Atmel hat für eine ganze µC-Familie eindeutig vordefinierte Symbolische Namen für alle E/A-Register und -Pins in einer controllerspezifischen INCLUDE-Datei abgelegt. Diese kann und sollte zur Erhöhung der Lesbarkeit verwendet werden.

```
.include    "m16def.inc."
              :
              :
            ; das vorige Beispiel lautet dann:
            LDI    R16, (1 << PA7) | (1 << PA6) | (1 << PA1) | (1 << PA0)
            LDI    R17, (1 << DDA7) | (1 << DDA6) | (1 << DDA5) | (1 << DDA4)
            OUT    PORTA, R16
            OUT    DDRA, R17
              :
              :
            IN     R18, PINA
```

Die Schreibweise (1 << PA7) bedeutet:

> Eine 1 auf das Port-Bit PA7, d.h. um 7 Bits (PA7), nach links schieben, also 1000 0000.

Durch "|" (ODER) können dann wie oben mehrere dieser Maskenbits zur endgültigen Maske kombiniert werden.

Der erhöhte Schreibaufwand wird durch die deutlich verbesserte Les- und Wartbarkeit mehr als aufgewogen.

Einzelne Bits der E/A-Register $00 bis $01F (0-31) können auch über spezielle Bitmanipulationsbefehle gesetzt, gelöscht (SBI, CBI, SBIC, SBIS) und getestet werden (BRxx), z.B. um im laufenden Betrieb einzelne Ausgabebits oder Konfigurationsbits ohne Seiteneffekte zu ändern bzw. abzufragen.

Beispiel: BCD-nach-ASCII-Konvertierung

Einlesen einer BCD-Zahl von Port A (unteres Nibble). Ausgabe als ASCII-Zahl ohne Parität in einer Endlosschleife auf Port D.

```
.include "m16def.inc"          ;get label definitions
.def    TEMP = R30             ;R30 working register

.CSEG                          ;CODE SEGMENT
        .ORG    $0000          ;Starting address (after
                               ;RESET)
INIT:   SER     TEMP           ;Configure
        OUT     DDRD, TEMP     ;Port D as output
        CLR     TEMP           ;Configure
        OUT     PORTA, TEMP    ;Port A as
        OUT     DDRA, TEMP     ;Tri-State input

LOOP:   IN      TEMP, PINA     ;Input byte from Port A
        ANDI    TEMP, $0F      ;Mask out low nibble
        ORI     TEMP, $30      ;Set ASCII to high nibble
        OUT     PORTD, TEMP    ;Output byte to Port D
        RJMP    LOOP           ;Endless polling loop
```

Anmerkung: Hier geschieht ausnahmsweise keine Stackpointer-Initialisierung, weil der Stack nicht gebraucht wird.

ASCII-Tabelle der druckbaren Zeichen

(American Standard Code for Information Interchange)

ASCII hex	ASCII dez	Zeichen	ASCII hex	ASCII dez	Zeichen	ASCII hex	ASCII dez	Zeichen
20	32	SPace	40	64	@	60	96	`
21	33	!	41	65	A	61	97	a
22	34	"	42	66	B	62	98	b
23	35	#	43	67	C	63	99	c
24	36	$	44	68	D	64	100	d
25	37	%	45	69	E	65	101	e
26	38	&	46	70	F	66	102	f
27	39	'	47	71	G	67	103	g
28	40	(	48	72	H	68	104	h
29	41	)	49	73	I	69	105	i
2A	42	*	4A	74	J	6A	106	j
2B	43	+	4B	75	K	6B	107	k
2C	44	,	4C	76	L	6C	108	l
2D	45	-	4D	77	M	6D	109	m
2E	46	.	4E	78	N	6E	110	n
2F	47	/	4F	79	O	6F	111	o
30	48	0	50	80	P	70	112	p
31	49	1	51	81	Q	71	113	q
32	50	2	52	82	R	72	114	r
33	51	3	53	83	S	73	115	s
34	52	4	54	84	T	74	116	t
35	53	5	55	85	U	75	117	u
36	54	6	56	86	V	76	118	v
37	55	7	57	87	W	77	119	w
38	56	8	58	88	X	78	120	x
39	57	9	59	89	Y	79	121	y
3A	58	:	5A	90	Z	7A	122	z
3B	59	;	5B	91	[	7B	123	{
3C	60	<	5C	92	\	7C	124	|
3D	61	=	5D	93	]	7D	125	}
3E	62	>	5E	94	^	7E	126	~
3F	63	?	5F	95	_	7F	127	DELete

Serielle Schnittstellen

USART: Synchrone und asynchrone serielle Vollduplex-Schnittstelle (vgl. V24/RS232), z. B. für Anschluss an einen PC etc.
(USART = universal synchronous/asynchronous receiver/transmitter)

TWI: Serieller Multi-Master-Bus (Two Wire Interface; Takt und Daten) mit bis zu 128 Slaves und 400 kBit/s

SPI: Synchrone serielle Vollduplex-Schnittstelle, z. B. zur schnelleren Datenkommunikation

asynchron: Bei der asynchronen Übertragung wird auf eine Taktleitung verzichtet und zur Synchronisation von Sender und Empfänger auf Byte-Ebene ein einfaches Protokoll mit Start- und Stopp-Bits (und implizitem Timing) genutzt.

synchron: Eine eigene Taktleitung erlaubt eine synchronisierte Übertragung mit hoher Datenrate bis zu mehreren MBit/s.

Die Taktgenerierung muss bei der Konfiguration des µC explizit konfiguriert werden.

Die Parallel-Seriell- bzw. Seriell-Parallel-Wandlung und die Anpassung an das verwendete Protokoll erfolgen in Hardware, also unabhängig von der Programmausführung.

USART, SPI, TWI verwenden Pins von Port B, Port C, Port D, die dann natürlich nicht mehr für die parallele Ein-/Ausgabe zur Verfügung stehen.

Timersystem: Zwei 8-Bit- und ein 16-Bit-Timer/Counter

Ein Timer ist ein Hardwarezähler, der per Software konfiguriert wird und dann programmunabhängig Impulse erzeugt oder zählt.
Durch die Auswahl von Taktquelle und Vergleicher kann das recht flexibel eingesetzt werden, z. B. für

- Zeitmessungen,
- Ereigniszählung,
- Frequenzgenerierung oder –messung,
- quasi-analoge Ausgabe per Pulsweitenmodulation,
- Timing-Vorgaben an die Software für periodische Aktivierung von Tasks oder für Verzögerungen.

Kern: Zähler (8 Bit bzw. 16 Bit), die vom Systemtakt (ggf. über Vorteiler) oder extern getaktet werden.

Typische Modi:

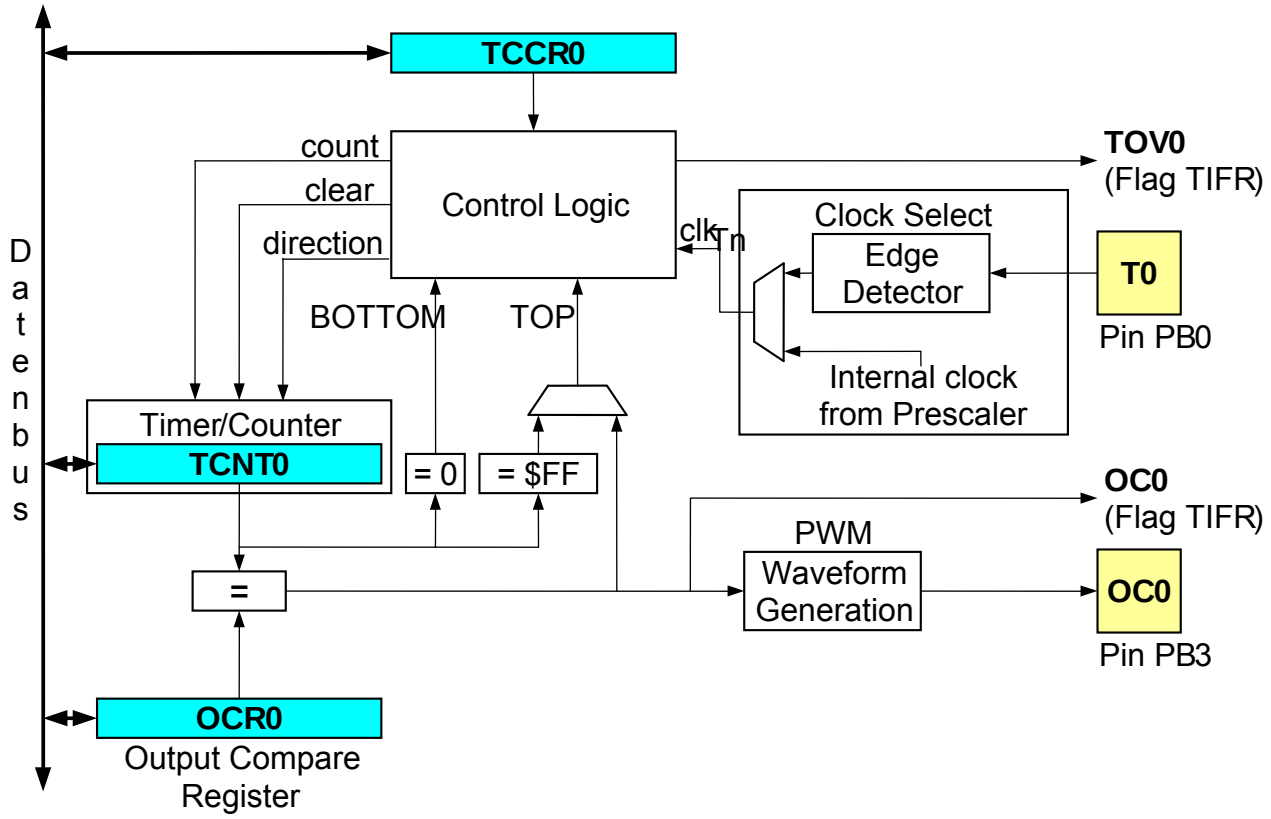
input-capture: Bei einer Eingangssignaländerung an dafür aktivierten Pins wird der aktuelle Zählerstand in spezielle Timerregister gespeichert (z. B. nutzbar für Impulsbreiten- oder Periodendauermessung).

output-compare: Hat der Zähler den in einem speziellen Output-Compare-Register gespeicherten Stand erreicht, wird ein dafür konfigurierter Pin gesetzt (z. B. nutzbar zur Erzeugung von Ausgabeimpulsen mit programmierbarer Dauer, Puslweitenmodulation).

Timer/Counter 0

8-Bit-Zähler, der durch den Prozessortakt (ggf. vorgeteilt) oder externe Ereignisse (Takte) hochgezählt werden kann.

Vom Prozessor aus ansprechbar über E/A-Register.



Zähler selbst (8 Bit, 0 ... 255):

$32 ($52) TCNT0 Timer/Counter 0

Kontrollregister für Betriebsarten:

$33 ($53) TCCR0 Timer/Counter Control Register 0

Mit:

7 ... 3	2	1	0
waveform generation mode	CS02	CS01	CS00

No clock (Timer/Counter stopped)	0	0	0
CLK (Processor Clock)	0	0	1
CLK/8 (Prescale)	0	1	0
CLK/64 (Prescale)	0	1	1
CLK/256 (Prescale)	1	0	0
CLK/1024 (Prescale)	1	0	1
T0 (PB0)-Pin, falling edge	1	1	0
T0 (PB0)-Pin, rising edge	1	1	1

Die TNC0-Betriebsarten sind über Bits 3 bis 7 von TCCR0 einstellbar (z. B. PWM).

Vergleichsregister mit Zählerstand:

$3C ($5C) OCR0 Output Compare Register 0

Flagregister (für alle Zähler gemeinsam):

$38 ($58) TIFR Timer/Counter Interrupt Flag Register

- **OCF0** Output Compare Flag 0 (Bit 1)

  wird gesetzt, wenn der Zählerstand den Wert im Vergleichsregister erreicht hat (TCNT0 = OCR0).

- **TOV0** Timer/Counter Overflow Flag 0 (Bit 0)

  wird gesetzt, wenn der Zähler überläuft, d. h. Wechsel in TCNT0 von $FF nach $00.

Flags können z. B. in einer Schleife abgefragt werden (Polling) oder einen Interrupt (s.u.) auslösen.

Automatisches Rücksetzen bei Ausführung des Interrupts oder Rücksetzen durch Schreiben einer *Eins* in das jeweilige Bit!

Interrupt-Maskierungsregister (für alle Zähler):

$39 ($59) TIMSK Timer/Counter Interrupt Mask Register

Lokale Freigabe (=1) / Sperren (=0) von Timer0-Interrupts durch:

- **OCIE0** Counter Output Compare Match Interrupt Enable 0 (Bit 1)
- **TOIE0** Timer/Counter Overflow Interrupt Enable 0 (Bit 0)

Timer 1 und **Timer 2** mit ähnlichen Betriebsarten und Registern, nur Timer 1 mit 16 Bit Auflösung.

Externe Taktung im entsprechenden Modus möglich, z. B. durch externe Ereignisse oder Taktgenerator für Echtzeituhr, oder Taktung per Software durch Pin-Toggle.

Beispiel 2: Ereigniszähler (Timer 0) mit Polling

An Port B, Pin PB0 (T0) anliegende Impulse sollen mit Timer 0 gezählt werden (L/H-Flanken). Nach ANZ = 3 Eingangsimpulsen ist ein Pegelwechsel (‘Toggle‘) an Pins PB6 und PB7 von Port B vorzunehmen.

```
.include “m16def.inc.“                  ;  Get label definitions
.def    TEMP    = R16                   ;  Label for working register
.equ    ANZ     = 3                     ;  Counts required for toggle

.CSEG   ; CODESEGMENT
        .ORG    $0000                   ;  Starting at address 0

        ; Prepare Counter 0 and Init Port B
INIT:   LDI     TEMP, (1<<CS02)|(1<<CS01)|(1<<CS00) ; Mode:
        OUT     TCCR0, TEMP             ;  External Event Counter T0 L/H
        LDI     TEMP, ANZ-1             ;  Set Compare Value of
        OUT     OCR0, TEMP              ;  Counter to ANZ-1
        LDI     TEMP, (1<<PB6)|(1<<PB7) ;  Set PinB 6,7
        OUT     DDRB, TEMP              ;  as Output, others tristate Input
        CLR     TEMP                    ;  Set Port B
        SBR     TEMP, $80               ;  to initial output
        OUT     PORTB, TEMP             ;

        ; Reset Counter
START:  CLR     TEMP                    ;  Set Timer/Counter 0
        OUT     TCNT0, TEMP             ;  to zero

        ; Polling Loop until ANZ Events Counted
POLL:   IN      TEMP, TIFR              ;  Read Timer-Flags
        SBRS    TEMP, OCF0              ;  Skip if Output Compare set
        RJMP    POLL                    ;  else polling loop

        ; Invert Bits 6,7 of Port B
        IN      TEMP, PORTB             ;  Get value of Port B
        COM     TEMP                    ;  Invert Bits
        ANDI    TEMP, (1<<PB6)|(1<<PB7) ;  Write inverted
        OUT     PORTB, TEMP             ;  Bits 6,7 back to Port B

        ; Clear Flag and Start Next Counting Round
        LDI     TEMP, (1<<OCF0)         ;  Clear Compare Flag
        OUT     TIFR, TEMP              ;  by writing a one into it
        RJMP    START                   ;  Endless loop when done
```

Analoge Schnittstelle

Analog/Digitalwandler → Eingabe

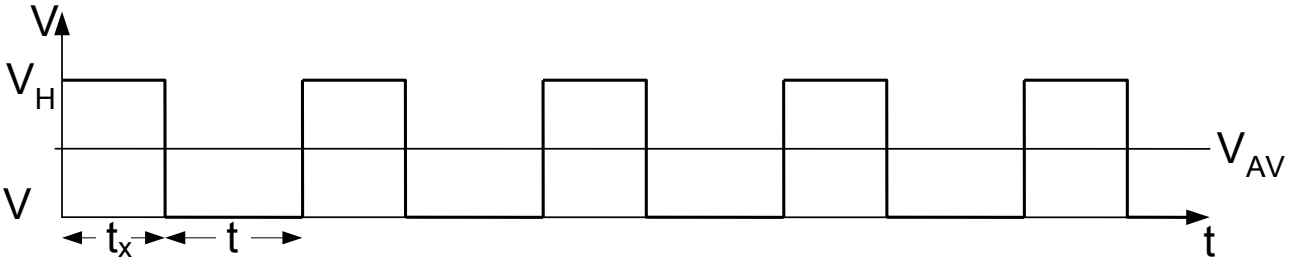
Der ATmega16 erlaubt die Umwandlung analoger Spannungen in 10-Bit-Digitalwerte für bis zu 8 Kanäle. Die zu wandelnden Spannungen werden an den Eingängen von Port A angelegt.
Wandlung mehrerer Kanäle im Zeitmultiplex
Eine Referenzspannung für den maximalen Wert V_{REF} wird an eigenem Pin angelegt (oder optional intern erzeugt).

Analoger Komparator

Vergleicht zwei analoge Spannungen, angelegt an Pins 2 und 3 von Port B, z. B. um einfache Analog-Digital-Wandler zu realisieren. Wenn die Spannung an PB2 höher ist als an PB3, wird das Statusflag ACO auf 1 gesetzt, sonst auf 0.

Digital/Analogwandler (über Timer) → Ausgabe

Quasi-analoge Ausgangsspannungen können durch Pulsweitenmodulation (PWM) aus von einem Timer generierten digitalen Signal gewonnen werden. Eine Glättung kann falls nötig durch externe (elektrische) Tiefpassfilter erfolgen.



$$V_{AV} = \frac{V_H \cdot t_x + V_L \cdot t_y}{t_x + t_y}$$

Weitere Anmerkungen:

Nach einem Reset sind alle Port-Pins als Eingang geschaltet (hochohmig). Sie müssen gesondert konfiguriert und freigegeben werden.

Mittels eines *Bootloader*-Programms im Flash-ROM kann ein Programm, z. B. für Updates des prozessorinternen Programms (*Firmware*), auch über die USART-Schnittstelle geladen werden (anderer Adressbereich, “Read-while-Write“).

