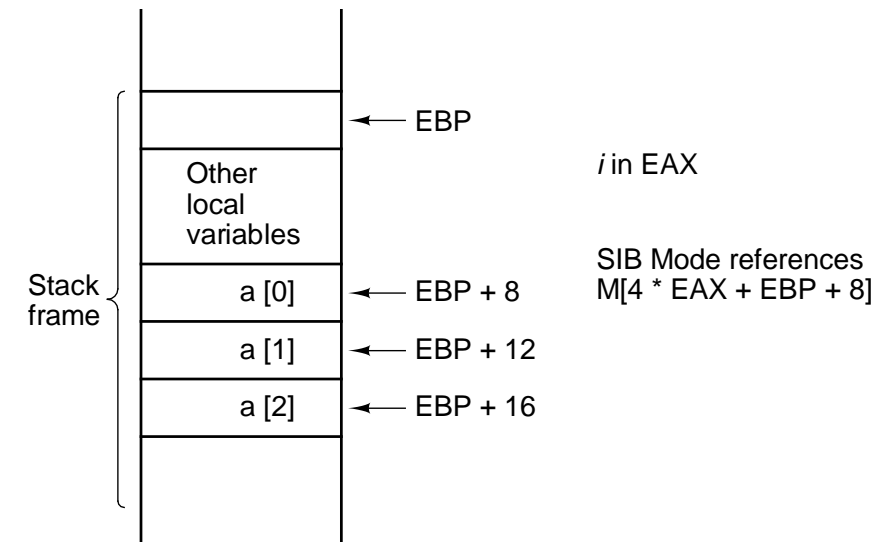
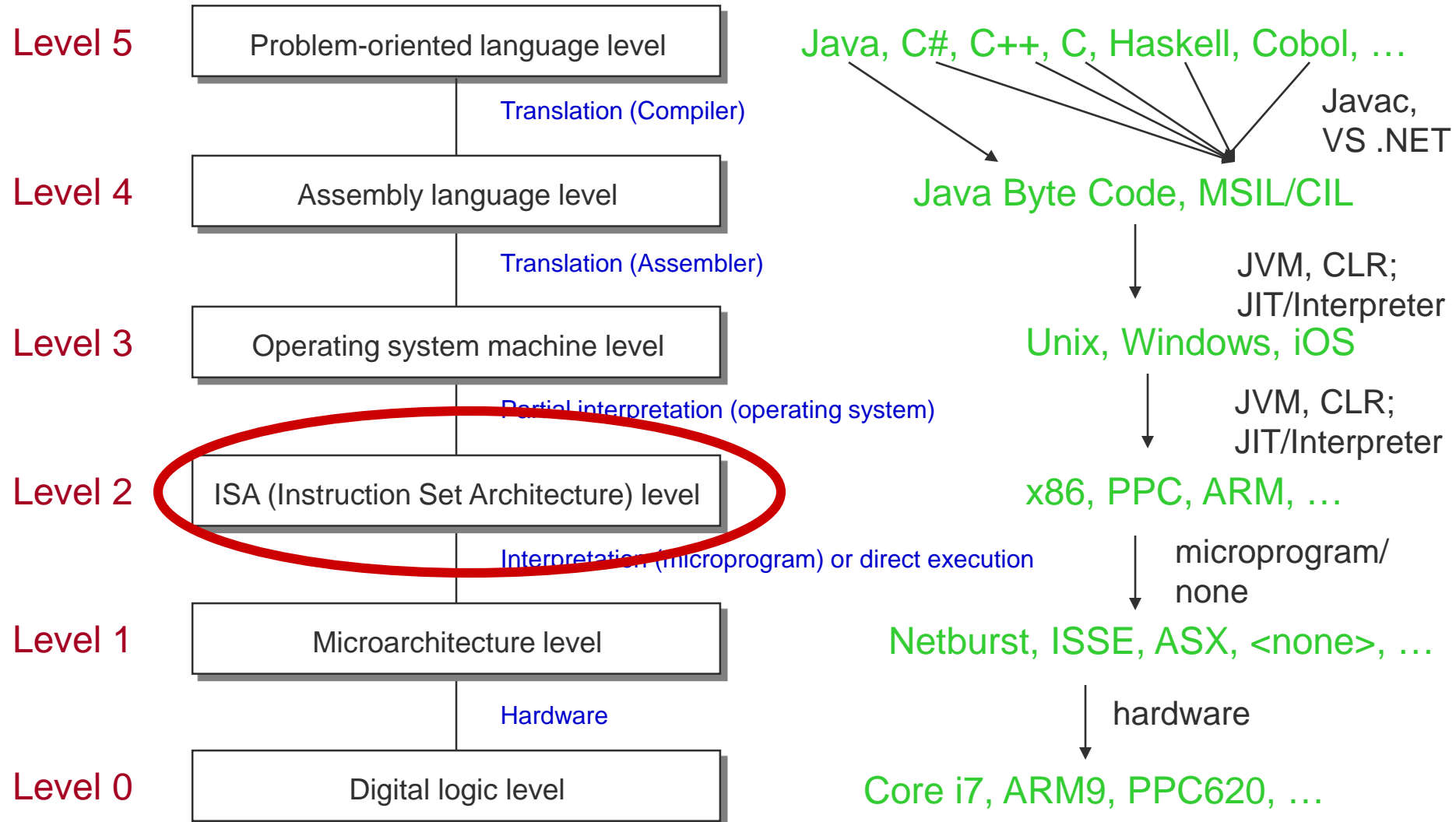


# TI II: Computer Architecture ISA and Assembly

**CISC vs. RISC**  
**Data Types**  
**Addressing**  
**Instructions**  
**Assembler**



## Where are we now?



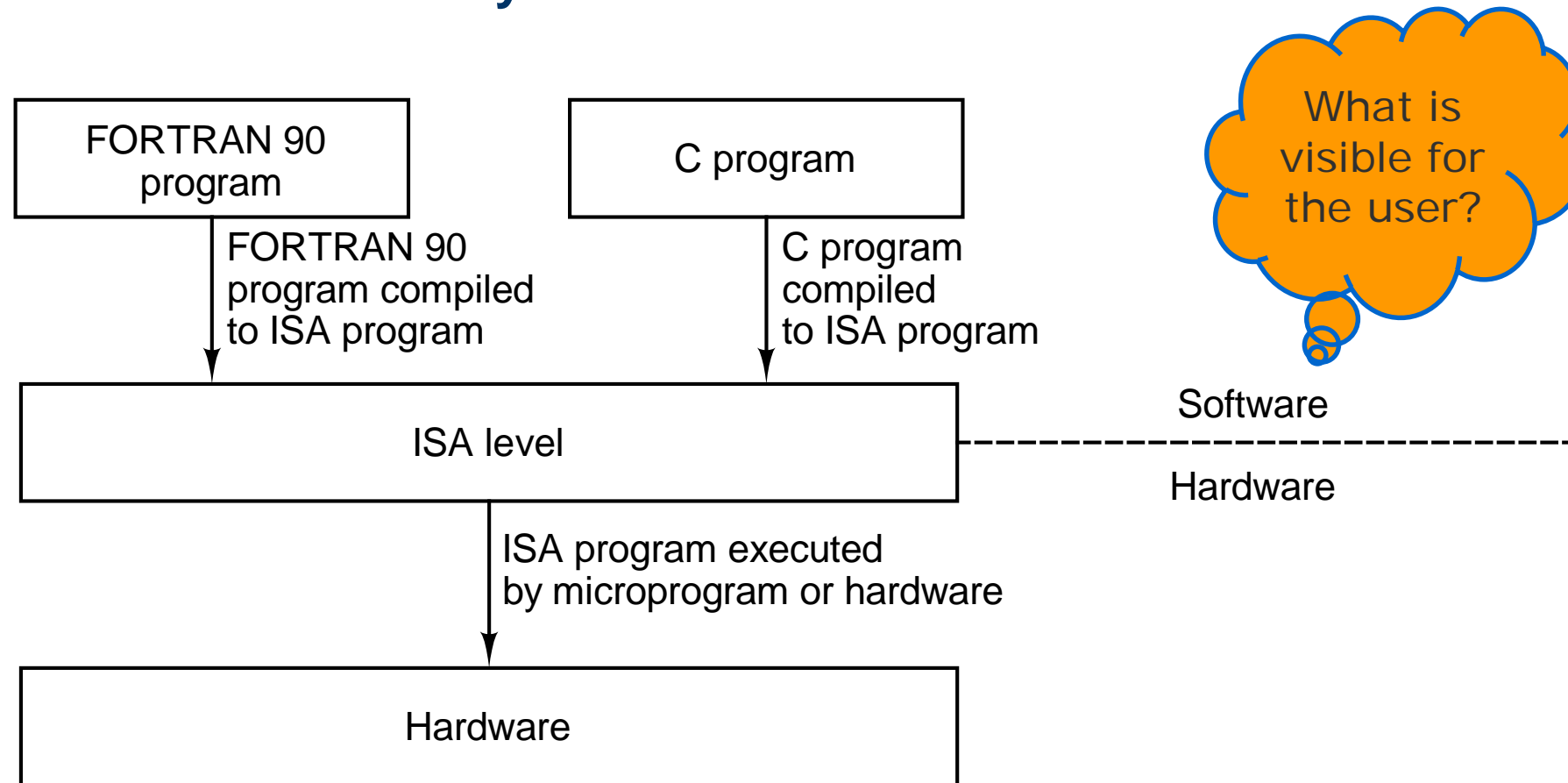
## Information

This chapter is mainly for some background information about ISAs, assembler etc.

The assignments plus tutorials teach you how to program with assembler

We skip the operation system part as this is covered in the course operating systems and networks

## The classical SW/HW boundary



## Execution of operations

Interpretation or compilation of instructions from the ISA-layer

- High-level languages translated into instructions from the ISA
- Pure RISC processors can directly execute ISA instructions
  - i.e. the hardware (HW) can execute these instructions
- More complex processors use **microprogramming**, flexibility and compatibility reasons:
  - hardware changes leave ISA unchanged
  - reprogramming to circumvent hardware problems
  - more powerful ISA – simpler compilers

# A selection of the Pentium II integer instructions

## Moves

MOV DST, SRC	Move SRC to DST
PUSH SRC	Push SRC onto the stack
POP DST	Pop a word from the stack to DST
XCHG DS1, DS2	Exchange DS1 and DS2
LEA DST, SRC	Load effective addr of SRC into DST
CMOV DST, SRC	Conditional move

## Arithmetic

ADD DST, SRC	Add SRC to DST
SUB DST, SRC	Subtract DST from SRC
MUL SRC	Multiply EAX by SRC (unsigned)
IMUL SRC	Multiply EAX by SRC (signed)
DIV SRC	Divide EDX:EAX by SRC (unsigned)
IDIV SRC	Divide EDX:EAX by SRC (signed)
ADC DST, SRC	Add SRC to DST, then add carry bit
SBB DST, SRC	Subtract DST & carry from SRC
INC DST	Add 1 to DST
DEC DST	Subtract 1 from DST
NEG DST	Negate DST (subtract it from 0)

## Binary coded decimal

DAA	Decimal adjust
DAS	Decimal adjust for subtraction
AAA	ASCII adjust for addition
AAS	ASCII adjust for subtraction
AAM	ASCII adjust for multiplication
AAD	ASCII adjust for division

## Transfer of control

JMP ADDR	Jump to ADDR
Jxx ADDR	Conditional jumps based on flags
CALL ADDR	Call procedure at ADDR
RET	Return from procedure
IRET	Return from interrupt
LOOPxx	Loop until condition met
INT ADDR	Initiate a software interrupt
INTO	Interrupt if overflow bit is set

## Boolean

AND DST, SRC	Boolean AND SRC into DST
OR DST, SRC	Boolean OR SRC into DST
XOR DST, SRC	Boolean Exclusive OR SRC to DST
NOT DST	Replace DST with 1's complement

## Shift/rotate

SAL/SAR DST, #	Shift DST left/right # bits
SHL/SHR DST, #	Logical shift DST left/right # bits
ROL/ROR DST, #	Rotate DST left/right # bits
RCL/RCR DST, #	Rotate DST through carry # bits

## Test/compare

TST SRC1, SRC2	Boolean AND operands, set flags
CMP SRC1, SRC2	Set flags based on SRC1 - SRC2

