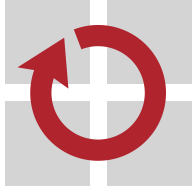
Übung zu Betriebssysteme

Interruptbehandlung

08. & 10. November 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

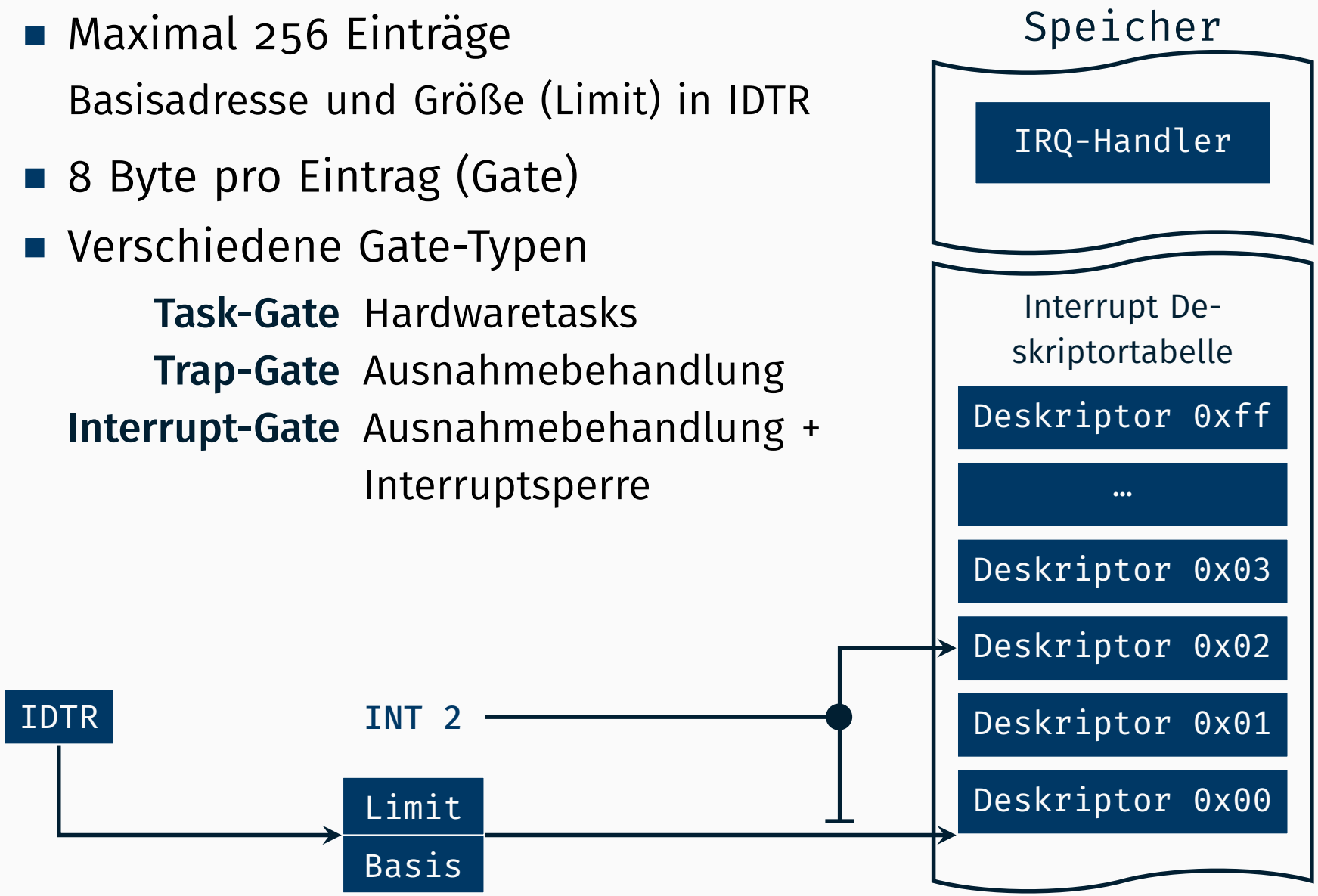
TECHNISCHE FAKULTÄT

Interrupts und Traps beim x86 Prozessor

Intel 80386: Interrupt Deskriptor Tabelle (IDT)