Der ATmega verfügt über die genannten Peripherie-Einheiten hinaus noch über weitere Elemente, die über Konfigurationsregister gesteuert werden:

- Watchdog-Timer (Erkennen von Fehlerzuständen der Software durch Überprüfen von Zeitbedingung, z.B. zur Absicherung von Echtzeitbetrieb)

- Bootloader Lock-Bits (zum Schutz vor Auslesen und/oder Veränderung eines geladenen Programms)

- EEPROM (zum Halten von Parametern über das Ausschalten der Versorgungsspannung hinaus)

  → nicht flüchtig

- JTAG-Interface (zum Programmieren und Debuggen)

- Boundary Scan-Interface (zum Testen der Hardware des Controllers mit seiner angeschlossenen Elektronik im eingebauten Zustand)

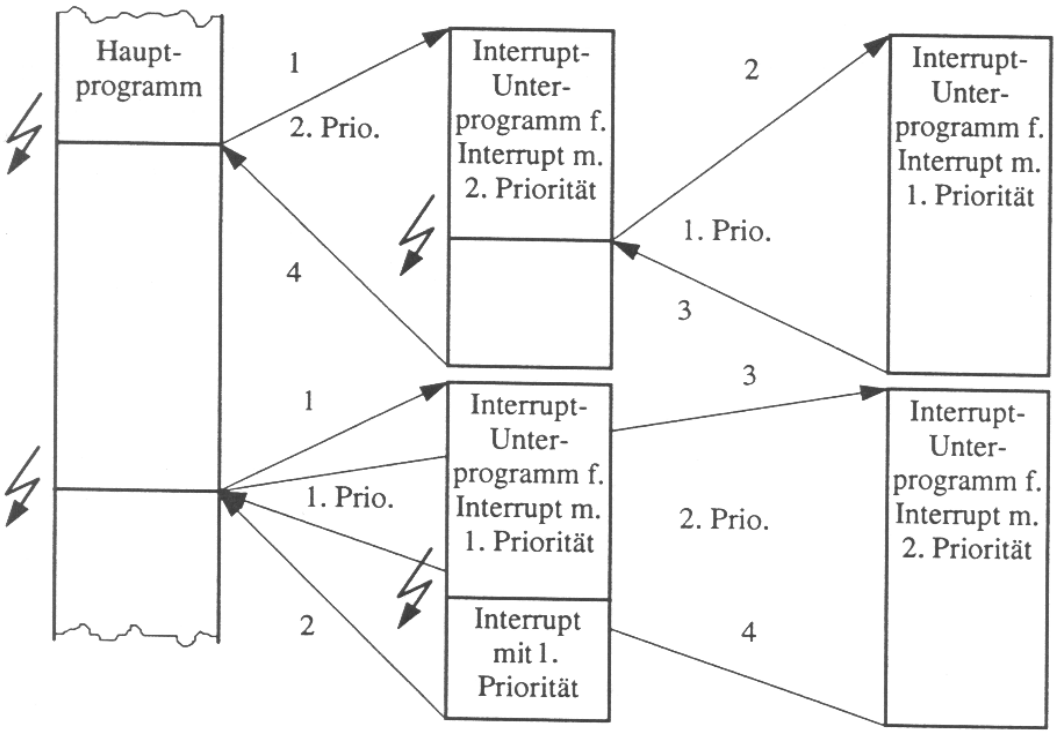
9.3 Interrupts (Unterbrechungen)

9.3.1 Allgemeines Interrupt-Prinzip

Ein Interrupt bedeutet die (asynchrone) Unterbrechung eines laufenden Programms durch ein externes (z. B. E/A-Bausteine) oder internes (z. B. Division durch Null) Ereignis. Der Prozessor verzweigt dann in eine bereitgestellte Routine (Interrupt-Service-Routine; Interrupt-Handler) und bedient die Unterbrechung.

Nach dessen Bedienung (Service) wird das unterbrochene Programm an der alten Stelle, also mit der nächsten Anweisung, fortgesetzt. *Das unterbrochene Programm merkt damit gar nicht, dass es unterbrochen worden ist! (Die Interrupt-Service-Routine muss dafür aber **seiteneffektfrei** sein!)*

Interrupt-Service-Routinen können selbst wieder durch höherpriore Interrupts unterbrochen werden (geschachtelte Interrupts).



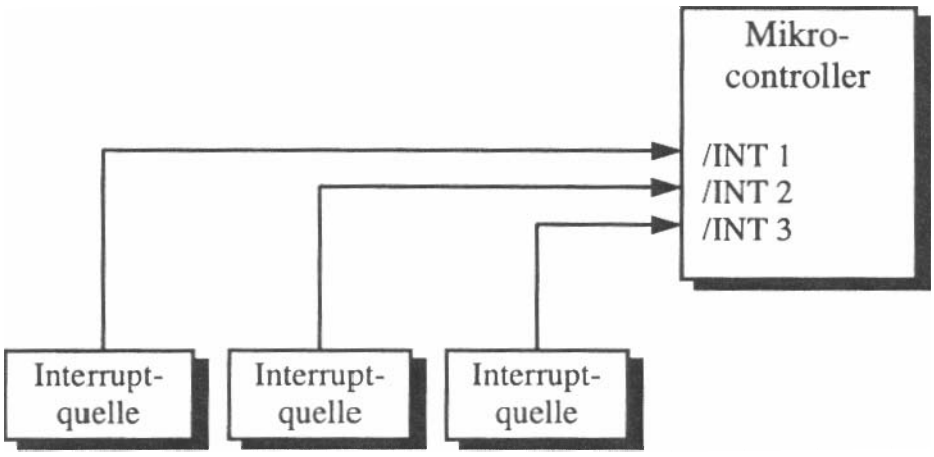
Interrupts sind unter Softwarekontrolle aktivierbar (freischalten) bzw. deaktivierbar (Sperren, *Maskierung* von Interrupts).

Besonders wichtige, hochpriore Interrupts sind je nach Prozessor auch nicht maskierbar (*nichtmaskierbare* Interrupts).

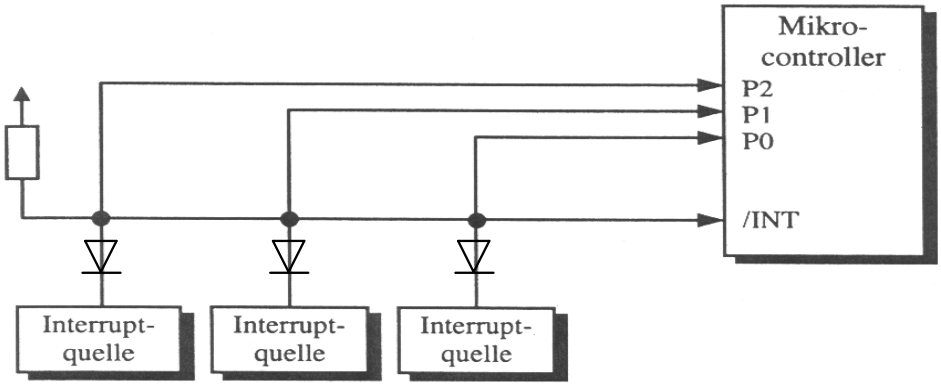
Interruptquellen

Externe Quellen (Interrupts):
E/A-Geräte wie Schalter, Signale, Tastaturen, Timer etc.

Zugeordnete Interruptquellen: für jede Quelle wird ein eigener Interrupt vorgesehen.



Gemeinsame Nutzung von Interrupts:
Liegt an einer gemeinsamen Interruptleitung ein Interrupt an, muss die Interrupt-Service-Routine durch Abfragen (‘Polling‘) der daran angeschlossenen Quellen ermitteln, welche den Interrupt verursacht hat. Bei mehreren gleichzeitigen Aktivierungen muss einer ausgewählt werden (z. B. nach Priorität, Round-Robin-Verfahren etc.).



Interne Quellen:

Interne Interrupts (auch **Traps** genannt) können automatisch durch die Prozessorhardware z. B. bei Fehlern wie *Division durch Null* oder *unerlaubter Opcode* selbst ausgelöst werden. Die Behandlung erfolgt wie bei externen Interrupts.

Durch spezielle Befehle können bei vielen Prozessoren interne Unterbrechungen auch durch den Programmierer ausgelöst werden (**Software-Interrupt**). Das ist oft eine beliebte Methode zum Aufruf von Betriebssystemroutinen (SVC: Supervisor Call; z. B. INT-Befehl bei 80x86).

Vorteil: Der Einsprung ins Betriebssystem ist unabhängig vom Programm. Und Schutzmechanismen können per Hardware beim Eintritt in die Serviceroutine deaktiviert werden (Supervisor Mode) und bei der Rückkehr zum Anwenderprogramm wieder aktiviert werden (User Mode).

Kontextwechsel:

Bei Auftreten einer Unterbrechung (extern oder intern) muss der Kontext des laufenden Programms (also der Prozessorstatus, d.h. mindestens der PC, ggf. auch Flags und Registerinhalte) gerettet und bei der Rückkehr wieder restauriert werden. Evtl. werden noch Schutzmechanismen deaktiviert und nachher wieder aktiviert.

Der (Prozessor-)Kontext wird in der Regel auf den Stack gerettet (vgl. Unterprogramme). Beim Eintritt des Interrupts geschieht dies automatisch per Hardware, beim Verlassen der Interrupt-Service-Routine durch einen speziellen Befehl (z.B. *RTI - Return from Interrupt*).

Die Größe des automatisch geretteten Kontextes ist je nach Prozessor unterschiedlich (zumindest PC, maximal alle Flags und Register).

Vektor-Interrupts:

Unterstützt ein Prozessor mehrere Interruptquellen, muss zu jeder die Startadresse der zugehörige Interrupt-Service-routine in Hardware zugeordnet werden. Dies geschieht in Hardware meist durch den Einsatz einer speziellen Einsprungtabelle (Interrupt-Vektortabelle), die an einer *festen* Adresse im Adressraum liegt.

In diese Tabelle müssen bei der Initialisierung die *Anfangsadressen* der Interrupt-Service-Routinen oder (Sprung-) Befehle dorthin (wie z.B. beim ATmega16) eingetragen werden.

Manche Prozessoren erlauben es, den Offset für die Tabelle (Vektornummer) bei der Unterbrechung vom Datenbus einzulesen, so dass ein E/A-Gerät seine eigene Vektornummer mitteilen kann (z. B. Motorola 680X0-Familie).

Interrupts können flanken- oder zustandsgesteuert sein. Im ersten Fall wird ein nicht unmittelbar bedienter Interrupt zwischengespeichert (RS-Flipflop) und auch dann noch bedient, wenn bei der nächst möglichen Interrupt-Bearbeitung das Interrupt-Signal nicht mehr ansteht.
Im zustandsgesteuerten Fall wird ein Interrupt nicht mehr bedient, wenn das auslösende Signal/Ereignis dann nicht mehr anliegt, aber der zugehörige Interrupt bearbeitet werden könnte.

Beachten: Verschiedene Prozessorfamilien weisen recht unterschiedliche Varianten der Interrupt-Behandlung auf!

9.3.2 Interruptsystem des ATmega16

Der ATmega16 hat ein recht einfaches Interruptsystem mit 21 maskierbaren Interrupts inkl. RESET, davon 3 externe Interrupts über die vordefinierten Pins INT0, INT1, INT2.

Es gibt keine nicht-maskierbaren Interrupts (außer RESET) und keine Software-Interrupts (Traps).

Software-Interrupts sind notfalls über die Pins INT0, INT1, INT2 (entsprechen PD2, PD3, PB2) durch Schreiben auf die Port-Pins emulierbar.

Prinzipieller Ablauf eines Interrupts (in Hardware)

(1) Unterbrechung des laufenden Programms nach dem gerade in Ausführung befindlichen Befehl.

(2) Automatisches Retten des PC auf den Stack (PUSH). Sperren weiterer Interrupts (Löschen des *Global Interrupt Enable I-Flags*).

(3) Ausführung desjenigen Befehls, der an der für den vorliegenden Interrupt festgelegten Stelle in der Interrupt-Vektor-Tabelle steht (Vektor-Interrupt). In der Regel ist das ein Sprungbefehl zur eigentlichen Interrupt-Service-Routine.

(4) Ausführung der Interrupt-Service-Routine.

(5) Abschluss der Interrupt-Service-Routine mit dem Befehl RETI, der Interrupts wieder global freischaltet (Setzen des I-Flags), den PC wieder vom Stack holt (POP) und damit das unterbrochene Programm an der alten Stelle fortsetzt (durch automatische Restauration des PCs).

Interrupt-Maskierung

global: I-F im Statusregister SREG;
manipulierbar durch Flagregisterbefehle
(z. B. CLI, SEI)

lokal: durch spezielle Bits in den E/A-Kontrollregistern
(spezifisch für die einzelnen E/A-Interrupts)

Interrupt-Vektortabelle (im Flash-ROM)

Vector No.	Program Address(2)	Source	Interrupt Definition
1 (highest p.)	$0000(1)	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	$0002	INT0	External Interrupt Request 0
3	$0004	INT1	External Interrupt Request 1
4	$0006	TIMER2 COMP	Timer/Counter2 Compare Match
5	$0008	TIMER2 OVF	Timer/Counter2 Overflow
6	$000A	TIMER1 CAPT	Timer/Counter1 Capture Event
7	$000C	TIMER1 COMPA	Timer/Counter1 Compare Match A
8	$000E	TIMER1 COMPB	Timer/Counter1 Compare Match B
9	$0010	TIMER1 OVF	Timer/Counter1 Overflow
10	$0012	TIMER0 OVF	Timer/Counter0 Overflow
11	$0014	SPI, STC	Serial Transfer Complete
12	$0016	USART, RXC	USART, Rx Complete
13	$0018	USART, UDRE	USART Data Register Empty
14	$001A	USART, TXC	USART, Tx Complete
15	$001C	ADC	ADC Conversion Complete
16	$001E	EE_RDY	EEPROM Ready
17	$0020	ANA_COMP	Analog Comparator
18	$0022	TWI	Two-wire Serial Interface
19	$0024	INT2	External Interrupt Request 2
20	$0026	TIMER0 COMP	Timer/Counter0 Compare March
21 (lowest p.)	$0028	SPM_RDY	Store Program Memory Ready

Notes: 1. When the BOOTRST Fuse is programmed, the device will jump to the Boot Loader address at reset, see ‘Boot Loader Support – Read-While-Write Self-Programming’

2. When the IVSEL bit in GICR is set, interrupt vectors will be moved to the start of the Boot Flash section. The address of each Interrupt Vector will then be the address in this table added to the start address of the Boot Flash section.

Bei gleichzeitig auftretenden Interrupts entsprechen die Interruptprioritäten der Reihenfolge in der Interruptvektor-Tabelle (RESET höchste Priorität, SPM_ READY niedrigste).

In die Tabelle müssen *Befehle* eingetragen werden (nicht nur Adressen (Vektoren) wie bei vielen anderen Prozessoren).

Der ***Stack Pointer*** muss natürlich vorher vom Hauptprogramm **initialisiert** worden sein.

Beispiel: Aufbau eines Programms mit Interrupts

```
            ; Interrupt Vector Table
$0000       JMP   INIT            ; JMP to Main Routine
                                  ; after RESET
              :
              :
              .
$0026       JMP   TIM0_COMP       ; Timer 0 Compare Match
              :
              :
              .
            ; Interrupt Service Routine
TIM0_COMP:    .                   ; Timer 0 Compare Match
              :
              :
            RETI
              :
              :
              .
            ; Main Program
INIT:       LDI   R16, high (RAMEND)  ; Init
            OUT   SPH, R16            ; Stackpointer to
            LDI   R16, low (RAMEND)   ; End of
            OUT   SPL, R16            ; RAM
              :
              :
              .
            LDI   R16, (1 << OCIE0)   ; Local Interrupt Enable
            OUT   TIMSK, R16          ; per byte operation
            SEI                       ; Global Interrupt Enable
              :
              :
              .

Oder:       IN    TEMP, TIMSK         ; Set Local Interrupt
(besser)    ORI   TEMP, (1 << OCIEO)  ; Enable bit only before
            OUT   TIMSK, TEMP         ; Global Interrupt Enable
```

Anmerkung:
Wird kein Interrupt verwendet, kann der Speicherbereich der Interruptvektor-Tabelle für Programme genutzt werden (nach RESET unmittelbarer Start an Adresse $0000).

Achtung:

Die Ausführung eines Interrupts rettet *keine* Register, nicht einmal das Statusregister SREG. Der Programmierer muss selbst **alle** genutzten Register retten und wieder restaurieren (PUSH/POP auf/vom Stack).

Vorteil: flexibler und schneller als das automatische Retten und Restaurieren des gesamten Prozessorstatus, da i. Allg. nicht alle Register betroffen sind

Nachteil: Programmieraufwand, Gefahr von Fehlern

Beispiel: Retten/Restaurieren von R16 und SREG

```
            ; Interrupt Service Routine
ISR:    PUSH  R16          ; Save R16
        IN    R16, SREG    ; and SREG
        PUSH  R16          ; on Stack

          :                ;       R16 and SREG
          .                ; ready for use

        POP   R16          ; Restore SREG
        OUT   SREG, R16    ; and R16 from Stack
        POP   R16          ; in LIFO order
        RETI               ; Leave Interrupt
```

Geschachtelte Interrupts

Die Ausführung eines Interrupts sperrt beim ATmega16 durch Löschen des I-Flags automatisch alle weiteren Interrupts. Für geschachtelte Interrupts muss das I-Flag in der Interrupt-Service-Routine also wieder explizit gesetzt werden.

Beispiel: Interrupt-Verarbeitung

Hauptprogramm:
Umwandlung von BCD-Zahl an Port A in ASCII-Zahl auf Port D (siehe Beispiel 1) in Endlosschleife.