## Strings

LODS	Load string
STOS	Store string
MOVS	Move string
CMPS	Compare two strings
SCAS	Scan Strings

## Condition codes

STC	Set carry bit in EFLAGS register
CLC	Clear carry bit in EFLAGS register
CMC	Complement carry bit in EFLAGS
STD	Set direction bit in EFLAGS register
CLD	Clear direction bit in EFLAGS reg
STI	Set interrupt bit in EFLAGS register
CLI	Clear interrupt bit in EFLAGS reg
PUSHFD	Push EFLAGS register onto stack
POPFD	Pop EFLAGS register from stack
LAHF	Load AH from EFLAGS register
SAHF	Store AH in EFLAGS register

## Miscellaneous

SWAP DST	Change endianness of DST
CWQ	Extend EAX to EDX:EAX for division
CWDE	Extend 16-bit number in AX to EAX
ENTER SIZE, LV	Create stack frame with SIZE bytes
LEAVE	Undo stack frame built by ENTER
NOP	No operation
HLT	Halt
IN AL, PORT	Input a byte from PORT to AL
OUT PORT, AL	Output a byte from AL to PORT
WAIT	Wait for an interrupt
SRC = source	# = shift/rotate count
DST = destination	LV = # locals

# COMPLEX INSTRUCTION SET COMPUTER (CISC)

# Complex Instruction Set Computer (CISC)

Entstehungsgründe für umfangreiche Maschinenbefehlssätze

- Geschwindigkeitsunterschied zwischen CPU und Hauptspeicher
- Mikroprogrammierung
- Kompakter Code
- Unterstützung höherer Programmiersprachen
- Aufwärtskompatibilität
- Marktstrategie



# Complex Instruction Set Computer (CISC)

## Gründe dafür

- Ausführung komplizierter Befehle ist immer noch schneller als die Ausführung von Programmen gleicher Funktion
- Mikroprogrammierung begünstigt komplizierte Befehle
- Komplizierte Befehle führen zu kürzeren Programmen, also schnelleres Laden
- Umfang des Befehlssatzes wird oft als Werbeargument verwendet
- Unterstützung höherer Programmiersprachen durch komplizierte Befehle (Direkte Abbildung: Sprachkonstrukt → Befehl)
- Unterstützung von Compilern durch entsprechende Befehle
- Unterstützung spezieller Einsatzgebiete

## Fazit

- Entwicklung von Hardware, Programmiersprachen und Einsatzgebieten begünstigt „komplizierte“ Befehle

# Complex Instruction Set Computer (CISC)

Gründe dagegen

- Schnellere Hauptspeicher (Argument der 80er, heute wieder problematisch!) und die Verwendung von Cache-Speichern beschleunigen die Programmausführung
- Mikroprogramme wurden immer komplizierter (immer geringere Unterschiede zwischen Programmierung und Mikroprogrammierung!)
- Ersetzung komplexer Maschinenbefehle durch mehrere einfache Maschinenbefehle
- Verlängerte Entwurfszeit
- Sehr komplexe Steuerwerke
- Sehr umfangreiche Mikroprogramme
- Nur relativ kleine Teile des großen Befehlssatzes werden häufig benutzt

## CISC – ursprüngliche Motivation

Systemprogramme in XPL auf IBM/360:

- 90% aller ausgeführten Befehle: 10 verschiedene Befehle
- 95% aller ausgeführten Befehle: 21 verschiedene Befehle
- 99% aller ausgeführten Befehle: 30 verschiedene Befehle

COBOL-Programme auf IBM/370:

- 90,28% aller ausgeführten Befehle: 26 verschiedene Befehle
- 99,08% aller ausgeführten Befehle: 48 verschiedene Befehle
- Nur 84 verschiedene Befehle wurden überhaupt benutzt

## The 10 most used instructions in SPECint92 for Intel x86

Instruction	Percentage [%]
load	22
conditional branch	20
compare	16
store	12
add	8
and	6
sub	5
move register-register	4
call	1
return	1
<b>Total</b>	<b>95</b>

## Grenzen der CISC Architekturen

Befehlsausnutzung (80/20 Regel)

- nur 20% der Befehle werden überwiegend benutzt
- Viele mächtige Befehle
- komplexes Befehlsformat
- Mikroprogrammierung

Kritisches Problem: Anzahl der Zyklen pro Instruktion (CPI)

- bei vielen CISC Architekturen ist  $CPI \gg 2$ 
  - Motorola MC68030:  $CPI = 4-6$
  - Intel 80386:  $CPI = 4-5$
- **ABER**: hochoptimiert in Pentium/Itanium – oft  $CPI \approx 1$

# REDUCED INSTRUCTION SET COMPUTER (RISC)

# Reduced Instruction Set Computer (RISC)

The instruction set consists of

- a few, absolutely necessary **instructions ( $\leq 128$ )** and
- **instruction formats ( $\leq 4$ )** with a
- **fixed instruction length** of 32 bit and only some
- **addressing modes ( $\leq 4$ )**.

This allows a much simpler implementation of the control unit and saves space on the chip for additional units.

Many general purpose registers, at least 32, are needed.

Memory access is only possible via special load and store instructions.

## Register/register architecture

Memory access is via load and store operations only.

All other instructions work on the CPU registers only, e.g., arithmetic operations load operands from registers and store results in registers only.

This basic principle is called

- **register/register architecture** or
- **load/store architecture** and is typical for many (original) RISC computers.



## Additional features of RISC computers

If possible all instructions should be implemented in a way that they finish **within a single processor cycle**.

Consequence: pure RISC processors do **not use** micro programming

- RISC processors introduced enhanced pipelining mechanisms (today, many processors use pipelining for the micro instructions, e.g., Pentium 4).

Furthermore, the early RISC processors had a software controlled pipeline (compilers inserted delay NOPs, introduced delayed jumps etc.) instead of special hardware.

### Aside

- PC processors like the Pentium 4 (and up) use micro programming, the internal micro architecture (netburst) is rather RISC, the ISA is CISC.

# RISC

## Gründe dafür

- Implementierung auf einem Chip
- Kurze Entwurfszeiten
- Hohe Taktraten, Pipelining
- Neue Verwendungsmöglichkeiten für die eingesparte Chipfläche

## Gründe dagegen

- Engpass der Speicheranbindung, heute wieder vergleichsweise langsame Speicher
- Chipfläche nicht mehr so kritisch wie früher

## Early RISC processors

### IBM 801 project

- Already 1975, Cocke, IBM research, Yorktown Heights

### MIPS project

- Started 1981, Hennessy at the University of Stanford
- The first fully functioning chip was finished in 1983 (NMOS VLSI)
- This project was the starting point of the MIPS corporation.

### Berkeley RISC project

- Started 1980, Patterson at UC Berkeley
- Origin of the SPARC processor
- Basic principle of overlapping register windows
- The instruction set contained only 31 instructions with a fixed length of 32 bit and only 2 instruction formats
- Only 3 addressing modes

## RISC-Rechner aus heutiger Sicht

... geblieben ist von der RISC-Idee im Wesentlichen

- das Befehlspipelining
- die Load/Store-Architektur
- ein großer Registersatz: z.B.
  - 32 allgemeine und
  - 32 Gleitpunkt-Register
- ein einheitliches Befehlsformat von z.B. 32 Bit
- die Verwendung weniger Adressierungsarten
- der Verzicht auf Mikroprogrammierung

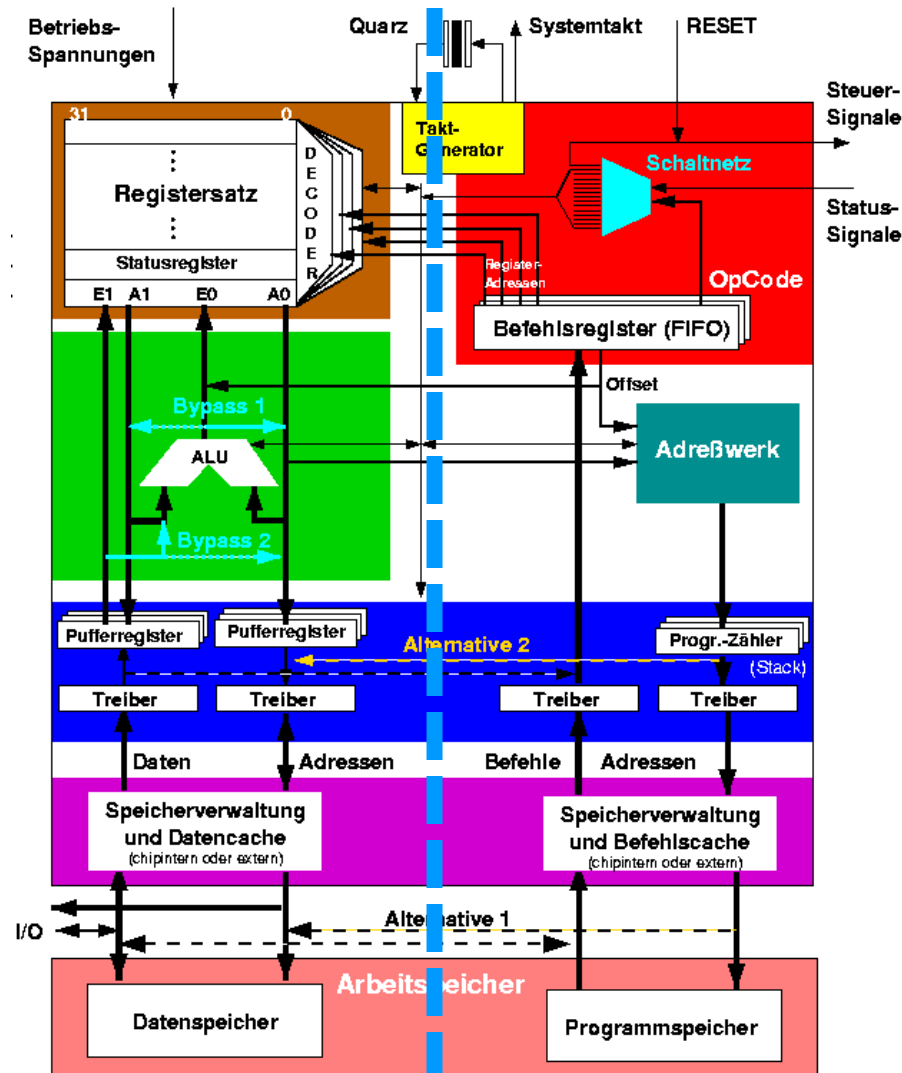
Anmerkung: der Kern eines Pentium 4 (und nachfolgende) folgt RISC-Prinzipien (nur eben für Mikrobefehle!)

# Aufbau eines (puristischen) RISC-Prozessors

Datenteil

Programmteil

Anm.: nicht mehr  
klassische  
von-Neumann  
Architektur!



# Unterschiede zwischen RISC- und CISC-Prozessoren

## Havard-Architektur

- getrennter Programm- und Datenspeicher, deshalb zwei Adress- und Datenbusse
- ➡ paralleles Holen von Operanden und Instruktionen

## Vereinfachende Varianten

1. zwei getrennte Bussysteme bis zu den Cache-Speichern, jedoch nur ein Arbeitsspeicher (niedrigere Kosten, heute Standard)
2. nur ein Bussystem wie bei Standard-Mikroprozessoren

# Unterschiede zwischen RISC- und CISC-Prozessoren

## Steuerwerk

- Festverdrahtet
- Das Befehlsregister als Warteschlange (FIFO) realisiert
- Für jede Pipeline-Stufe ist dort ein Register vorhanden
- Die OpCodes jeder Stufe können vom Schaltnetz des Steuerwerks ausgewertet werden

## Registersatz

- besteht aus einer großen Anzahl von Registern
- erlaubt gleichzeitige Auswahl von 3 bis 4 Registern
  - z.B. 4 Port Registersatz: gleichzeitiges Schreiben (E0, E1) und Lesen (A0, A1) von jeweils 2 Registern

# Unterschiede zwischen RISC- und CISC-Prozessoren

## Rechenwerk

- Besitzt eine Load/Store-Architektur. Die Operanden werden über 2 Operandenbusse aus dem Registersatz herbeigeführt, das Ergebnis (noch im selben Taktzyklus) über den Ergebnisbus in den Registersatz geschrieben.
- Normalerweise gibt es keine direkte Verbindung zwischen ALU und Systemdatenbus, Datentransfer läuft über die Register (Load/Store-Architektur).

