungsaddresse für die Interruptbehandlung (z.B. `irq_entry_6`)

Present: Eintrag aktiv (`1`) oder inaktiv (`0`)

Descriptor Privilege Level

Storage Segment: `0` für Interrupt und Traps

Mode: 16-bit (`0`) oder 32-bit (`1`)

Type: Task (`5`), Interrupt (`6`) oder Trap (`7`)?

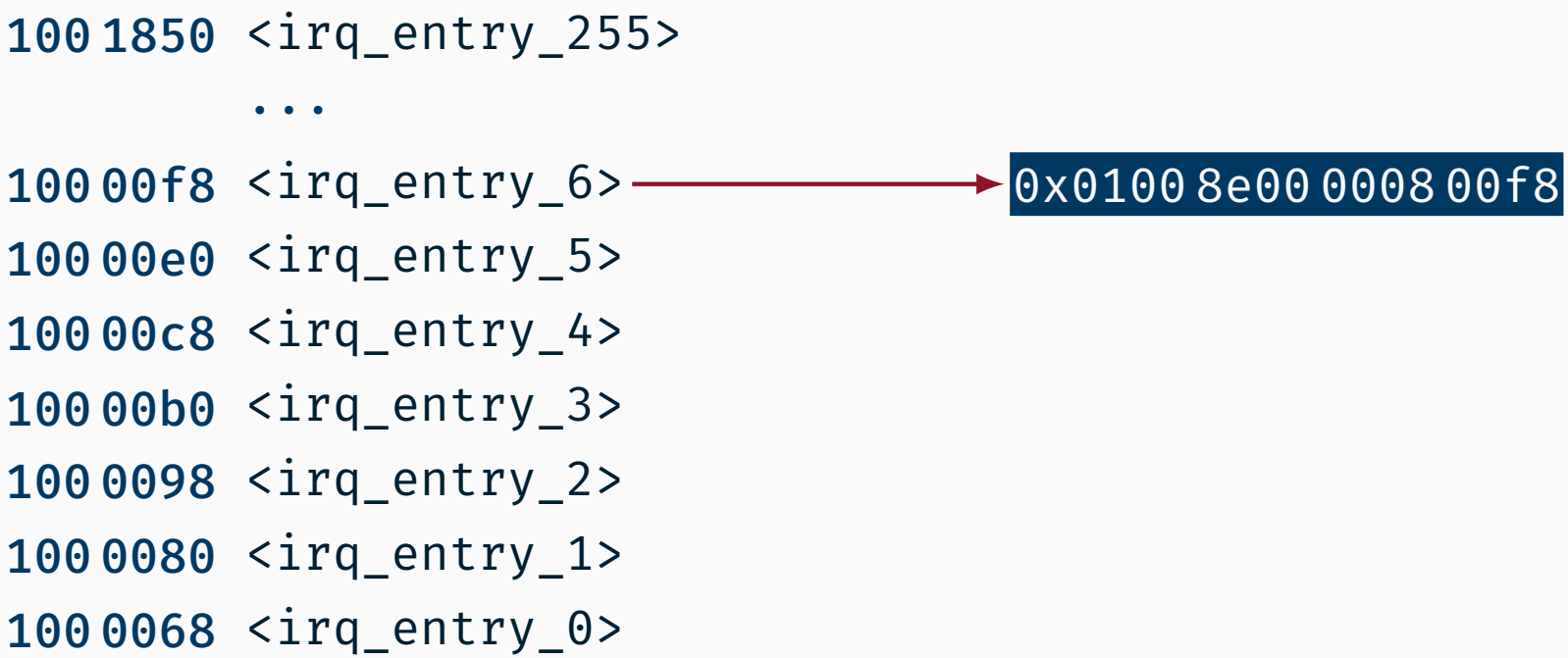
Unused – muss `0` sein

Selector: Codesegment, in das beim Interrupt gewechselt wird (i.d.R. Kernel-Codesegment)

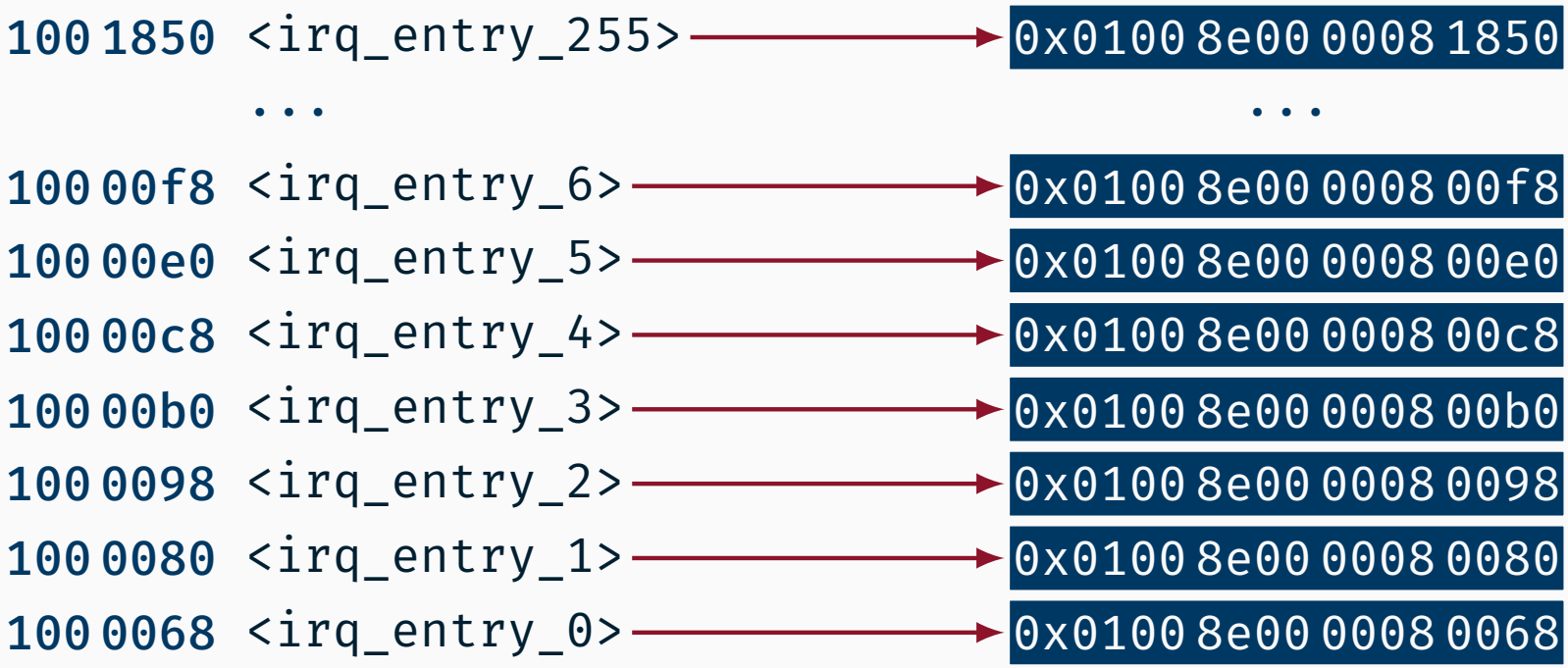
Offset (low): *unterer* Teil der Einsprungsaddresse für die Interruptbehandlung

für **`100 00f8 <irq_entry_6>`** → **`0x0100 8e00 0008 00f8`**

Interrupt Deskriptor Tabelle (IDT)



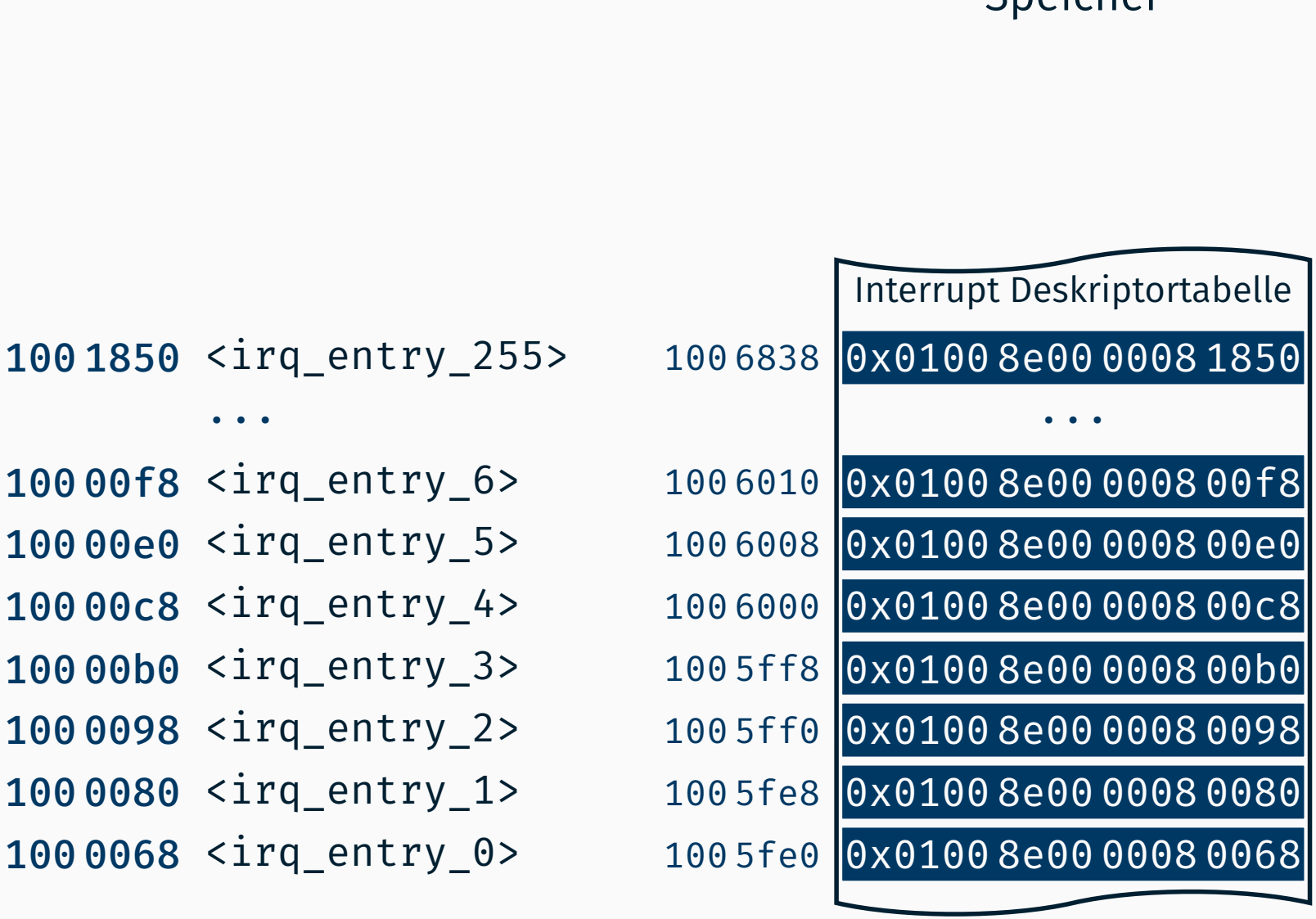
Interrupt Deskriptor Tabelle (IDT)



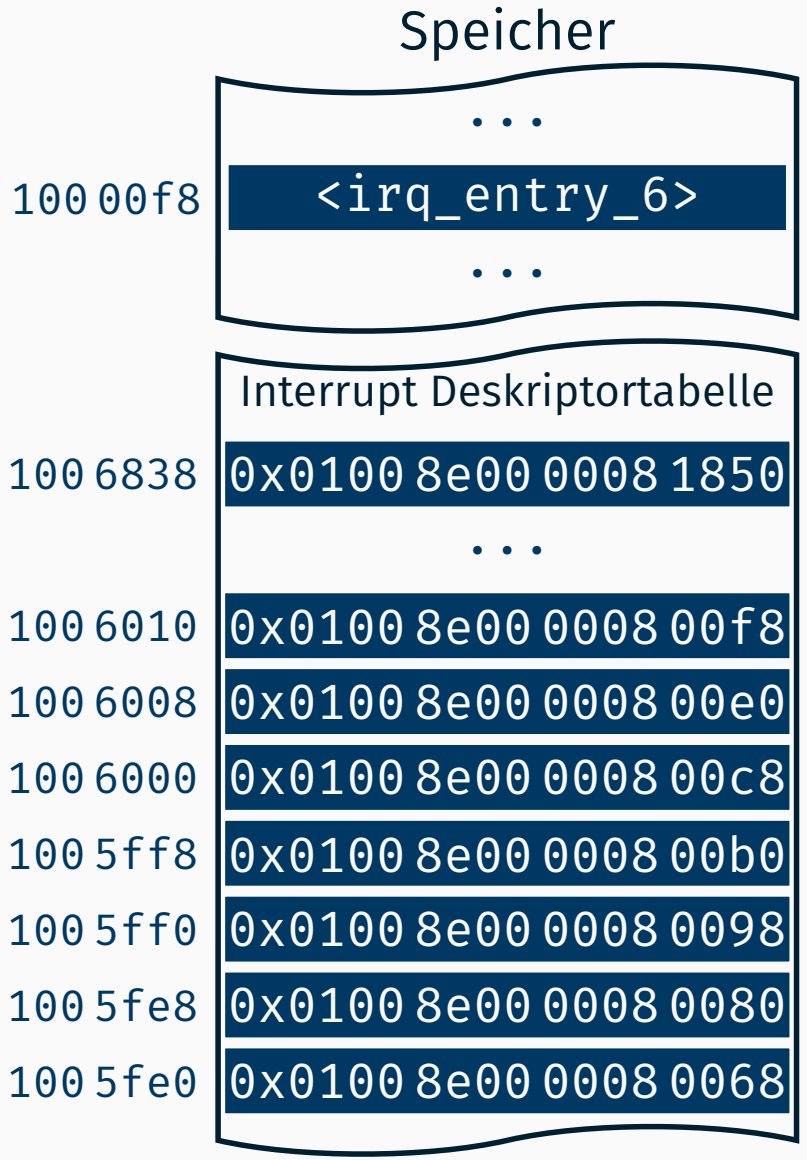
Interrupt Deskriptor Tabelle (IDT)

```
100 1850 <irq_entry_255>
         ...
100 00f8 <irq_entry_6>
100 00e0 <irq_entry_5>
100 00c8 <irq_entry_4>
100 00b0 <irq_entry_3>
100 0098 <irq_entry_2>
100 0080 <irq_entry_1>
100 0068 <irq_entry_0>
```

Interrupt Deskriptor Tabelle (IDT)

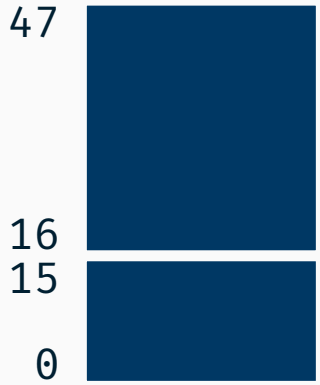


Interrupt Deskriptor Tabelle (IDT)



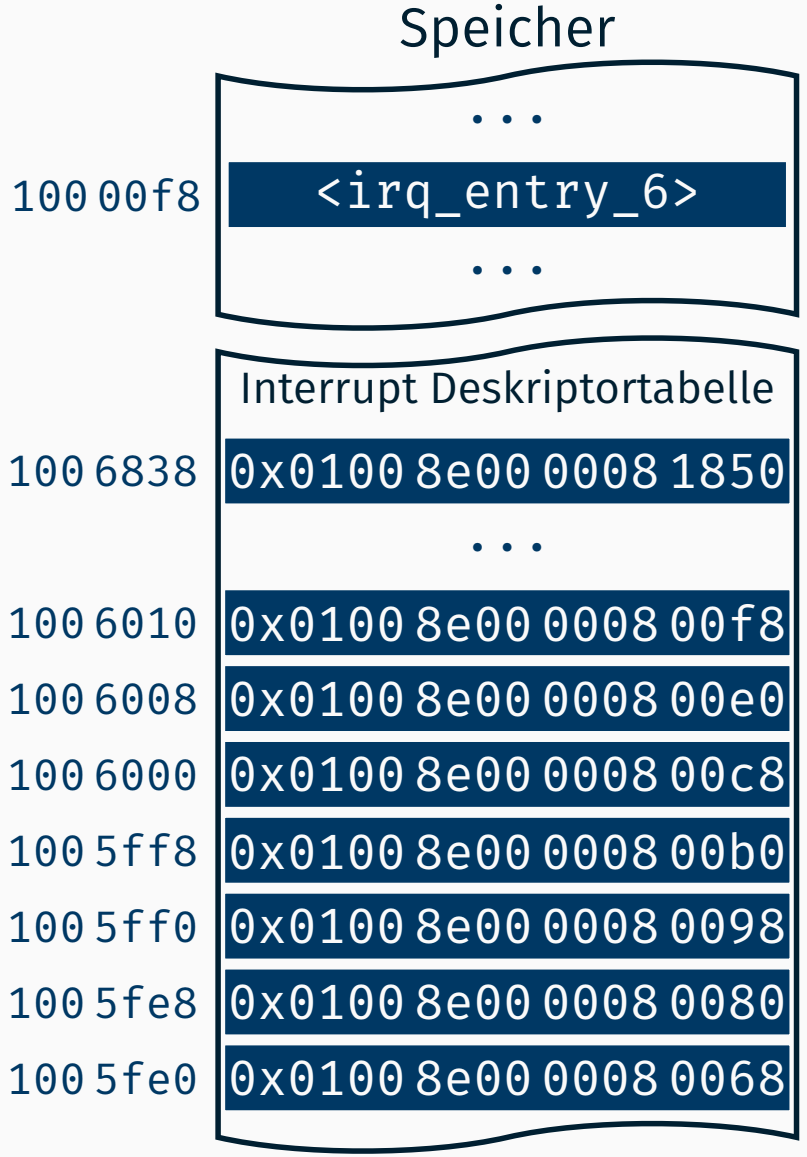
Interrupt Deskriptor Tabelle (IDT)