Hintergrundprogramm:
Zählt Impulse (in Hardware) an PB0 (T0 im Timer/Counter 0) und invertiert PB6, PB7 nach ANZ Impulsen (Beispiel 2) durch die *Interrupt-Service-Routine* quasi nebenläufig und ereignisgesteuert.

```
.include “m16def.inc.“          ;  get label definitions

.def   TEMP   =  R16            ;  R16 working register
.equ   ANZ    =  3              ;  Number of events to be counted

; INTERRUPT VECTOR TABLE
.CSEG                                       ; CODE SEGMENT
            .ORG $0000                      ; RESET Address
            JMP       INIT                  ; JMP to MainProg
            .ORG $0026                      ; Timer0 Compare
            JMP       T0Comp                ; JMP to ISR

; MAIN PROGRAM: Initialization
            .ORG $0028                      ; first addr after IVT
INIT:       LDI       TEMP, high(RAMEND)
            OUT       SPH, TEMP             ; Init SP to
            LDI       TEMP, low (RAMEND)
            OUT       SPL, TEMP             ; RAMEND
            RCALL     InitT0                ; Init event counter
            SBI       TIMSK, OCIE0          ; Enable Timer0 Compare inter.
                                            ; locally
            SEI                             ; Enable Global Interrupt
```

```
; Program Code BCDtoASCII CONVERSION from Example 1
START:  SER     TEMP            ;  Configure
        OUT     DDRD, TEMP      ;  PortD as Output
        CLR     TEMP            ;  Configure
        OUT     PORTA, TEMP     ;  Tri-State
        OUT     DDRA, TEMP      ;  Input for PortA
LOOP:   IN      TEMP, PINA      ;  Input Byte from PortA
        ANDI    TEMP, $0F       ;  Mask out low Nibble
        ORI     TEMP, $30       ;  Set ASCII to high Nibble
        OUT     PORTD, TEMP     ;  Output Byte to Port D
        RJMP    LOOP            ;  Endless polling loop

; EVENT COUNTER from Example 2
        ; SUBROUTINE Prepare Counter 0 and Init Port B
InitT0: LDI     TEMP, (1<<CS02)|(1<<CS01)|(1<<CS00)
        OUT     TCCR0, TEMP     ;  Mode: Event Counter T0 L/H
        LDI     TEMP, ANZ-1     ;  Set Compare Value of
        OUT     OCR0, TEMP      ;  Counter to ANZ-1
        LDI     TEMP, (1<<PB6)|(1<<PB7)  ;  Set PinB 6,7
        OUT     DDRB, TEMP      ;  as Outp., others Inp./Hi-Z
        CLR     TEMP            ;  Set Port B
        SBR     TEMP, $80       ;  to initial output
        OUT     PORTB, TEMP     ;
        ; Init Counter to zero and return
        CLR     TEMP            ;  Set Timer/Counter 0
        OUT     TCNT0, TEMP     ;  to zero
        RET
        ; INTERRUPT SERVICE ROUTINE for T0 Compare
        ; Save Context on Stack
T0Comp: PUSH    TEMP            ;  Save TEMP (R16)
        IN      TEMP, SREG      ;  and SREG
        PUSH    TEMP            ;  on Stack
        ; Invert Bits 6,7 of Port B
        IN      TEMP, PORTB     ;  Get value of Port B
        COM     TEMP            ;  Invert Bits
        ANDI    TEMP, (1<<PB6)|(1<<PB7)  ;  Write inverted
        OUT     PORTB, TEMP     ;  Bits 6,7 back to Port B
        ; Reset Counter
        CLR     TEMP            ;  Set Timer/Counter 0
        OUT     TCNT0, TEMP     ;  to zero
        ; Restore Context and return
        POP     TEMP            ;  Restore
        OUT     SREG, TEMP      ;  TEMP
        POP     TEMP            ;  and SREG
        RETI
```

9.4 Kontrollsysteme für komplexere Automatisierungsanwendungen

9.4.1 Kernprobleme

Industrieanlagen

Ausdehnung über mehrere hundert Meter, mehrere tausend Sensor-/Aktor-Signale, raue Umgebung, d.h. starke Verschmutzung, Feuchtigkeit, Störstrahlungen, Vibrationen etc.

Produktion / Fertigung

Fertigungsstraße mit z. B. 30 Industrierobotern in einer Anlage; jeder setzt über 1000 Schweißpunkte pro Stunde

Roboter in der Autofertigung und Verpackungsindustrie (Kuka Roboter GmbH)

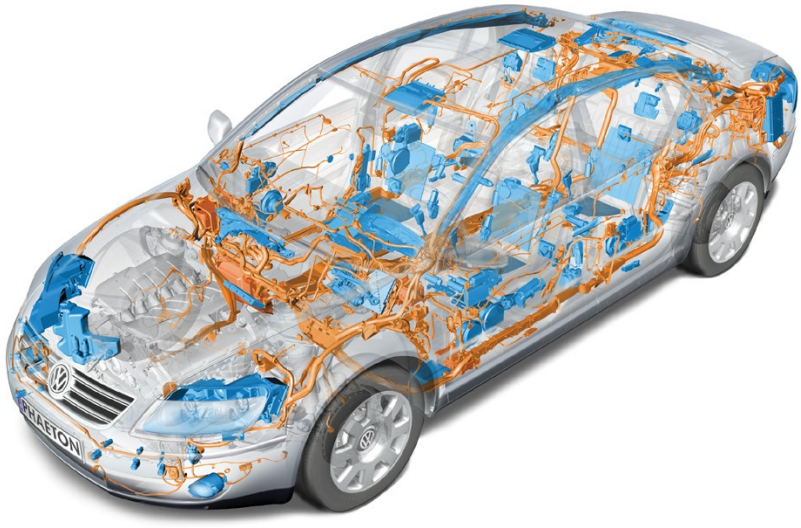
Containerschiffe

Bis über 8000 Datensignale, die kontinuierlich überwacht werden müssen

Containerschiff mit 6000 Containern, 320 m Länge (OSS, Dänemark)

Kraftfahrzeuge

30 - 120 Mikrorechner verschiedener Hersteller und bis zu über 50 Sensoren im Fahrzeug verteilt, mehrere km Kabel;
ca. 60% der Innovationen im Kraftfahrzeugbereich durch Mikroelektronik und Software, zunehmender Kostenanteil der Elektronik (derzeit ca. 20 → 24 %), sehr großer Kostendruck



Bordnetz des VW Phaeton

Generelle Anforderungen:

- **Beherrschbarkeit der Komplexität**
(große Menge an Sensoren, Aktoren, nebenläufigen Tasks und Prozessoren, Interaktionen, Fehlerquellen, …)

- **Echtzeitproblematik**
  - konsistente Sicht auf den Zustand des Gesamtsystems
    - synchrones, d.h. zeitgleiches Erfassen aller Sensorinformationen
    - stark unterschiedliches Zeitverhalten verschiedener Systemkomponenten und Tasks (Millisekunden bis Minuten)
  - zeitgerechte, d.h. rechtzeitige Reaktion im ganzen System

→ **Gleichzeitigkeit und Rechtzeitigkeit**

Anmerkung: Das Einhalten von Zeitgrenzen muss bei harter Echtzeit **immer** garantiert werden können.
Deshalb sind kurze Reaktionszeiten oft wichtiger als ein großer Befehlsdurchsatz.

→ Einfach nur schnell sein reicht nicht!

- **Verlässlichkeit** ist die allgemeine Eigenschaft eines Systems, bezüglich der Anforderungen vertrauenswürdig zu sein.

  → Als Oberbegriff schließt diese Eigenschaft Aspekte für den Betrieb wie Zuverlässigkeit, Verfügbarkeit, Sicherheit, Diagnostizierbarkeit, Datenintegrität und Zugriffschutz ein.

  → Für den Entwurf gelten vergleichbare Anforderungen an die Spezifikation und deren Umsetzung. Das impliziert auch verlässliche Entwick-

lungstools und einen sauberen (Software-)Entwurfsprozess mit mehreren Kontrollmechanismen (z.B. Code Reviews, Bug-Tracking, …).

- **Zuverlässigkeit** ist die Eigenschaft eines Systems, die angibt, wie verlässlich eine dem Produkt zugewiesene Funktion in einem Zeitintervall zur Laufzeit erfüllt wird, also ggf. auch beim Auftreten von Störungen und Fehlern wie Komponentenausfall; meist durch die Überlebenswahrscheinlichkeit beschrieben.

- **Verfügbarkeit** ist die Wahrscheinlichkeit, dass ein System (bei möglichen Ausfällen) in einem Zeitintervall tatsächlich zur Verfügung steht.

- **Kostenreduktion** der Hardware und Software und deren Entwicklung

In vielen Anwendungen gelten noch **weitere Anforderungen** hinsichtlich:

- Platzbedarf
- Stromverbrauch
- Erwärmung
- Entwicklungszeit
- Entwurfssicherheit
- Wiederverwendbarkeit bestehender Komponenten (Hardware, Software, …)
- Strahlungsfestigkeit
- Vibrationsfestigkeit
- Explosionsschutz
- lange Verfügbarkeit von Ersatzteilen
- Zweitanbieter
- Herstellerbindung
- . . .

9.4.2 Rechnerarchitekturen für Automatisierungssysteme

Modularisierung

Wie in anderen Bereichen auch wird für komplexere Automatisierungsaufgaben das Komplexitätsproblem der Rechnersysteme durch Modularisierung gelöst (*divide-and-conquer*). Der Entwickler kann sich dabei auf die einzelne Modulfunktionalität konzentrieren. Ein Modul muss sich auch hier nahtlos in den Gesamtkontext einfügen und mit seiner Umgebung kommunizieren, d.h. funktionale und informationstechnische Schnittstellen einhalten.

Um eine aufwändige (sternförmige) Verkabelung zu vermeiden und bei einer großen räumlichen Ausdehnung, wird in vielen Anwendungsgebieten, wie z.B. einer Fertigungsstraße, einem Containerschiff und auch in Kraftfahrzeugen, die Automatisierungsaufgabe auf **mehrere Rechner** lokal oder zentral verteilt.

Häufig erfolgt auch eine **räumliche Trennung** von Rechner und Sensorik/Aktorik, wobei die Sensor- und Aktorinformation über Feldbus-Systeme übertragen werden.

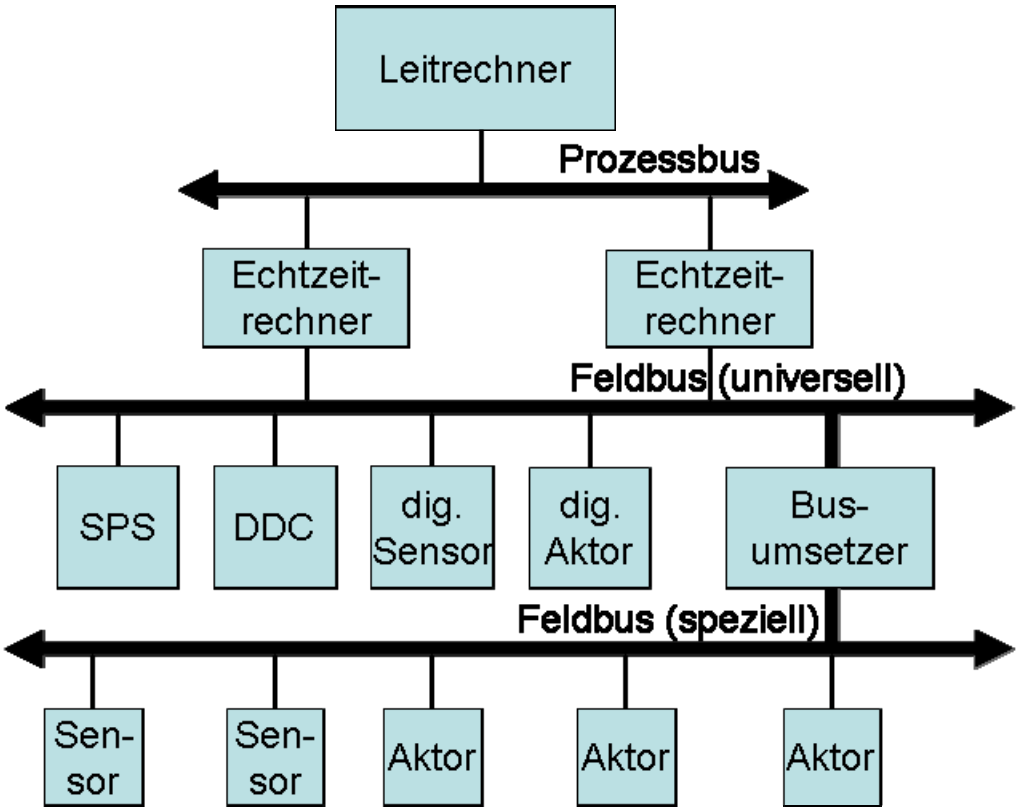
Dieses Konzept wird auch (z.B. in Kraftfahrzeugen) ganz gezielt für eine deutliche Kostenreduktion ausgenutzt.

Denn die Modularisierung erlaubt die **Wiederverwendung** von

- Funktionalitäten (Entwicklungskosten) und
- Hardware (Reduktion der Herstellungskosten durch große Stückzahlen)

Hierarchisierung

Weil auf verschiedenen Verarbeitungsebenen auch verschiedene Anforderungen an das Zeitverhalten (Echtzeitfähigkeit) und die Menge der ausgetauschten Daten gestellt werden, wird insbesondere in Industrieanlagen eine hierarchische Struktur von verschiedenen Rechnern verwendet, die sich an der Aufgabenhierarchie orientiert.



- Leitrechner: Planung und Koordinierung von Teilaufgaben und Überwachung des Gesamtsystems; leistungsfähige Workstation/Großrechner; große Datenmengen (Dateien), keine oder nur weiche Echtzeit

- Echtzeitrechner: Bearbeitung von zeitkritischen Teilaufgaben in harter Echtzeit; Leistung auf PC-Niveau, Industrie-PCs, spezielle Echtzeitrechner, Echtzeit-Betriebssysteme; paketförmiger Datentransfer (kBytes) unter harten Echtzeitbedingungen (z.B. Feldbusse, Echtzeit-Ethernet)

- SPS = Speicherprogrammierbare Steuerung:

  ursprünglich industrielle Realisierung von Schaltwerken mit zyklischer Abfrage **digitaler** Sensorwerte und Schaltzustände;

  Einzeldaten-Transfer, Echtzeit je nach Anwendung im ms-Bereich;

  Ursprünglich Programmierung mit eigenen Programmiermethoden (Koppelplan, Anweisungsliste), die sich teils an einer Relais-Realisierung orientierten.

  Heute Realisierung mittels Mikrocontroller, Programmierung auch grafisch oder in Hochsprachen (C), Kombination mit analogen Funktionen wie Schwellwertüberwachung und einfache Regelungen;

  Leistung heute i.d.R. auf Mikrocontroller- bis PC-Niveau

- DDC = Direct Digital Control:

  Komponente ähnlich wie eine SPS, aber primär für die Verarbeitung von **analogen** Größen; vorzugsweise für Steuerungs- und Regelungsaufgaben.

  Die Unterschiede zwischen DDC und SPS verschwimmen heute immer mehr, weil in beiden Fällen vergleichbare Rechnerhardware, nur andere Softwarekonzepte verwendet werden.

9.4.3 Feldbus-Systeme

Auf der untersten (Prozessleit-)Ebene geschieht die physikalische Ankopplung an die jeweilige technische Anwendung, dem so genannten „*Feld*“, durch Sensoren und Aktoren.

Man spricht von einem *Feldbus*, wenn die Sensoren bzw. Aktoren nicht direkt mit einem Rechner, sondern über einen (echtzeitfähigen) Bus verbunden sind.

Ziel: Vermeidung sternförmiger Verdrahtung, Reduktion des Verdrahtungsaufwands, Erhöhung von Flexibilität, Erweiterbarkeit und Wartbarkeit, Kostenreduktion

Parallele Busse sind meist nur für die Übertragung innerhalb von wenigen Metern geeignet. Für industrielle Anwendungen und in Kraftfahrzeugen werden deshalb praktisch ausschließlich serielle Busse, die Feldbusse, verwendet. Manche Feldbusse können Daten auch über mehrere hundert Meter bis Kilometer übertragen.

Die wichtigsten Vertreter der Feldbusse sind:

- CAN-Bus (Kraftfahrzeuge)
- Flexray-Bus (fehlertolerierender Bus, KFZ)
- I^2C-, SPI-, ASI-Bus (low cost Aktor-Sensor-Busse)
- Profibus (Industrie)
- Industrial Ethernet (mit geregeltem Zugriff für Echtzeitfähigkeit; parallel dazu auch nicht-echtzeitgebundener Datentransport)
- zunehmend auch (echtzeitfähige) Funkverbindungen wie WLAN, Zigbee, M2M-Kommunikation (für weniger kritische Applikationen)

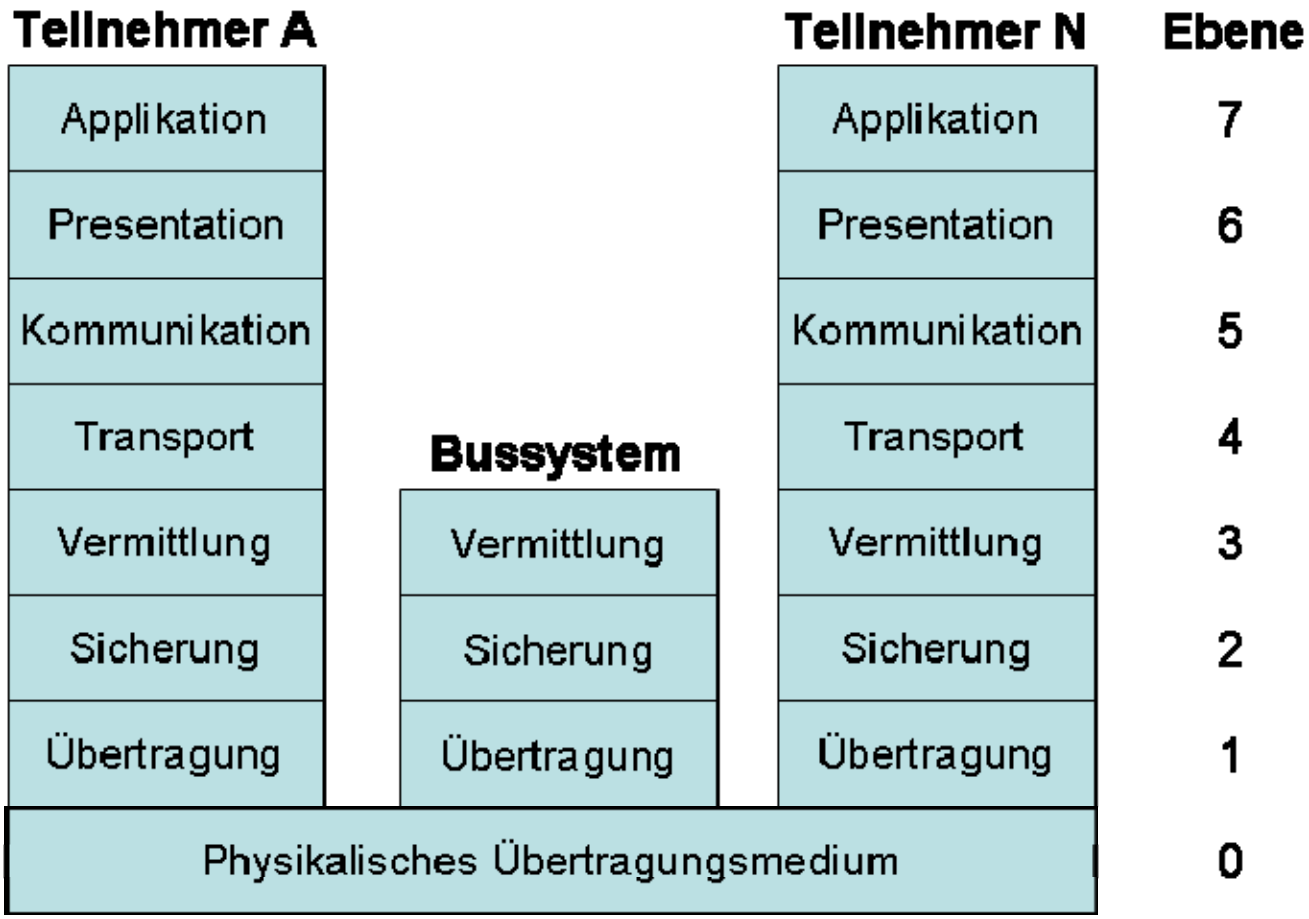
Durch Feldbusse und Funkverbindungen wird der Verdrahtungsaufwand oftmals drastisch reduziert. Und mehrere Rechner können (von verschiedenen Stellen) auf die gleichen Informationen zugreifen.
Allerdings steigen die Probleme mit dem Zugriffsschutz („Sicherheit“).