## Ausnahme

- Register-Bypass zur Vermeidung von Pipeline-Hemmnissen (forwarding techniques)



## The future of RISC?

### Today

- Again, processors much faster than RAM/interconnection
- Frequent load/stores as bottleneck
- Integration of > 2 billion transistors on a single chip feasible

### Thus

- Development of VLIW (Very Large Instruction Word) processors
- HP/Intel Itanium, very short pipeline, compiler does most of the work, powerful ISA, less memory accesses
- **Commercial Failure!**

### The future?

- RISC considered harmful? Not in embedded systems...
- Will legacy stay there forever...
  - seems so with Intel 64 and similar...

Examples of ISAs

**PENTIUM, SPARC, JVM**

## Overview: Pentium, SPARC, JVM

The following processors serve as **examples** for different ISAs

### Pentium

- Originates from the classical x86 CISC architectures
- Still CISC to the outside, but many RISC features inside
- Other CISC examples: Athlon, many old processors (VAX, IBM, ...)

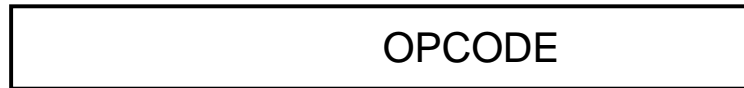
### UltraSPARC (Ultra Scalable Processor Architecture)

- Originates from the early RISC projects (like the MIPS processor)
- Still RISC, although extended in many ways
- Can be found in, e.g., SUN computers, industry control systems
- Other RISC examples: Alpha, MIPS, Power, PowerPC

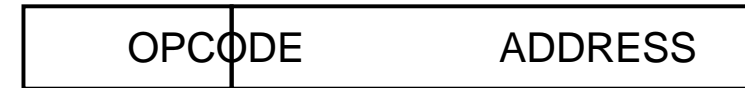
### JVM (Java Virtual Machine)

- Either seen as virtual processor or real HW (e.g., picoJava)
- Stack machine (operations take place on a stack)
- Heavily biased by Java
- Other virtual machine examples: CLR, P-Code

# Instruction formats



(a)



(b)



(c)



(d)

Example:  $C := A + B$

a) Zero-address instruction

- stack architectures: `push A; push B; ADD; pop C`

b) One-address instruction

- Accumulator implicitly operand and result: `load A; ADD B; st C`

c) Two-address instruction

- One operand becomes result: `ADD B,A; move A,C`

d) Three-address format

- `ADD C,A,B`

# Adressierungsarten

## Adressierungsarten

- die verschiedenen Möglichkeiten eines Prozessors die Adresse eines Operanden oder eines Sprungziels im Speicher zu berechnen

## Früher

- Adresse der Operanden und Sprungziele absolut im Befehl vorgegeben

## Nachteile

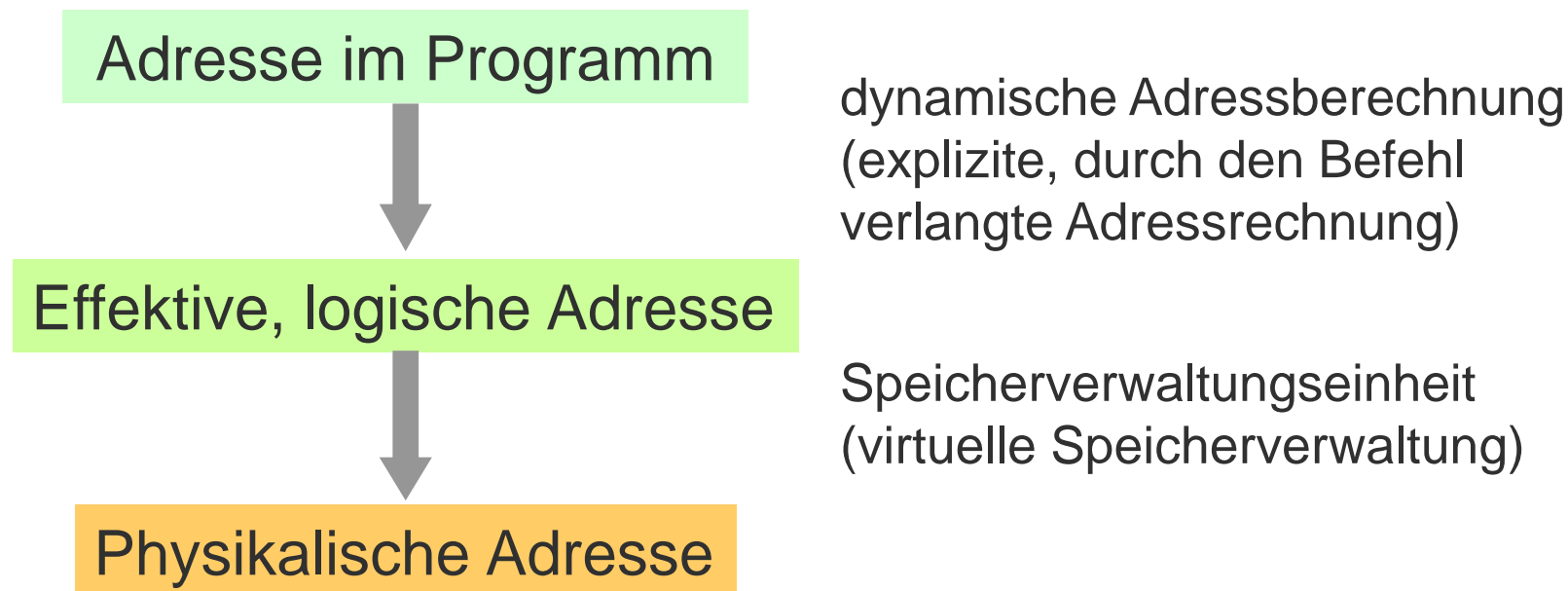
- absolute Adressen müssen bereits zur Programmierzeit festgelegt werden
  - ➔ Programme sind lageabhängig im Speicher
- Bei Tabellenzugriffen im Speicher muss die Adresse im Befehl geändert werden
  - ➔ keine Festwertspeicher als Programmspeicher möglich

# Adressierungsarten

## Abhilfe

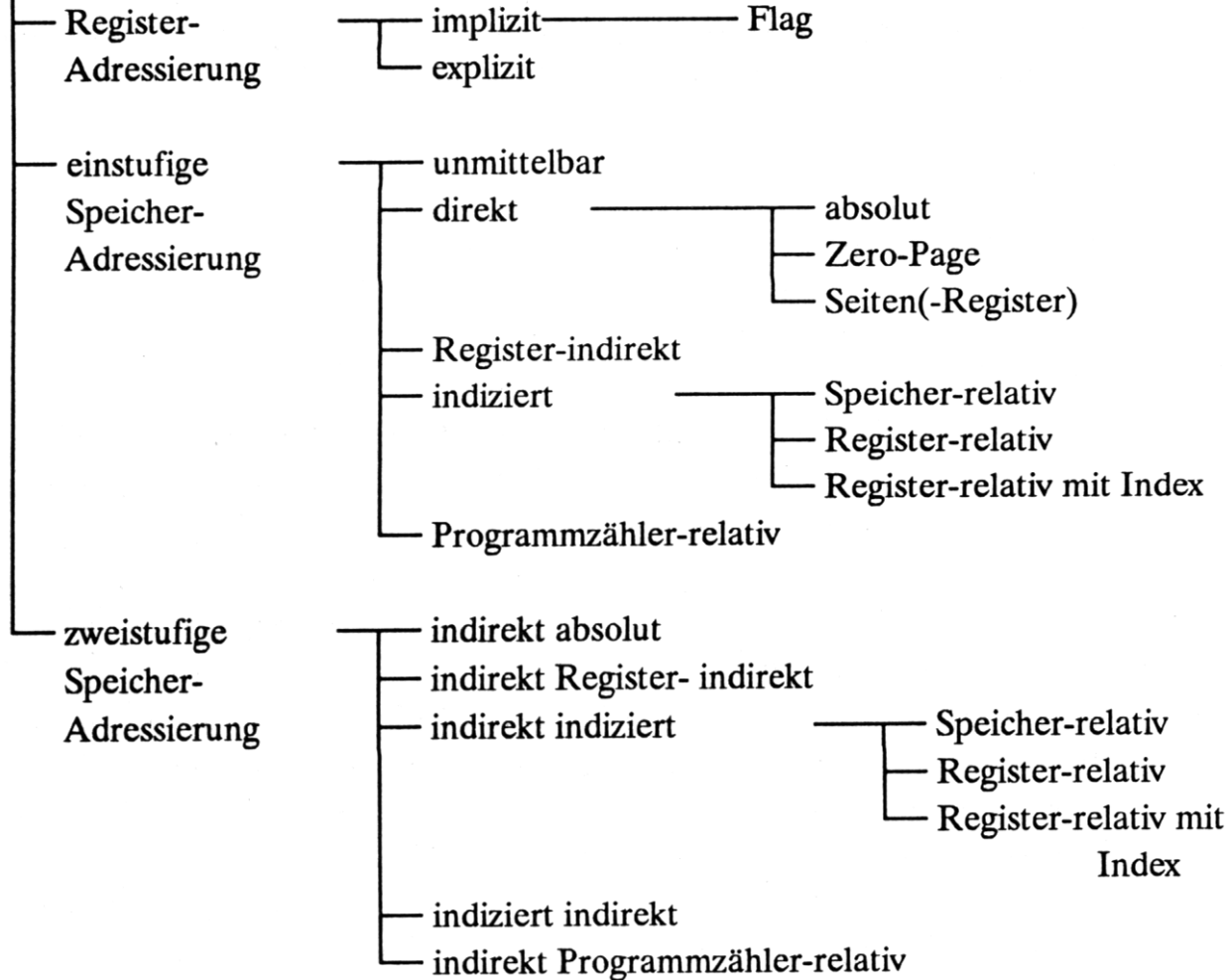
- Adresse wird zur Laufzeit berechnet (dynamische Adressberechnung)

## Ablauf der Adressberechnung

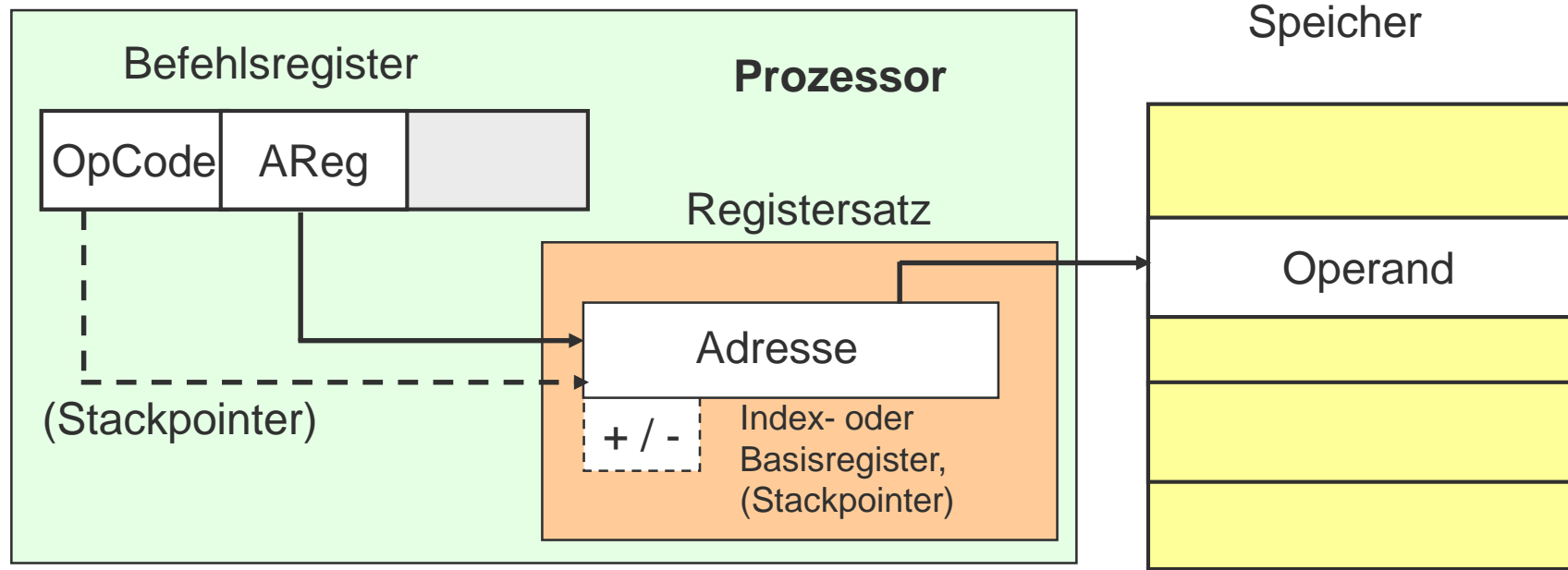


# Adressierungsarten - Überblick

## Adressierungsarten



# Register-indirekte Adressierung



## Beispiel:

**LD R1, (A0) (*load*)**

*(Lade das Register R1 mit dem Inhalt des durch das Adressregister A0 gegebenen Speicherwortes)*