IDT-Register `idtr`



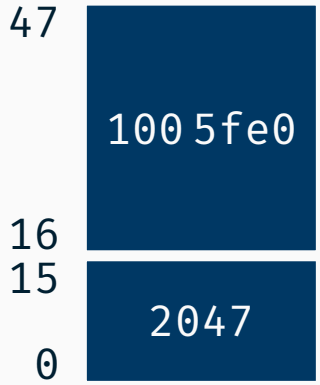
Basis: Startadresse der IDT

Limit: Bytes *Einträge * 8 - 1*



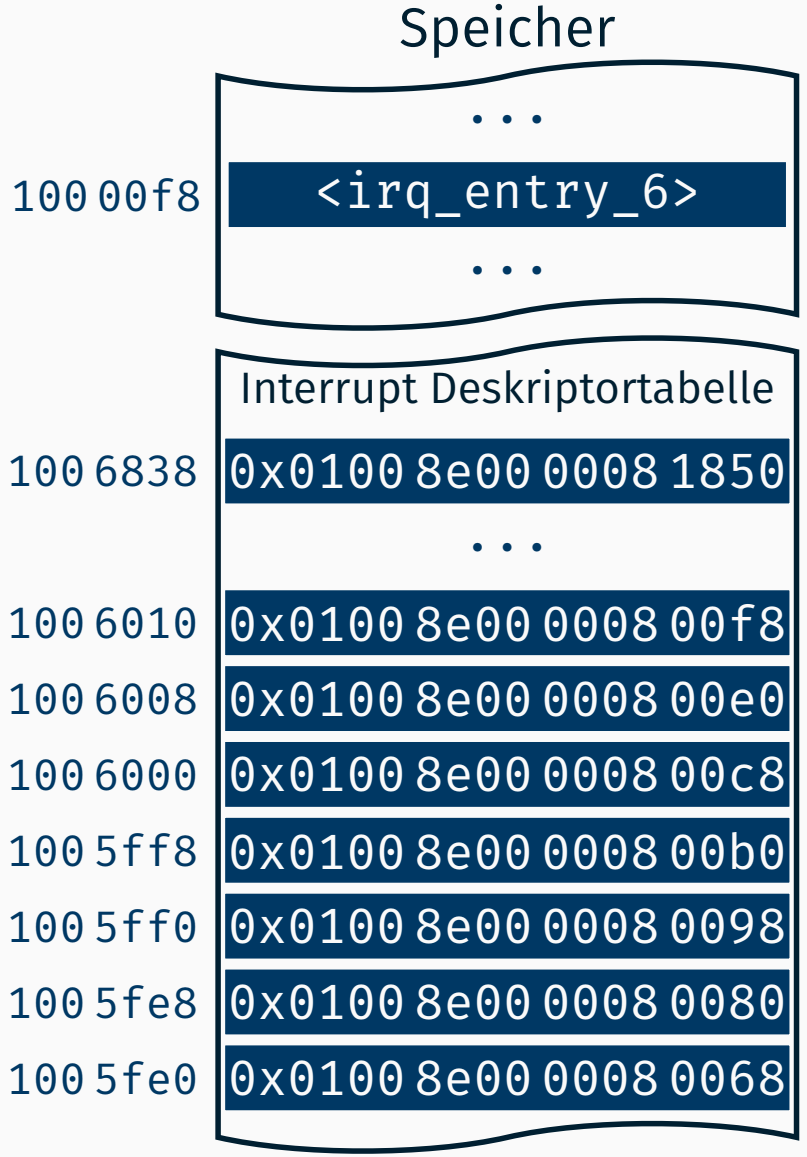
Interrupt Deskriptor Tabelle (IDT)

IDT-Register `idtr`



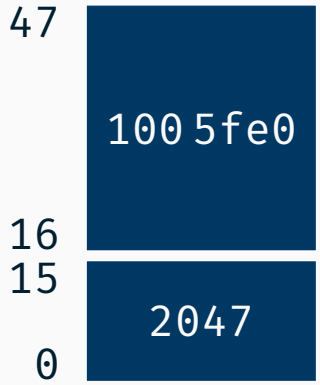
Basis: Startadresse der IDT

Limit: Bytes *Einträge * 8 - 1*



Interrupt Deskriptor Tabelle (IDT)

IDT-Register `idtr`



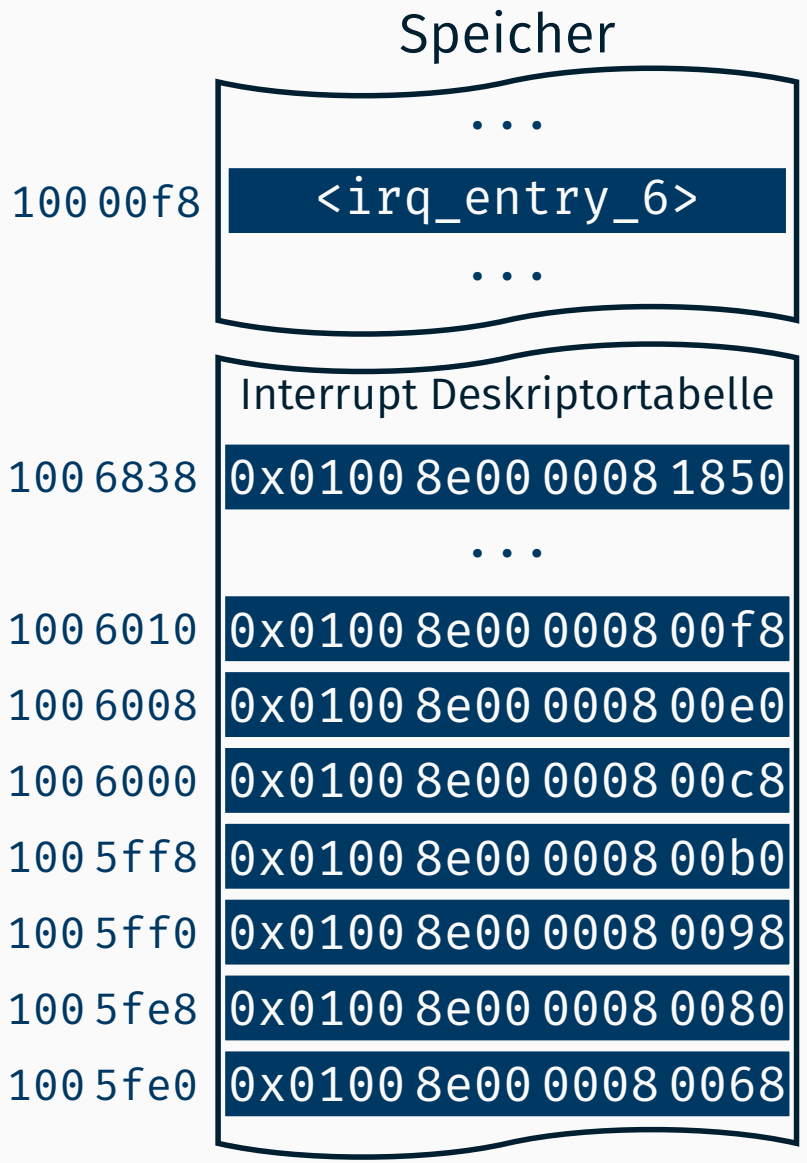
Basis: Startadresse der IDT

Limit: Bytes *Einträge * 8 - 1*

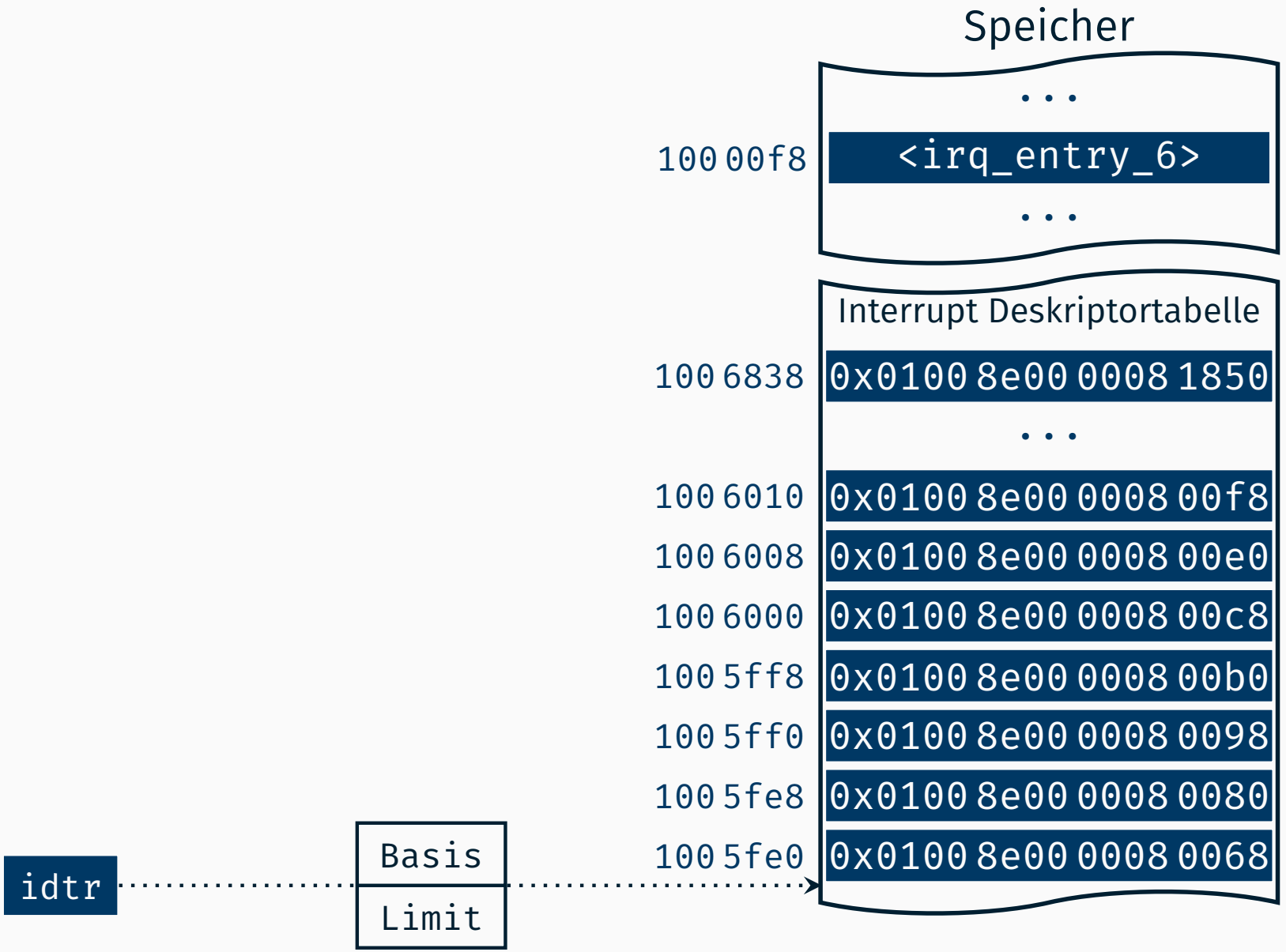
Instruktionen:

`lidt` in Register laden

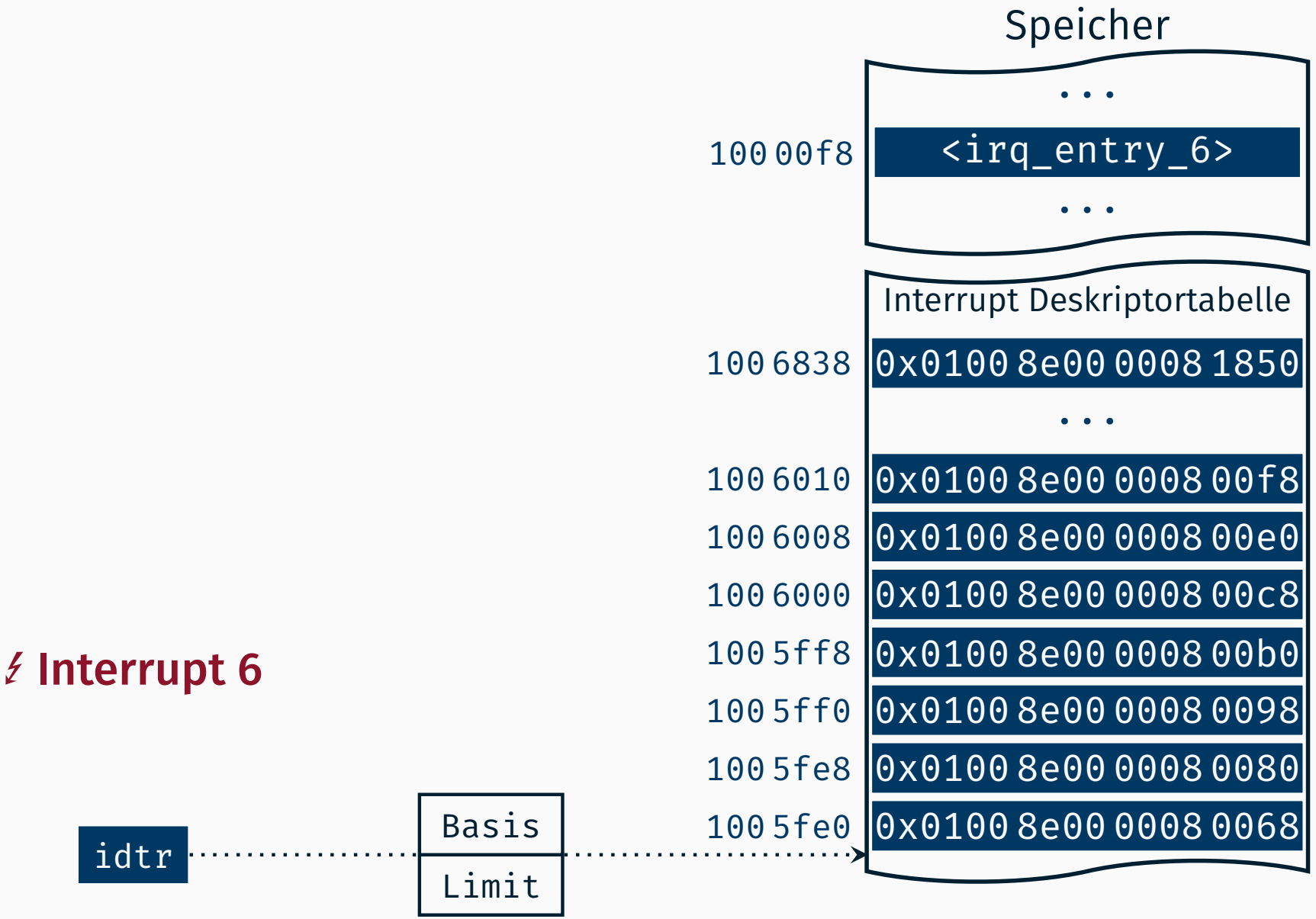
`sidt` aus Register lesen



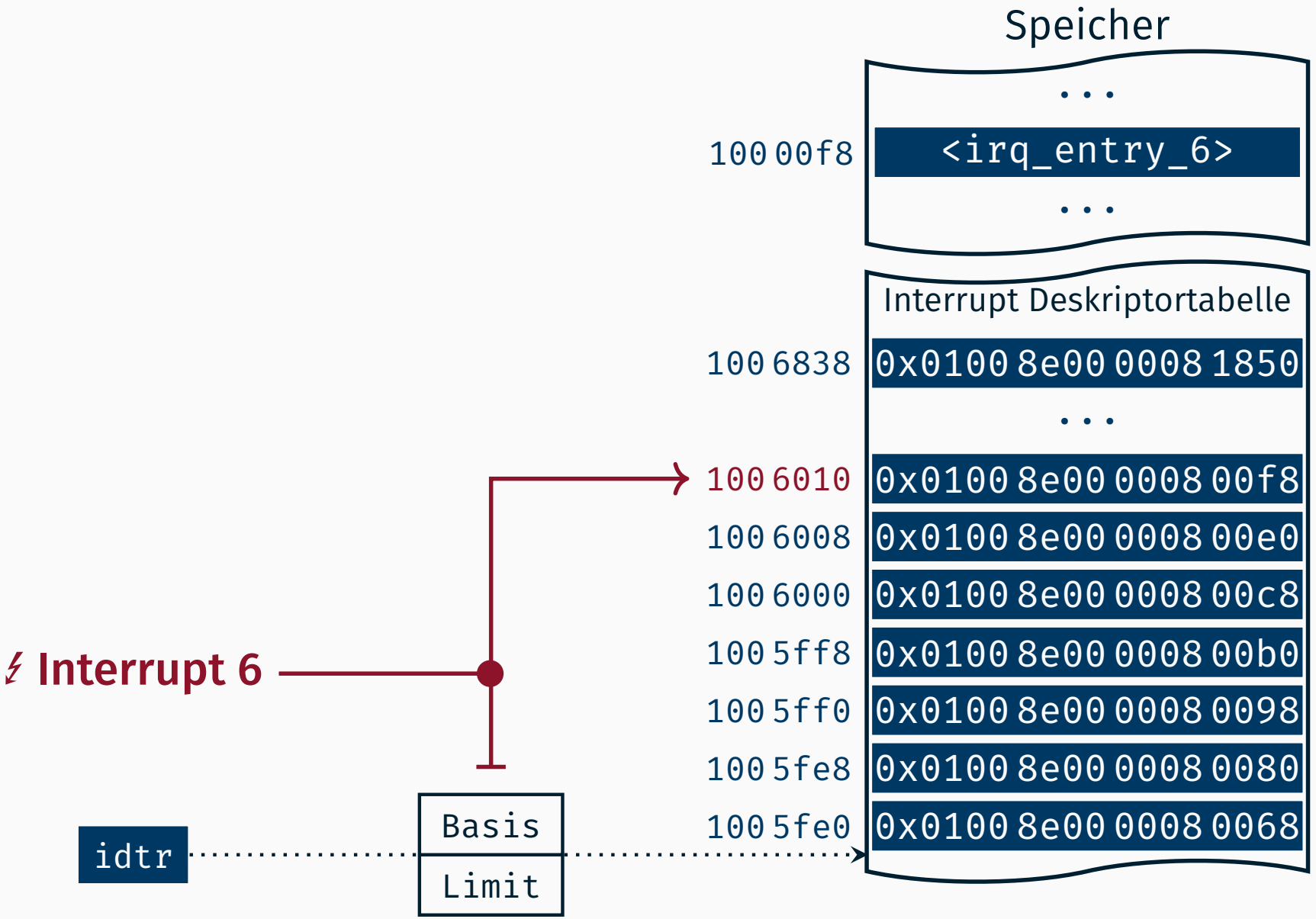
Interrupt Deskriptor Tabelle (IDT)