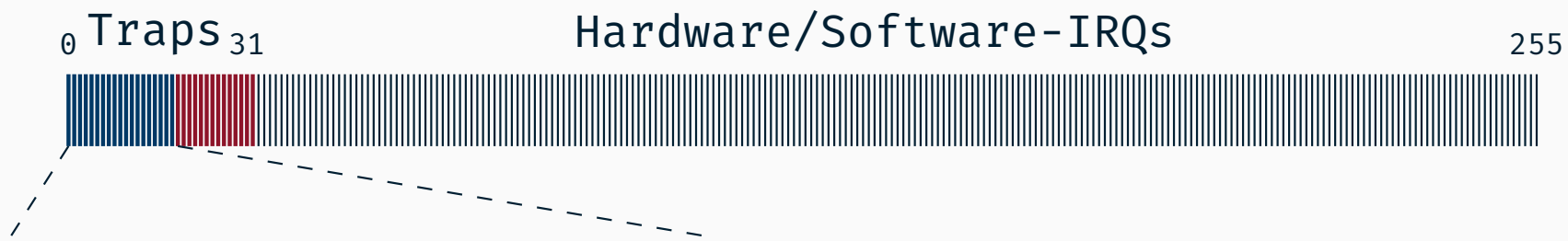
- Maximal 256 Einträge
  Basisadresse und Größe (Limit) in IDTR
- 8 Byte pro Eintrag (Gate)
- Verschiedene Gate-Typen

Task-Gate	Hardwaretasks
Trap-Gate	Ausnahmebehandlung
Interrupt-Gate	Ausnahmebehandlung + Interruptsperre



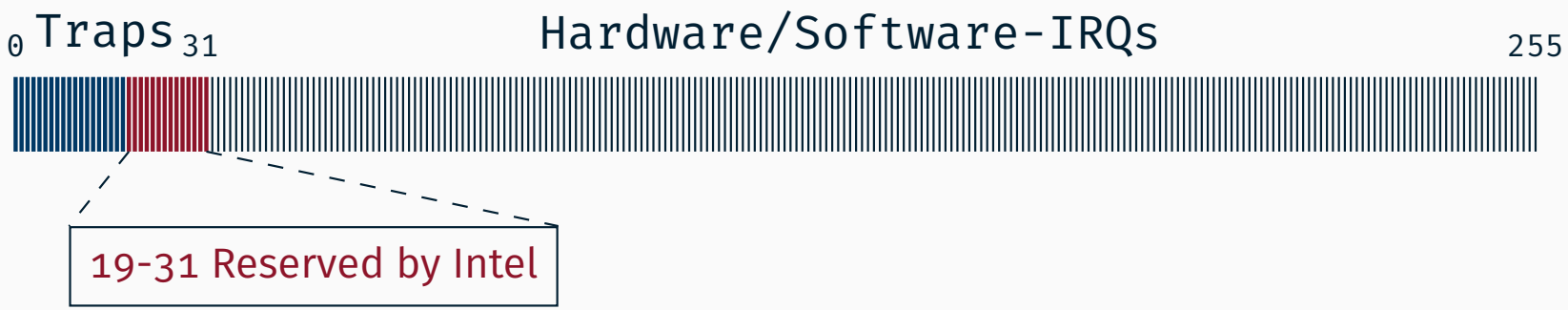
Intel 80386: Interruptvektoren in der IDT

Intel 80386: Interruptvektoren in der IDT

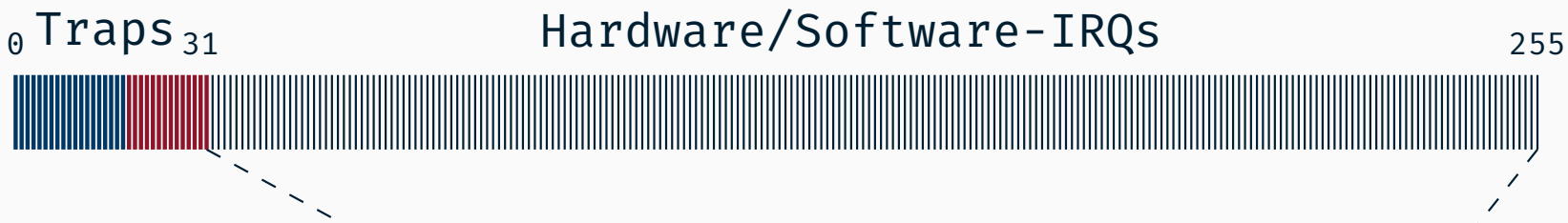


0	**Division-by-Zero**
1	Debug Exception
2	Non-Maskable Interrupt(NMI)
3	**Breakpoint (INT 3)**
4	Overflow Exception
5	Bound Exception
6	**Invalid Opcode**
7	FPU not Available
8	Double Fault
9	Coprocessor Segment Overrun
10	Invalid TSS
11	Segment not Present
12	Stack Exception
13	**General Protection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