# PROCEDURES, TRAPS, INTERRUPTS & CO.

# Procedures, Traps, Interrupts & Co.

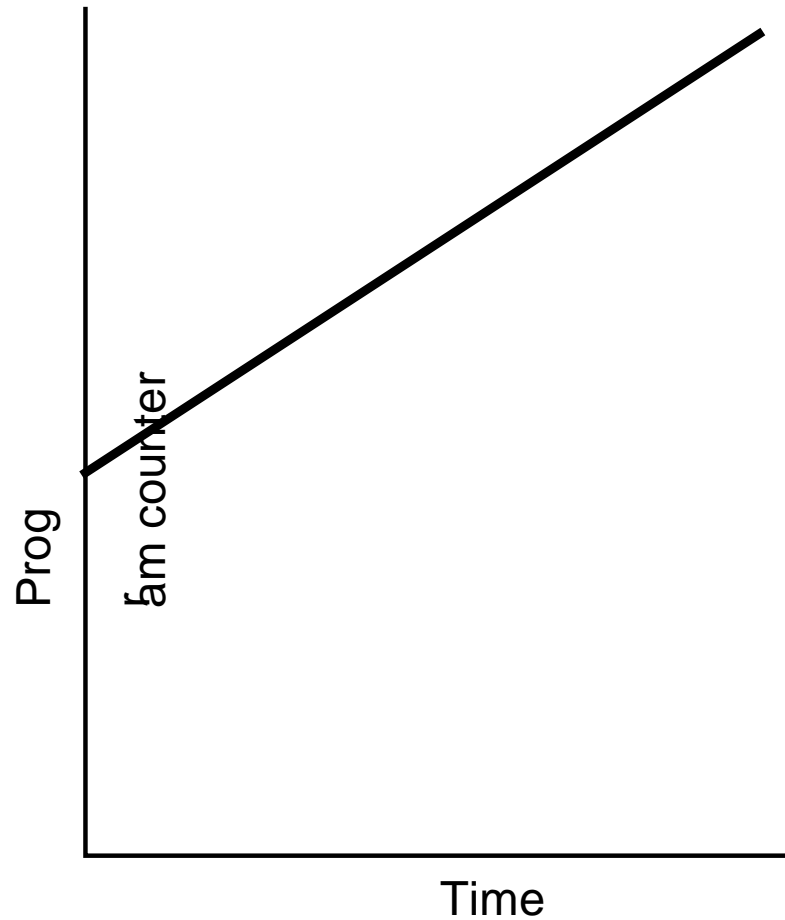
Many reasons for non-linear program execution

- Jumps, branches
- Procedure calls, subroutines, method invocation
- Multithreading, parallel processes, co-routines
- Hardware interrupts (processor external reasons)
- Traps, software interrupts (processor internal reasons)

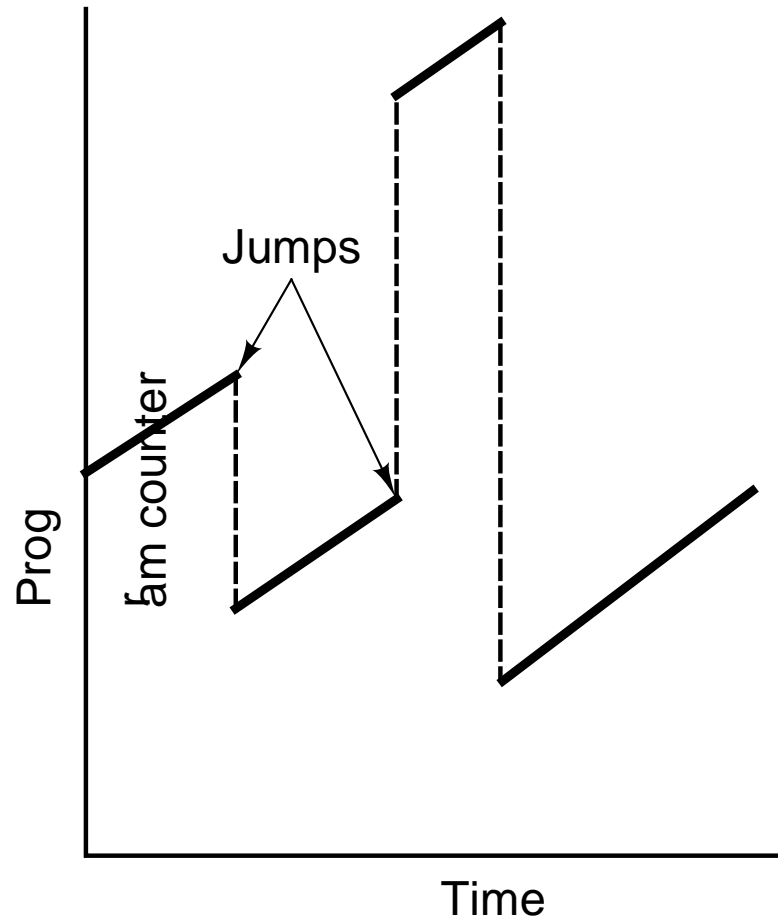
Non linear program execution is the normal case!

- And invalidates standard cache content ...
  - Trace caches can help (more later)

# Program execution

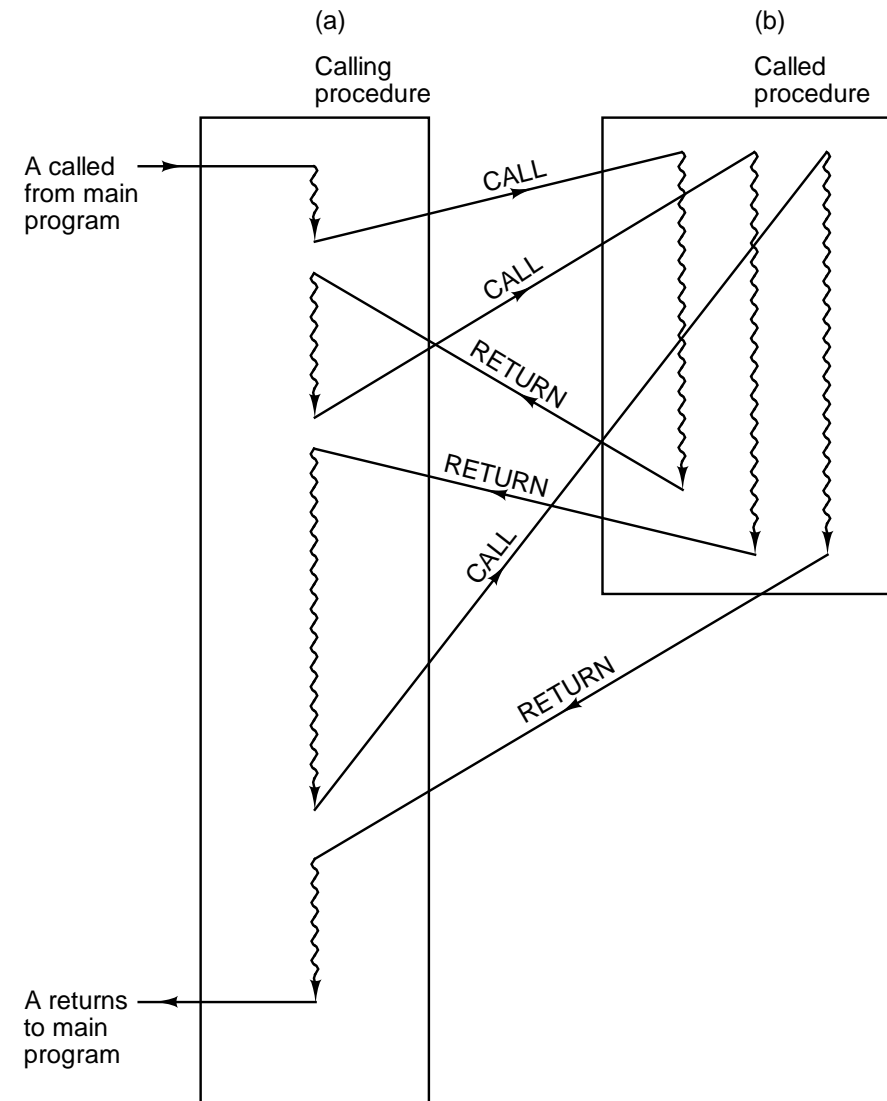


Linear, without branches

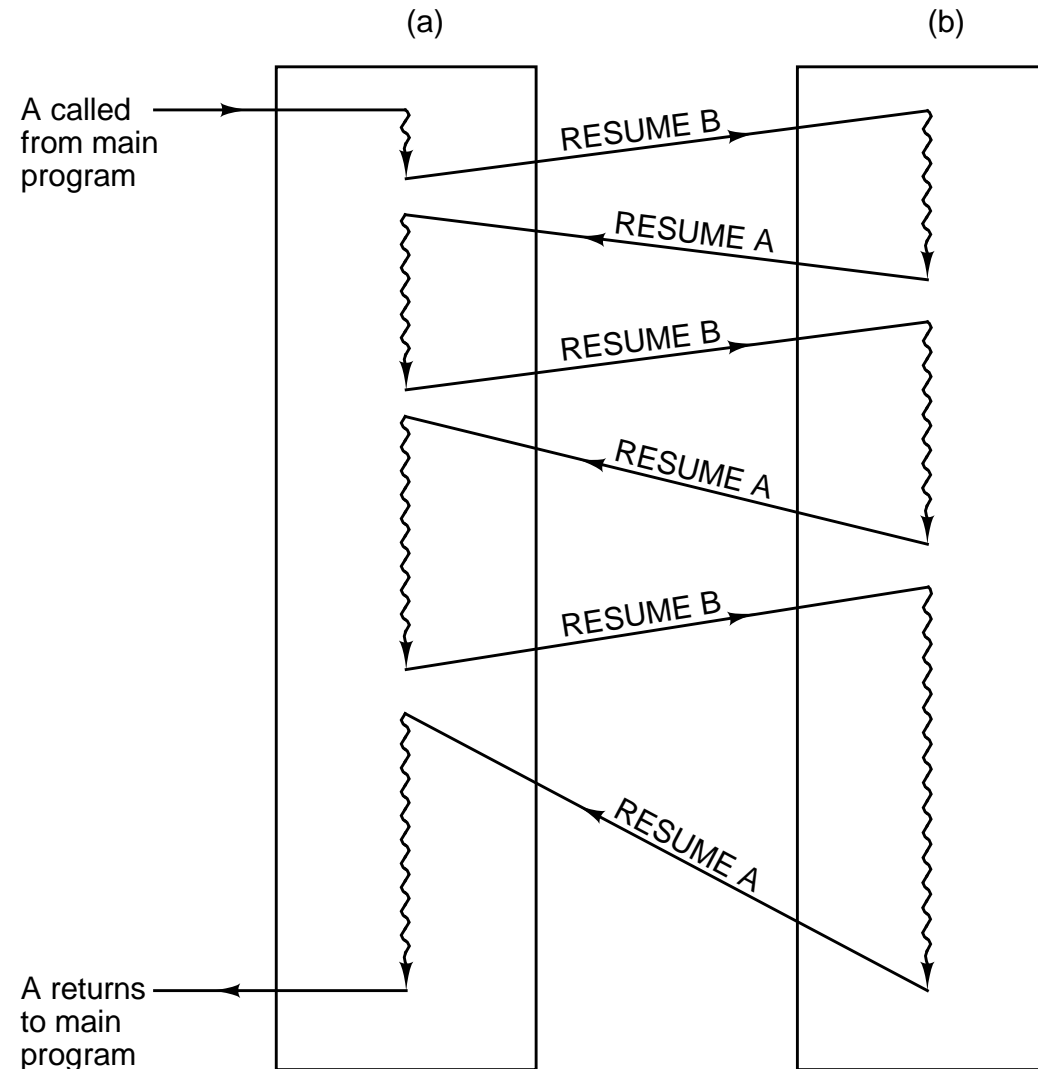


With branches

# Procedure call (subroutine, method, ...)



# Co-routine call (parallel process, multithreading,...)



# Behandlung von Ausnahme-Situationen

Während des Betriebs eines Mikroprozessorsystems können Ausnahme-Situationen (Exceptions) auftreten