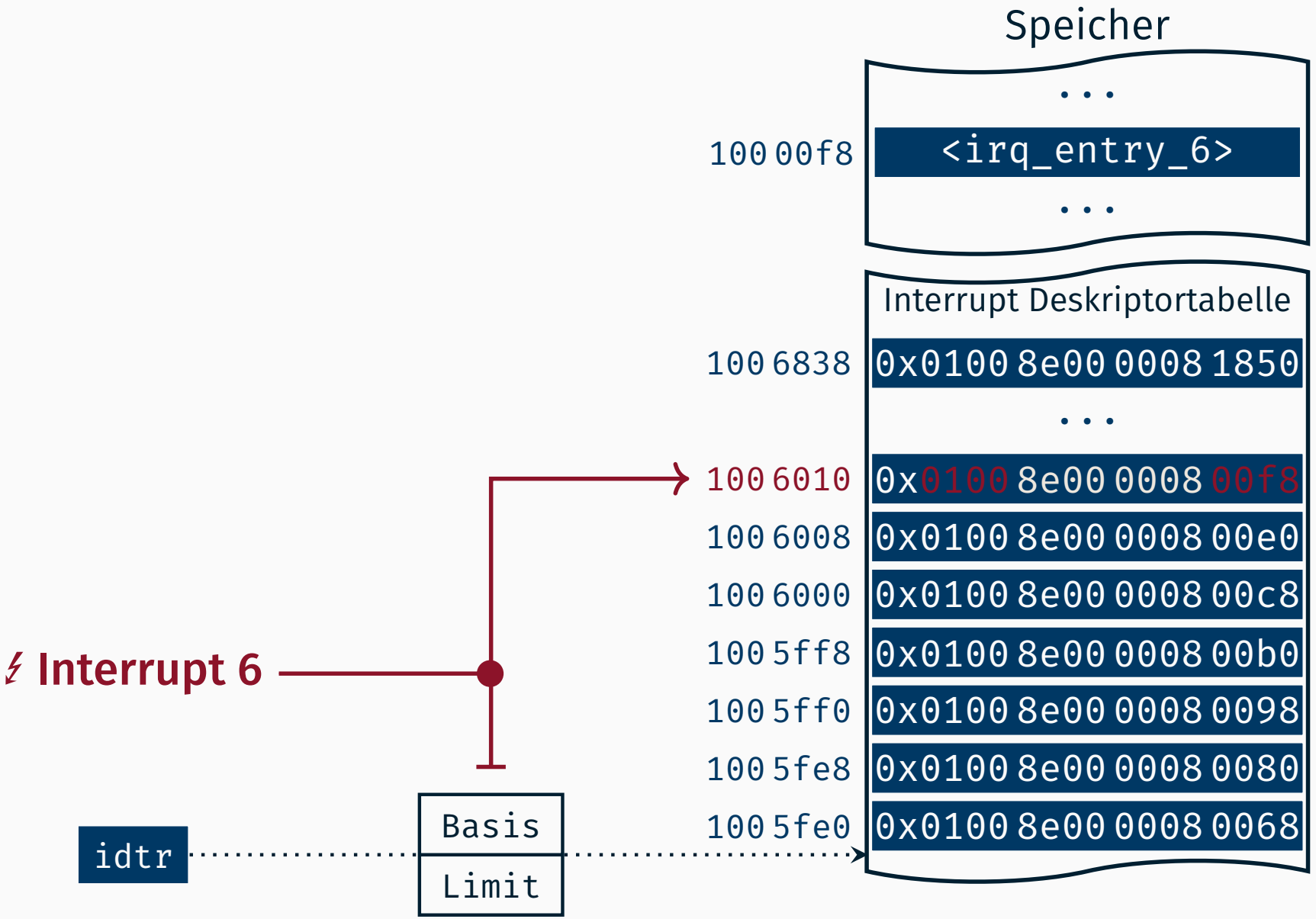
Interrupt Deskriptor Tabelle (IDT)



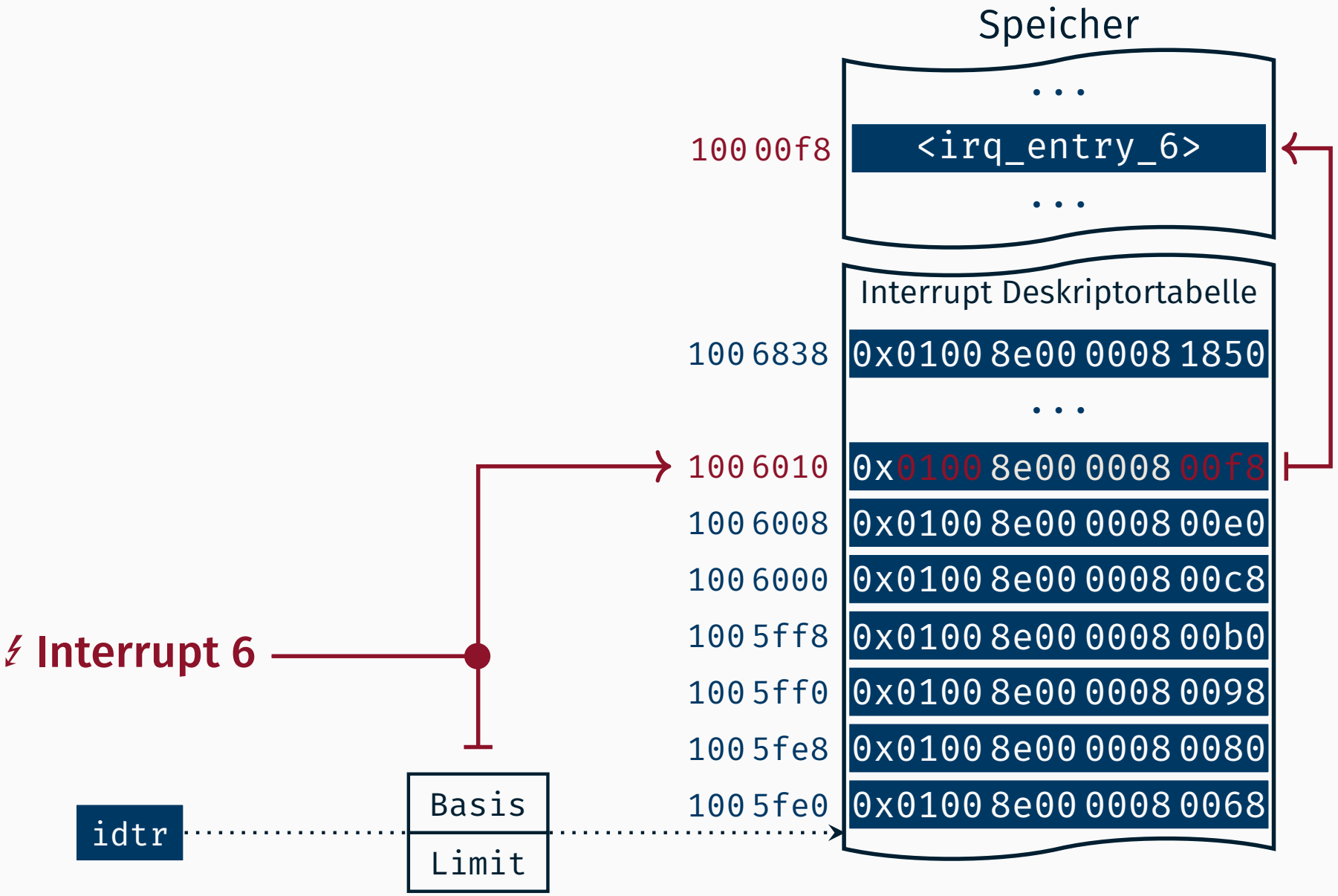
Interrupt Deskriptor Tabelle (IDT)



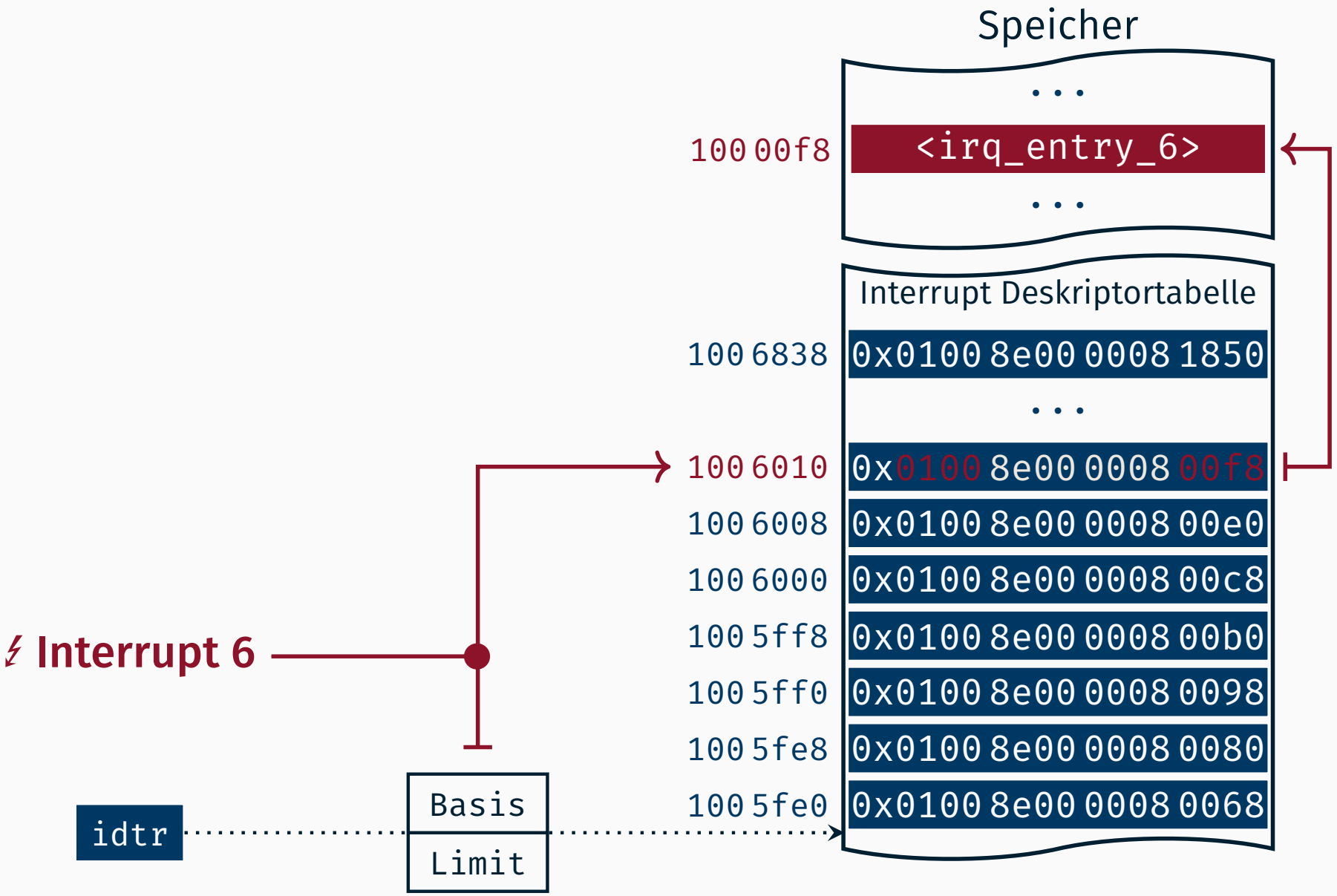
Interrupt Deskriptor Tabelle (IDT)



Interrupt Deskriptor Tabelle (IDT)



Interrupt Deskriptor Tabelle (IDT)



Externe Interrupts

Unterbrechungen durch externe Geräte bei x86-CPUs

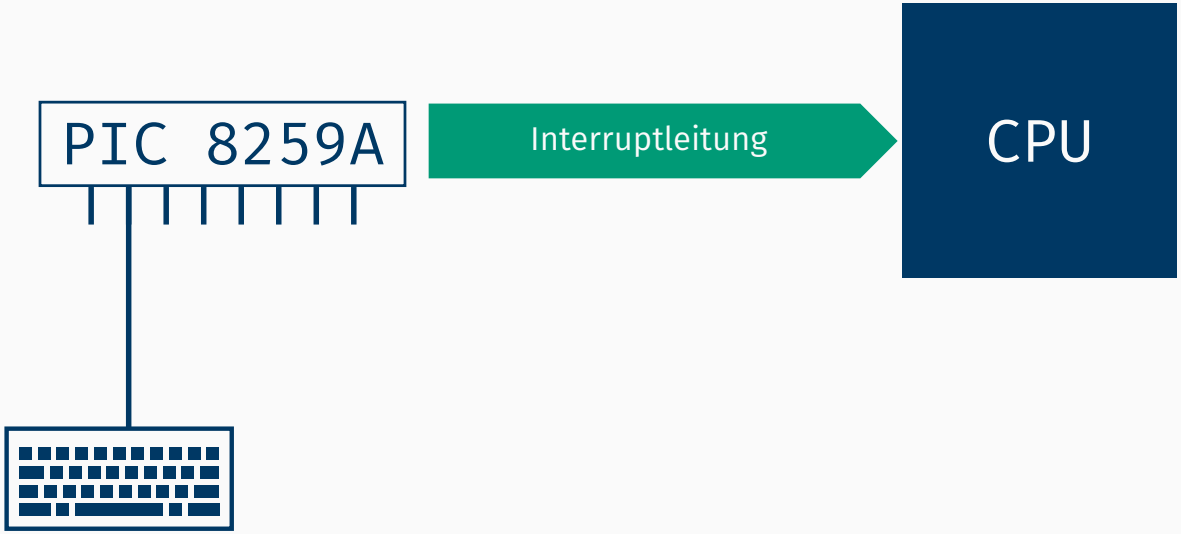
Unterbrechungen durch externe Geräte bei x86-CPUs

- Anschluss von mehreren externen Geräten durch [Programmable Interrupt] Controller