Intel 80386: Interruptvektoren in der IDT



Intel 80386: Interruptvektoren in der IDT



32-255 Einträge für IRQs

- Softwareauslösung mit `int <vec#>`
- Hardwareauslösung durch externe Geräte
  APIC löst zugeordnete Unterbrechung aus

Hardware-IRQs bei x86-CPUs

Traditionell nur eine INT-Leitung (`INT`) an der CPU

- `INT` kann durch Prozessorbefehle maskiert werden
  - **`cli`** (Clear Interrupt Flag) Interruptleitung sperren
  - **`sti`** (Set Interrupt Flag) Interruptleitung freigeben
- Anschluss von mehreren externen Geräten durch externen Controller
  - traditionell ist dies der Programmable Interrupt Controller (`PIC 8259A`)
  - bietet u.a. keine Unterstützung für Mehrprozessorsysteme

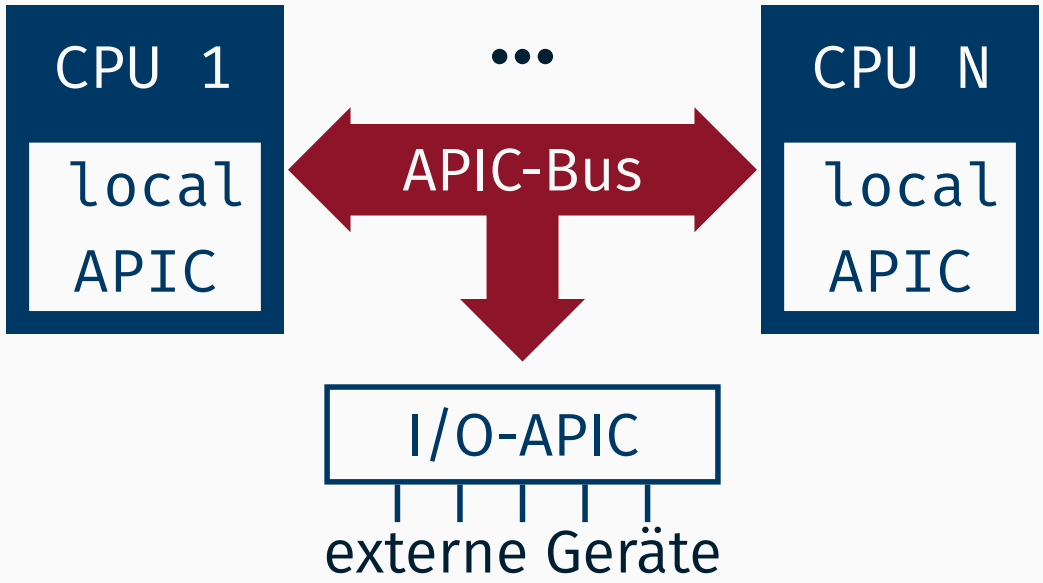
Hardware-IRQs bei x86-CPUs

APIC-Architektur bei „modernen“ PCs

- mehr Anschlussmöglichkeiten für externe Geräte (24 vs. 8 bzw. 15)
- freie Konfigurierbarkeit des Interruptvektors pro Gerät möglich
- wird für Multiprozessorsysteme auf jeden Fall benötigt
- funktioniert natürlich auch in einem Einprozessorsystem