## Ursachen

- Fehler im Betriebssystem bei der Ausführung des Anwenderprogramms oder Fehler der Hardware
- Wunsch externer Systemkomponenten, die Aufmerksamkeit des Prozessors zu erhalten

Eine solche Ausnahme-Situation erfordert eine vorübergehende Unterbrechung oder gar den Abbruch des laufenden Programms

# Behandlung von Ausnahme-Situationen

Die Ausnahme-Behandlung erfolgt durch eine Ausnahmeroutine (Interrupt Service Routine)

Die Auswahl und Aktivierung der Ausnahmeroutine  
wird durch eine Hardware-Komponente im Steuerwerk unterstützt: Unterbrechungs-System (Interrupt System)

Die Ausnahmeroutine hat Ähnlichkeit mit dem Aufbau eines Unterprogramms

Es gibt jedoch wesentliche Unterschiede

## Ausnahmeroutine/Unterprogramm

### Aktivierung

- *call subroutine* bei Unterprogramm
- *INT*-Befehl oder Hardware-Aktivierung bei Ausnahmebehandlung

### Beendigung

- *ret*-Befehl bei Unterprogrammen (*return from subroutine*)
- *reti*-Befehl bei Ausnahmebehandlung (*return from interrupt*)

Einsprungadresse ins Unterprogramm direkt im Programm, bei Ausnahmebehandlung über Interrupt-Tabelle

Unterprogrammaufruf sichert meist nur PC auf den Stack, Ausnahmebehandlungs-Aufruf meist auch das PSW



## Ausnahmeroutine/Unterprogramm

Unterprogrammaufrufe werden immer durchgeführt, die meisten Ausnahmebehandlungen werden nur aktiviert, falls das Interrupt Enable Bit im PSW gesetzt ist

Ursachen für Ausnahmebehandlungen

- Prozessorexterne Ursachen (asynchrone Ereignisse)
- Prozessorinterne Ursachen (synchrone Ereignisse)

# Prozessorexterne Ursachen

## RESET

- Rücksetzen des Mikrorechnersystems, z.B. ausgelöst durch Taste, Schwankungen der Betriebsspannung, Watch-Dog, ...

## HALT

- Anhalten des Prozessors, z.B. zur Vermeidung von Zugriffskonflikten auf dem Systembus bei DMA-Zugriffen

## ERROR

- Aufruf einer Fehlerbehandlungsroutine, z.B. bei Bus-Fehlern

## Interrupt

- Unterbrechungsanforderung von einem peripheren Gerät, z.B. um verfügbare Daten eines Eingabegerätes anzukündigen
- 2 Arten: maskierbar/nicht maskierbar (NMI)

# Prozessorinterne Ursachen

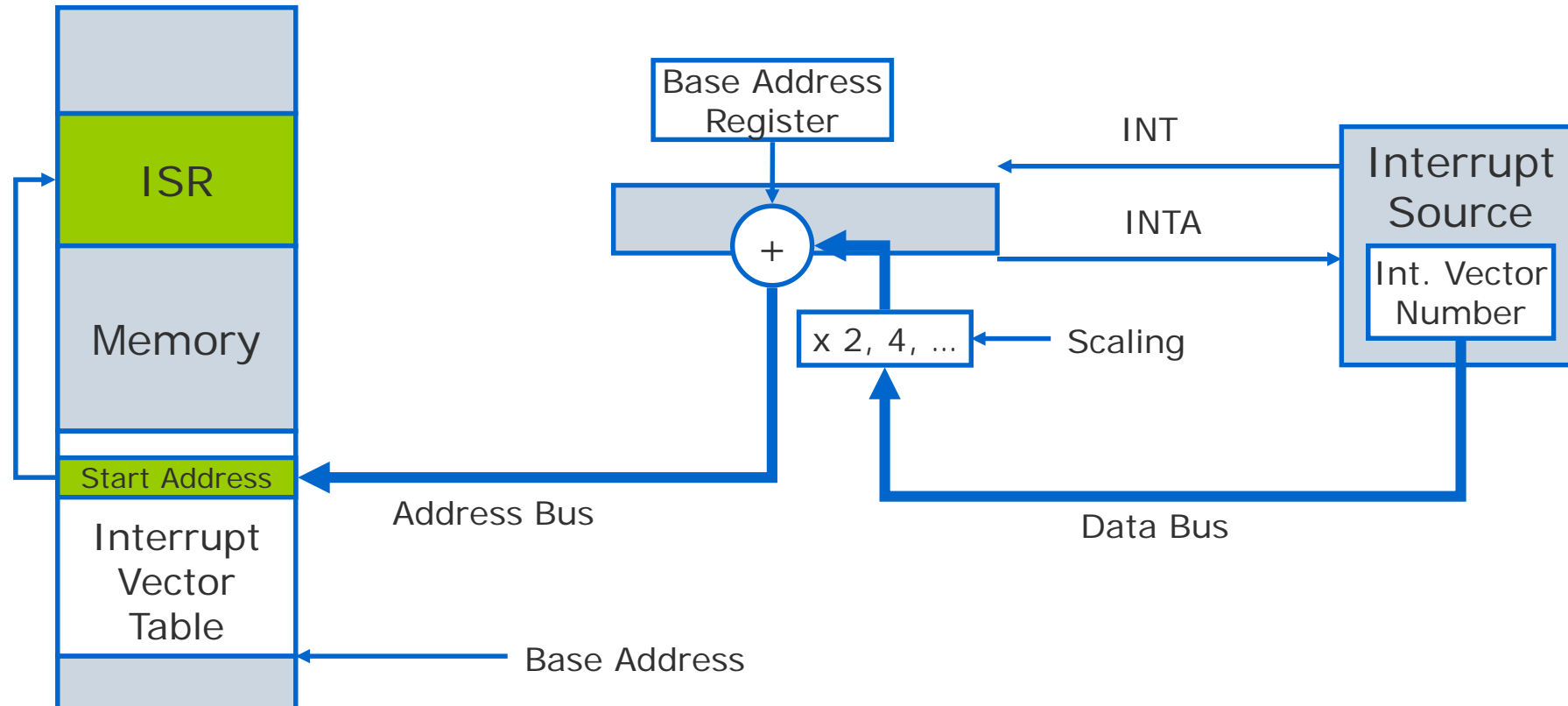
## Software Interrupts

- durch INT-Befehl im Programm ausgelöste Unterbrechungen

## Traps

- Ausnahmesituationen durch prozessorinterne Ereignisse, z.B. Overflow, Divide by Zero, Stack Overflow

## Calculation of the start address of an Interrupt Service Routine (ISR)



## Ablauf einer ISR I

1. Interrupt-Aktivierung
2. Beenden des gerade in Ausführung befindlichen Befehls
3. Feststellen, ob Software-Interrupt oder interner/externer Hardware-Interrupt vorliegt
4. Feststellen, ob das Interrupt Enable Bit gesetzt ist  
➔ Interrupt zugelassen
5. Falls HW-Interrupt: Quelle des Interrupts finden, INTA-Leitung (Interrupt Acknowledge) aktivieren
6. PSW und PC auf den Stack sichern
7. Interrupt Enable Bit rücksetzen (und damit weitere Unterbrechungen verhindern)

## Ablauf einer ISR II

8. Startadresse der Interrupt-Service-Routine ermitteln und in den PC laden
9. Interrupt Service Routine ausführen:
  - Meist werden zuerst die benutzten Register auf den Stack gesichert
  - Interrupt Enable Bit wieder setzen
  - Eigentliche Routine
  - Am Ende der Interrupt Service Routine: IRET-Befehl
10. PSW und PC werden wiederhergestellt und mit dem unterbrochenen Programm wird fortgefahren.

**Achtung: Ist eine ISR zu groß blockiert der Rechner!**

## Interrupt-Vektortabelle

Oft in Festwertspeicher an spezieller Speicheradresse (z.B. ab Adresse 0000:0000 beim 80X86)

Enthält die Startadressen der Behandlungsroutinen

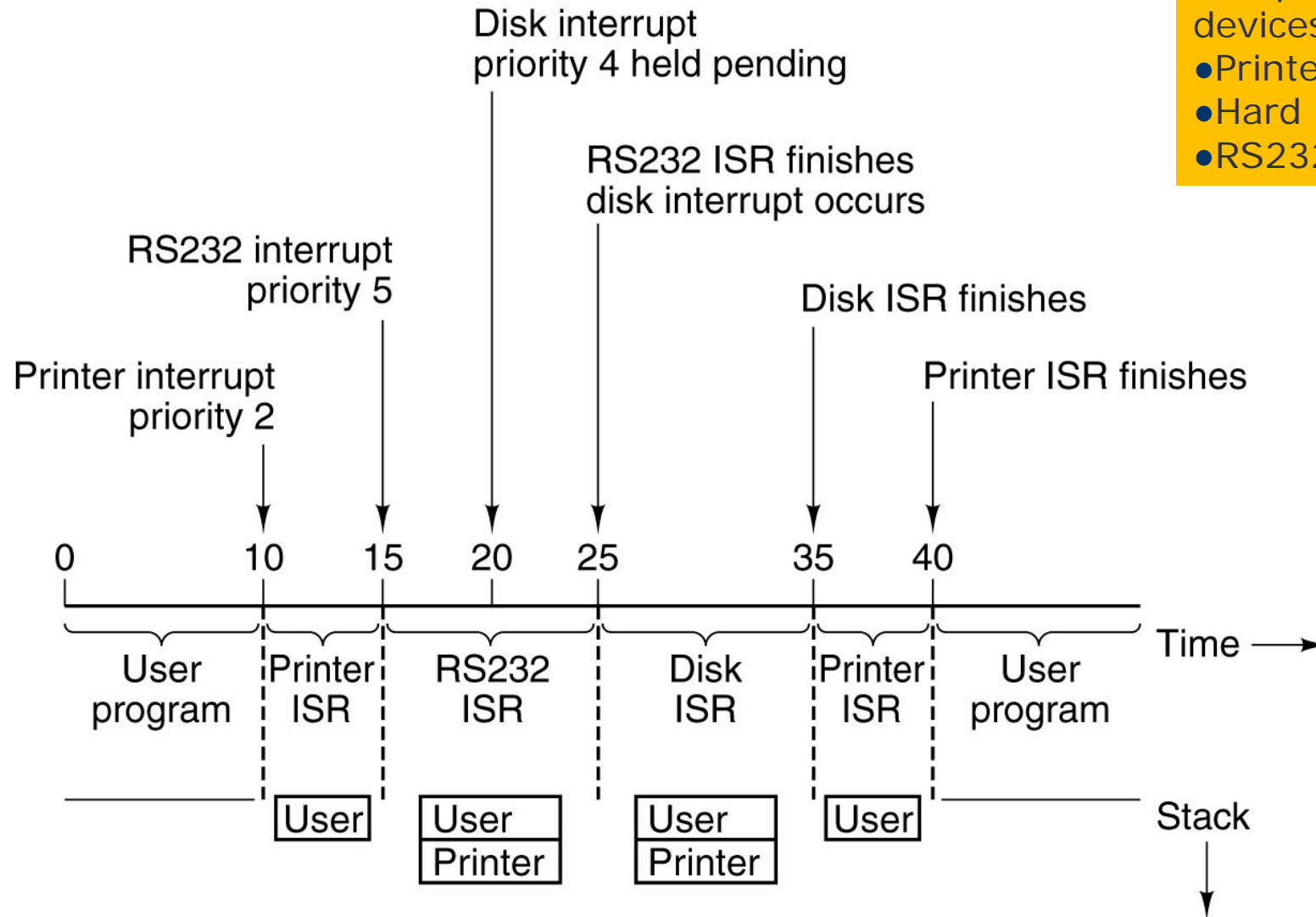
Interrupt-Quelle liefert bei Interrupt eine Interrupt-Vektor-Nummer (IVN), welche den Eintrag in der Interruptvektor-Tabelle charakterisiert

# Beispiel einer Vektortabelle

Index	Ausnahmesituation	
0	<i>divide by 0</i>	Division durch 0
1	<i>overflow</i>	Zahlenbereichsüberschreitung
2	<i>array bound check</i>	Indexbereichsüberschreitung
3	<i>invalid opcode</i>	illegaler Befehlscode
4	<i>SVC (supervisor call)</i>	Betriebssystem-Aufruf
5	<i>privilege violation</i>	unerlaubter Aufruf einer privilegierten Operation
6	<i>trace</i>	Einzelschritt-Modus
7	....	
-	....	(weitere Traps)
i	....	
i+1	RESET	Rücksetzen
i+2	BERR	Busfehler
i+3	NMI	nicht maskierbarer Interrupt
-	....	
k	....	(weitere Ausnahmesituationen)
k+1	<i>error</i>	Coprozessor-Fehler
-	....	(weitere Coprozessor-Meldungen, z.T. über spezielle Statusleitungen)
l	....	
l+1	<i>page fault</i>	Seitenfehler (s. Kapitel 4)
-	....	(weitere Fehler der Speicherverwaltung)
m	....	
m+1	....	(reserviert für zukünftige Erweiterungen und für Testzwecke des Herstellers)
-	....	
n	....	
n+1	<i>user vectors</i>	(durch Systementwickler frei zuzuordnende, maskierbare Interrupts)
-	....	
255	....	