Unterbrechungen durch externe Geräte bei x86-CPUs

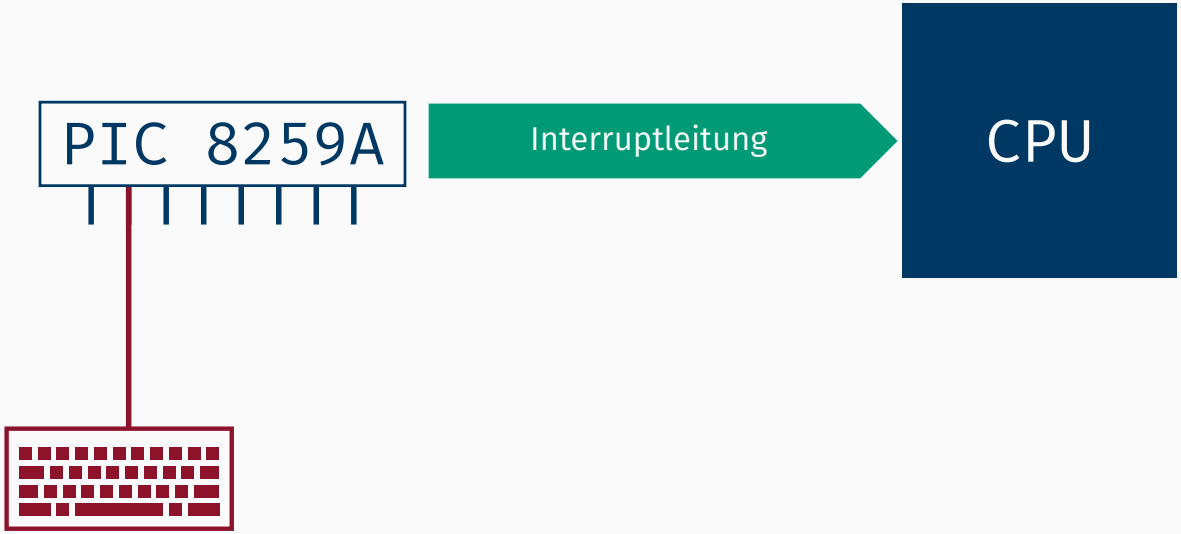
- Anschluss von mehreren externen Geräten durch [Programmable Interrupt] Controller
  - feste Prioritätenreihenfolge
  - Interruptnummern bedingt änderbar (Vielfaches von 8)
  - Konfiguration über IO-Ports

Unterbrechungen durch externe Geräte bei x86-CPUs



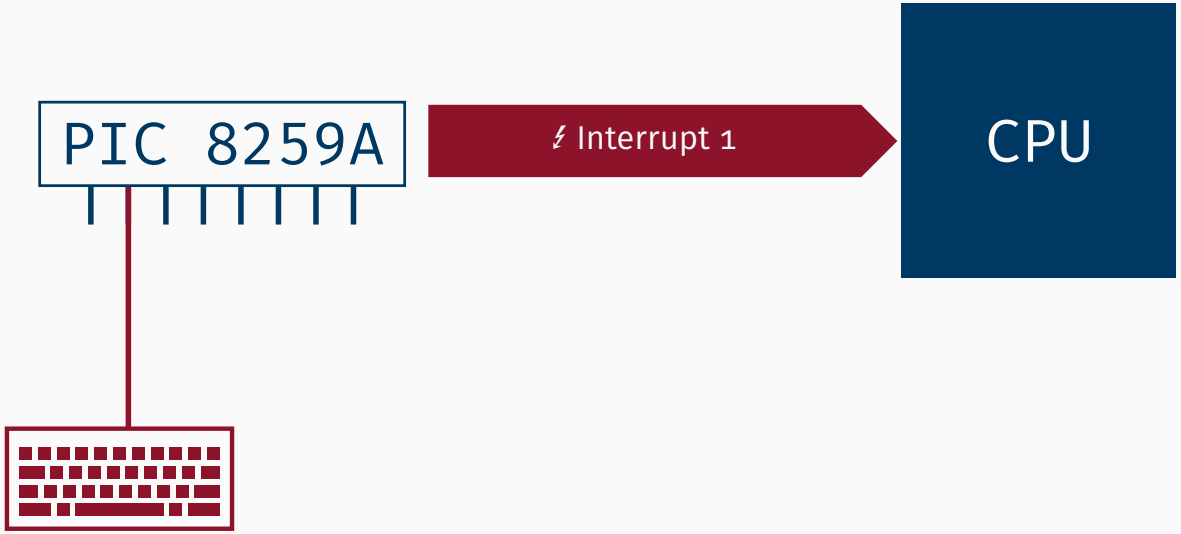
- Anschluss von mehreren externen Geräten durch [Programmable Interrupt] Controller
  - feste Prioritätenreihenfolge
  - Interruptnummern bedingt änderbar (Vielfaches von 8)
  - Konfiguration über IO-Ports

Unterbrechungen durch externe Geräte bei x86-CPUs



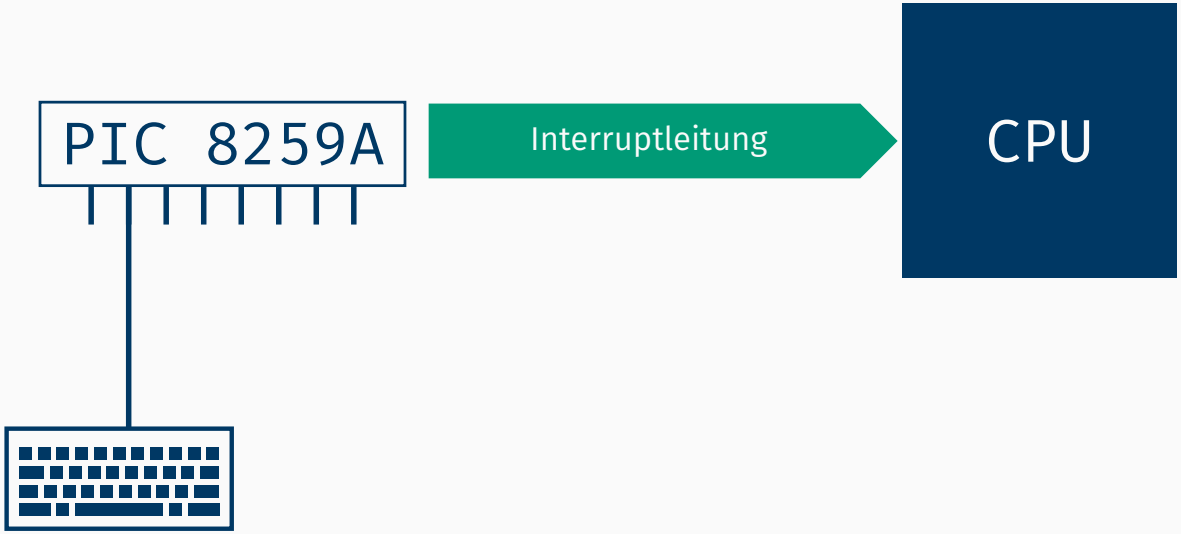
- Anschluss von mehreren externen Geräten durch [Programmable Interrupt] Controller
  - feste Prioritätenreihenfolge
  - Interruptnummern bedingt änderbar (Vielfaches von 8)
  - Konfiguration über IO-Ports

Unterbrechungen durch externe Geräte bei x86-CPUs



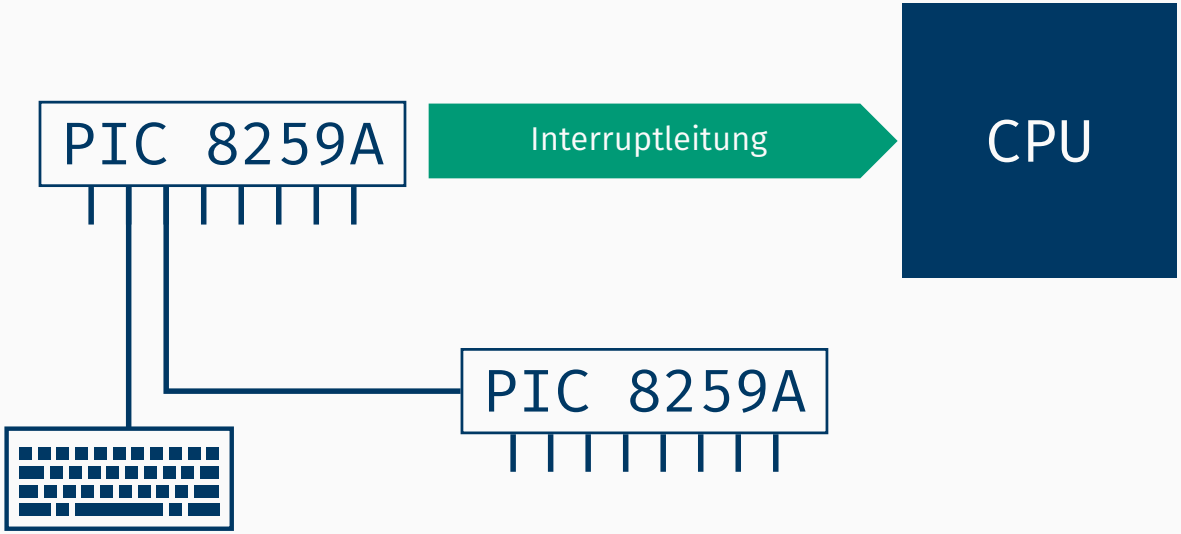
- Anschluss von mehreren externen Geräten durch [Programmable Interrupt] Controller
  - feste Prioritätenreihenfolge
  - Interruptnummern bedingt änderbar (Vielfaches von 8)
  - Konfiguration über IO-Ports

Unterbrechungen durch externe Geräte bei x86-CPUs



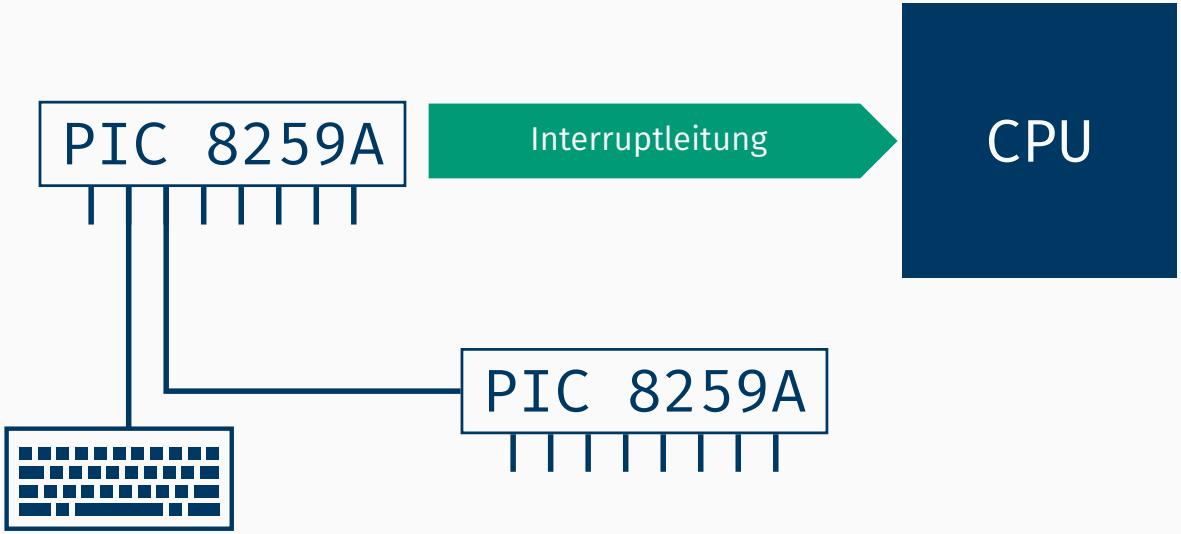
- Anschluss von mehreren externen Geräten durch [Programmable Interrupt] Controller
  - feste Prioritätenreihenfolge
  - Interruptnummern bedingt änderbar (Vielfaches von 8)
  - Konfiguration über IO-Ports

Unterbrechungen durch externe Geräte bei x86-CPUs



- Anschluss von mehreren externen Geräten durch [Programmable Interrupt] Controller
  - feste Prioritätenreihenfolge
  - Interruptnummern bedingt änderbar (Vielfaches von 8)
  - Konfiguration über IO-Ports
- Erweiterung von 8 auf 15 Geräte durch Kaskadierung

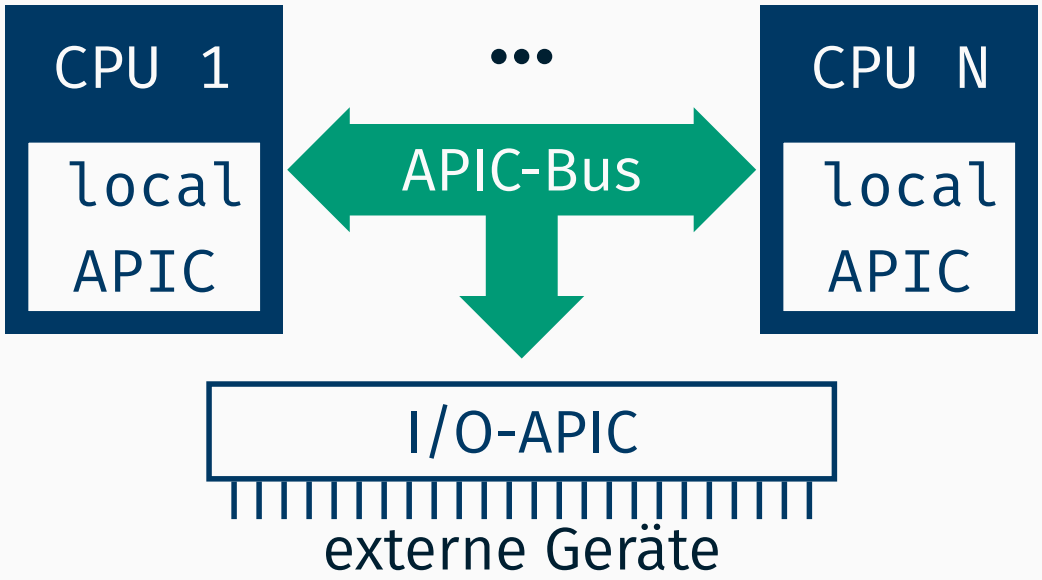
Unterbrechungen durch externe Geräte bei x86-CPUs



- Anschluss von mehreren externen Geräten durch [Programmable Interrupt] Controller
  - feste Prioritätenreihenfolge
  - Interruptnummern bedingt änderbar (Vielfaches von 8)
  - Konfiguration über IO-Ports
- Erweiterung von 8 auf 15 Geräte durch Kaskadierung
- Nicht für Mehrprozessorsysteme geeignet