Die *universellen Feldbusse* verbinden Echtzeitrechner mit dezentralen Geräten. Die *speziellen Aktor-Sensor-Feldbusse* dienen für einfache Verbindungen zu Sensoren und Aktoren.

Weil sich aber mehrere Teilnehmer (Sensoren, Aktoren, Rechner) das Übertragungsmedium (Bus) teilen und damit Anwender die Produkte mehrerer Hersteller einsetzen können, müssen die Kommunikationspartner interoperabel sein, d.h. problemlos Daten austauschen können.

Dazu muss die Kommunikation nach vordefinierten Regeln erfolgen. Deshalb konnten sich nur die Feldbusse international behaupten, die standardisiert wurden. Das betrifft

- die Datenformate und Protokolle,
- die elektrischen Signalpegel und
- die mechanischen Verbindungen.



OSI-Referenzmodell für die Schichten
für die Netzwerk-Kommunikation

Das OSI-Referenzmodells (Open System Interconnection) standardisiert den Aufbau der Protokolle für eine Netzwerk-Kommunikation.

Aus Gründen der Echtzeitfähigkeit und Kosten werden oft nicht alle sieben Schichten unterstützt. Bei der häufig streng geregelten Zugriffskontrolle müssen sie das auch nicht.

Die Ebenen 1 und 7 sind immer vorhanden. Oft wird auch die Ebene 2 unterstützt, um bei Störungen ein Nachrichtenpaket (durch die Hardware gesteuert) zu wiederholen.

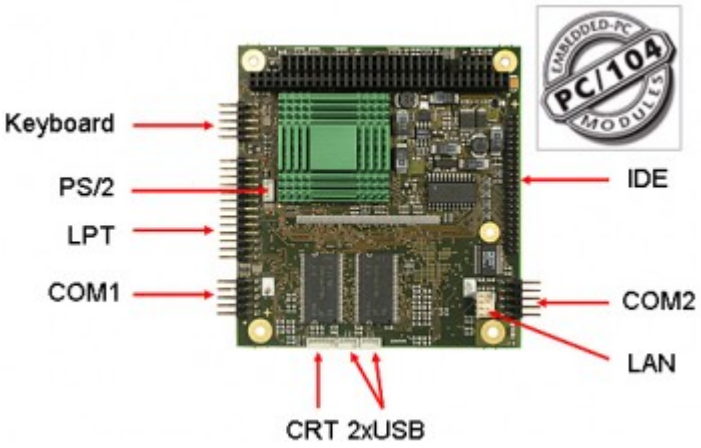
Falls mehrere Teilnehmer an einem Bus individuell adressiert werden sollen, ist auch Ebene 3 erforderlich.

9.4.4 Spezielle Kontrollerhardware

Durch die Ankopplung der Sensorik/Aktorik über Feldbusse müssen nicht notwendigerweise Mikrocontroller, sondern können auch Rechner ohne integrierte Peripherieschnittstellen eingesetzt werden.

Weil auf den unteren Automatisierungsebenen i.d.R. nur die Automatisierungsaufgabe (ohne User-Interface) läuft und keine großen Dateien gehandhabt werden müssen, sind verschiedene Hardware-Plattformen weit verbreitet:

- Industrie-PCs vorzugsweise für komplexere Aufgaben, bei denen es nicht so sehr auf die Kosten, den Platzbedarf und den Stromverbrauch ankommt.



  Industrie-PCs verfügen i.d.R. über ein (Echtzeit-) Betriebssystem und die gleichen Hardware-Schnittstellen wie ein normaler PC, sind aber robuster aufgebaut und für den Betrieb in einer rauen Umgebung ausgelegt. Sie können modular mit dedizierten Schnittstellen erweitert werden.

- Signalprozessoren für datenstrom-basierte Anwendungen wie Filteralgorithmen und Bildverarbeitung, bei denen es auf eine große arithmetische Rechenleistung (bis zu mehreren GFLOPS) bei geringem Platz- und Strombedarf und geringen Kosten ankommt.

- Mini-Rechner als guter Kompromiss aus Rechenleistung (mehrere hundert MHz), Platz, Kosten und Stromverbrauch; häufig mit einem abgespeckten Linux oder WindowsCE® als Betriebssystem. Solcher Rechner werden wegen der geringen Kosten (< 100 €) auch gern als Embedded Linux-System und Web-Client eingesetzt.



Beispiel:	Verdex XL6P	ECO820	Adelaide
CPU:	Marvell® PXA270, 600MHz	ATMEL AT91RM9200, 180 MHz	Freescale i.MX31 ARM-11, 532 MHz
Memory:	128MB RAM 32MB Flash	Flash ROM 16 or 32 MB SDRAM 32 or 64 MB	NAND-Flash ROM 64 or 128 MB DDR-SDRAM 64, 128 MB
Interfaces:	USB host CCD camera signals	SD / MMC cards CF cards RS232 IrDA / FIR I²C SPI SIM 100 MBit Ethernet 1 x USB Host 1 x USB Device	SD / MMC / SDIO cards PCMCIA / CF cards RS232 IrDA / FIR I²C SPI SIM 100 MBit Ethernet 1 x USB OTG 1 x USB Host 1 x USB Device 10 Bit A/D Converter
Multimedia Interfaces:			Programmable LCD Controller Up to 1024 x 1024 x 16 bpp Supports STN, Color STN, TFT Touch Controller (4-

			Wire) 3D-Accelerator MPEG-4 Encoder Camera Interface Audio I/O (16 Bit Stereo)
Size:	80mm x 20mm	48 x 65 mm	54 x 85,5mm (credit card size)
Power Draw:	250 mA	170 mA	600 mA
Software:	Linux Kernel 2.6	Windows CE Linux	Windows CE Linux
Price (> 1000 Pieces)	100 €	64 €	119 €

- Kombination einer CPU mit Hardware-Beschleunigern

Kernidee:

Auslagern von rechenintensiven Operationen in Hardware zur Entlastung der CPU

→ Hardware (FPGAs) als Akzelerator für schnelle Datenstrombearbeitung (hohe Bandbreite, kurze Latenz), aber einfache, relativ starre Algorithmen

Weitere Möglichkeiten durch FPGA:

- Erweiterung des Befehlssatzes
- Erweiterung um eine Floating-Point-Einheit

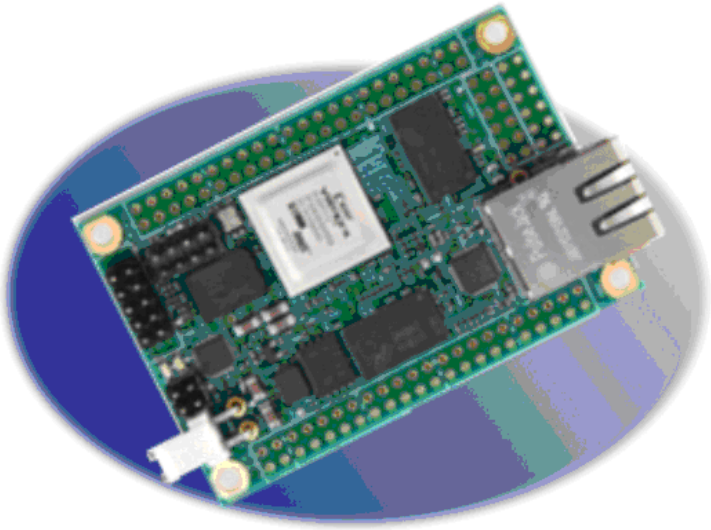
Alternative Sicht/Ansatz:

Die Hauptverarbeitung läuft in Hardware (FPGA). Ein Prozessor übernimmt den Teil einer Aufgabe, der komplexe Entscheidungen/Abläufe enthält und viele Variablen auszuwerten hat.

→ Prozessor entlastet die Hardware und vermeidet aufwändige Schaltwerke und Verdrahtung;

flexible, aber langsamere Befehls-(strom) verarbeitung

Beispiel: Suzaku SZ410-Board mit Xilinx XC4VFX12-FPGA



Hohe Flexibilität durch Xilinx Virtex-4 FX-FPGA mit integrierter PowerPC-CPU (350 MHz) **onchip** und Standardanschlüsse des Boards.

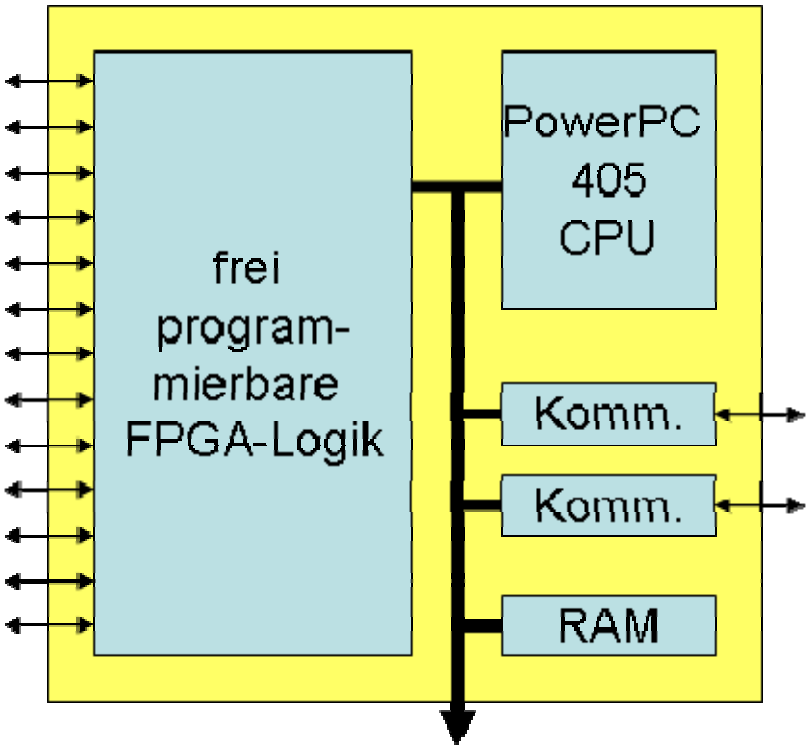
Linux Ready unterstützt Linux Kernel 2.6 als Standard Betriebssystem. Durch Linux ist eine umfangreiche Quellcode-Basis gegeben.

Network Ready durch integrierten Ethernet-Anschluss (10Base-T/100Base-TX).

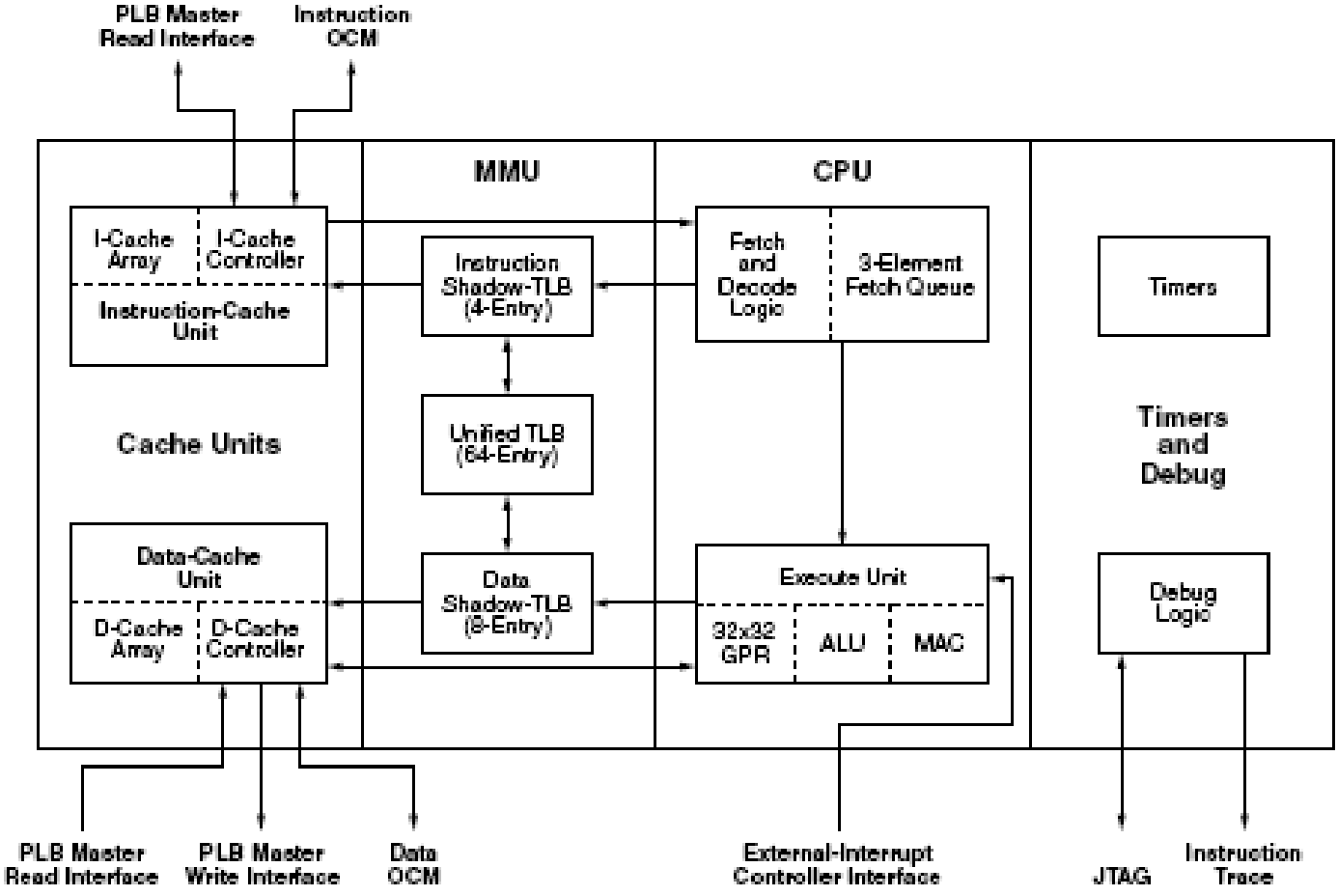
Sonstige Daten:
- 8 Mbyte SPI FLASH
- 2x32 Mbyte DDR2 RAM
- RS-232 COM Port (OPB-IP in FPGA)
- 86 General Purpose IO Pins
- Grösse: 72 x 47 mm

Eingesetztes FPGA: Xilinx XC4VFX12

- 90 nm-Technologie
- PowerPC 405-CPU
- 32 DSP-Slices
- frei programmierbarer Logikteil (12.312 Logikzellen, 10.944 Register, 87.552 RAM-Zellen)
- 2 Ethernet-MAC (10/100/1000 MHz)
- bis zu 320 I/O-Pins



Prinzipschaltbild des Xilinx XC4VFX12-FPGAs von Xilinx



Blockschaltbild der integrierten PowerPC 405-CPU

9.5 Hochleistungsrechner

Schnelle Verarbeitung von häufig wechselnden Anwendungen, vielfach quasi-parallele Bearbeitung nebenläufiger Tasks mit mehreren Usern und hoher Wechseldynamik.

→ Ziel: Möglichst hoher Befehlsdurchsatz

9.5.1 CISC- und RISC-Architekturen

CISC-Architekturen (Complex Instruction Set Computer)

CISC-Architekturen waren das Standard-Konzept in den 70er Jahren. Zu dieser Zeit galt (noch):

- Hauptspeicher war noch klein, teuer und langsam.
- Cache-Speicher gab es nicht.
- Deshalb war der Ansatz, möglichst viele mächtige Operationen mit einem Befehl (Opcode) zu aktivieren, um
  - den langsamen Speicherzugriff zu kompensieren,
  - kürzere Programme zu erhalten.
- Mit Hilfe der **Mikroprogrammierung** konnten immer komplexere Befehlssätze leicht implementiert werden.
- **Familienkonzepte:** (Beispiel: IBM System/360):

  Die abstrakte Struktur auf der Ebene der Maschinensprache (Register, Maschinenbefehlssatz) bleibt gleich, es existieren aber verschiedene Hardware-Implementierungen und -Realisierungen (unterschiedliche Hardware-Funktionseinheiten, verschiedene Technologien).

  Durch Mikroprogrammierung wird die gleiche abstrakte Maschinenstruktur ("Architektur" nach Blaauw) auf unterschiedlicher Hardware **emuliert.**

Argumente für CISC

Reichere Befehlssätze mit vielen verschiedenen, teils mächtigen Befehlen, Adressierungsarten und Spezialregistern

→ höhere Programmiersprachen durch immer komplexere Befehle besser unterstützen (*vertikale Verlagerung*),

→ kürzere Programme und schnellerer Programmtransfer (wichtig, als Hauptspeicher noch klein, teuer und langsam waren),

→ Compiler vereinfachen (heute kaum noch relevant)

→ die **"semantische Lücke"** schließen als ein Weg aus der "**Software-Krise**".

Software-Krise: (70er Jahre) stark steigende Softwarekosten im Vergleich zu den Hardwarekosten,

Schwierigkeiten bei der Beherrschung großer Softwaresysteme.

Semantische Lücke:

Höhere Programmiersprache:		Maschinensprache:
WHILE X <> 10 DO	← **sem. Lücke** →	LDA X
.		CMP #10
.		BNE ENDE
.		.
.		.
.		.
END		ENDE;

CISC-Prozessoren

CISC-Prozessoren unterstützen sowohl die Ausführung höherer Programmiersprachen (*Compiler-Schnittstelle* für Java, C/C++ etc.) als auch Multitasking-Betriebssysteme (*Betriebssystem-Schnittstelle* für UNIX o. ä.).