## Time sequence of multiple interrupts



Computer with 3 I/O devices

- Printer, prio 2
- Hard disc, prio 4
- RS232, prio 5

## Behandlung mehrerer Interrupt-Quellen

Interrupt-Quellen werden durch Interrupt Controller zyklisch abgefragt (Interrupt-Flag im Statusregister des Controllers).

Komponenten mit gesetztem Interrupt-Flag

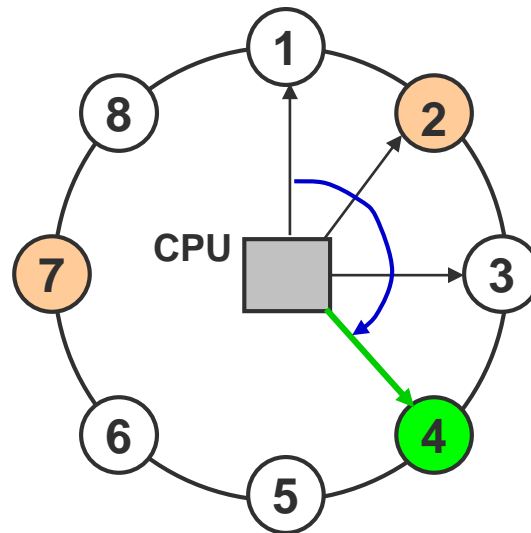
- ➡ Abfrage abbrechen und die entsprechende Interruptroutine abrufen.

Nach (und auch während) der Abarbeitung eines Interrupts können weitere Anforderungen auftreten.

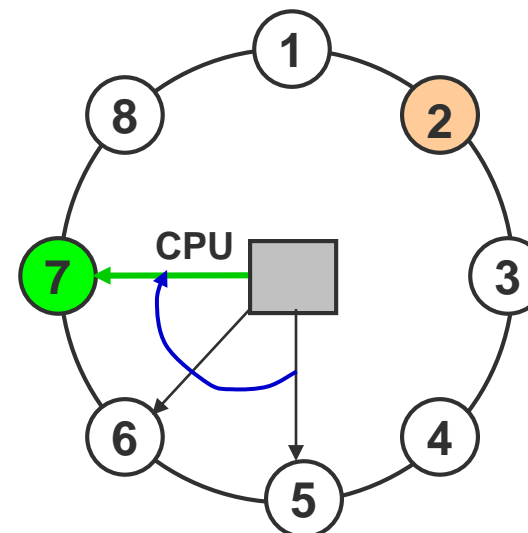
- ➡ Zwei Alternativen zur Behandlung


## Polling: 1. Variante

Zyklische Abfrage wird bei der Komponente fortgesetzt, die in der vorgegebenen Reihenfolge der zuletzt bedienten Komponente folgt → Alle Komponenten haben die gleiche Chance, bedient zu werden („faire“ Prozessor-Zuteilung)



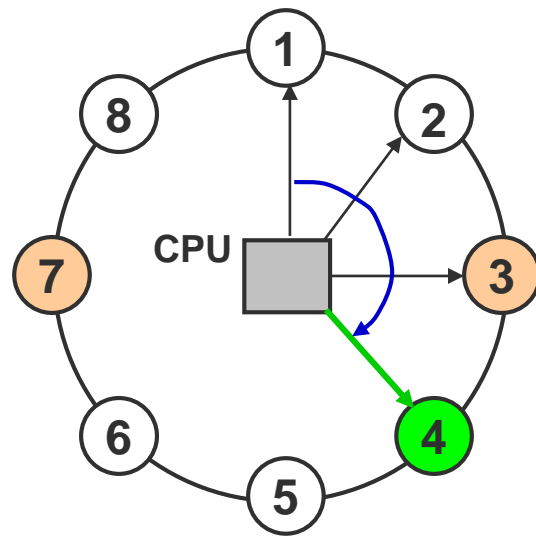
 Aktuell bearbeitete Interrupt-Anforderung



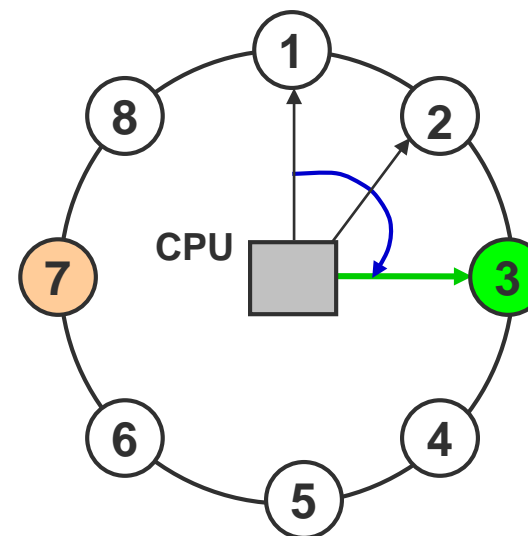
 Interrupt-Anforderungen während der Bedienung von Komponente 4


## Polling: 2. Variante

Zyklische Abfrage beginnt immer mit der eindeutig festgelegten ersten Komponente → Verschiedenen Komponenten werden verschiedene Prioritäten zugeordnet. Komponenten mit hoher Priorität werden schneller bedient.



 Aktuell bearbeitete Interrupt-Anforderung



 Interrupt-Anforderungen während der Bedienung von Komponente 4

# Polling

## Prioritäten beim 80286:

Priorität		Ausnahmesituation
0	RESET	Rücksetzen, Initialisieren
1	TRAPS INT	Ausnahme bei Befehlsausführung Software-Interrupt
2	TRACE	Einzel-Schritt-Ausführung
3	NMI	nicht maskierbare Interrupts
4	...	Coprozessor-Fehler
5	IRQ	maskierbare Interrupts

## Nachteil des Polling-Verfahrens:

Die Priorisierung und Identifizierung von Interrupts durch die zyklische Abfrage (Software) ist sehr zeitaufwendig.

## Daisy-Chain-Verfahren

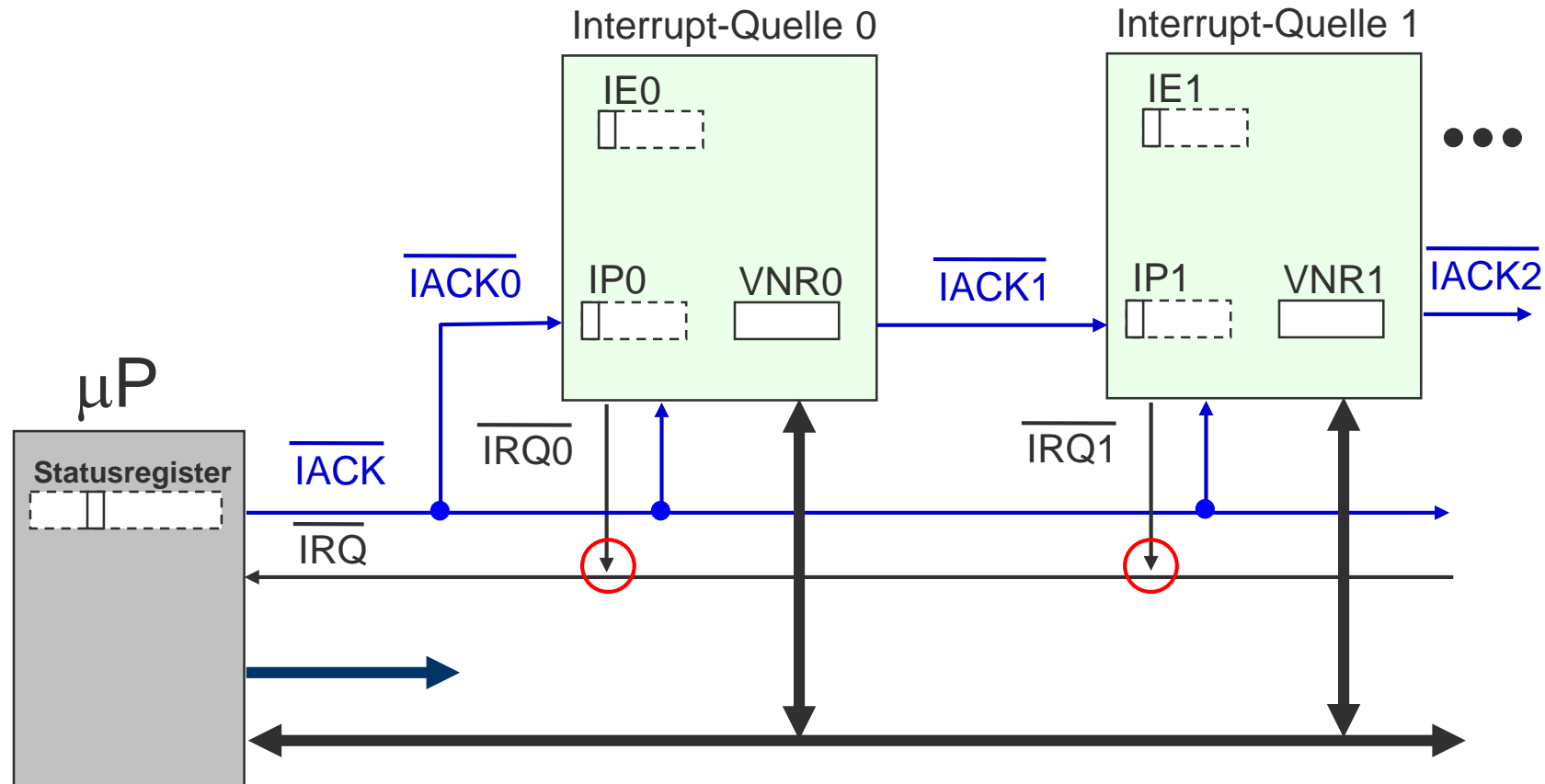
Priorisierung und Identifizierung von Interrupts wird durch eine **Zusatzhardware** durchgeführt.

Zusammenschalten von Interrupt-Quellen zu einer Prioritätskette (Interrupt-Daisy-Chain).

Jede Interrupt-Quelle hat eine Priorisierungsschaltung (dezentrale Priorisierung), die mit der des Vorgängers und Nachfolgers mit Signalleitungen verbunden ist.

**Erste Quelle** der Kette hat die **höchste Priorität**. Die **Priorität** der anderen Quellen **nimmt** mit jedem Glied in der Kette um **eine Stufe ab**.

# Daisy-Chain-Verfahren



## Daisy-Chain-Verfahren

Anforderungen der einzelnen Quellen werden durch ODER zusammengeschaltet (wired or) und über den Interrupt-Eingang IRQ (Kreissymbol) zum Prozessor geführt.

Prozessor leitet (bei gesetzten Interrupt-Enable-Flag) einen Interrupt-Zyklus durch das Aktivieren von IACK ein.