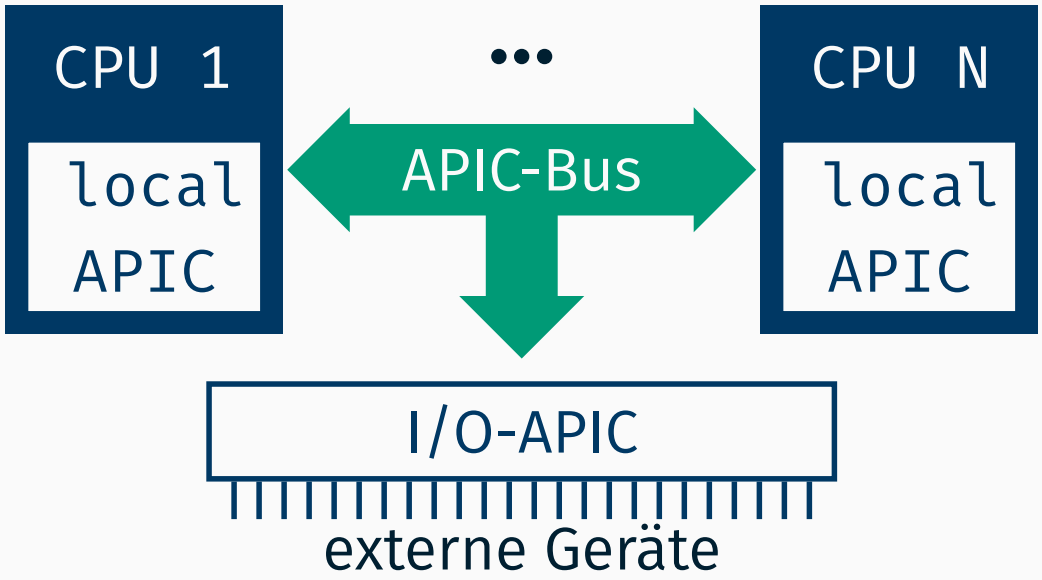
Externe Interrupts mit dem APIC

Aufbau der APIC-Architektur



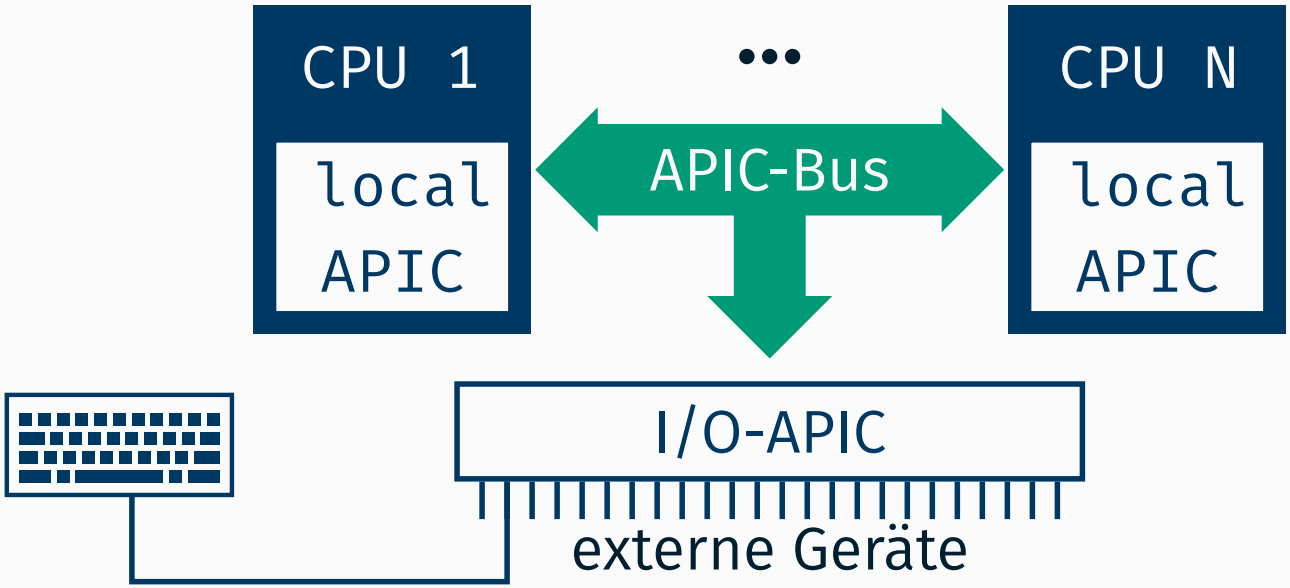
Aufteilung in lokalen APIC und I/O APIC

Aufbau der APIC-Architektur



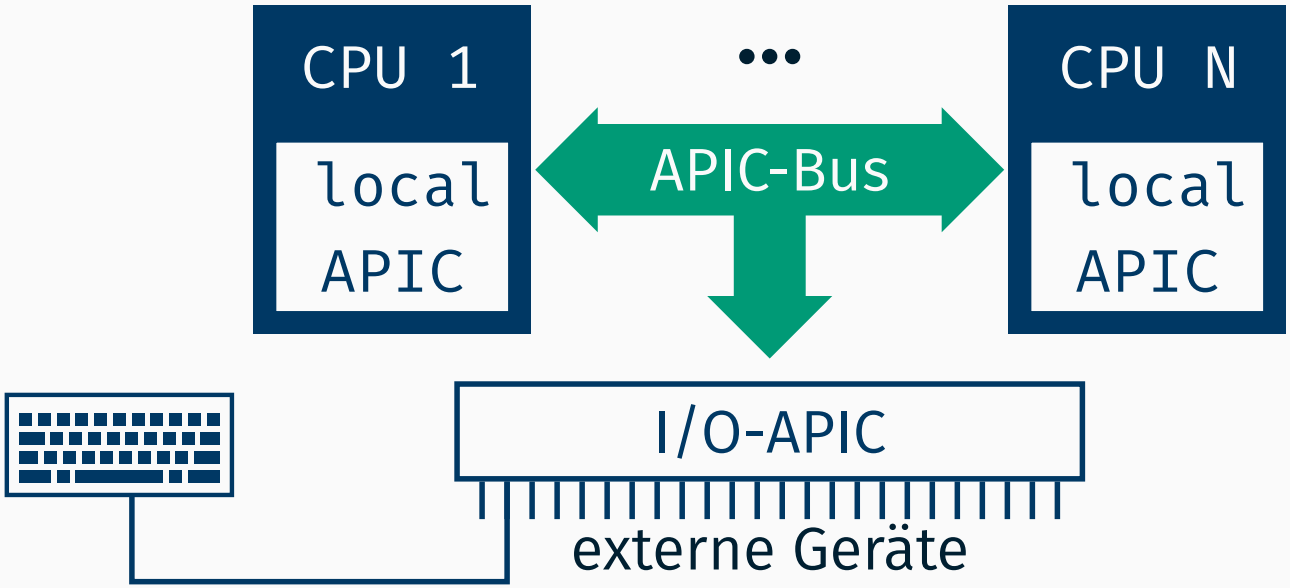
I/O-APIC dient zum Anschluss der Geräte

Aufbau der APIC-Architektur



I/O-APIC dient zum Anschluss der Geräte

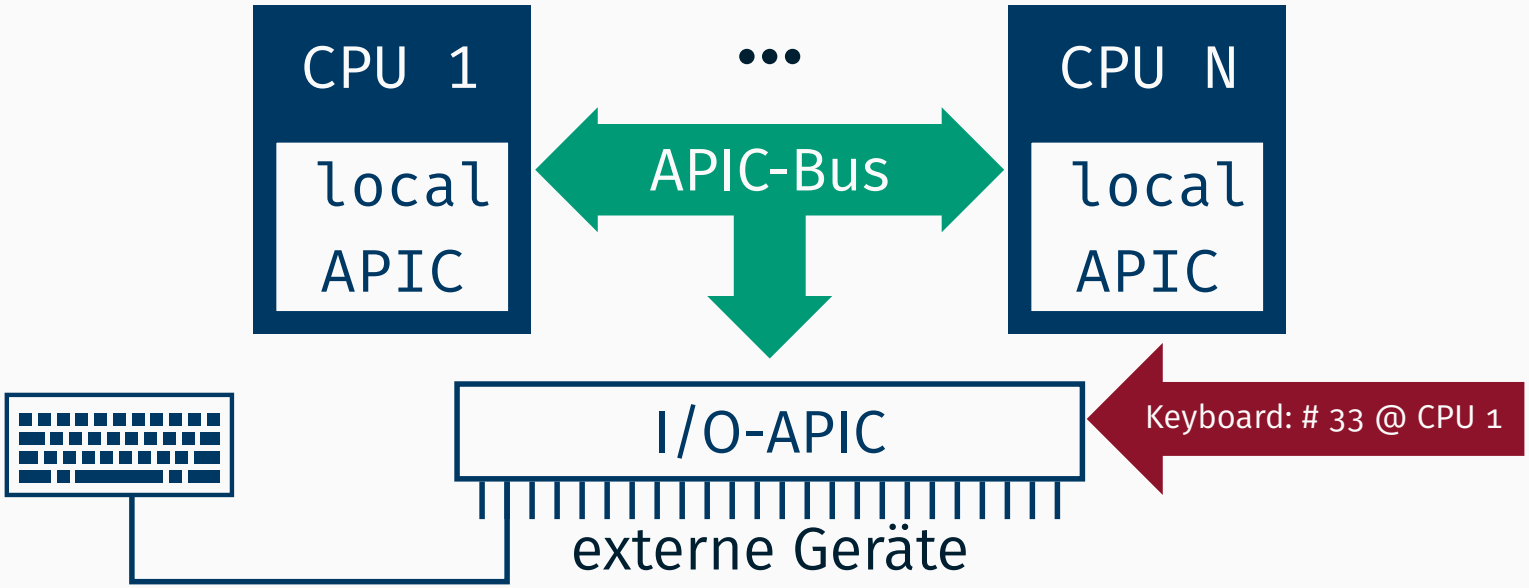
Aufbau der APIC-Architektur



I/O-APIC dient zum Anschluss der Geräte

- Zuweisung von beliebigen Vektornummern
- Aktivieren und Deaktivieren von einzelnen Interruptquellen
- Zuweisung von Zielprozessoren für einzelnen Interrupts in MP-Systemen

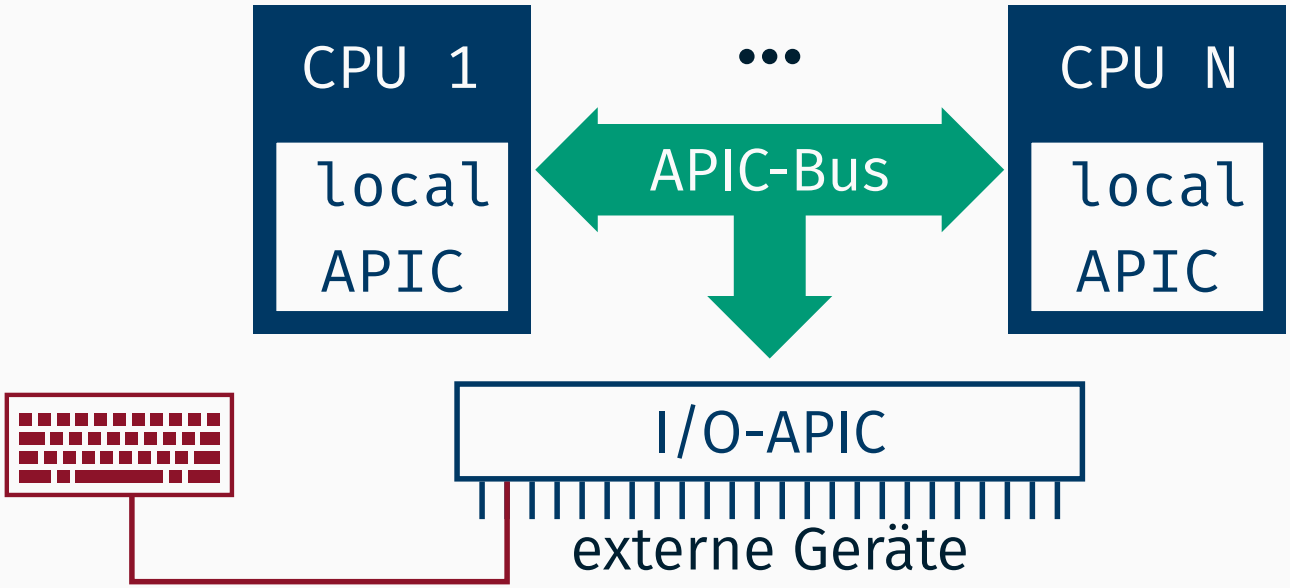
Aufbau der APIC-Architektur



I/O-APIC dient zum Anschluss der Geräte

- Zuweisung von beliebigen Vektornummern
- Aktivieren und Deaktivieren von einzelnen Interruptquellen
- Zuweisung von Zielprozessoren für einzelnen Interrupts in MP-Systemen

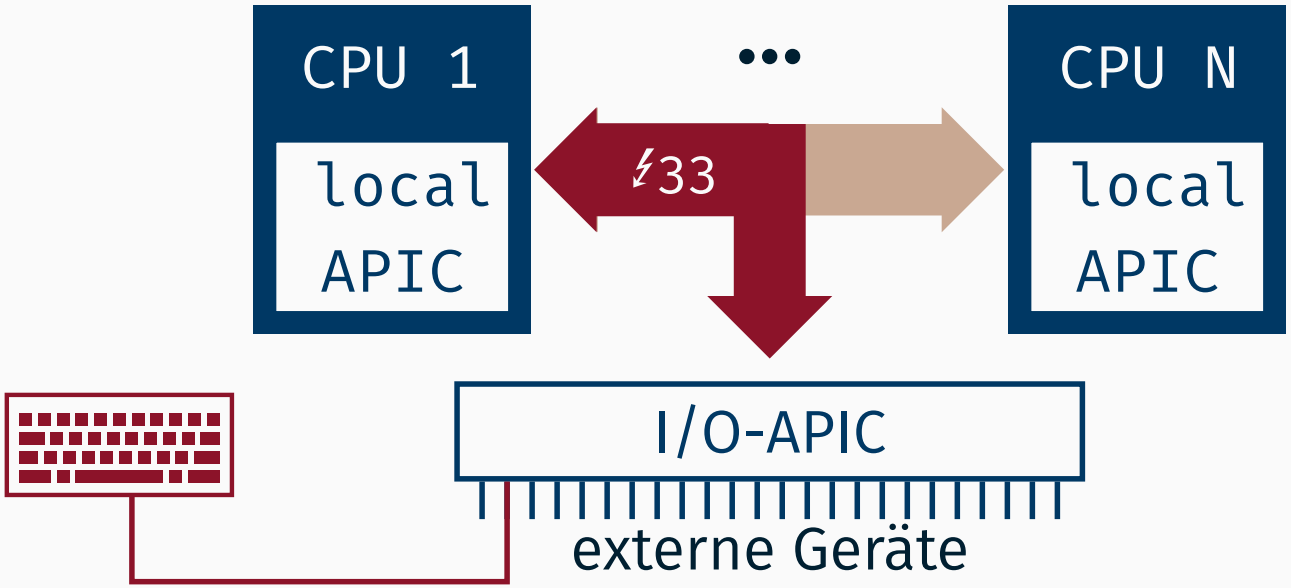
Aufbau der APIC-Architektur



I/O-APIC dient zum Anschluss der Geräte

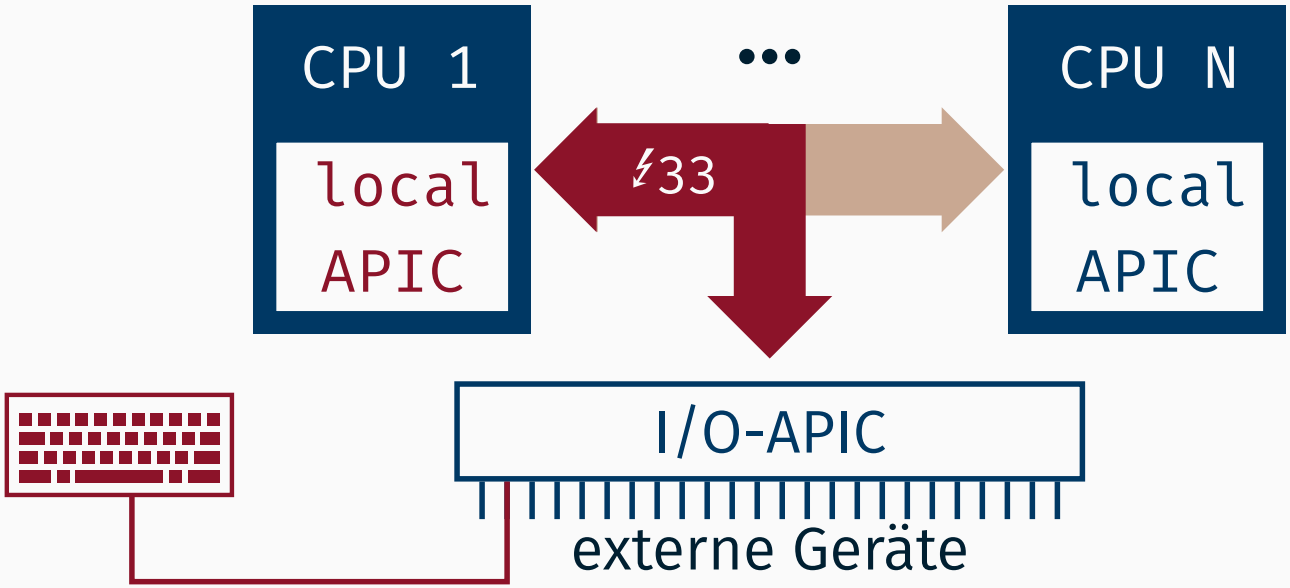
- Zuweisung von beliebigen Vektornummern
- Aktivieren und Deaktivieren von einzelnen Interruptquellen
- Zuweisung von Zielprozessoren für einzelnen Interrupts in MP-Systemen

Aufbau der APIC-Architektur



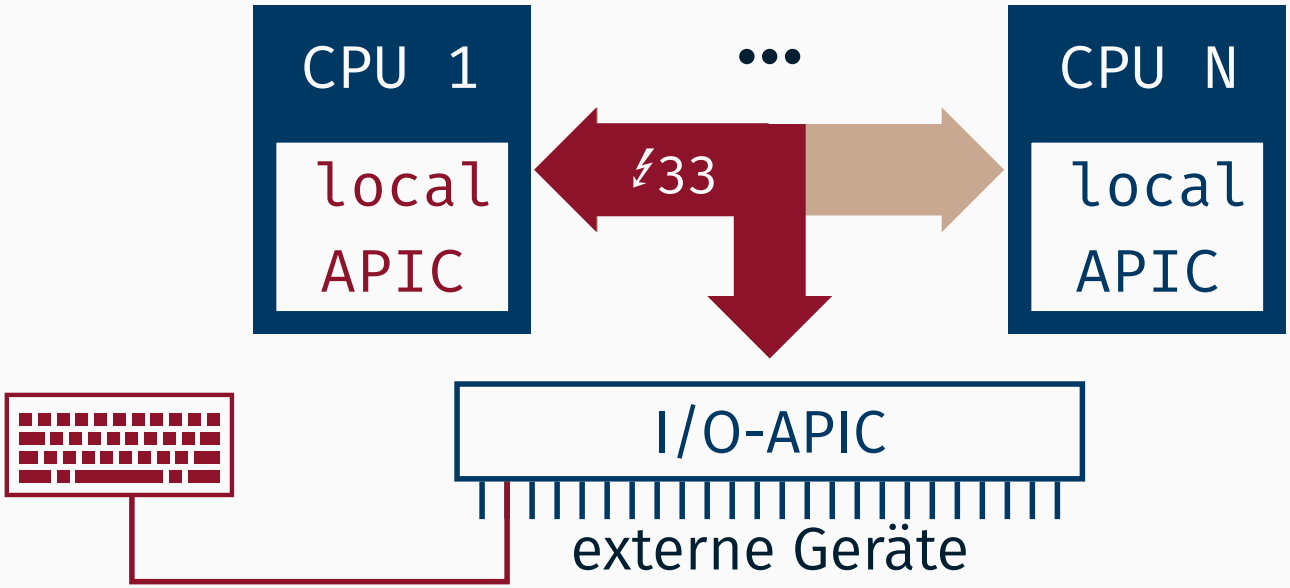
Interrupts werden zu Nachrichten auf dem APIC-Bus