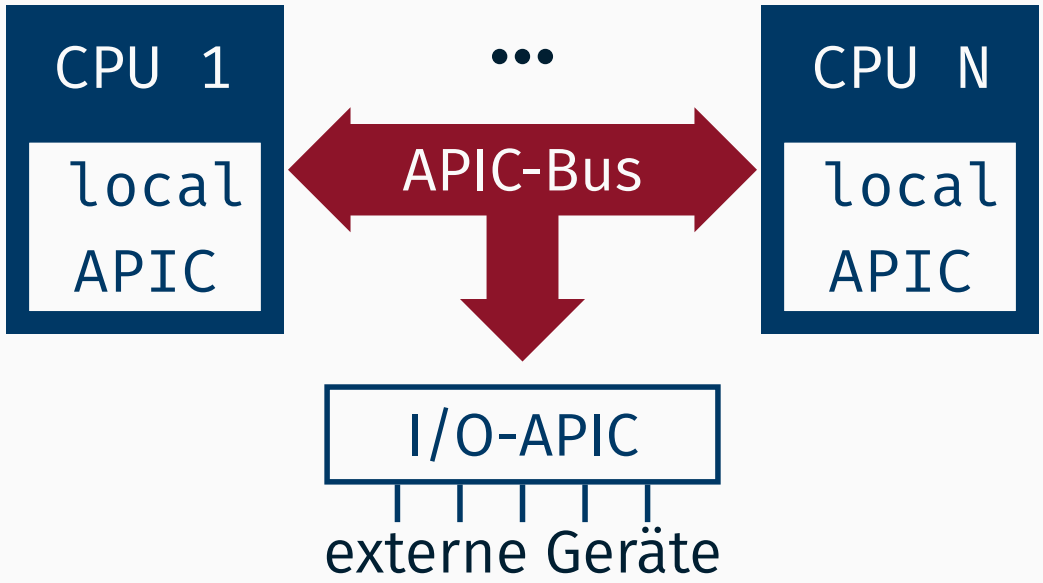
Externe Interrupts mit dem APIC

Aufbau der APIC-Architektur



Aufteilung in lokalen APIC und I/O APIC

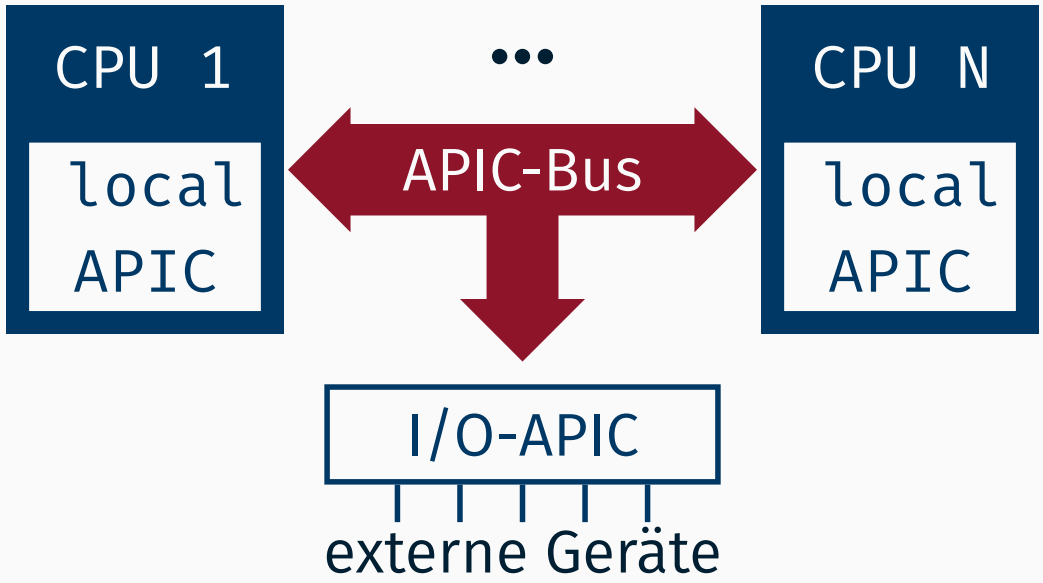
Aufbau der APIC-Architektur



I/O-APIC dient zum Anschluss der Geräte

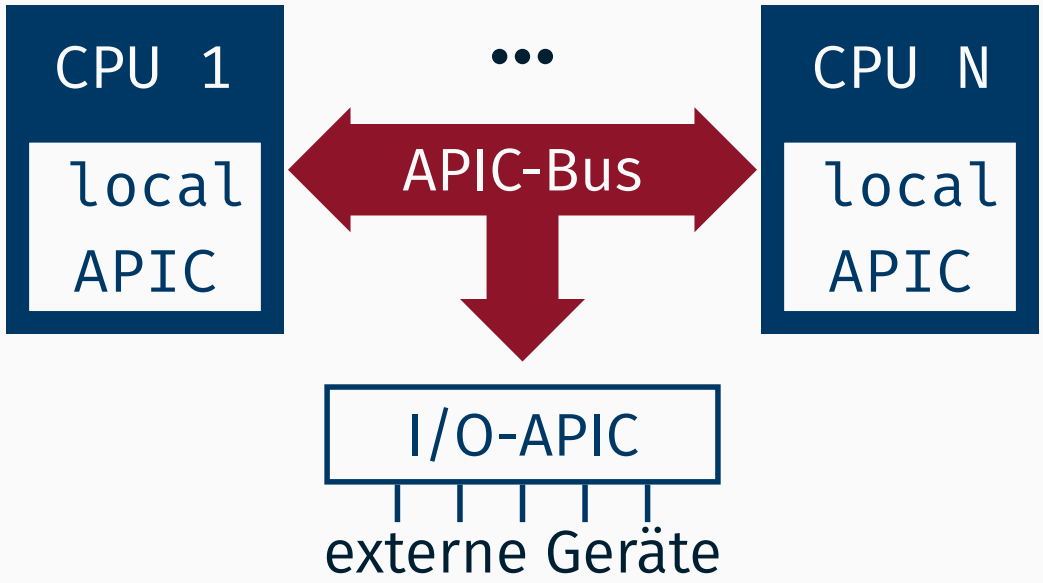
- Zuweisung von beliebigen Vektornummern
- Aktivieren und Deaktivieren von einzelnen Interruptquellen