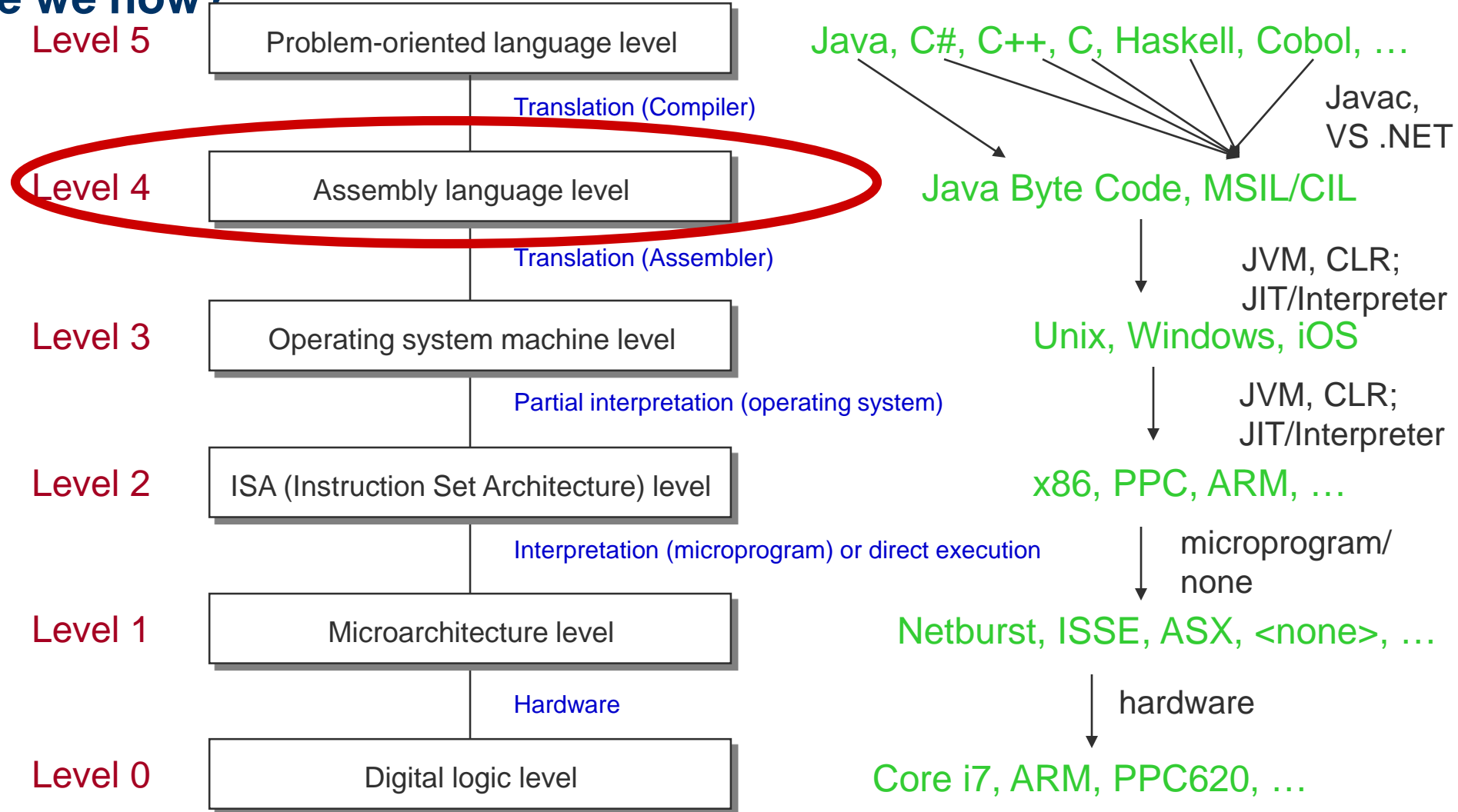
IACK wirkt direkt auf die Interrupt-Quellen und auf den IACK-Eingang der ersten Quelle in der Kette.

Eine Quelle, die während  $IACK = 1$  eine Anforderung anmeldet, verhindert die Weitergabe des aktiven Pegels von IACK an die nachfolgenden Kettenglieder.



# ASSEMBLER

## Where are we now?



# Compiler vs. Assembler

## Assembler

- Source: symbolic representation of a machine language (assembly language)
- Destination: numerical representation of the machine language (instructions from ISA)
- Examples: inline assembler in Visual Studio, MASM, ilasm, asm (gcc, Linux), MMIXal, nasm, ...

## Compiler

- Source: high-level language (depends on the definition of „high“ ...), e.g., C, Java, C#, Cobol, Modula, C++, ...
- Destination: assembler language or (built-in assembler) numerical representation of the machine language (instructions from ISA)
- Examples: C#-Compiler in Visual Studio, gcc, cc, javac, ...

## Assembler language

- Pure assembler language: 1:1 mapping onto ISA instructions
- But additionally: symbolic names, addresses, labels

## Reasons for an assembler level

### Full access to HW features

- (almost) all registers are exposed to the assembler language, all flags can be read or set, many „hidden“ features can be used
  - E.g., try accessing the Pentium performance counters from within Java

### Performance

- Optimized code for special purposes
  - Real-time: exact number of CPU cycles can be counted, guaranteed access times to registers, deterministic response times of sub-routines (again: try Java and real-time – and see what a garbage collector does...)
  - Low memory footprint: no useless overhead, optimized loops, etc.

### But much harder to program in assembler ....

- Thus typically combined with, e.g., C – only performance critical parts of a program will be tuned via assembler

# Examples for assembler: $N = I + J$

## Pentium

### FORMULA:

```
MOV EAX,I      ; register EAX = I
ADD EAX,J      ; register EAX = I + J
MOV N,EAX      ; N = I + J
```

```
I   DW   3      ; reserve 4 byte initialized to 3
J   DW   4      ; reserve 4 byte initialized to 4
N   DW   0      ; reserve 4 byte initialized to 0
```

## SPARC

### FORMULA:

```
SETHI %HI(I),%R1      ! R1 = high-order bits of the address of I
LD [%R1+%LO(I)],%R1    ! R1 = I
SETHI %HI(J),%R2      ! R2 = high-order bits of the address of J
LD [%R2+%LO(J)],%R2    ! R2 = J
NOP                    ! wait for J to arrive from memory
ADD %R1,%R2,%R2        ! R2 = R1 + R2
SETHI %HI(N),%R1      ! R1 = high-order bits of the address of N
ST %R2,[%R1+%LO(N)]    ! N = I + J
```

```
I:  .WORD 3      ! reserve 4 byte initialized to 3
J:  .WORD 4      ! reserve 4 byte initialized to 4
N:  .WORD 0      ! reserve 4 byte initialized to 0
```

# Pseudoinstructions

Pseudoinstructions or assembler directives

- Help a lot for assembler programming
- Depend on designer of the assembler, not the ISA

Examples (MASM for Pentium)

- |           |  |              |
|-----------|--|--------------|
| - DB      | allocate storage for one or more (initialized) | bytes        |
| - DW      | allocate storage for one or more (initialized) | 32 bit words |
| - PROC    | start a procedure                              |              |
| - MACRO   | start a macro definition                       |              |
| - INCLUDE | fetch and include another file                 |              |
| - IF      | start conditional assembly based on a given    | expression   |
| - PUBLIC  | export a name defined in the module            |              |

# Macros vs. procedures

Example macro: swap P, Q

```

SWAP      MACRO
            MOV EAX,P
            MOV EBX,Q
            MOV Q,EAX
            MOV P,EBX
        ENDM
    
```

## Differences

	Macro	Procedure
When is the call made?	During assembly	During execution
Is the body inserted into the object program every place the call is made?	Yes	No
Is a call instruction inserted into the object program and later executed?	No	Yes
Must a return instruction be used after the call is done?	No	Yes
How many copies of the body appear in the object program?	One per macro call	1

Macros are „textually“ inserted into the assembler program each time a call is made, formal parameters are converted into actual parameters (i.e., the above macro can be used for SWAP A,B as well as SWAP X,Y)

# The assembler process

Step-by-step translation does not work

- Forward references: symbols used before being defined ...

Solution: two-pass translator or single-pass plus conversion into intermediate format

- Pass: reading the source

First step

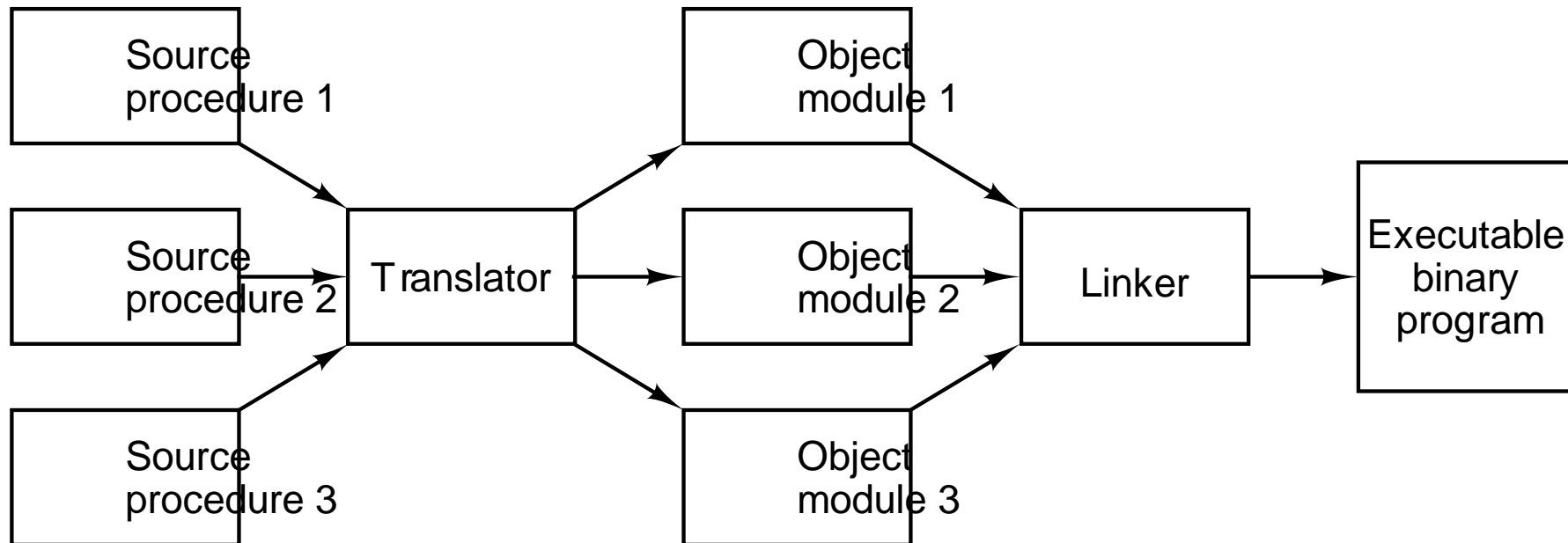
- Check syntax
- Create a symbol table, opcode table, literal table
- Check instruction length (opcode, operands, ...)