Aufbau der APIC-Architektur



Empfang durch Local APIC

Aufbau der APIC-Architektur



Empfang durch Local APIC

- Verbindet eine CPU mit dem APIC-Bus
- Liest Nachrichten vom APIC-Bus und unterbricht die CPU
- Muss Interrupts explizit quittieren (ACK)

Programmierung des Intel I/O-APIC

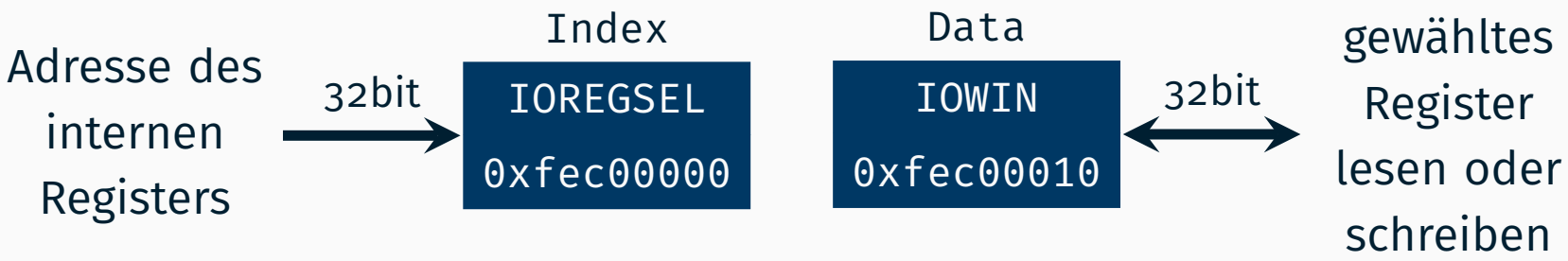
Zugriff auf die internen Register über memory-mapped Ein-/Ausgabe

- Jedoch keine direkte Abbildung von internen Registern auf Adressen
- „Umweg“ über ein Index- und Datenregister

Programmierung des Intel I/O-APIC

Zugriff auf die internen Register über memory-mapped Ein-/Ausgabe

- Jedoch keine direkte Abbildung von internen Registern auf Adressen
- „Umweg“ über ein Index- und Datenregister

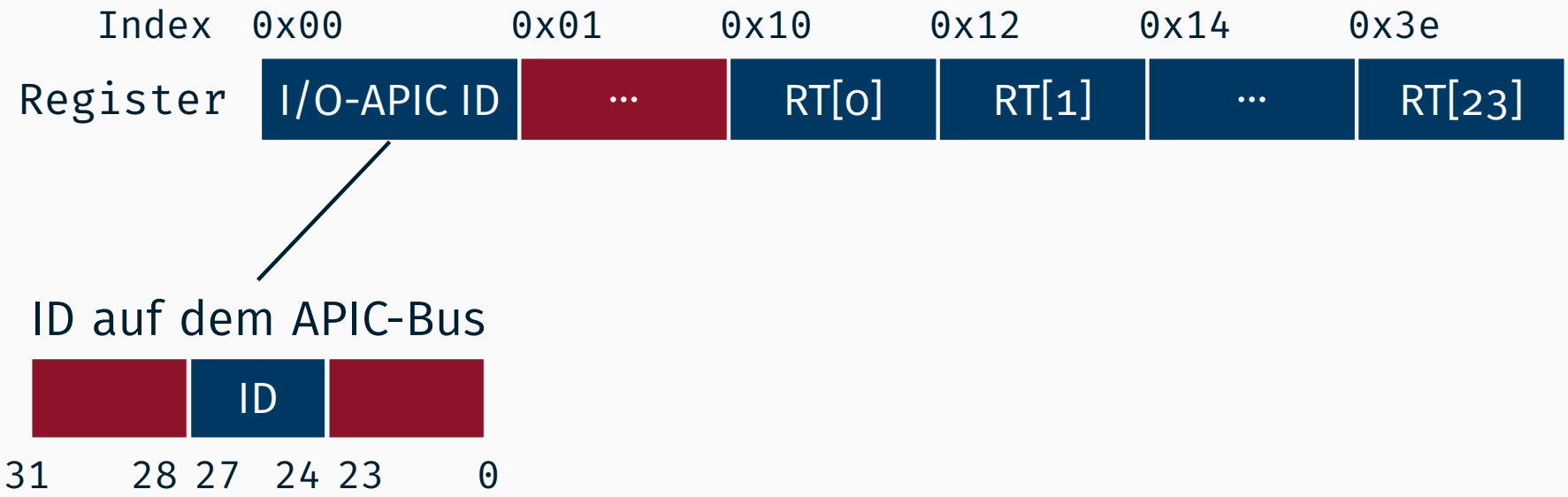


Programmierung des Intel I/O-APIC (2)

Interne Register des I/O-APICs

Programmierung des Intel I/O-APIC (2)

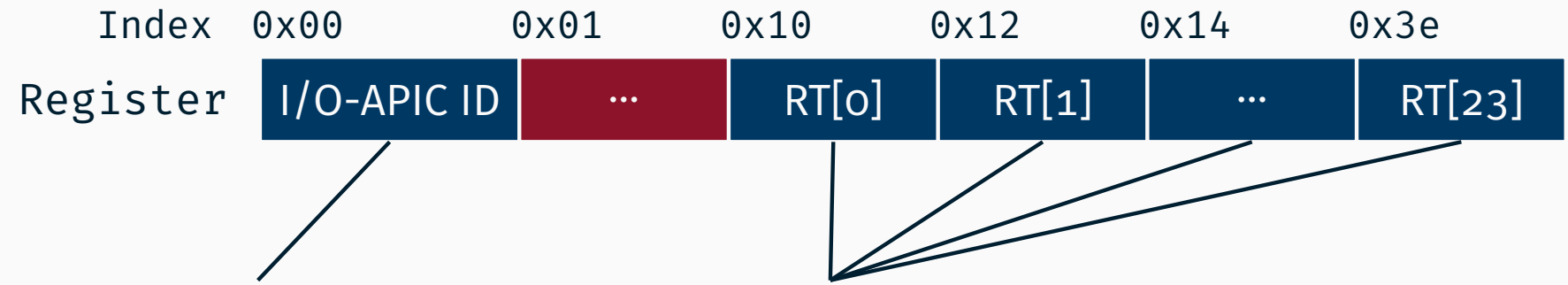
Interne Register des I/O-APICs



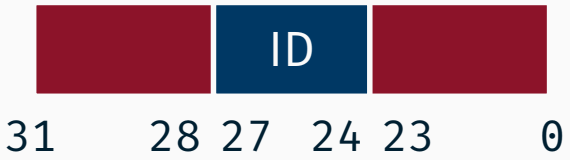
ID auf dem APIC-Bus

Programmierung des Intel I/O-APIC (2)

Interne Register des I/O-APICs



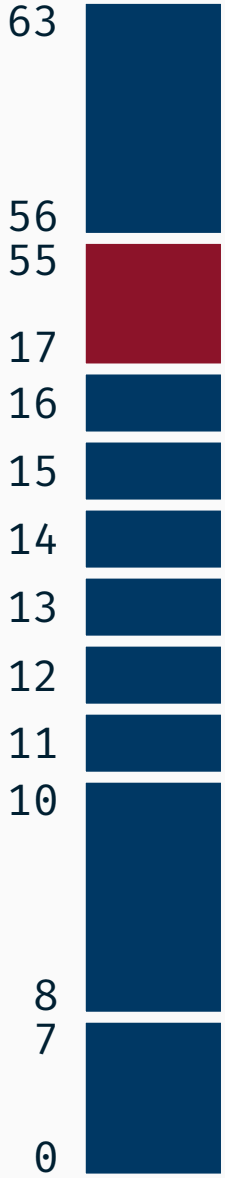
ID auf dem APIC-Bus



Redirection Table

- Ein Eintrag für jede Interruptquelle
- Konfiguration dadurch pro Interruptquelle
- Zwei interne Register pro Eintrag (64bit)

Aufbau eines Redirection Table Eintrags



Destination Field: Zieladresse des IRQs
- bei `Dest. Mode == Physical` APIC ID der Ziel-CPU
- bei `Dest. Mode == Logical` Gruppe von Ziel-CPUs

reserviert

Interrupt-Mask: Interrupt aktiv (`0`) oder inaktiv (`1`)
Trigger Mode: Flanken-(`0`) oder Pegelsteuerung (`1`)
Remote IRR: Art der erhaltenen Bestätigung
Interrupt Polarity: Active High (`0`) bzw. Active Low (`1`)
Delivery Status: Interrupt Nachricht noch unterwegs?
Destination Mode: Physical (`0`) oder Logical (`1`) Mode
Delivery Mode: Modus der Nachrichtenzustellung, z.B.
- `0` **Fixed** – Signal allen Zielprozessoren zustellen
- `1` **Lowest Priority** – CPU mit niedrigster Priorität ist Ziel

Interrupt Vektor: Nummer in der Vektortabelle (32-255)

Programmierung des APIC Systems

Wo ist welches Gerät angeschlossen?

- Das kann evtl. unterschiedlich sein von Rechner zu Rechner!
- Steht in der Systemkonfiguration, heutzutage typischerweise ACPI
- Klasse `APICSystem` stellt die relevanten Teile diese Informationen bereit

  `APICSystem::getIOAPICSlot` liefert für jedes Gerät den Index in die Redirection Table (Parameter: siehe `enum Device` in `apicsystem.h`)

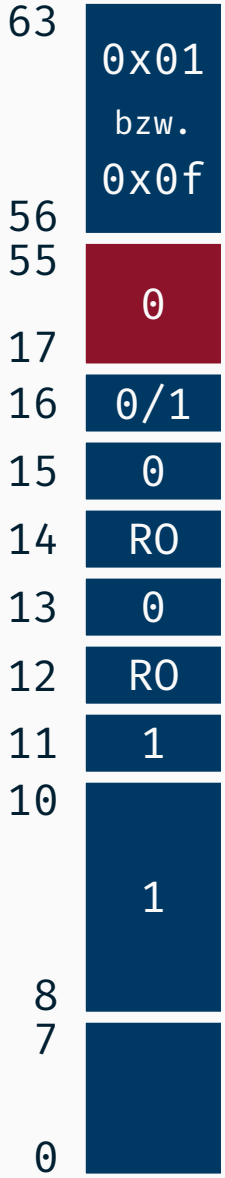
  `APICSystem::getIOAPICID` liefert die ID des IOAPICs

Programmierung des APIC Systems

Adressierung der APIC Nachrichten in OOStuBS/MPStuBS

- Zusammenspiel mehrerer Faktoren
  - `Destination Mode`, `Destination Field` und `Delivery Mode` im I/O-APIC
  - Prozessor Priorität in den Local APICs der einzelnen CPUs
- Ziel: Gleichverteilung der Interrupts auf alle CPUs
  - Priorität der Prozessoren im Local APIC fest auf `0` einstellen
  - Im I/O-APIC `Lowest Priority` als Delivery Mode verwenden
  - Verwendung des `Logical Destination Mode`; bis zu 8 CPUs adressierbar
  - `Destination Field`: Bitmaske mit gesetztem Bit pro aktivierter CPU

Redirection Table Einträge in OOStuBS/MPStuBS



Destination Field: Zieladresse des IRQs

bei `Dest. Mode == Physical` APIC ID der Ziel-CPU

bei `Dest. Mode == Logical` Gruppe von Ziel-CPUs

reserviert

Interrupt-Mask: Interrupt aktiv (`0`) oder inaktiv (`1`)

Trigger Mode: Flanken-(`0`) oder Pegelsteuerung (`1`)

Remote IRR: Art der erhaltenen Bestätigung

Interrupt Polarity: Active High (`0`) bzw. Active Low (`1`)

Delivery Status: Interrupt Nachricht noch unterwegs?

Destination Mode: Physical (`0`) oder Logical (`1`) Mode

Delivery Mode: Modus der Nachrichtenzustellung, z.B.

`0` **Fixed** – Signal allen Zielprozessoren zustellen

`1` **Lowest Priority** – CPU mit niedrigster Priorität ist Ziel

Interrupt Vektor: Nummer in der Vektortabelle (32-255)

(RO: Read Only)

Zusammenfassendes Beispiel: Keyboard Interrupt in OO/MPStuBS

Vorbereitung

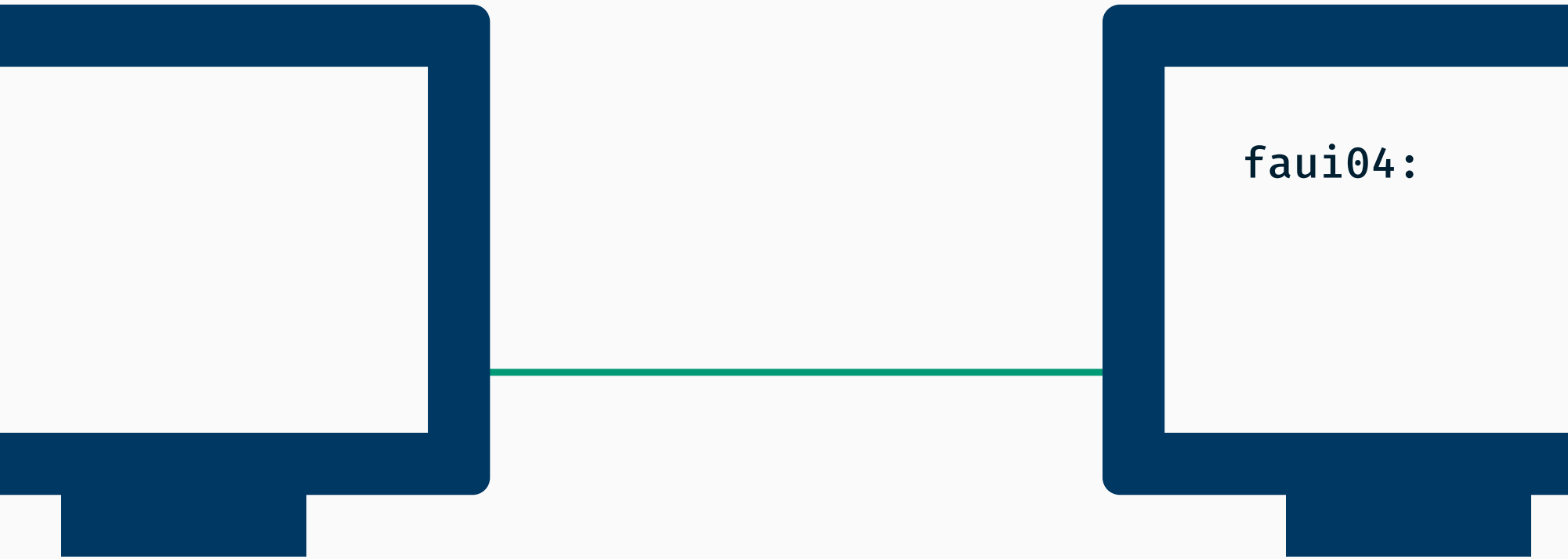
- **I/O APIC** initialisieren
  - **I/O APIC ID** setzen
  - Einträge in **Redirection Table** initialisieren (deaktivieren)
- **Keyboard** konfigurieren
  - Anmelden bei der **Plugbox**
  - Tastaturslot herausfinden und den entsprechenden Eintrag in der **Redirection Table** konfigurieren und aktivieren
  - Tastaturbuffer leeren
- **Interruptbehandlung** erstellen
  - Einsprungsroutine `irq_entry` mit Aufruf zu `guardian` schreiben [wird in der Vorgabe bereits erledigt]
  - Eintragen in die **Interrupt Description Table** (`idt`) und diese in das Register `idtr` laden [ebenfalls erledigt]
  - Behandlung in `guardian` mittels **Plugbox**
- Interrupts mit `CPU::enable_int()` aktivieren