- Zuweisung von Zielprozessoren für einzelnen Interrupts in MP-Systemen

Aufbau der APIC-Architektur



Interrupts werden zu Nachrichten auf dem APIC-Bus

Aufbau der APIC-Architektur



Empfang durch Local APIC

- 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

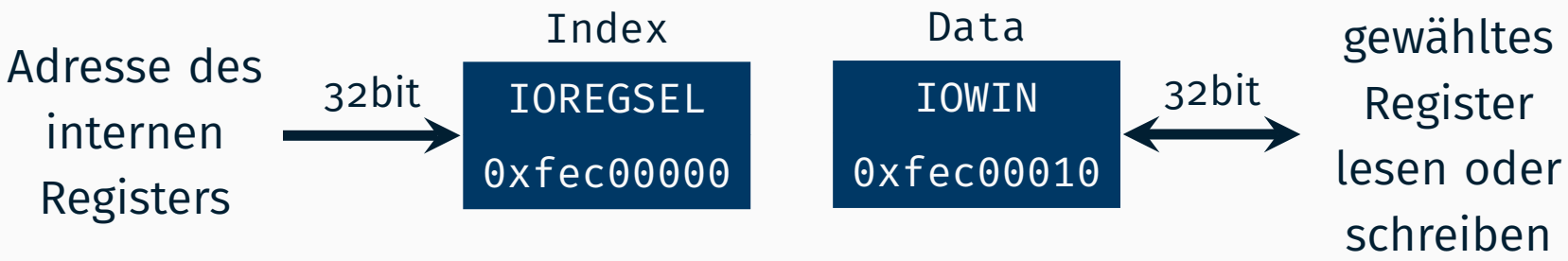
Zugriff auf die internen Register über memory-mapped Ein-/Ausgabe