Compiler-Schnittstelle
(bzw. Anwenderprogrammierungs-Schnittstelle)

- Datentypen und -repräsentation
- Maschinenbefehle und Befehlsformate
- Adressierungsarten
- Operationen auf Daten
- Programmflusskontrolle

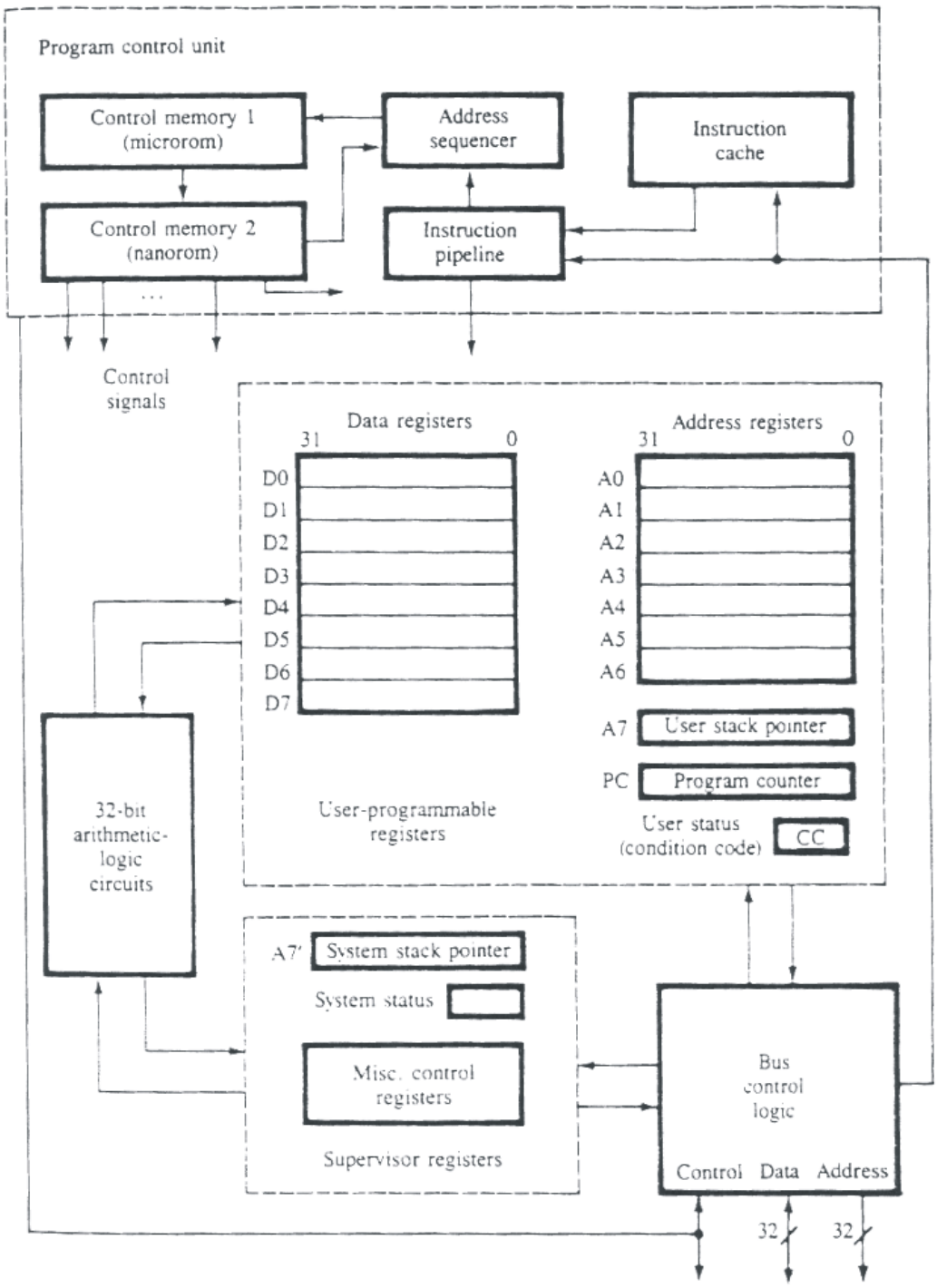
Betriebssystem-Schnittstelle

- Prozessverwaltung (Multitasking, Sicherheit, …)
- Speicherverwaltung (Datentransfer, virtueller Speicher, …)
- Ein/Ausgabe-Verwaltung (Interrupts etc.)
- Multiprozessor-Unterstützung (Kommunikation, Synchronisation, …)

Wegen des komfortablen Befehlssatzes sind CISC-Prozessoren auch zur reinen Assemblerprogrammierung geeignet.

Beispiel: CISC-Prozessor MC68020

Blockdiagramm



- 32-Bit-Registermaschine
- je 8 getrennte 32-Bit-Daten- und Adressregister
- Adresslänge 32 Bit, Byte-Adressierung, 4 GByte Adressraum
- Eineinhalbadress-Maschine (mit Ausnahmen)
- 2 Betriebsarten: User u. System Mode m. eigenen Stackpointern u. Status-Registern
- Mikroprogrammsteuerung mit Nanoprogrammierung, on-Chip Befehlscache u. Pipelining der Befehlsausführung
- CISC-Befehlssatz (ca. 120 Befehle, 16 Adressierungsarten, variables Befehlsformat)

Dynamische Aufruffrequenz von 68000-Befehlen

Instruction Class	Dynamic Frequency	
Move	32.85	
Branch/jump	15.58	
Decrement and branch	8.37	8.37
Compare/test	10.95	
Arithmetic/logical	16.31	
Add, Subtract, AND, OR	11.47	
Multiply	0.58	0.58
Divide	0.13	0.13
Shift and rotate	3.25	
Extend sign	0.88	
Subroutine	6.14	
Branch/jump to subroutine	1.69	
Return from subroutine	1.69	
Link/unlink	2.76	2.76
Miscellaneous	7.53	
Bit operation	0.22	0.22
Clear	2.69	
Load and push effective address	4.25	
Set conditional	0.37	
Total	100.00	12.06

Komplexe Befehle (rechte Spalte) machen nur ca. 12% der tatsächlich ausgeführten Befehle aus!

9.5.2 RISC-Prozessoren

RISC-Architekturen (Reduced Instruction Set Computer)

Erste RISC-Maschinen:

IBM 801	1979	IBM Research (Cocke)
RISC I, II	1982/83	University of Berkley (Patterson/Sequin)
MIPS	1983	Stanford University (Hennessy)

Motivation:

- Messungen bei Compilern zeigten:
  - ca. 80 % des generierten Codes verwendet nur ca. 20 % der Maschinenbefehle (statische Häufigkeit).
  - einfache Befehle und Adressierungsarten dominieren.
- Komplexe Befehle erfordern komplexe Steuerwerke (Entwurf) und lange Mikroprogramme mit entsprechend großer Ausführungszeit.
- Viele (komplexe) Befehle verbrauchen viel Chipfläche für den Mikro- (und Nano-)Programmspeicher (Kosten).
- VLSI-Technik steigert die Größe und Geschwindigkeit des Hauptspeichers (insbesondere in Verbindung mit Caches).
- Optimierende Compiler erzeugen sehr effizienten Code (besonders für einfache Maschinenbefehlssätze).
- Einfache Befehlssätze lassen sich besonders effizient in VLSI-Technik implementieren (feste Ablaufsteuerungen mit hoher Taktrate, mehr Platz für Register und zur Nutzung von chipinterner Parallelarbeit).

RISC-Philosophie

RISC ist keine konkrete Rechnerarchitektur, sondern mehr eine Philosophie, nach der Prozessoren entworfen werden.

Es gibt eine Reihe typischer RISC-Techniken, die sich in fast allen RISC-Prozessoren zur Verschlankung der Hardware wieder finden.

- Verwende möglichst einfache und möglichst wenige Maschinenbefehle **(reduzierte Befehlssätze)**, die sich sehr schnell in Hardware implementieren lassen.
  - nur die unbedingt notwendigen Befehle in Hardware vorsehen, den Rest emulieren
  - wenig Adressierungsarten, fehlende emulieren
  - wenig, einfache Befehlsformate mit einheitlicher Länge
- Verwende einen großen, universell verwendbaren Registersatz.
- Schaffe Befehle, die möglichst in einem Takt abgearbeitet werden (bei internem Pipelining).
- Verwende **optimierende Compiler**, um den zugehörigen Maschinencode zu erzeugen. (Assemblerprogrammierung ist i. Allg. nicht mehr vorgesehen!)

vgl.: Philosophie ‘sparsamer‘ Programmiersprachen

Beispiele

Erste RISC-Prozessoren:

	IBM 801	RISC I	MIPS
Year	1980	1982	1983
Number of instructions	120	39	55
Micro-control memory size	0	0	0
Instruction sizes (bits)	32	32	32
Technology	ECL MSI	NMOS VLSI	NMOS VLSI
Execution model	reg-reg	reg-reg	reg-reg

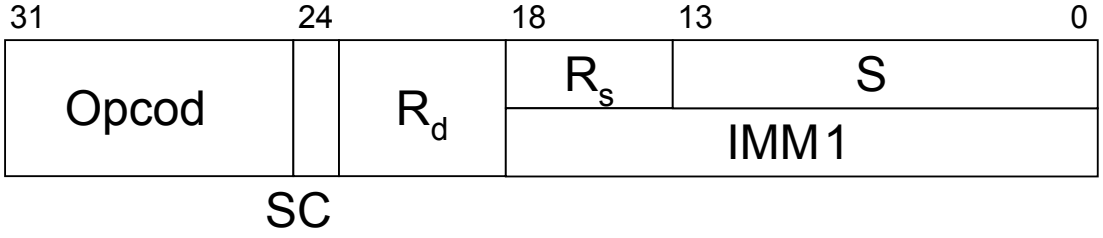
Vergleich CISC- / RISC-Prozessoren:

	CISC		RISC	
Characteristic	**MC68000**	**Z8000**	**RISC II**	**MIPS**
Year of introduction	1980	1979	1983	1983
Basic instructions	56	110	39	55
General registers	15	14	138	16
Addressing modes	14	12	3	4

Befehlssatz der Berkley RISC II (39 Befehle)

Arithm. /Log. Befehle	Speicher-zugriffs-befehle		Ablaufkontrolle und sonstige Befehle	Adressierungsarten
ADD	LDXW	STXW	CALLX	Arithm./Log. Operationen:
ADDC	LDRW	STRW	CALLR	$R_d \leftarrow R_s$ op S2
SUB	LDXHU		CALLI	
SUBC	LDRHU		RET	
SUBI	LDXHS	STXH	RETI	Load/Store-Operationen:
SUBCI	LDRHS	STRH	JMPX	a) Register-indiziert (x): EA = (R_S) + S2
AND	LDXBU		JMPR	
OR	LDRBU		GETPSW	b) PC-relativ (r): EA = (PC) + IMM19
XOR	LDXBS	STXB	GETLPC	
SLL	LDRBS	STRB	PUTPSW	EA = Effektivadresse PC = Programmzähler übrige Operanden: s. Befehlsformat
SRL				
SRA	LDHI			

RISC II-Befehlsformate (32 Bit für alle Befehle)



RISC-II Befehlsformat:

- SC = Set Condition Code Flag,
- R_d = 5-Bit-Zielregister-Adresse,
- R_S = 5-Bit-Quellenregister-Adresse,
- S2 = 5-Bit-Quellenregister-Adresse oder 14-Bit-Absolutwert,
- IMM19 = 19-Bit-Absolutwert (nur PC-relative Adressierung oder LDHI)

Architekturmerkmale von RISC-Prozessoren

Nicht notwendigerweise sind immer alle, aber typischerweise folgende Kennzeichen implementiert:

- möglichst wenige Befehle und Adressierungsarten

  (typ. ca. 50 Befehle, 2 Adressierungsarten: PC-relativ und indiziert).
  Neue Befehle nur aufnehmen, wenn sie **zur Laufzeit** eine deutliche Geschwindigkeitssteigerung **im Mittel** bringen.

- Einheitliches Befehlsformat:

  1 Wort = 1 Befehl (einfache Dekodierung)

- Ein-Zyklus-Maschinenbefehle:

  Möglichst alle Befehle laufen in einem Taktzyklus ab.

- Load/Store-Architektur:

  Nur über Load/Store-Befehle Zugriff auf Hauptspeicher. Alle anderen Befehle: Register – Register.

- Reguläre Hardware-Struktur:

  leichte VLSI-Realisierbarkeit.

- keine Mikroprogrammierung:

  festverdrahtete Ablaufsteuerung (einfaches Steuerwerk) für hohe Taktraten.

- Aufwandsverlagerung in Compiler:

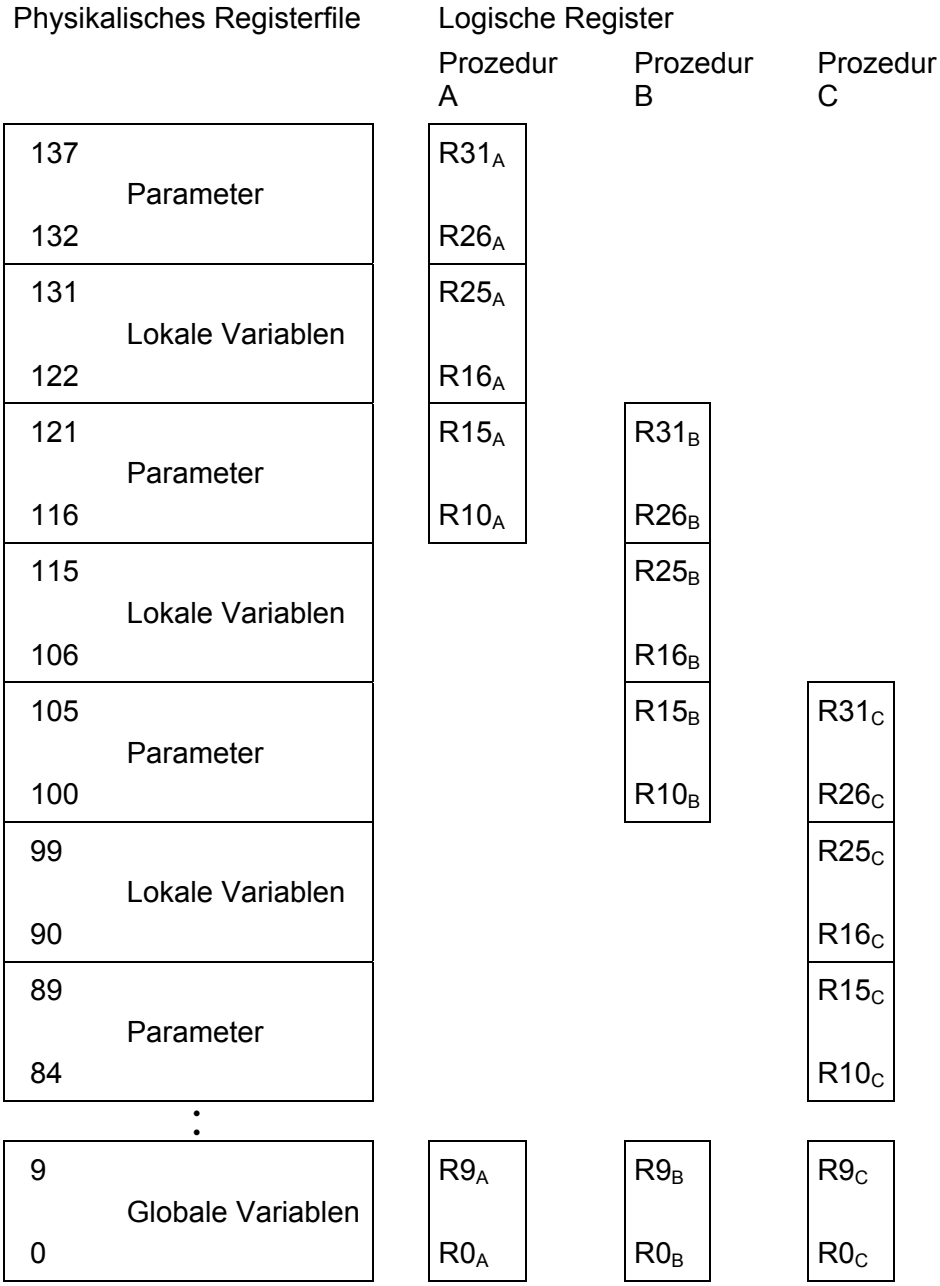
  optimierender Compiler erzeugt Code *zur Kompilierzeit*.

9.5.3 Typische RISC-Techniken

Mehrzweck-Registerdateien (typ. 16-32 Reg.)

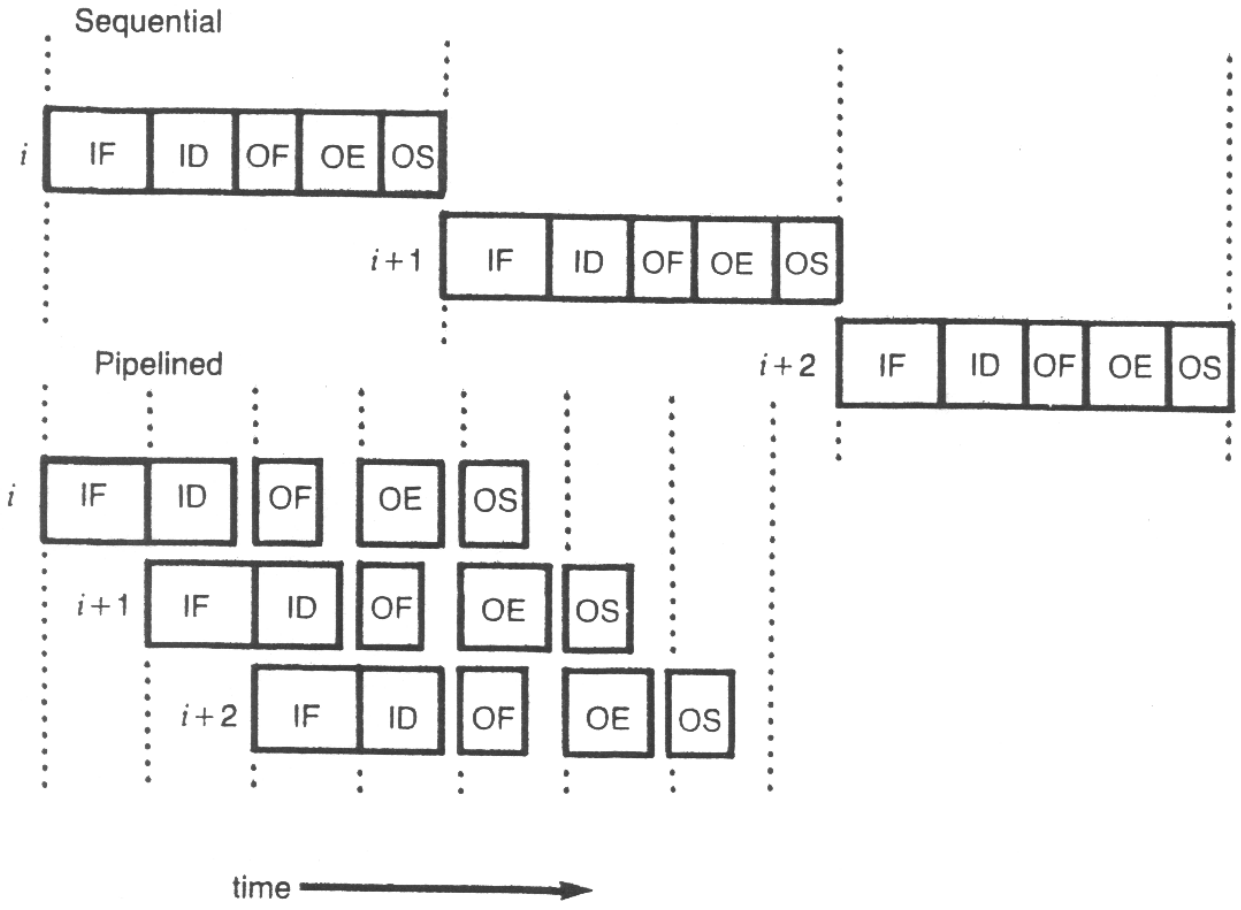
z. B. Berkely RISC II:

138	32-Bit-Register
32	zu jedem Zeitpunkt sichtbar
10	global, 2*6 Parameterübergabe



Bei jedem (geschachtelten) Prozeduraufruf wird das Registerfenster zur Parameterübergabe weiter geschoben. Nur bei einem Überlauf werden Parameter in den Hauptspeicher ausgelagert (→ nur noch 15% Load/Store Befehle).