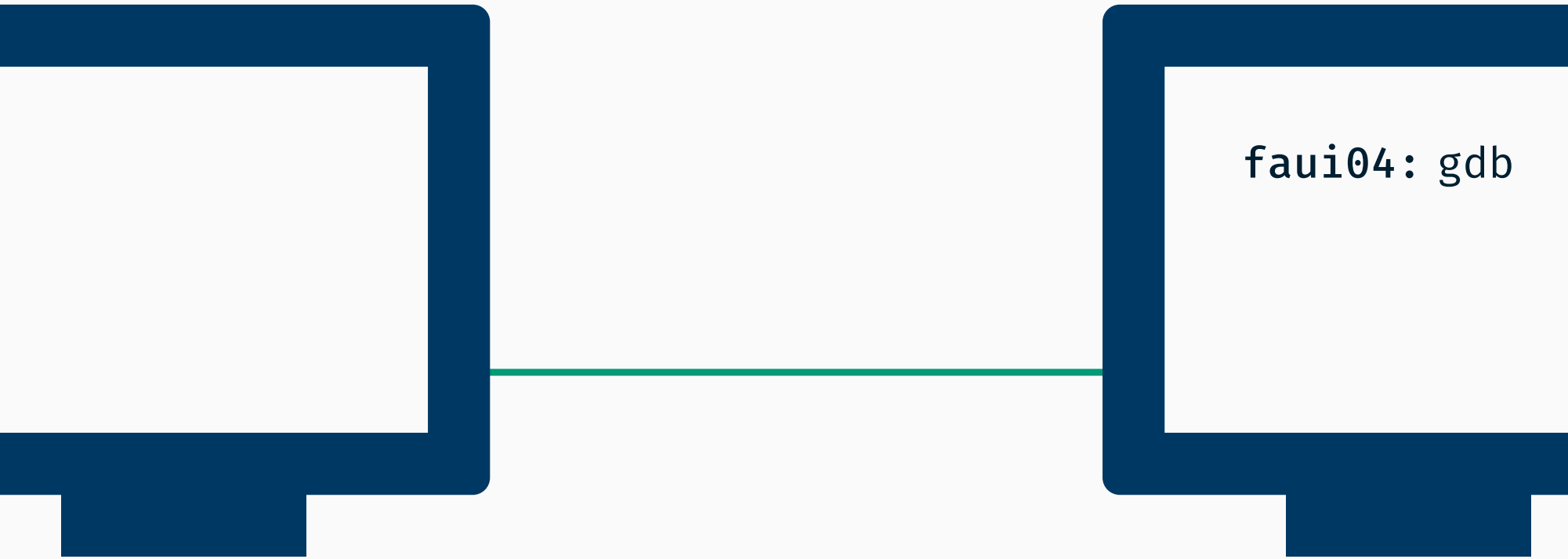
Ablauf

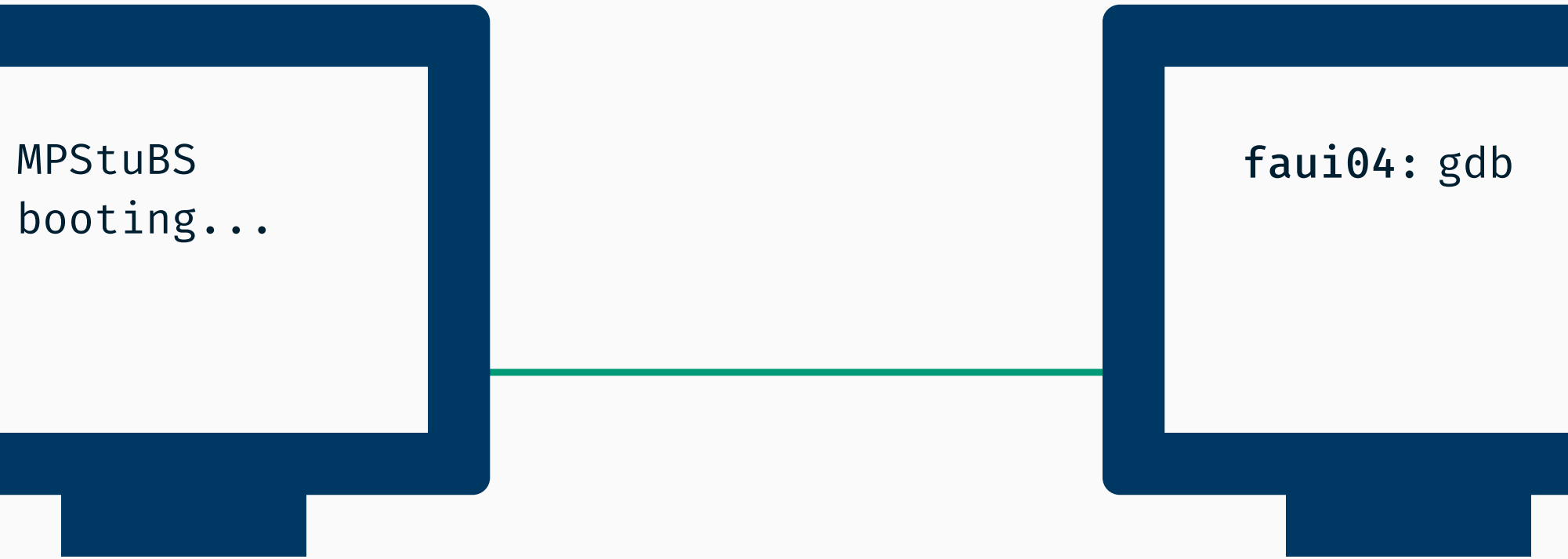
1. **Tastendruck** – Tastaturprozessor (in der Tastatur) meldet seriell (via **PS/2**) an Tastaturcontroller
2. **Tastaturcontroller** aktiviert Interruptleitung zu **I/O APIC**
   2.1 Anhand der **Redirection Table** wird Aktion gewählt
   2.2 Nachricht auf **APICBUS** mit Interruptnr. 33 und Ziel-CPU
3. entsprechender **LAPIC** empfängt Nachricht vom **APICBUS** und unterbricht CPU
4. **CPU** führt Unterbrechungsbehandlung aus
   4.1 Mittels Register `idtr` wird der entsprechende Eintrag in der **Interrupt Description Table** ausgewählt und in die Einsprungsroutine gesprungen
   4.2 Einsprungsroutine `irq_entry_33` sichert Register und ruft `guardian` auf
   4.3 `guardian` behandelt mittels **Plugbox** den Interrupt
5. **LAPIC** quittiert die Behandlung

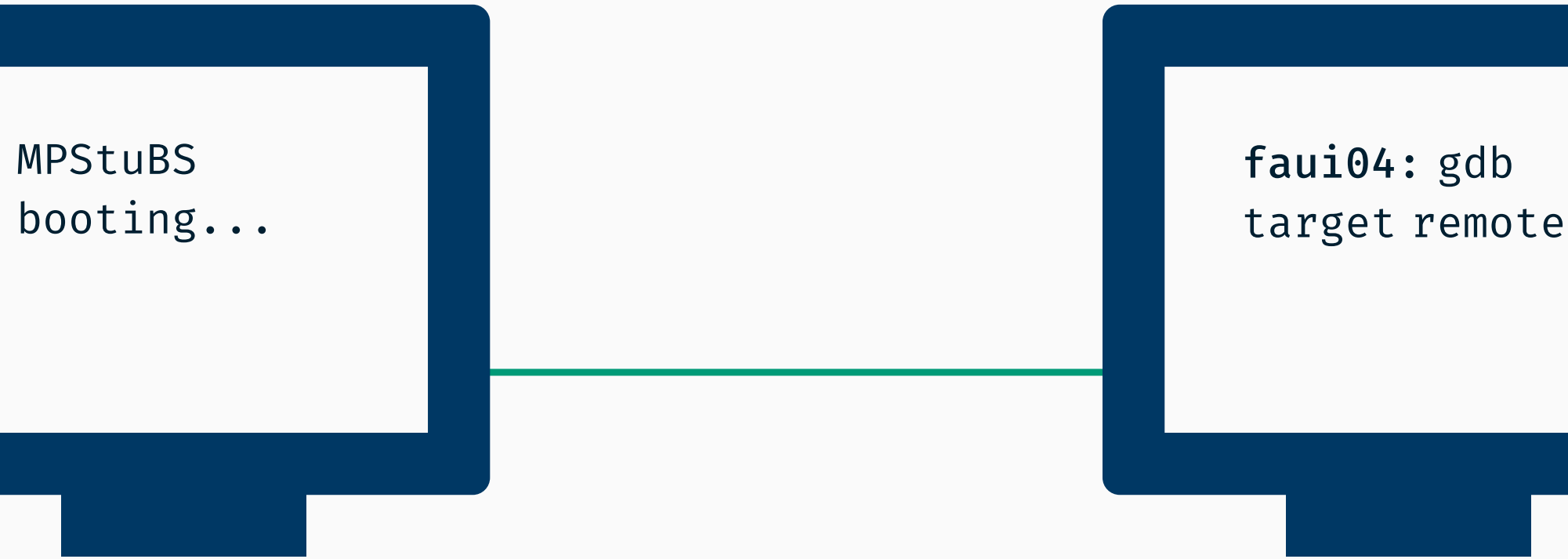
Remotedebugging mit GDB

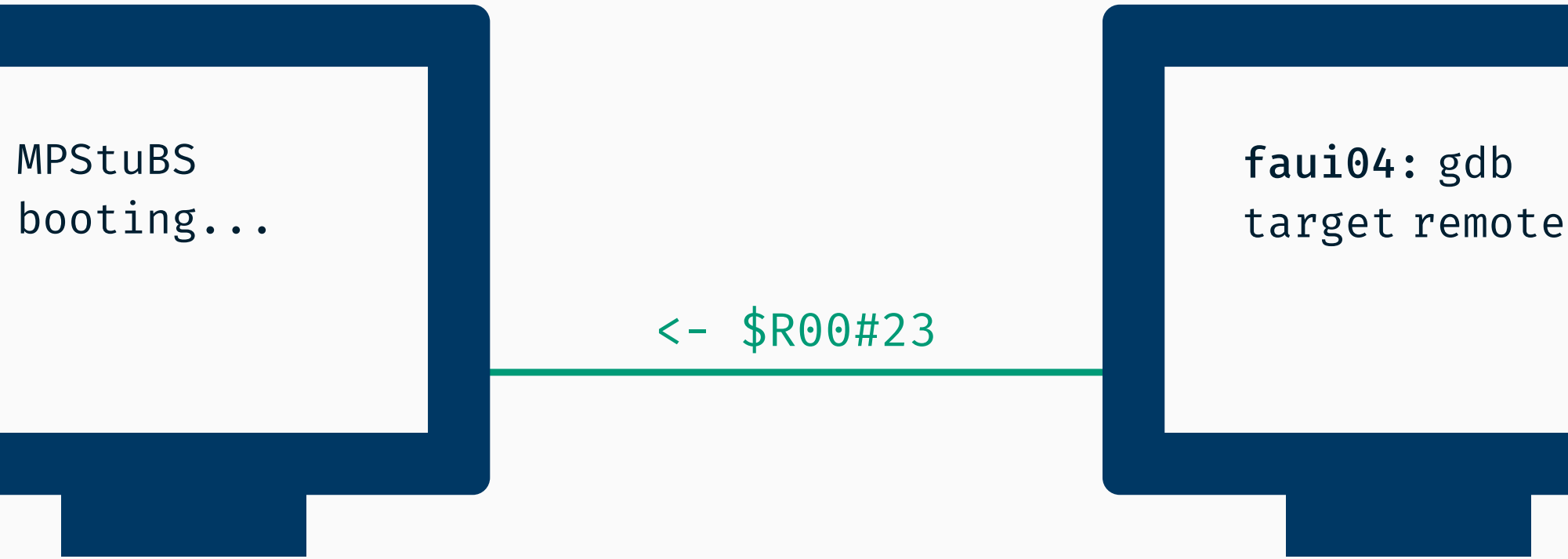

faui04:


faui04:

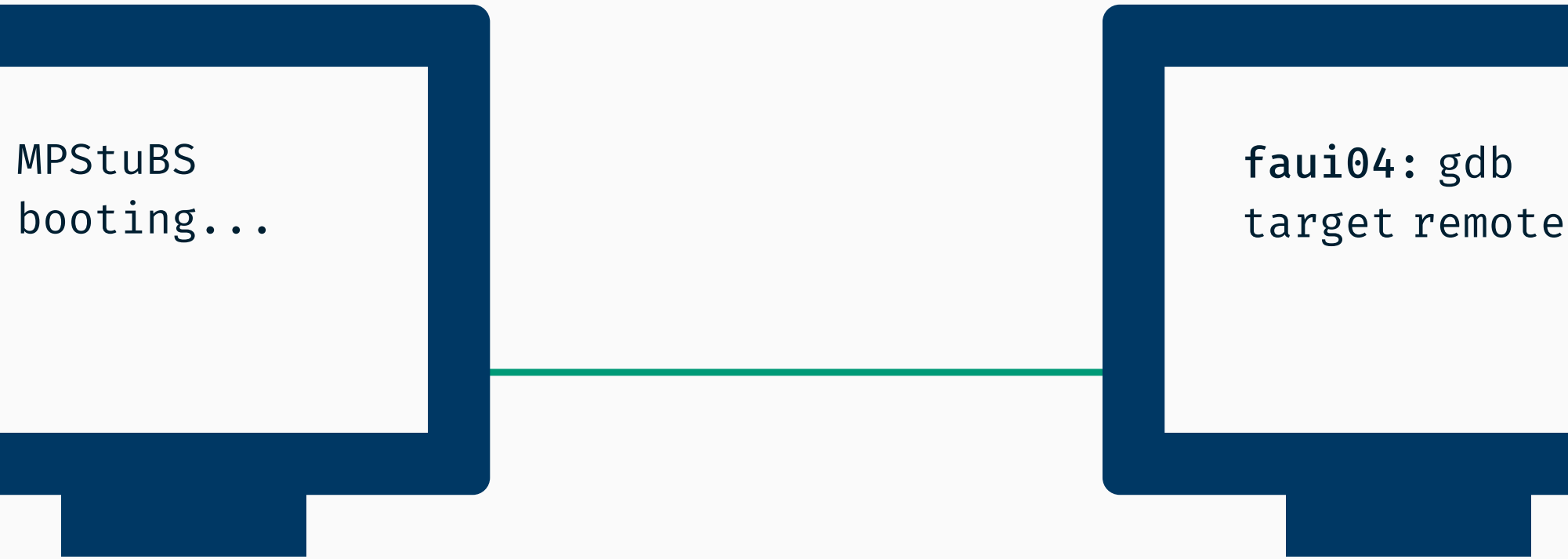

faui04: gdb

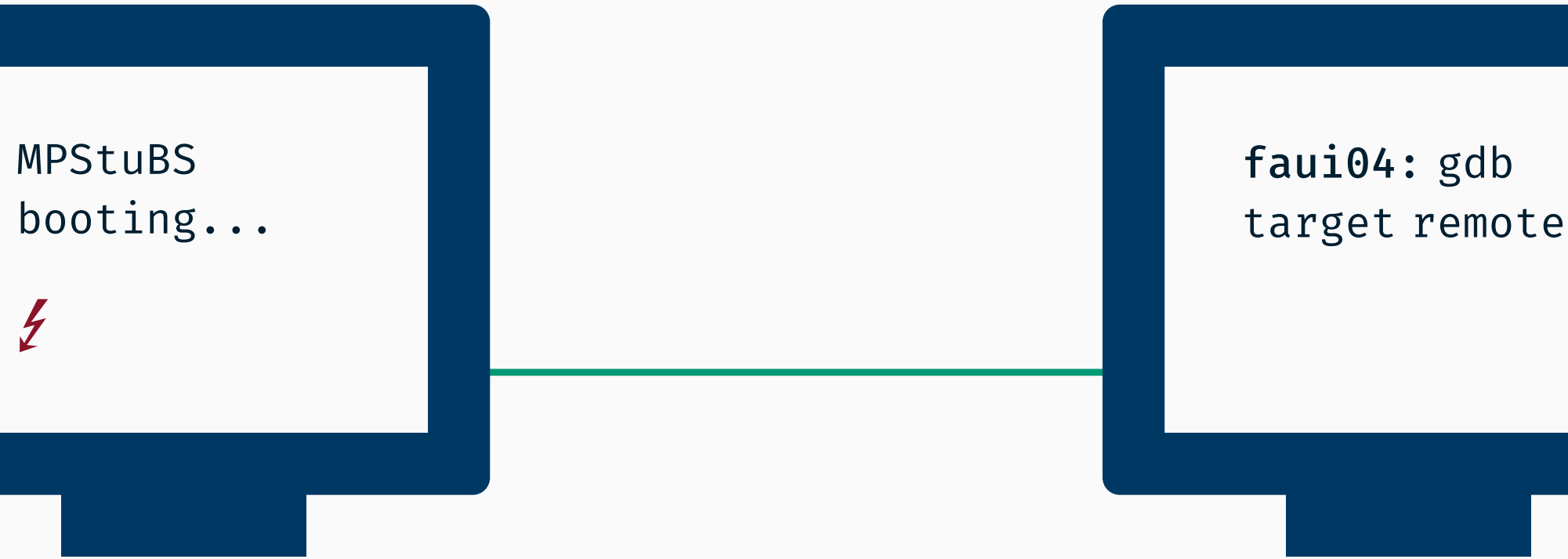

MPStuBS
booting...
faui04: gdb


MPStuBS
booting...
faui04: gdb
target remote

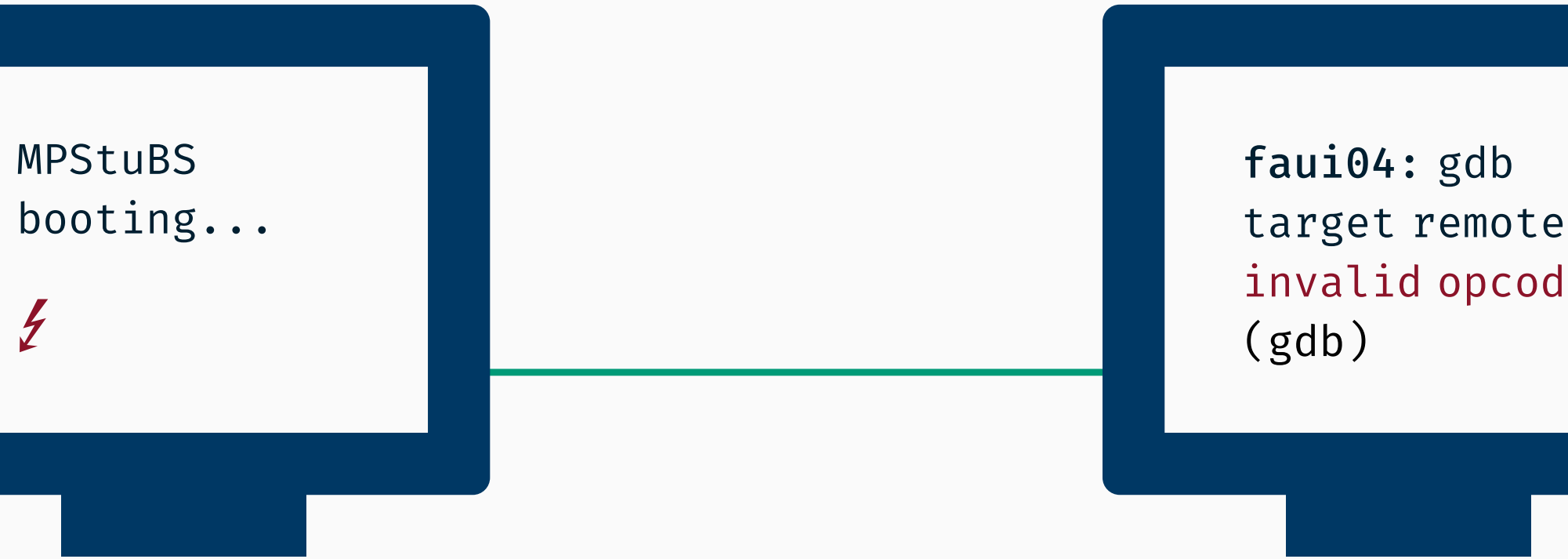

MPStuBS
booting...
<- $R00#23
faui04: gdb
target remote

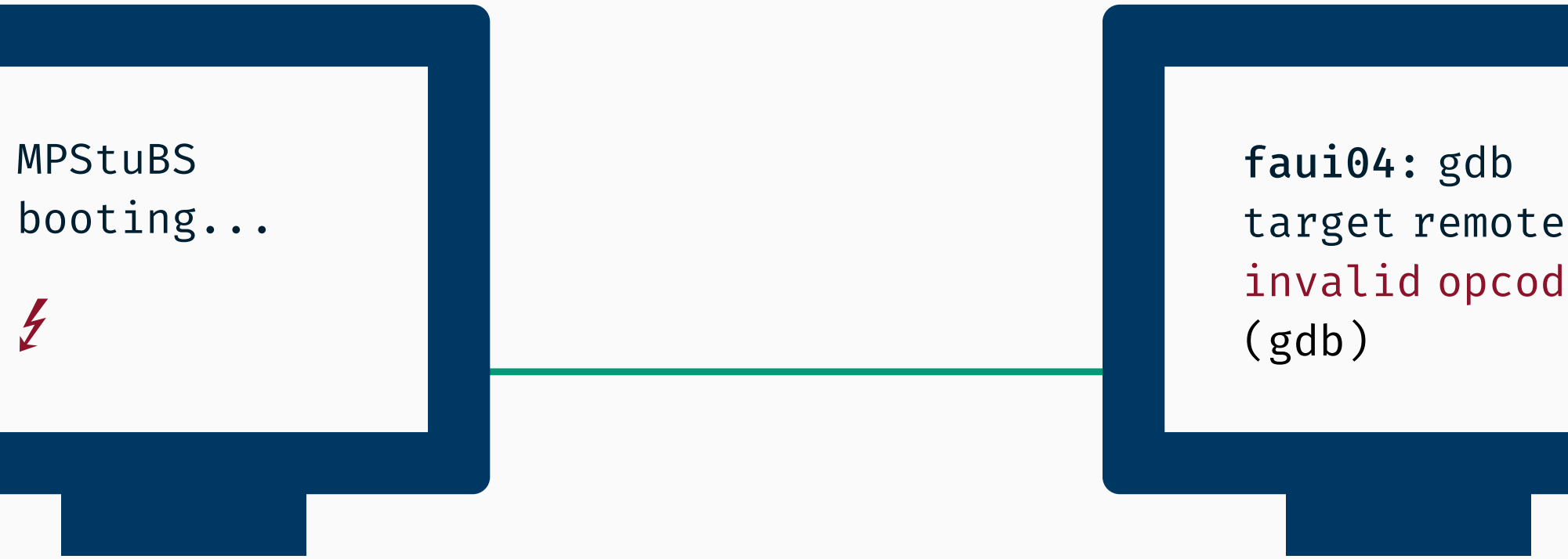

MPStuBS
booting...
+ ->
faui04: gdb
target remote


MPStuBS
booting...
faui04: gdb
target remote

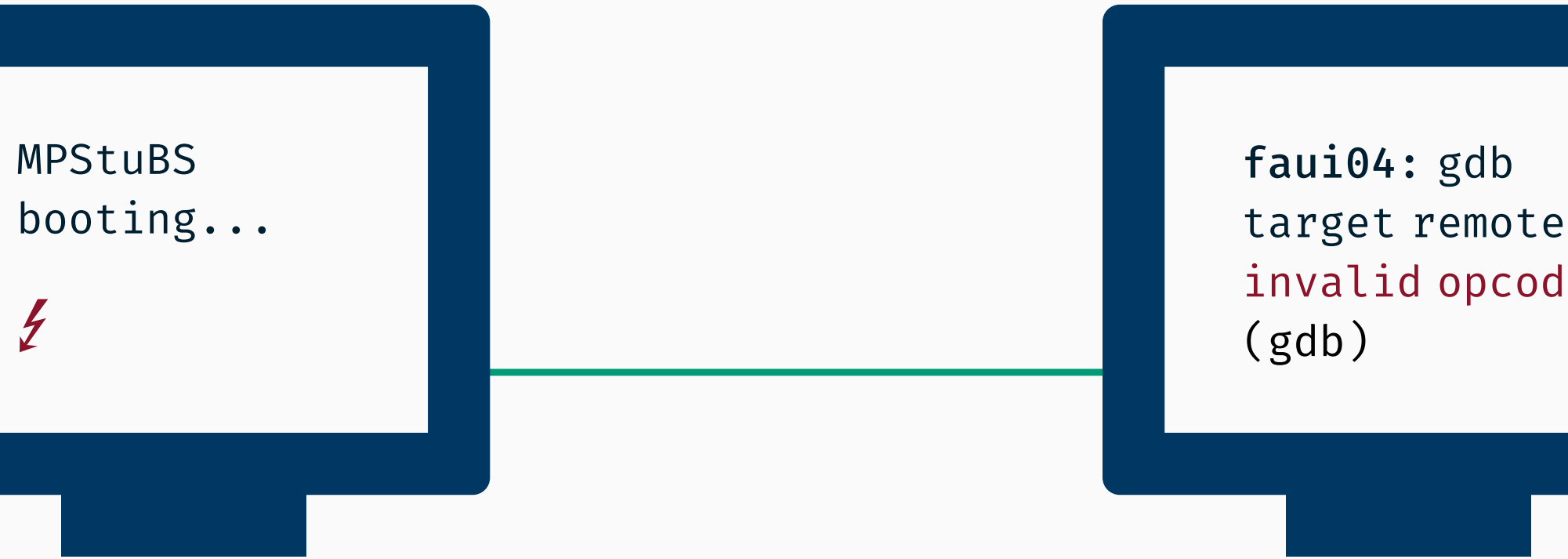

MPStuBS
booting...
faui04: gdb
target remote


MPStuBS
booting...
?S8 ->
faui04: gdb
target remote

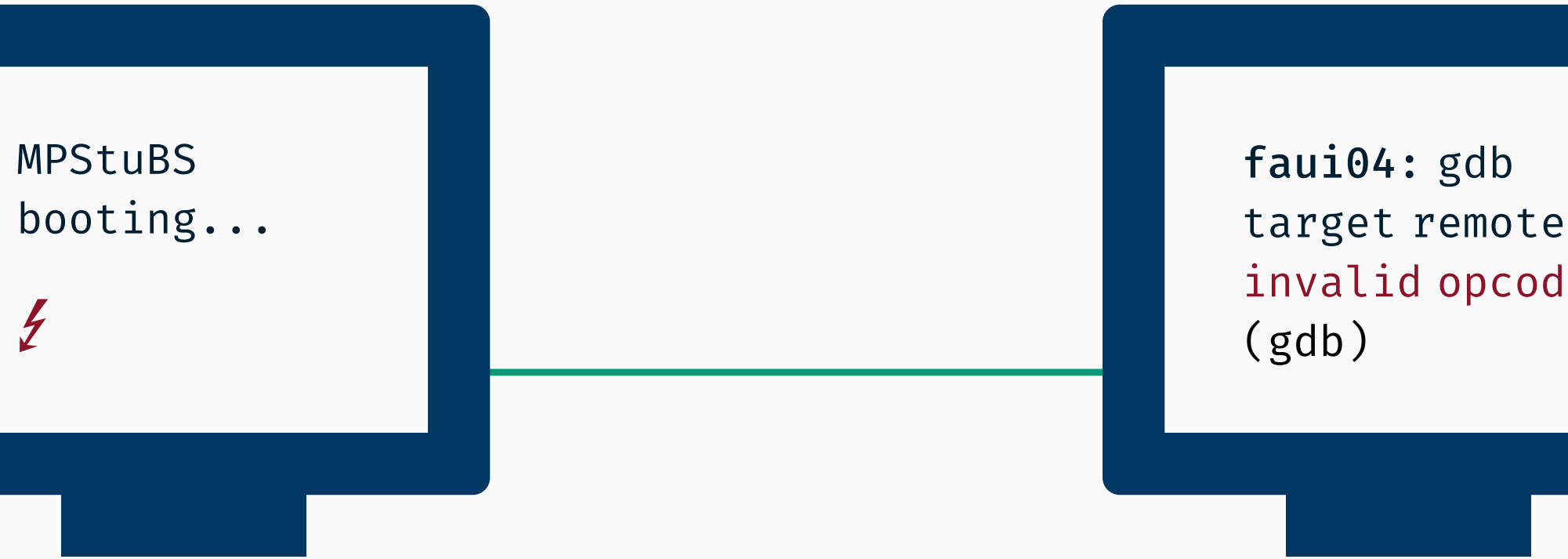

MPStuBS
booting...
faui04: gdb
target remote
invalid opcod
(gdb)



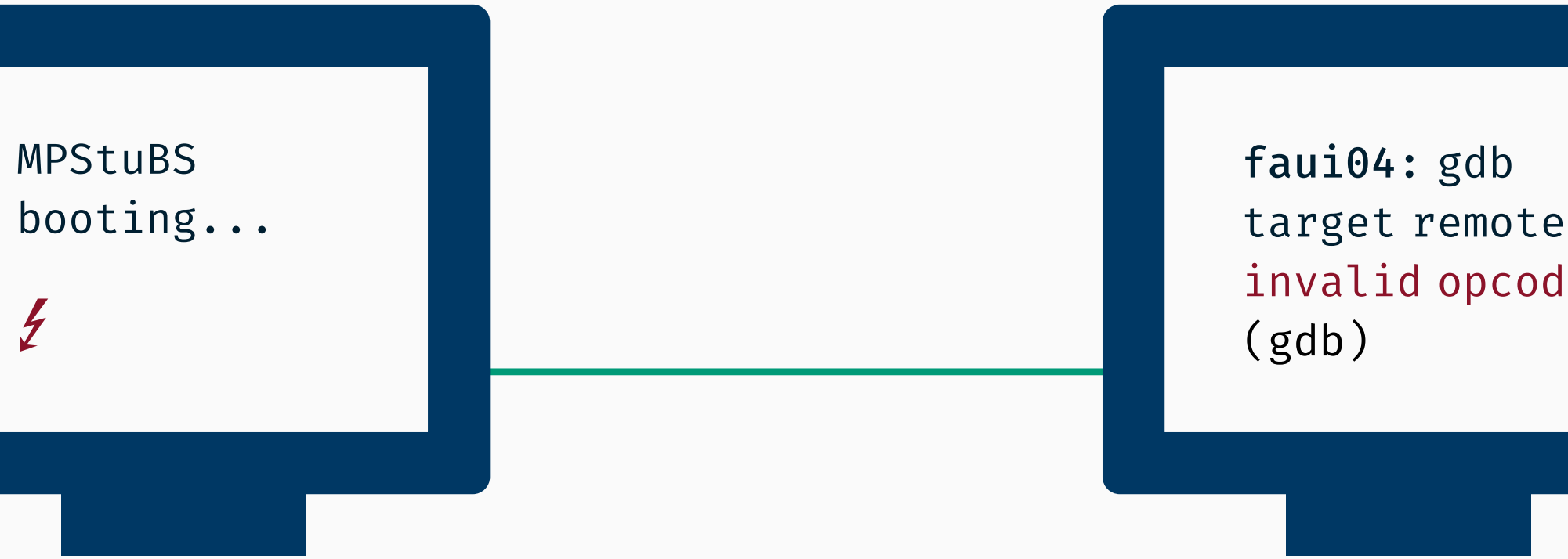
- benötigt serielle Schnittstelle (von Aufgabe 1)



- benötigt serielle Schnittstelle (von Aufgabe 1)
- Protokoll ist bereits implementiert

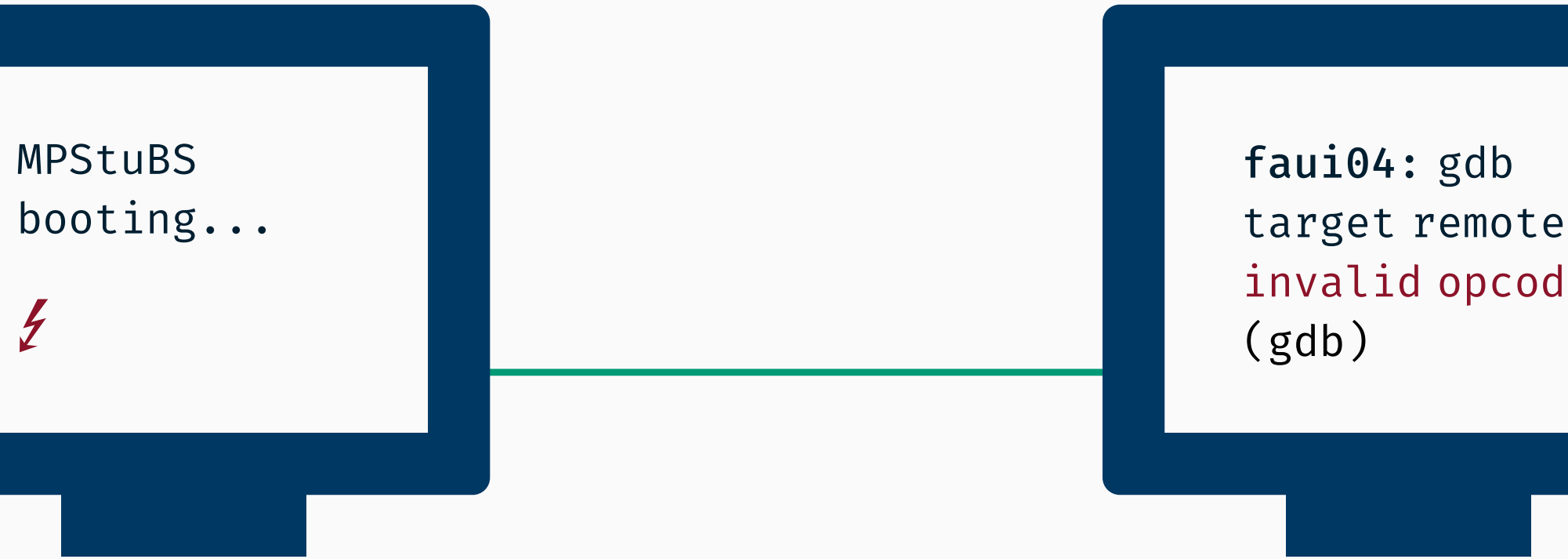


- benötigt serielle Schnittstelle (von Aufgabe 1)
- Protokoll ist bereits implementiert
- verwendet eigene IRQ-Handler für Traps



- benötigt serielle Schnittstelle (von Aufgabe 1)
- Protokoll ist bereits implementiert
- verwendet eigene IRQ-Handler für Traps

→ „nur“ die IDT muss bearbeitet werden



- benötigt serielle Schnittstelle (von Aufgabe 1)
- Protokoll ist bereits implementiert
- verwendet eigene IRQ-Handler für Traps

→ „nur“ die IDT muss bearbeitet werden

- aber ist freiwillig

Fragen?

Zur Erinnerung:
Nächste Woche (16. & 19. November)
Abgabe von Aufgabe 1 im Huber-CIP
(keine Tafelübung!)