- 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

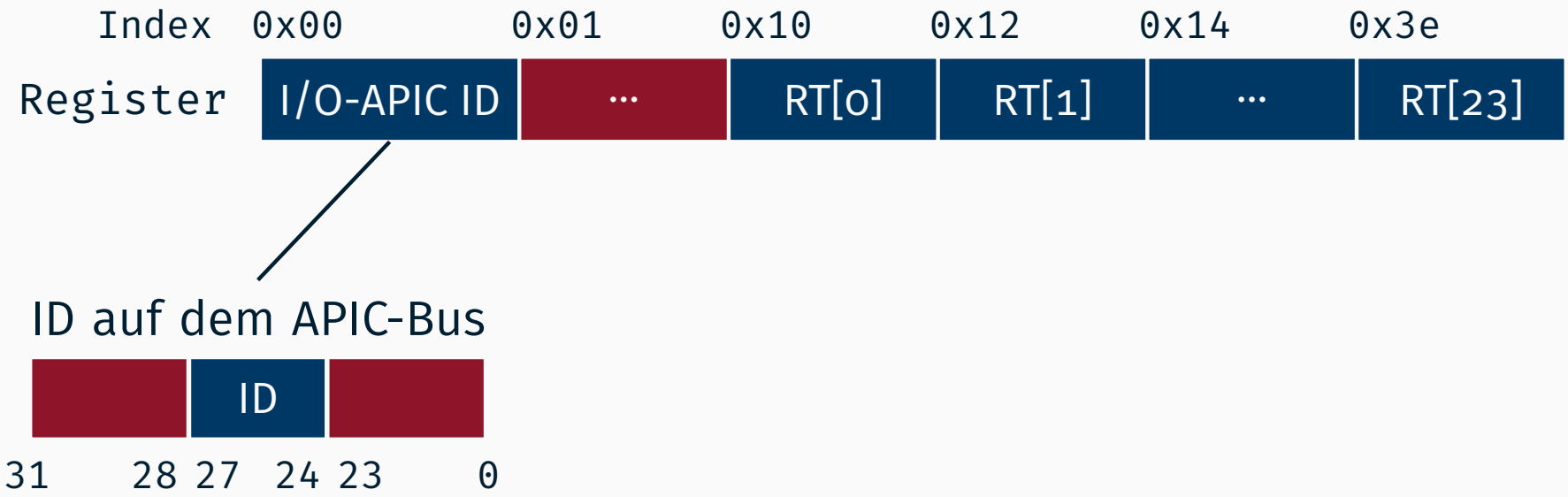


Programmierung des Intel I/O-APIC (2)

Interne Register des I/O-APICs

Programmierung des Intel I/O-APIC (2)

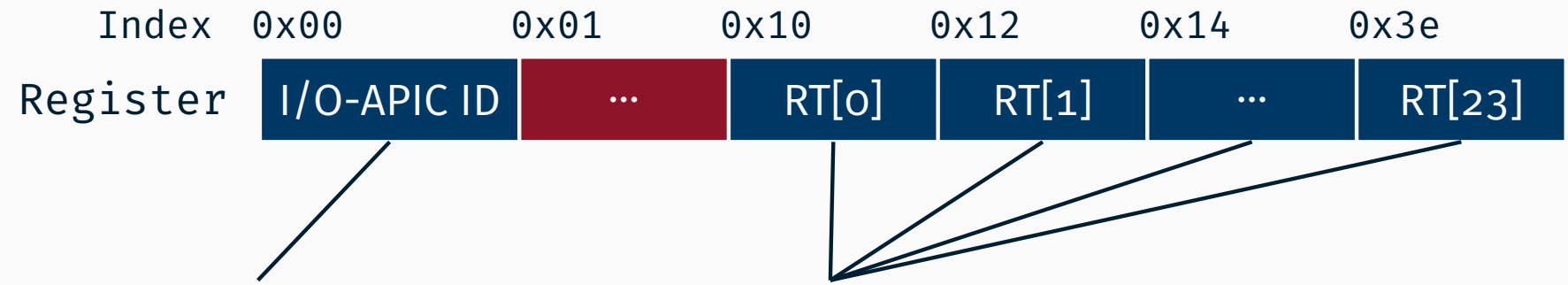
Interne Register des I/O-APICs



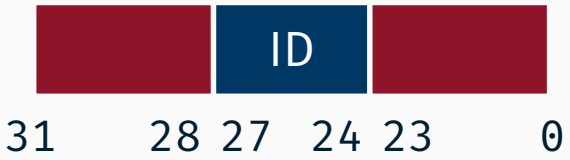
ID auf dem APIC-Bus

Programmierung des Intel I/O-APIC (2)

Interne Register des I/O-APICs



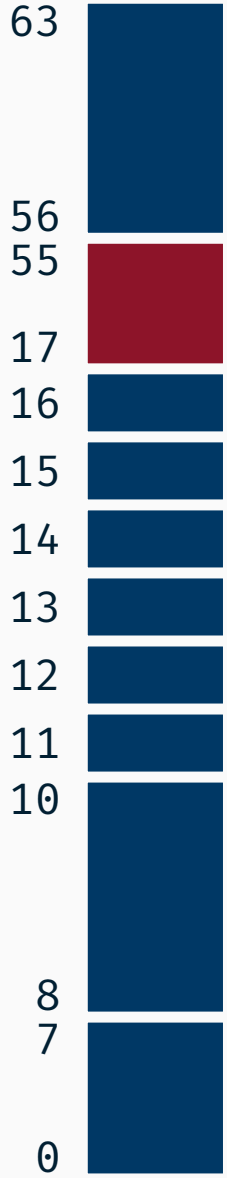
ID auf dem APIC-Bus



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

- APIC ID der Ziel-CPU falls Dest. Mode == Physical
- Gruppe von Ziel-CPUs falls Dest. Mode == Logical

reserviert

Interrupt-Mask: Interrupt aktiv (`0`) oder inaktiv (`1`)
Trigger Mode: Flanken-(`0`) oder Pegel-(`1`)steuerung
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`)