Phasen-Pipelining



Zerlegen des Befehlszyklusses in mehrere Phasen, z. B.:

IF	Instruction Fetch (Befehl holen)
ID	Instruction Decode (Befehl decodieren)
OF	Operand Fetch (Operand holen)
OE	Operation Execute (Operation ausführen)
OS	Operand Store (Ergebnis abspeichern)

Statt sequentieller Ausführung dieser Phasen Überlappung ähnlich wie bei einem Fließband bei der Auto-Fertigung (Pipelining).

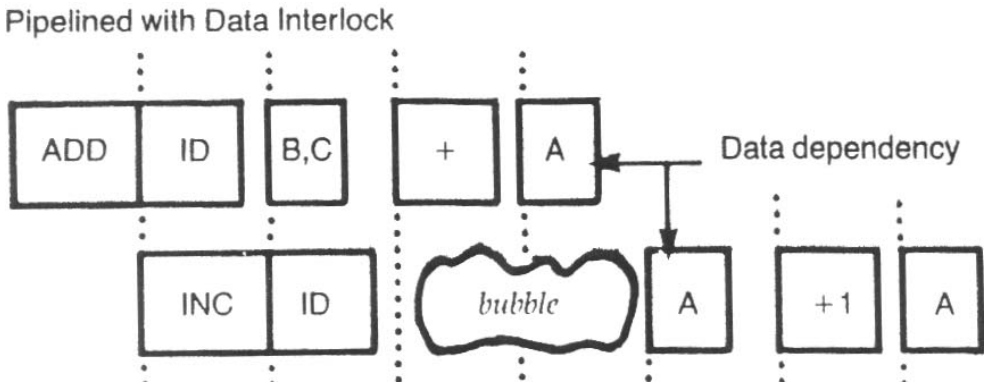
Ein einzelner Befehl wird nicht schneller ausgeführt, aber der *Durchsatz* an Befehlen erhöht sich um maximal die Anzahl der Pipeline-Stufen.

Pipeline-Konflikte

Datenfluss-Konflikt durch Datenabhängigkeiten (Data Interlock):

Beispiel:

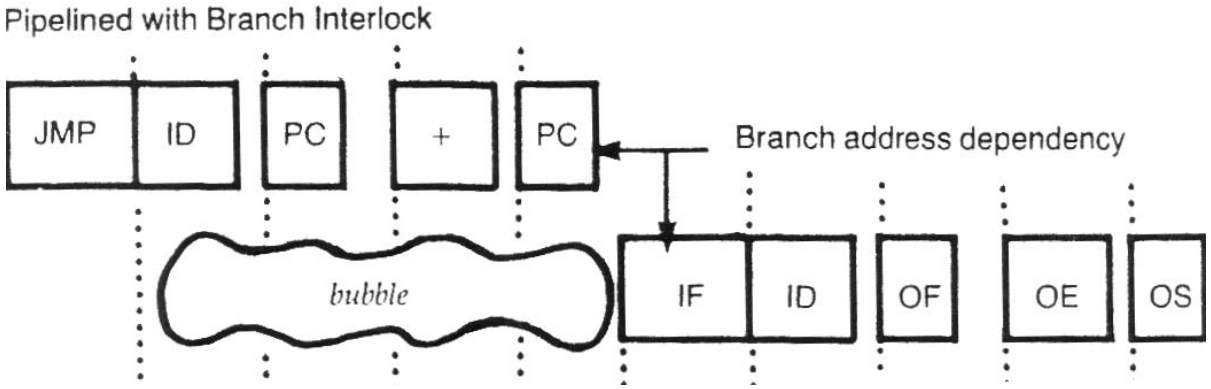
ADD	**A**, B, C	;	A ← B + C
INC	**A**	;	A ← A + 1



OF-Phase vom INC-Befehl kann erst ausgeführt werden, wenn OS-Phase von ADD beendet ist.

⇒ Pipeline bleibt für 2 Phasen leer (Blase).

Steuerfluss-Konflikt bei Verzweigungen (Branch Interlock)

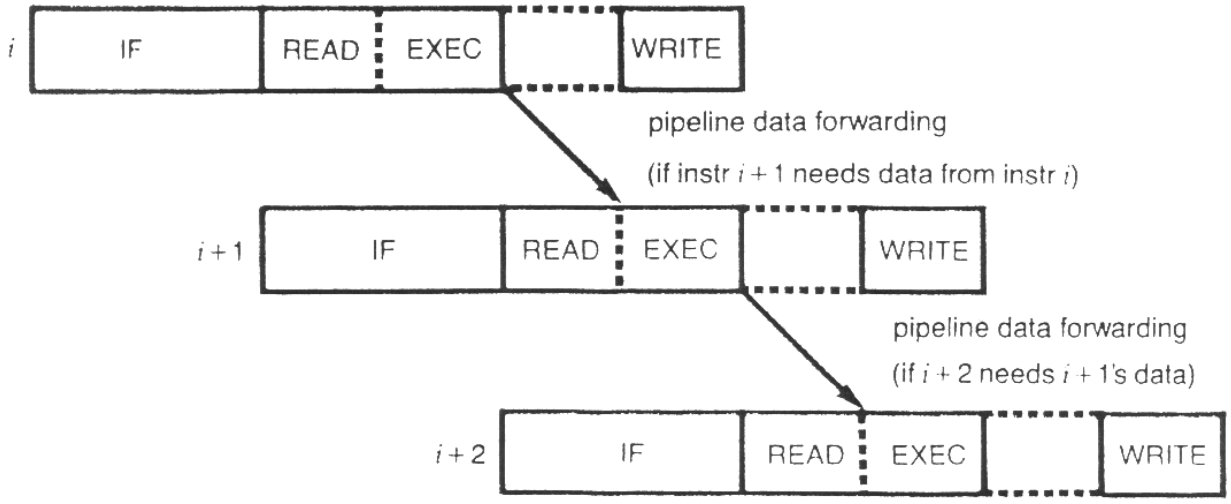


Bei einem Sprungbefehl steht das Sprungziel erst in der Phase OS fest, wenn die Zieladresse in den PC geschrieben wird. Erst dann kann der nächste Befehl von dieser Adresse in die Pipeline geladen werden.

⇒ Es entsteht eine Blase von 4 Phasen.

RISC-Prozessoren: Optimierung von Phasenpipelining

- schnellerer Pipeline-Takt durch einfachen Befehlsaufbau.
- Vermeidung von Datenflusskonflikten durch Compiler (ggf. Befehle umsortieren).
- Data Forwarding (RISC II)



Operand kann vor dem Rückspeichern (WRITE = Operand Store = OS) prozessorintern schon an den nächsten Befehl weitergegeben werden, so dass sich dessen EXEC-Phase mit der WRITE-Phase überlappt und keine Blase mehr entsteht.

- Delayed Load (vgl. Delayed Branch)

  Bei einem Ladebefehl darf nicht sofort im nächsten Befehl auf den geladenen Operanden zugegriffen werden, sondern erst im übernächsten (Blase wird vermieden, dafür müssen Befehle lokal umsortiert oder ggf. ein NOP-Befehl eingeschoben werden).

- **Steuerfluss-Konflikte**

"Delayed Branch": Verzweigung wird erst ab dem übernächsten Befehl wirksam

Beispiel:

Address	Traditional Branch	Delayed Branch	Optimized, Delayed Branch
100	LOAD X, R1	**LOAD X, R1**	**LOAD X, R1**
101	ADD 1, R1	ADD 1, R1	**BRANCH L**
102	**BRANCH L**	**BRANCH L**	ADD 1, R1
103	ADD R1, R2	**NOP**	ADD R1, R2
104	SUB R3, R2	ADD R1, R2	SUB R3, R2
105	L: STORE R1, Y	SUB R3, R2	L: STORE R1, Y
106		L: STORE R1, Y	

Oft kann durch Umordnen der Befehle das 'Loch' nach dem Branch-Befehl durch einen Befehl, der eigentlich vor dem Branch hätte ausgeführt werden sollen, sinnvoll genutzt werden (siehe Optimized Delayed Branch).

Wenn dies nicht möglich ist, muss ein NOP-Befehl eingeschleust werden.

Für Assemblerprogrammierer sehr lästig, für optimierende Compiler kein Problem.

Delayed Load: analog für Ladebefehle.

Laut Messungen ist der Wirkungsgrad von Delayed Branch bzw. Delayed Load ca. 90%, d. h. nur in 10% der Fälle muss ein NOP-Befehl eingefügt werden!

9.5.4 Vor- und Nachteile von RISC

+ ca. 2-5 mal **schneller** als CISC bei gleicher Technologie durch:

  - höheren Takt durch einfache Befehle
  - optimiertes Pipelining
  - viele Register

  Vorsicht: RISC-MIPS ≠ CISC MIPS
  (MIPS: **M**illion **I**nstructions **P**er **S**econd)

+ kürzerer VLSI-Entwicklungszyklus
  (einfachere, reguläre Hardware-Struktur)

+ günstiges Preis/Leistungsverhältnis

- komplexere, optimierende Compiler
  (Aufwand, Compilierzeit)

- ca. 50-70 % längerer Maschinencode

- keine Kompatibilität zu bisherigen (CISC-)Rechnern auf Maschinenprogrammebene

- aufwändigere Assemblerprogrammierung

Derzeitiger Stand:

RISC hat sich durchgesetzt.

Einsatz in PCs, Workstations, Parallelrechnern, zunehmend auch Mikrocontrollern.

Moderne CISC-Prozessoren haben mehr und mehr RISC-Techniken übernommen (z. B. Intel Pentium-Familie).

RISC-Prozessoren sind besonders geeignet für die Implementierung mit neuen Technologien (z. B. Galliumarsenid).

9.6 Spezielle Prozessorarchitekturen

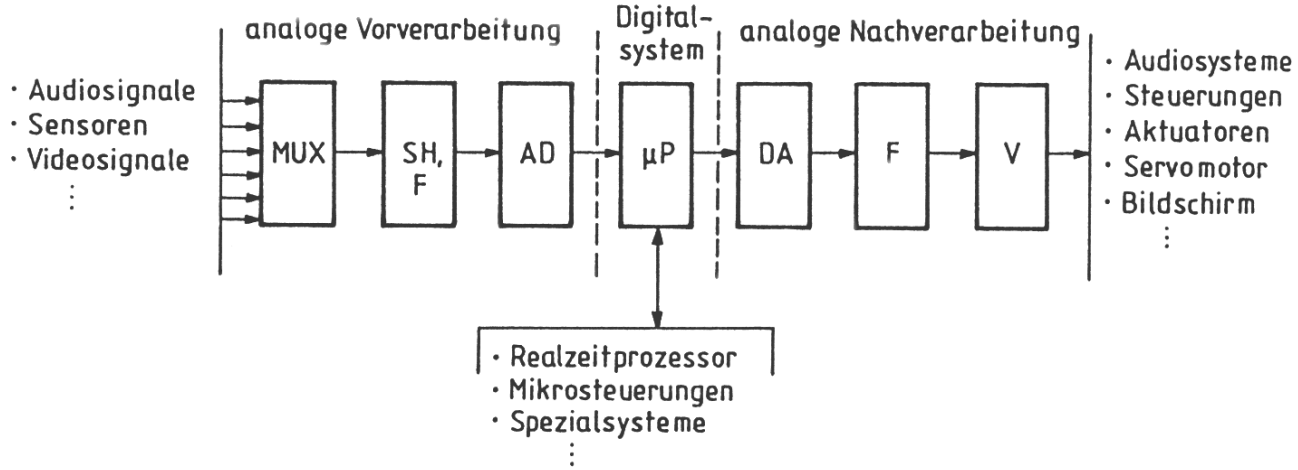
9.6.1 Digitale Signalprozessoren

Signalprozessoren[1] (DSP - Digitale Signalprozessoren) sind Prozessoren, die für die Datenstrom-Verarbeitung analoger Signale bzw. Signalflüsse unter sehr harten Echtzeitanforderungen dienen.

Typische Anwendungsfelder sind:

Signalverarbeitung wie Filterung, Rauschunterdrückung, Frequenzanalyse, De-/Kompression in der Spracherkennung, Telekommunikation, Bildverarbeitung, Nachrichtentechnik, Konsumerelektronik, Computerindustrie (z. B. Laserdrucker), Multimedia, Automobilelektronik (z. B. aktive Fahrwerke, ABS), Navigationssysteme, Industrie (Messwertverarbeitung, Robotersteuerung, Steuer- und Regelungstechnik), Medizintechnik (z. B. EKG, Ultraschall, CT, MRT), Militär (Radar, Sonar, Geschosslenkung, Kommunikationsabsicherung), digitales Radio usw.

Typischer Aufbau eines Signalverarbeitungssystems:

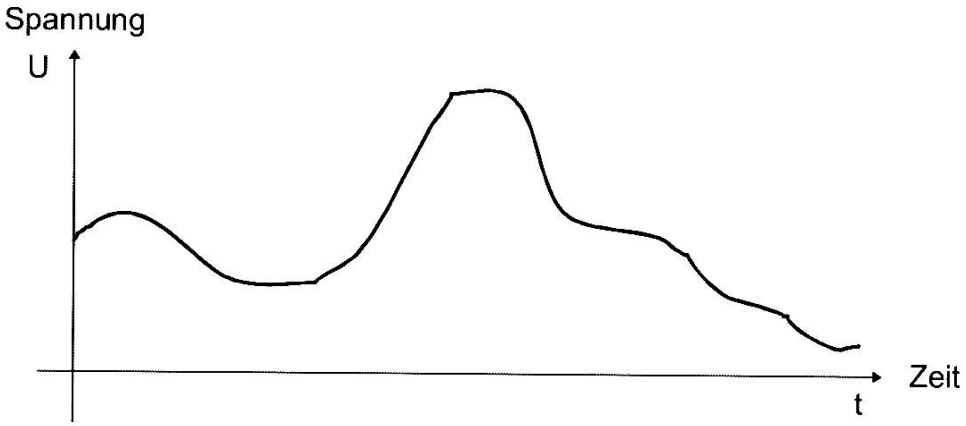


mit: MUX : Analog-Multiplexer, DA : DA-Wandler
SH : Abtastschaltung, F : Filter
AD : AD-Wandler, V : Verstärker
µP : Mikroprozessor, Signalprozessor

[1] Literatur z.B.: Bähring, H.: Mikrorechner – Band I: Mikroprozessoren und Digitale Signalprozessoren. Springer, Berlin, Heidelberg, New York, 2002

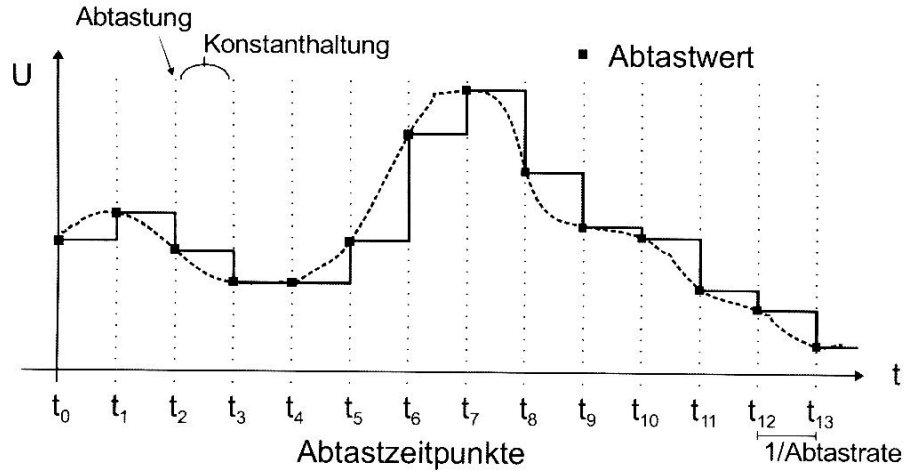
Verarbeitungsstufen eines analogen Signals

Analoges Signal: zeit- und wertkontinuierlich



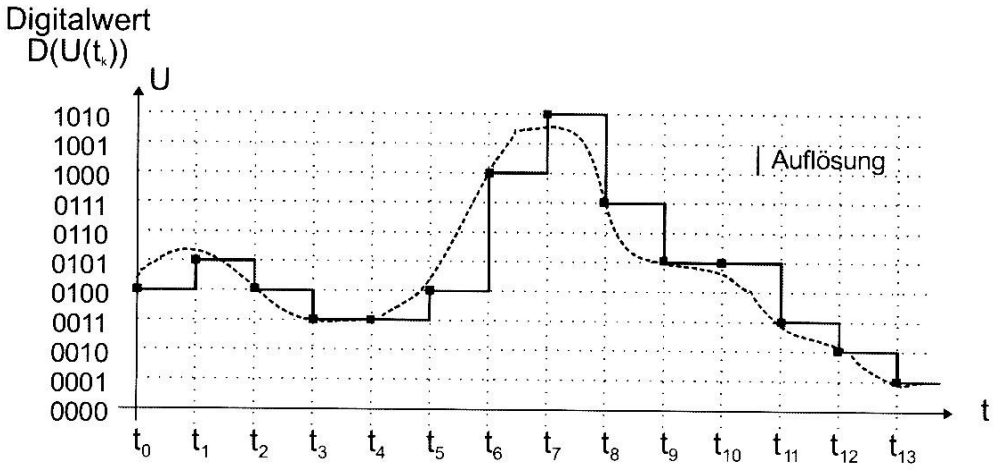
Abtastung:

zeitliche Quantisierung: wertkontinuierlich und zeitdiskret



AD-Wandlung:

Amplituden-Quantisierung: wert- und zeitdiskret



- Die *Abtastung* führt zu einer **Zeitdiskretisierung**.