Second step

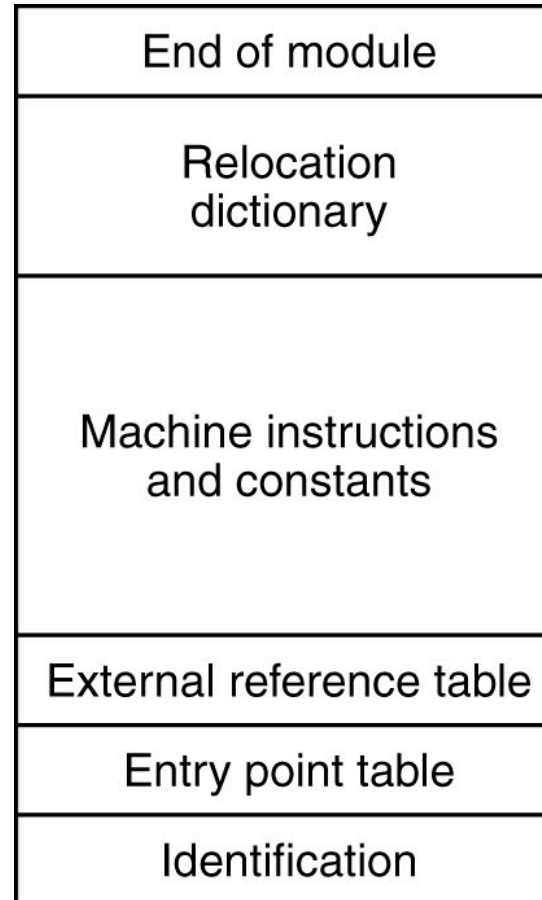
- Generate object code (\*.o, \*.obj, ...)
- Generate information for linker



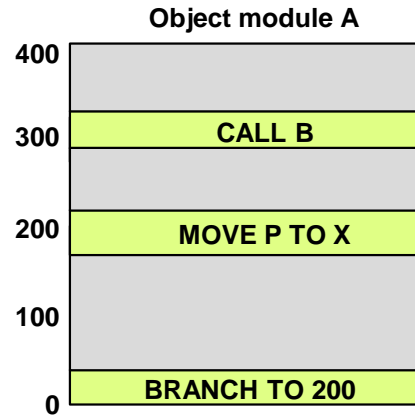
## Generation of an executable binary program



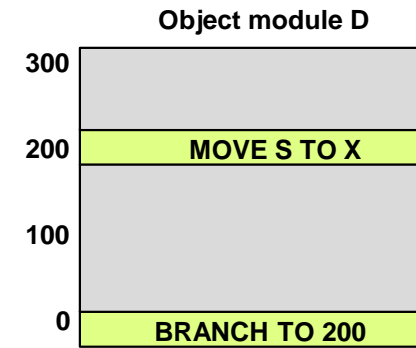
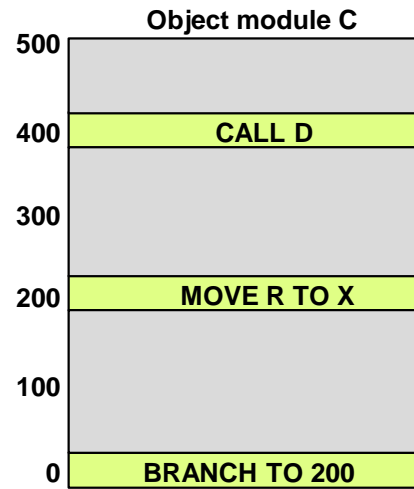
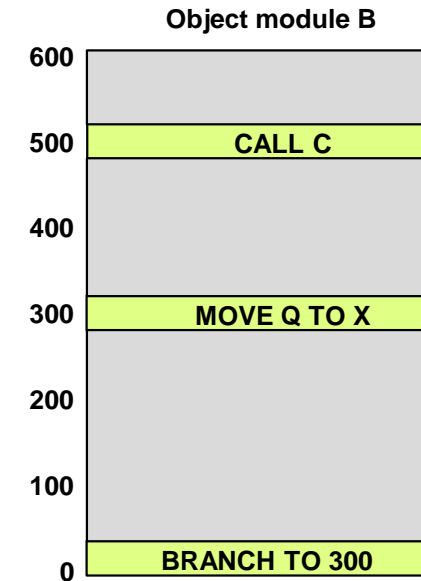
# Structure of an Object Module



# Example object modules

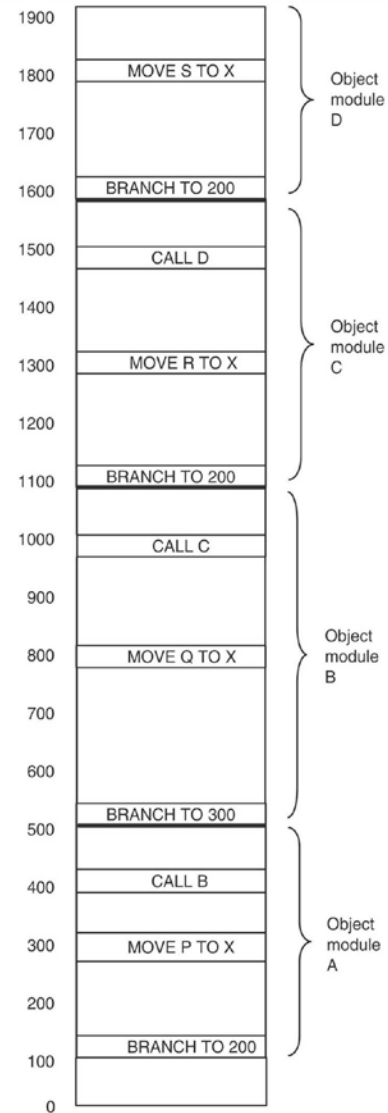


Each module has its own address space starting at 0

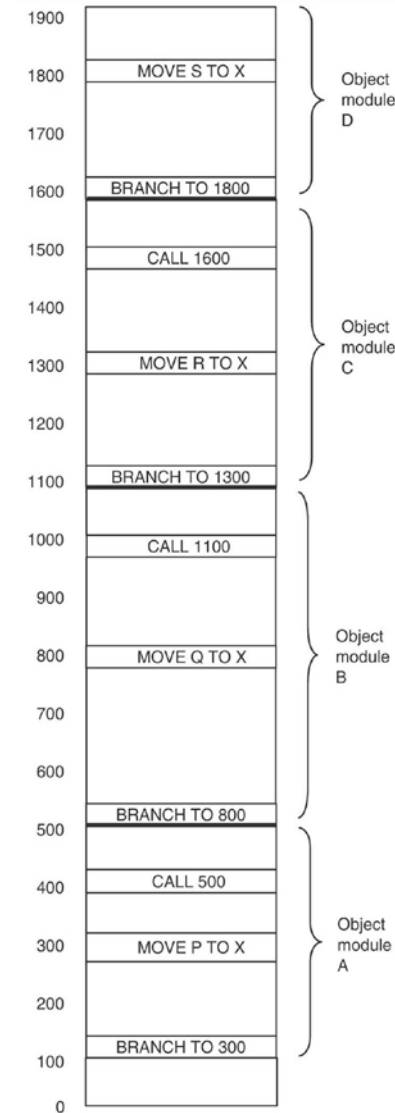


# Objects before/after linking and relocation

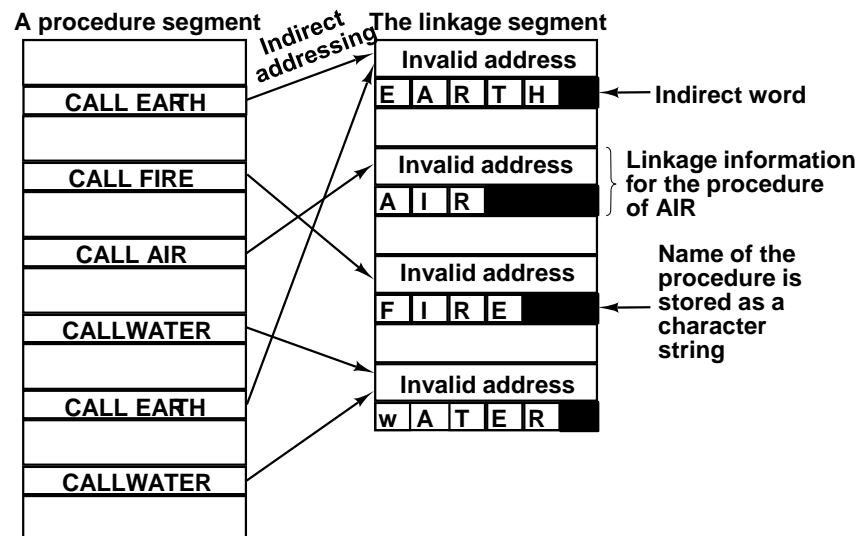
Before  
relocation and  
linking



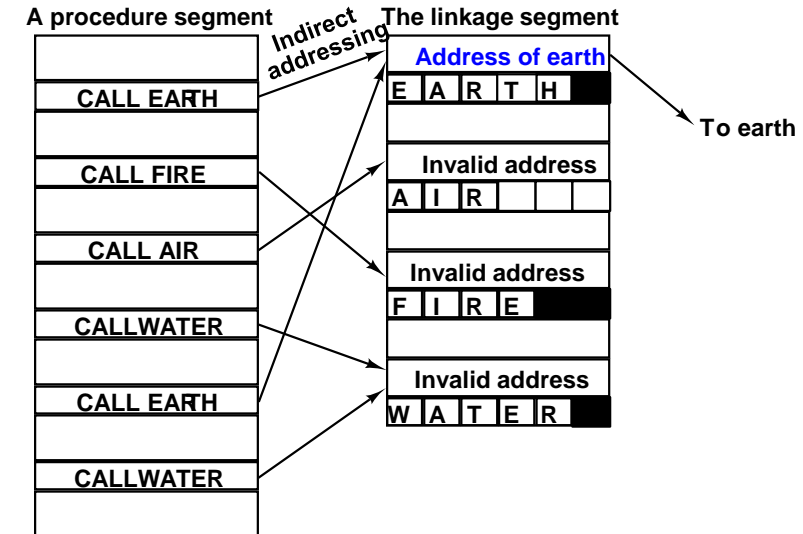
After  
relocation and  
linking



# Dynamic linking



Before EARTH is called

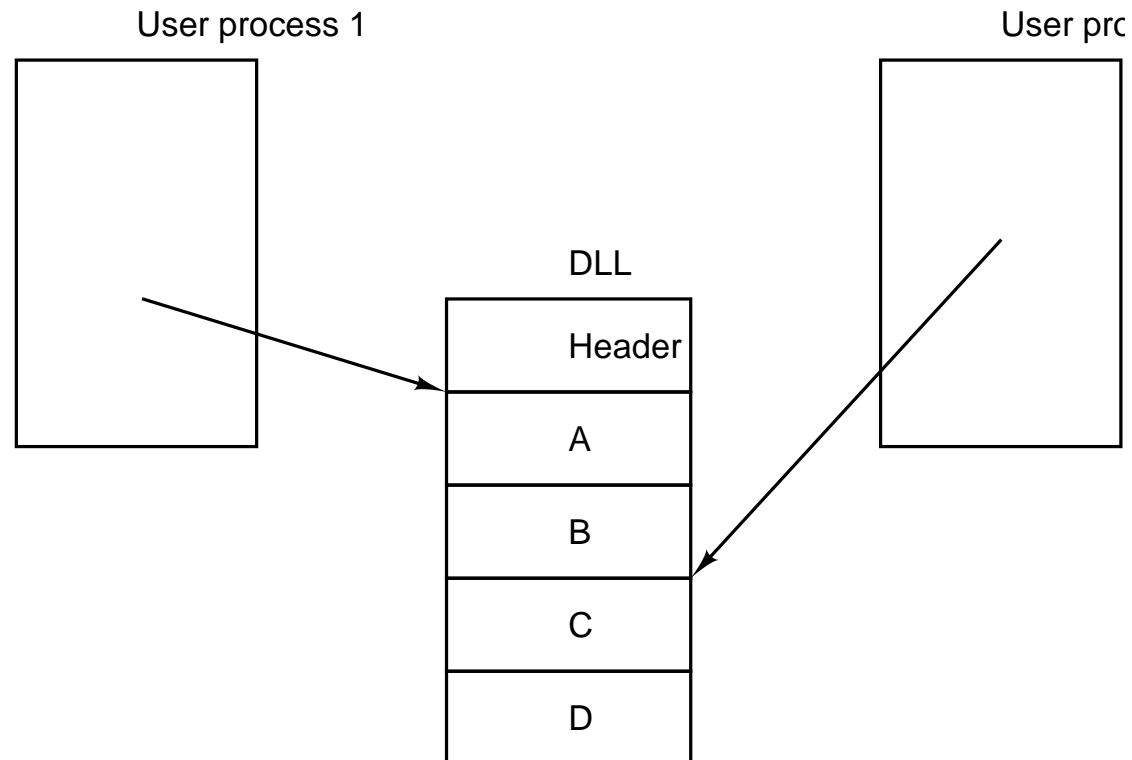


After EARTH has been called and linked

# Shared libraries

DLL (Dynamic Link Library, Windows), shared library (Unix)

- Save a lot of memory as they appear only once
- Many processes „share“ the same code (instructions)



# Summary

Soft-/Hardware boundary

Complex Instruction Set Computer (CISC)

Reduced Instruction Set Computer (RISC)

Examples of ISA

- Pentium
- SPARC
- JVM

Instructions formats

Addressing formats

Types of instructions

Procedures, Traps, Interrupts & Co.

Assembler