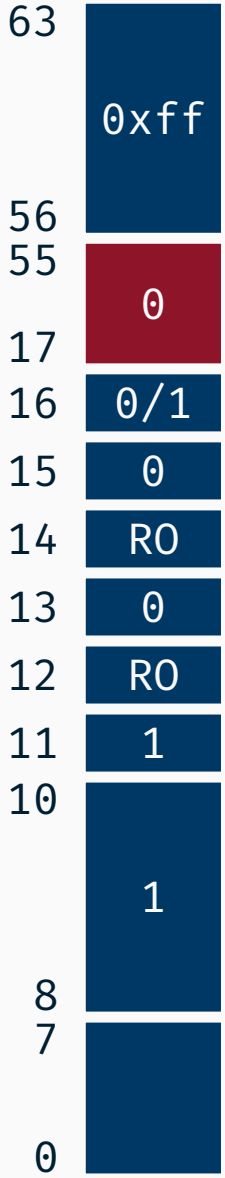
  `APICSystem::getIOAPICID` liefert die ID des IOAPICs

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 auf `0xff`: Potentiell jede CPU kann Empfänger sein

Redirection Table Einträge in OOStuBS/MPStuBS



Destination Field: Zieladresse des IRQs

- APIC ID der Ziel-CPU falls Dest. Mode == Physical
- Gruppe von Ziel-CPUs falls Dest. Mode == Logical

reserviert

Interrupt-Mask: Interrupt aktiv (0) oder inaktiv (1)
Trigger Mode: Flanken-(0) oder Pegel-(1)steuerung
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)

Einschub: Ansprechen von einzelnen Bits

- Bitmasken und logischen Bitoperationen

Einschub: Ansprechen von einzelnen Bits

- Bitmasken und logischen Bitoperationen
- Bitfelder in C/C++

```
struct cga_attrib {
    unsigned char fg : 4,
                  bg : 3,
                  bl : 1;
};

struct cga_attrib a;
a.fg = 15;
a.bg = 1;
a.blink = 1;
```

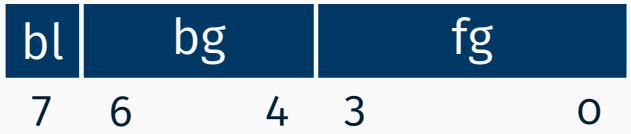
Einschub: Ansprechen von einzelnen Bits

- Bitmasken und logischen Bitoperationen
- Bitfelder in C/C++

```
struct cga_attrib {
    unsigned char fg : 4,
                  bg : 3,
                  bl : 1;
};

struct cga_attrib a;
a.fg = 15;
a.bg = 1;
a.blink = 1;
```

Layout mit GCC auf x86:



Zustandssicherung beim Auftreten von Interrupts

- Minimaler zu sichernder Zustand?

Zustandssicherung beim Auftreten von Interrupts

- Minimaler zu sichernder Zustand?
- CPU sichert automatisch
  - Condition Codes (**eflags**)
  - Aktuelles Code Segment (**cs**)
  - Programmzeiger/Rücksprungadresse (**eip**)

Zustandssicherung beim Auftreten von Interrupts

- Minimaler zu sichernder Zustand?
- CPU sichert automatisch
  - Condition Codes (**eflags**)
  - Aktuelles Code Segment (**cs**)
  - Programmzeiger/Rücksprungadresse (**eip**)

- Wiederherstellung des ursprünglichen Prozessorzustandes durch Befehl `iret`

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

Kontextsicherung in Hochsprachen

Aufteilung der Register in 2 Subsets

Kontextsicherung in Hochsprachen

Aufteilung der Register in 2 Subsets

- 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

Kontextsicherung in Hochsprachen

Aufteilung der Register in 2 Subsets

- 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**

Kontextsicherung in Hochsprachen

Aufteilung der Register in 2 Subsets

- 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!