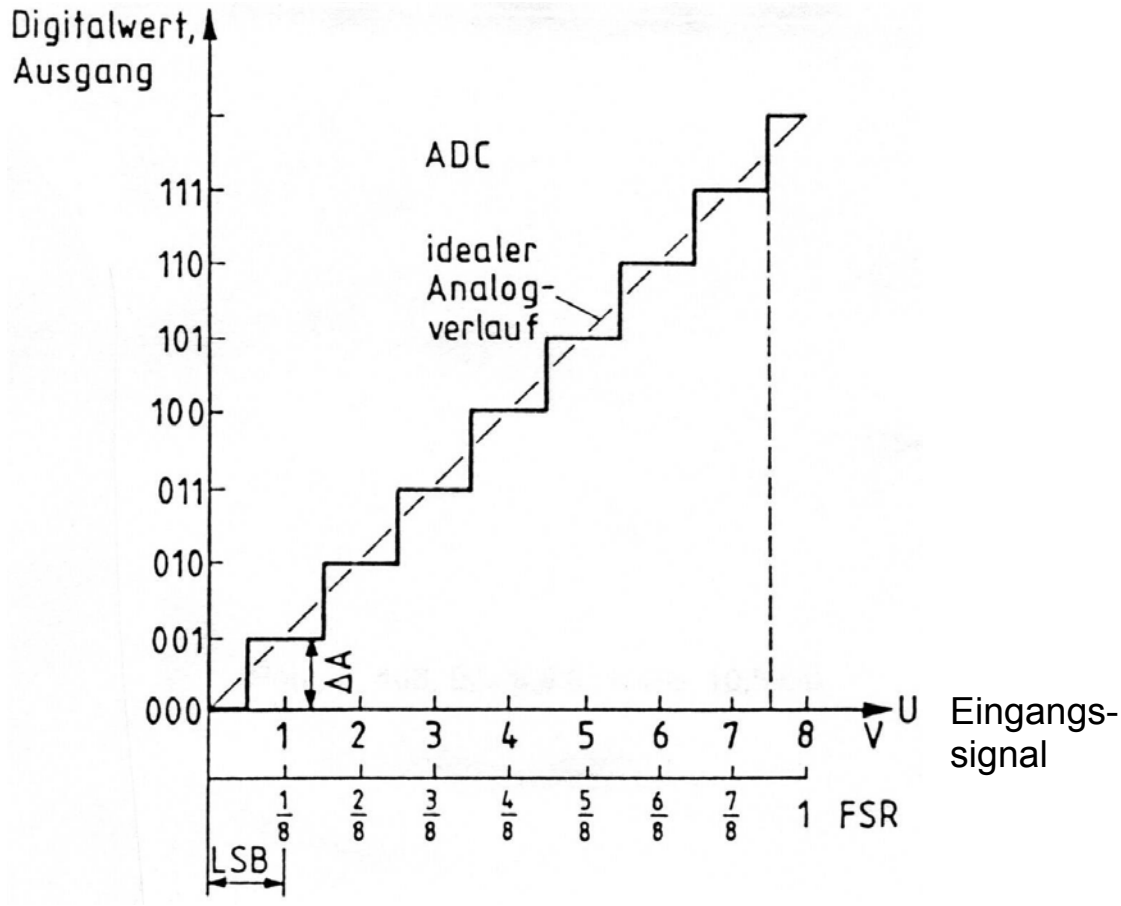
  Die *Abtastrate* f_A richtet sich nach der höchsten zu verarbeitenden Frequenz f_{max} des analogen Signals.

  $$\boxed{f_A > 2 \cdot f_{\max}}$$ (Shannon´sches Abtasttheorem)

- Die *Digitalisierung* bewirkt eine **Wertediskretisierung** (Quantisierung). Dabei ist jedem diskreten Amplitudenniveaus ein *Digitalwert* zugeordnet.

  Die *Auflösung* entspricht dem kleinsten im Digitalwort zu erkennenden Signalunterschied des analogen Signals. Sie ist i. Allg. konstant über den gesamten Wertebereich. Die Wandlerkennlinie ist also im Idealfall linear.

  <u>Wandlerkennlinie:</u>



Typische Signalverarbeitungsalgorithmen sind Fensterung, Fourierreihen-Entwicklung, diskrete Fourier- und Cosinus-transformation, Filterung.

Beispiel: Digitale Filter

FIR-Filter (Finite Impulse Response)

$$y(n) = \sum_i c_i \cdot x(n-i)$$

IIR-Filter (Infinite Impulse Response)

$$y(n) = \sum_i c_i \cdot x(n-i) + \sum_i d_i \cdot y(n-i)$$

mit c_i, d_i: applikationsspezifische Filterkoeffizienten

Die meisten Signalverarbeitungsalgorithmen basieren auf folgendem Teilalgorithmus:

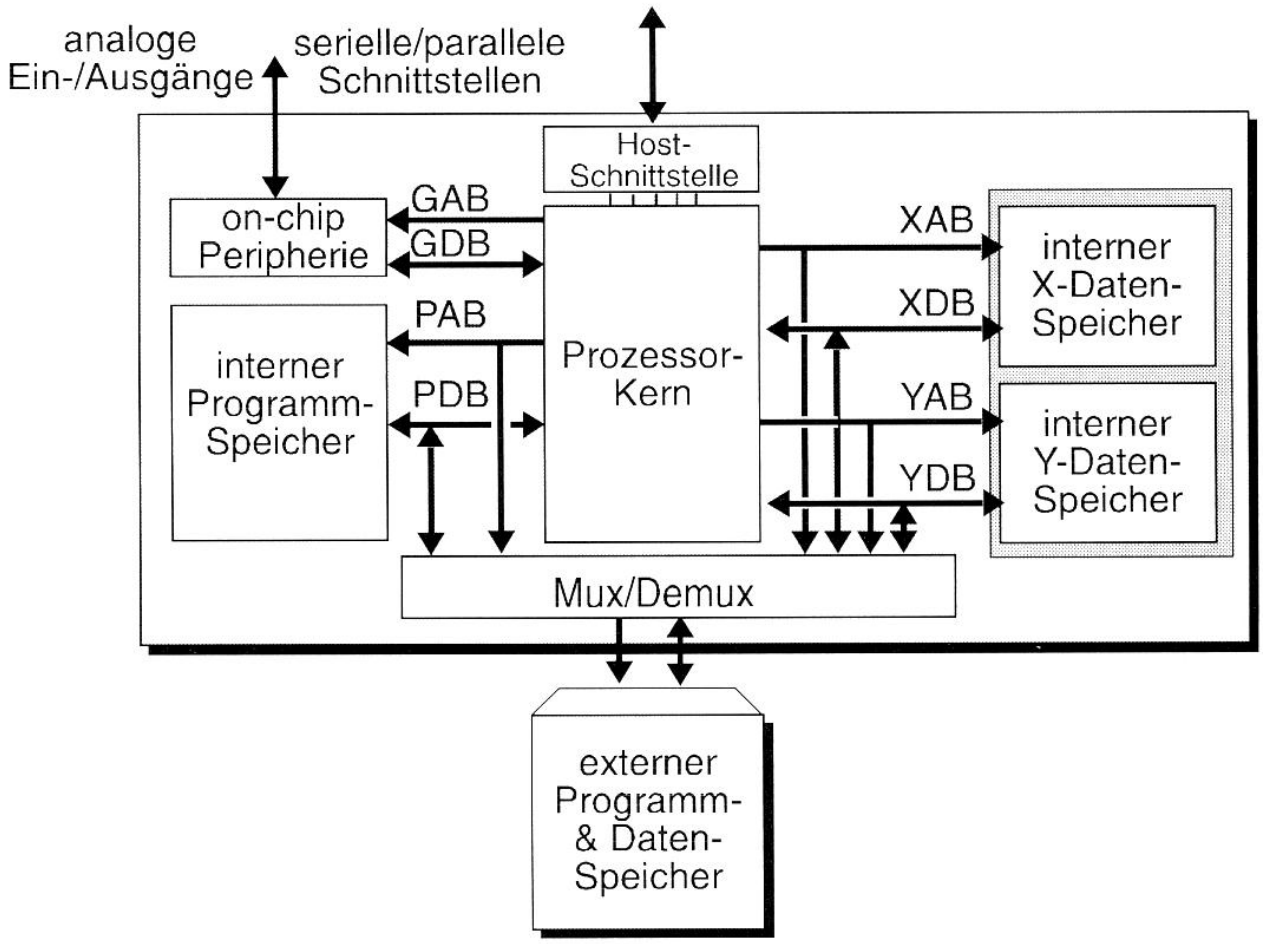
$$S(n) = a(n) \cdot b(n) + S(n-1),$$

wobei $a(n)$ und $b(n)$ Faktoren im Zyklus n und $S(n\text{-}1)$ die Teilsumme aus dem vorangegangenen Zyklus sind.

Daraus leiten sich folgende Anforderungen ab:

- Multiplikation und Addition möglichst in einem Takt
- effizienter Transport von mindestens 2 Operanden ($a(n)$, $b(n)$) und Ergebnis ($S(n)$)

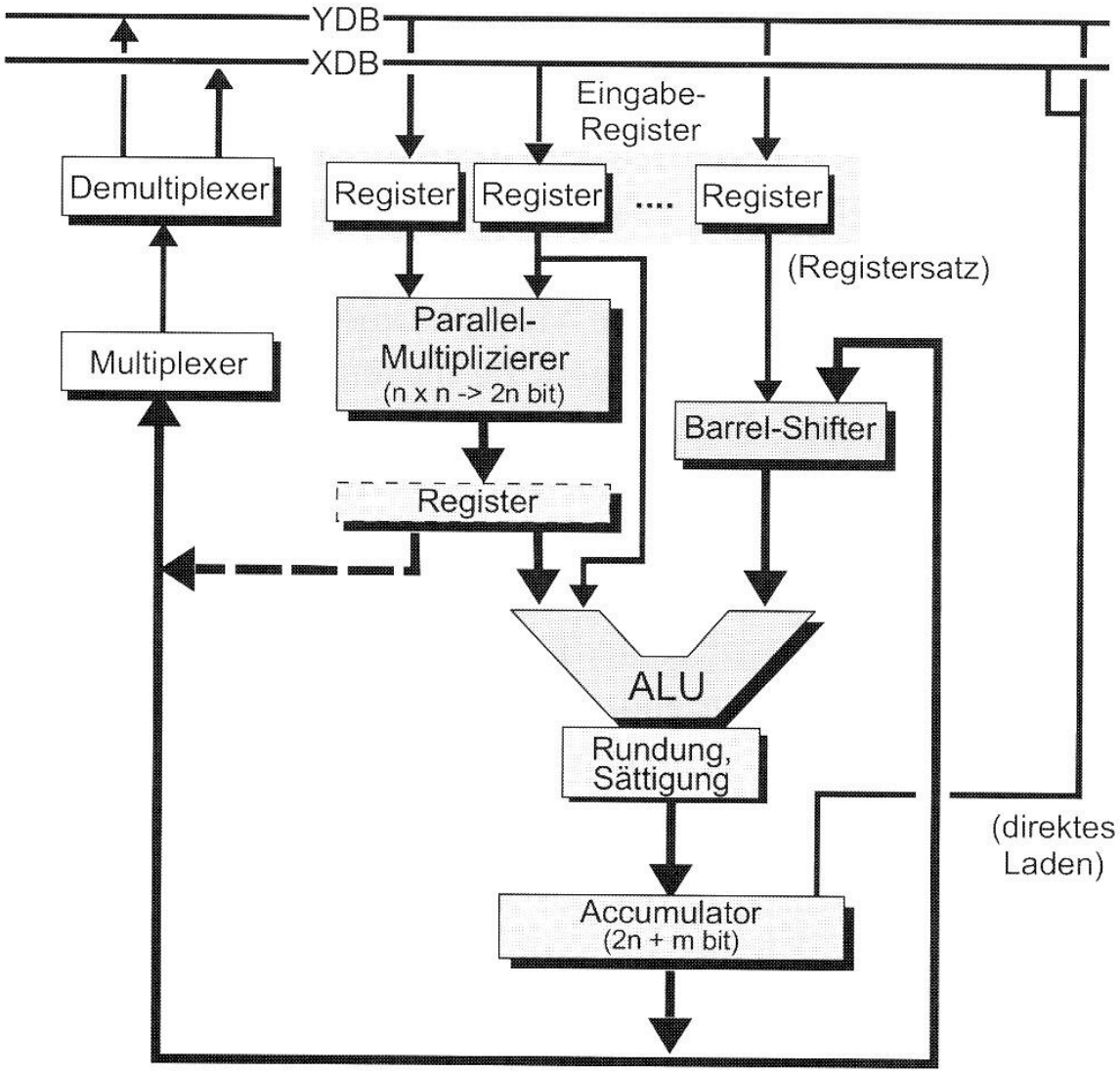
Grundarchitektur eines Signalprozessors:



Die typische Architektur moderner DSPs besitzt:

- Multiplizierer-Akkumulator-Einheit (MAC)
- Harvard-Architektur mit mindestens zwei (bis zu 5) Datenbussen und oft auch zwei (oder mehr) Datenspeichern
- oft mehrere Adressrechenwerke
- spezielle Adressierungsarten und Befehle

Struktur eines DSP-Rechenwerkes:



Je nach DSP wird Integer-, Festpunkt-, Fließkomma-Arithmetik eingesetzt.

9.6.2 Parallelverarbeitung

Klassifikation nach Flynn

SISD Single Instruction, Single Data	**SIMD** Single Instruction, Multiple Data
MISD Multiple Instruction, Single Data	**MIMD** Multiple Instruction, Multiple Data

- SISD: z.B. Mikrocontroller, Desktop-Rechner, DSPs

- SIMD: z.B. Vektorrechner, Feldrechner, Systolische Felder

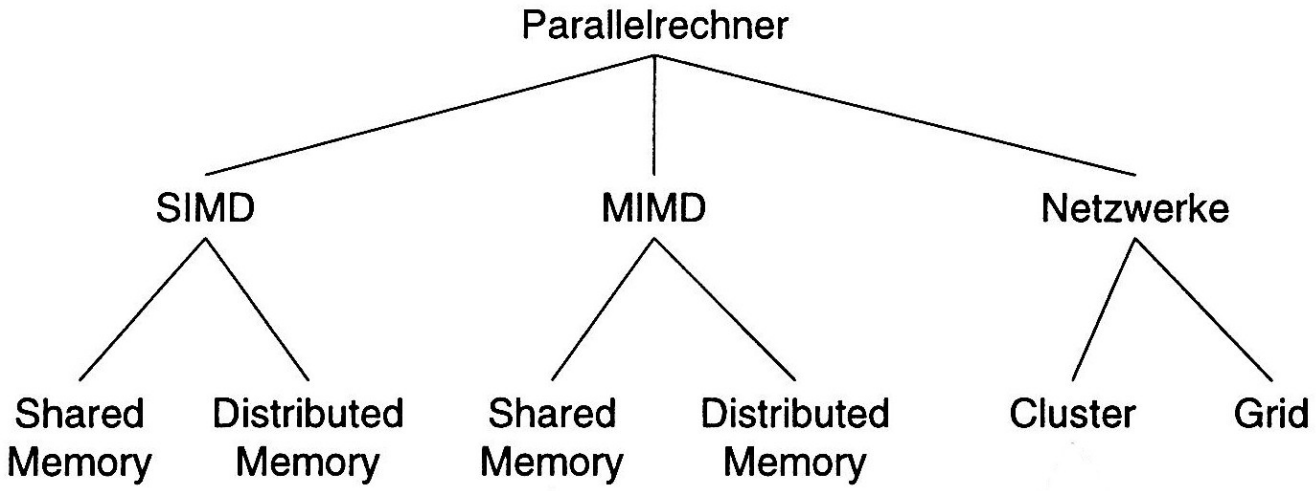
- MIMD: z.B. Multiprozessoren, Multicore-CPUs, Rechencluster

- MISD: praktisch nicht gebräuchlich

Parallelrechner

Das Ziel von Parallelrechnern ist es, die Verarbeitung größerer Aufgaben wie Wetter- oder Crashsimulationen zu beschleunigen, indem die Aufgaben so in kleinere Teilaufgaben zerlegt werden, dass diese parallel auf mehreren Recheneinheiten bearbeitet werden können.

Klassifikation von Parallelrechnern

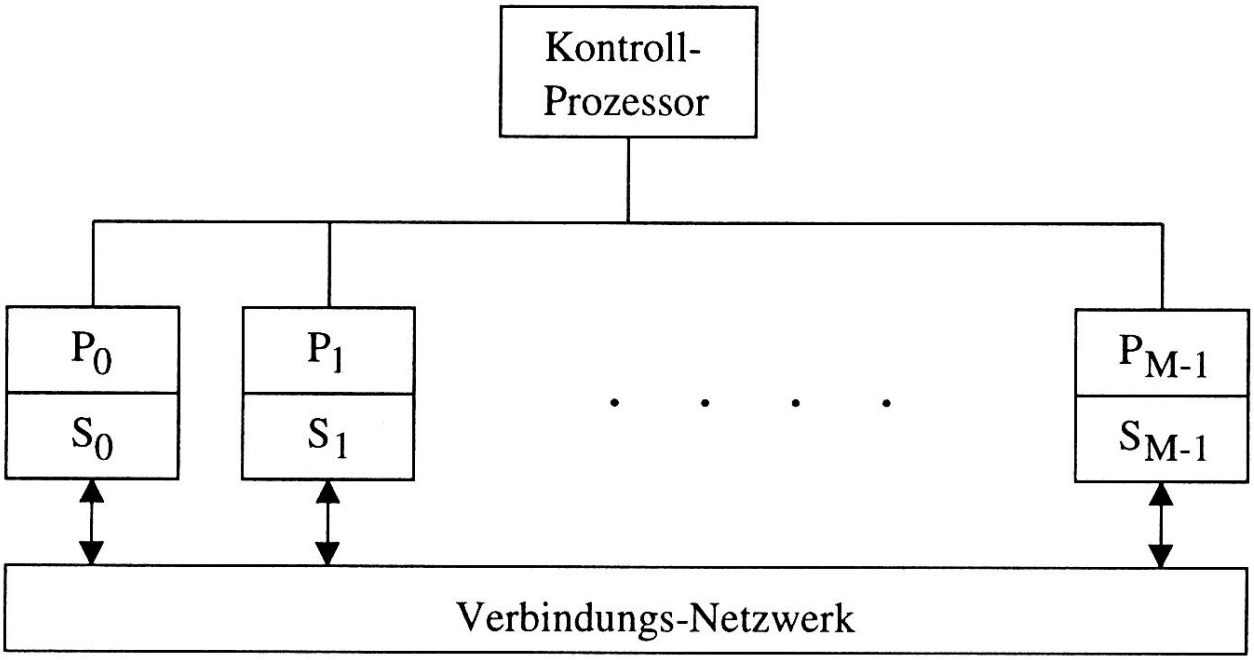


Die historisch ersten Modelle von Parallelrechnern waren SIMD-Modelle, d. h. einzelne Instruktionen werden gleichzeitig auf viele Daten angewendet.