Umsetzung in OOStuBS/MPStuBS

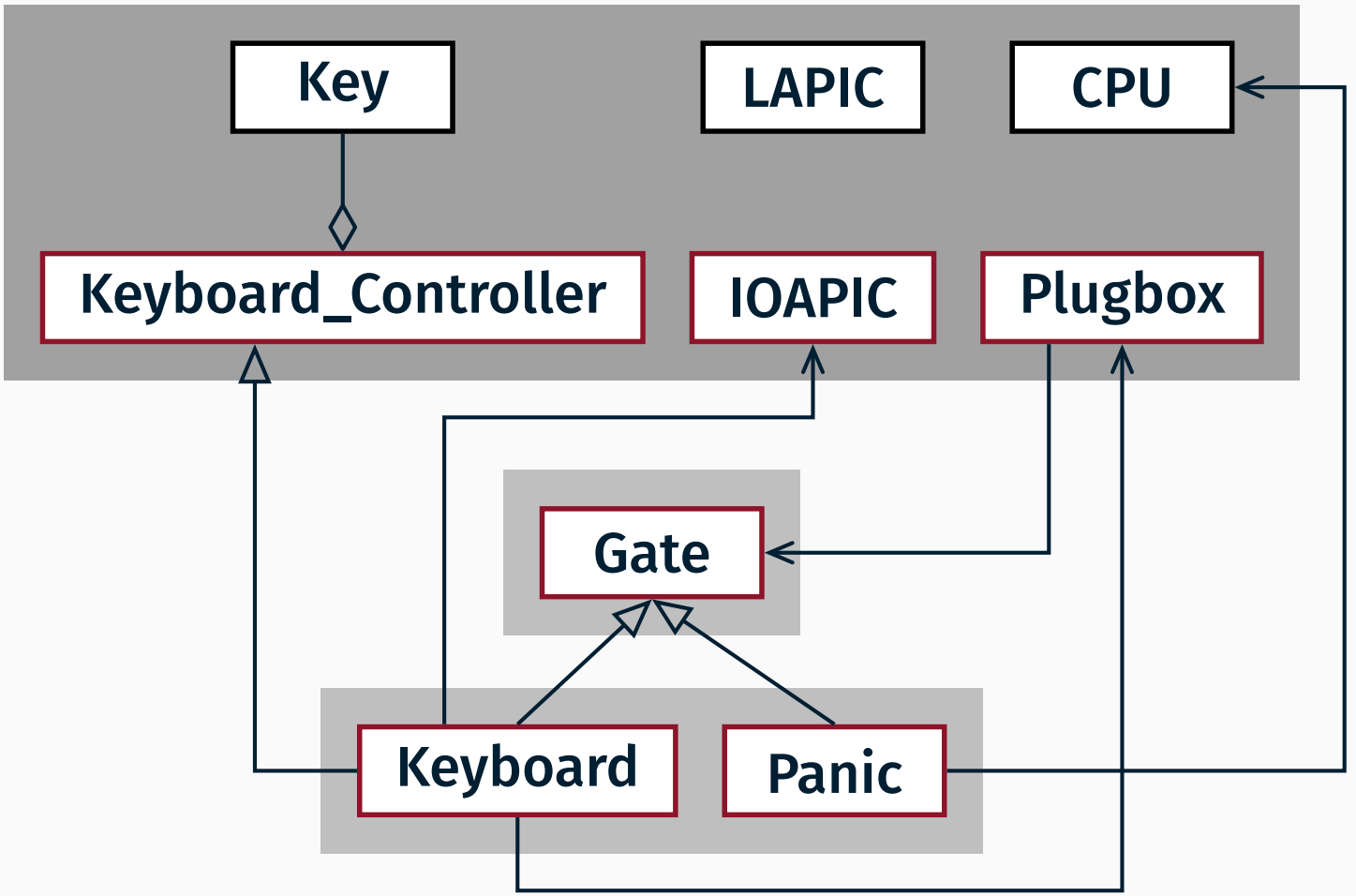
Interrupthandler in OOStuBS/MPStuBS

- Behandlung startet in der Funktion `guardian(int)`
  - bekommt die IRQ-Nummer als Parameter:

```
01 void guardian(unsigned int vector) {
02     // IRQ-Handler (Gate) aktivieren
03 }
```

- Interrupts sind während der Abarbeitung gesperrt
  - können mit `sti/CPU::enable_int()` manuell wieder zugelassen werden
  - werden automatisch wieder zugelassen, wenn `guardian(int)` terminiert
- Die eigentlichen (spezifischen) IRQ-Handler
  - sind Instanzen der Klasse Gate
  - werden an- und abgemeldet über die Klasse Plugbox

Zu implementierende Klassen



Fragen?

Zur Erinnerung:
Nächste Woche (15. & 17. November)
Abgabe von Aufgabe 1 im Huber-CIP
(keine Tafelübung!)