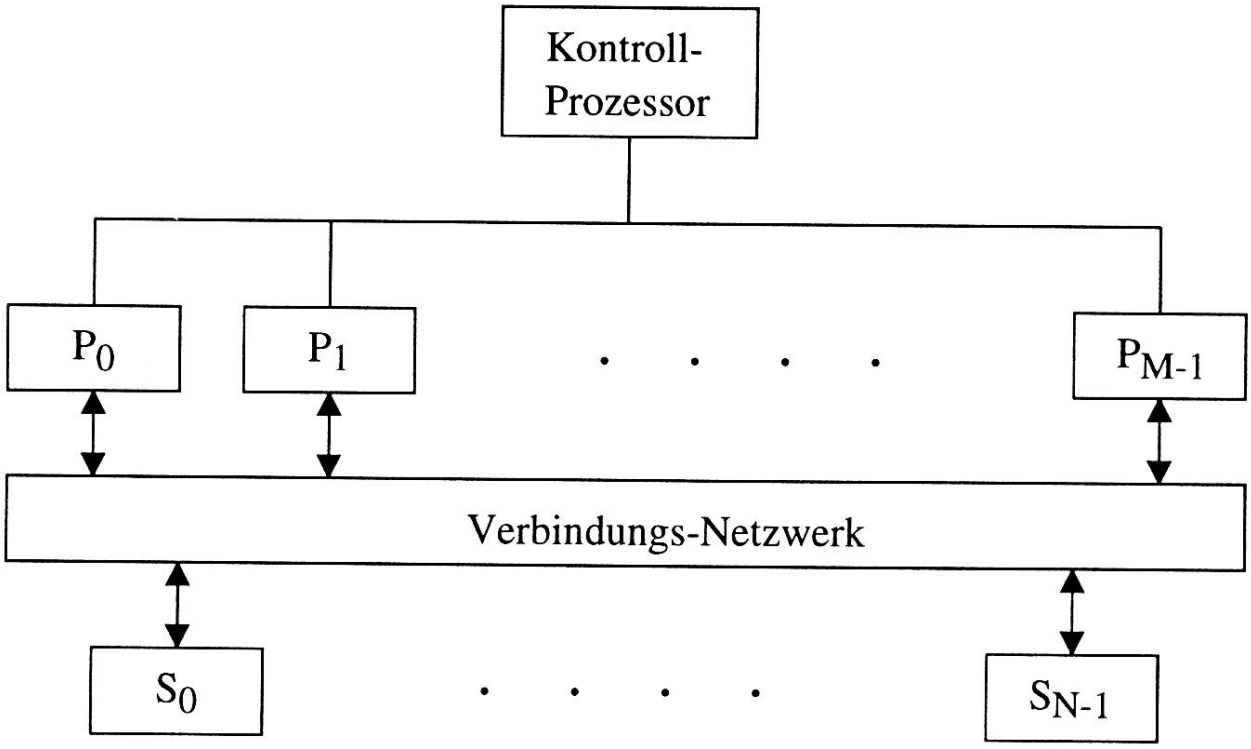
Die bekannteste SIMD-Parallelrechnerarchitektur sind *Feldrechner* (Array Computer) mit einem Feld von identischen Verarbeitungseinheiten (Processing Elements P), die alle mindestens über eine eigene ALU verfügen und zentral durch einen Kontrollprozessor gesteuert werden.

Die Verarbeitungseinheiten können linear oder matrixförmig angeordnet sein.

Feldrechner mit lokalen Speichern (Distributed Memory)



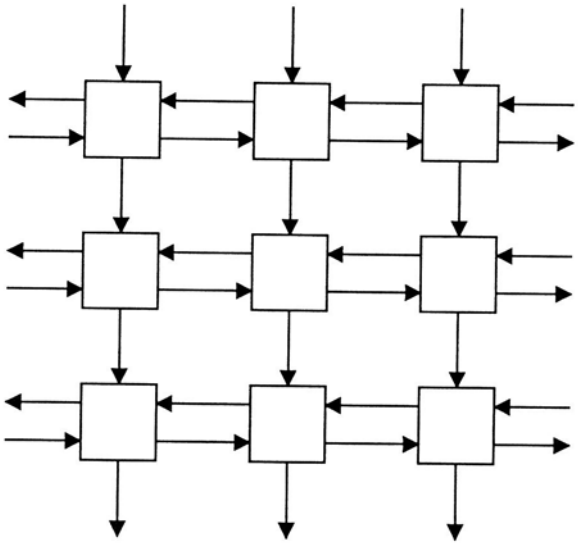
Feldrechner mit globalem Speicher (Shared Memory)



Systolische Felder

Systolische Felder sind typischerweise SIMD-Rechner.

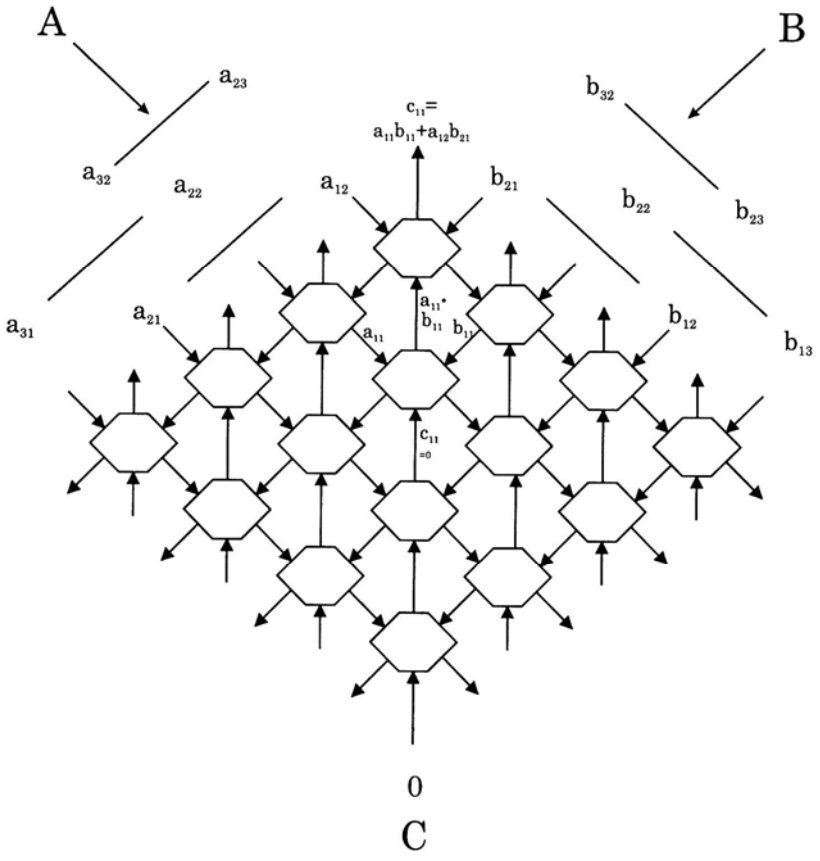
Das Modell eines systolischen Feldes geht von einem Datenstrom aus, der wie eine Welle analog zum systolischen Druck im Blutkreislauf durch ein Feld von einfachen, gleichen Verarbeitungseinheiten fließt.



Typische Anwendungen sind Suchen, Vergleichen, Sortieren, Pattern Matching, Matrix- und Datenbank-Operationen, Bildverarbeitung

Systolische Felder sind aufgrund ihrer regulären Struktur aus vielen relativ einfachen Elementen mit lokalem Datenaustausch gut für eine VLSI-Implementierung geeignet.

Beispiel: Systolisches Feld zur Matrix-Multiplikation



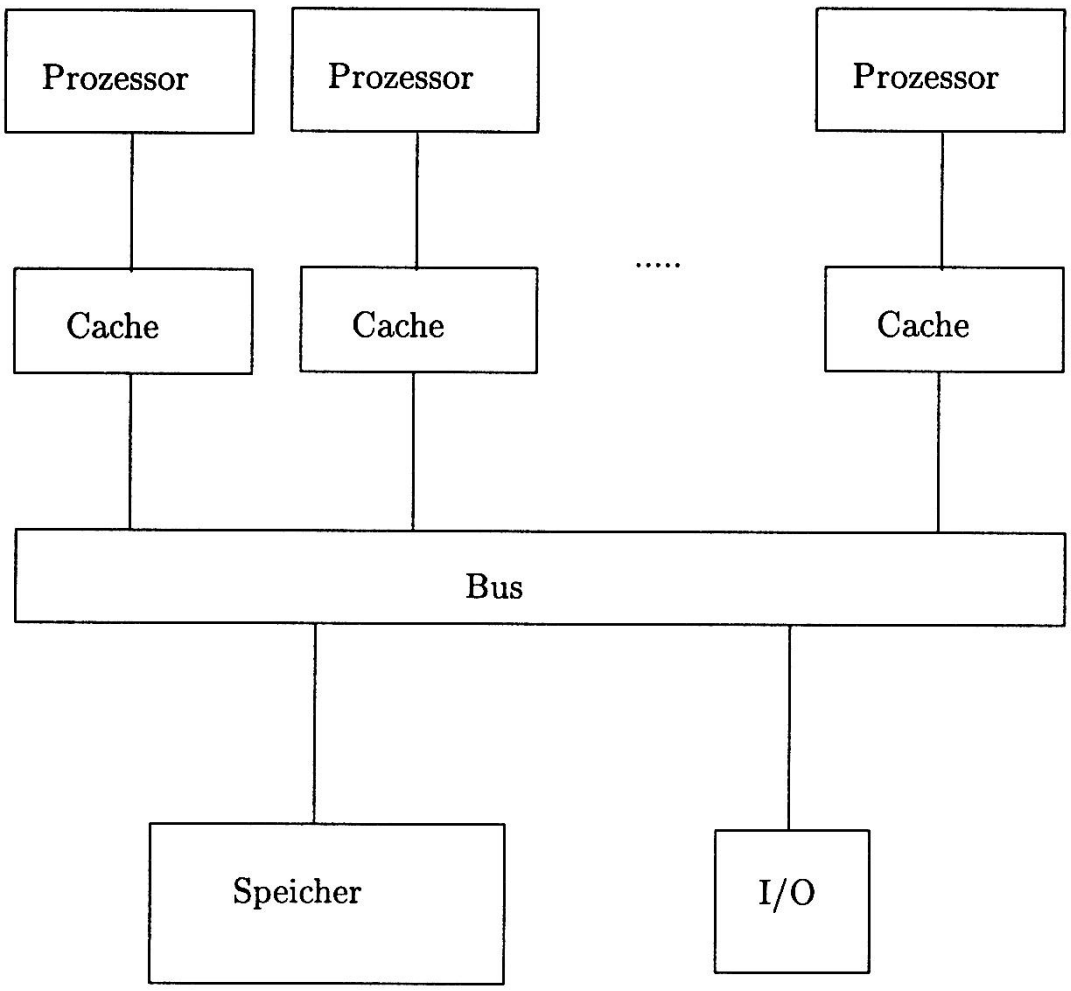
MIMD-Rechner

Die zentrale Idee der MIMD-Rechner ist es, durch Zusammenschalten mehrerer (Standard-)Rechner bzw. Prozessoren, die jeweils eigene (Teil-)Programme koordiniert abarbeiten, einen neuen, leistungsfähigeren Rechner zu schaffen.

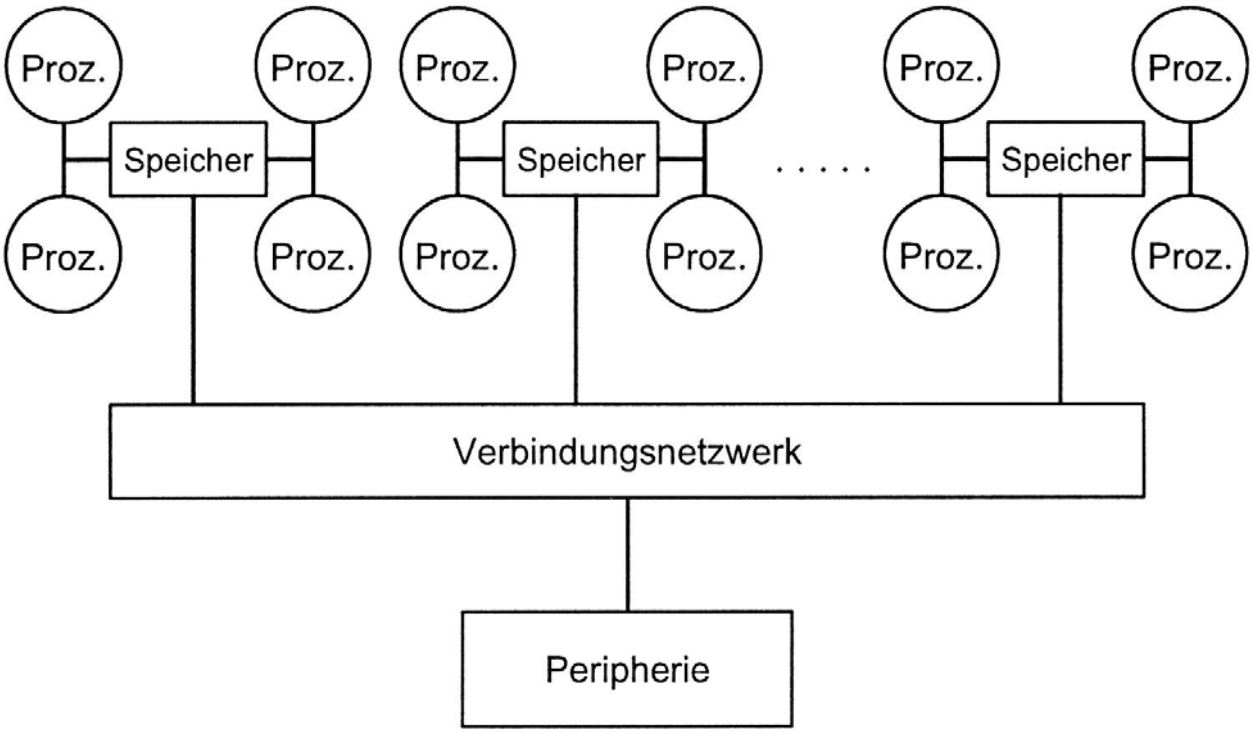
MIMD-Rechner werden unterschieden je nach dem

- wie auf gemeinsame Daten zugegriffen wird,
- wie die Parallelarbeit der einzelnen Prozessoren koordiniert wird.

MIMD-Rechnerstruktur mit einem Bus



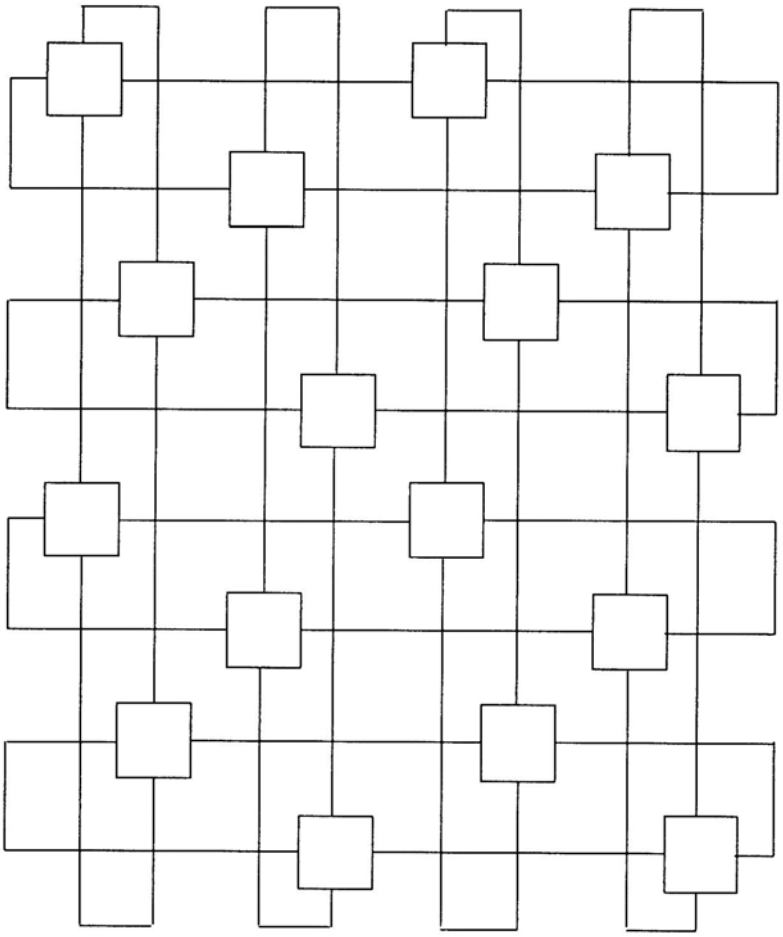
MIMD-Rechnerstruktur aus Prozessor-Clustern mit gemeinsamen Speicher



Beim Zugriff auf gemeinsam genutzte Daten werden speichergekoppelte (shared memory) und nachrichtengekoppelte (message passing) Systeme unterschieden.

Eine weitere Unterscheidung von Parallelrechnern erfolgt nach der verwendeten Verbindungsstruktur, z. B. Bussysteme und Punkt-zu-Punkt-Verbindungen, bzw. der Verbindungstopologie.

MIMD-Rechnerstruktur mit 4*4-Torus-Verbindung



Parallelrechner sind meist für einen bestimmten Zweck entworfen, oft sogar für eine spezielle Anwendung. Das betrifft sowohl den Aufbau der einzelnen Verarbeitungseinheiten als auch die Speicher- und Kommunikationsarchitektur usw.

Aus Kostengründen wird oft versucht, dabei weitestgehend Standardkomponenten einzusetzen, z.B. *Cluster* aus Standard-PCs, die über ein spezielles Verbindungsnetzwerk zu einem Parallelrechner zusammengeschaltet werden (*Cluster Computing*).

Die Software läuft i.d.R. unter einer zentralen Kontrolle (z. B. Farmer-Worker-Prinzip).

Aktuelle Entwicklungen

Grid Computing

Ein *Grid* stellt für die Bearbeitung einer Anwendung (oder im Multi-User-Betrieb auch mehreren Anwendungen) ein paralleles, verteiltes System für die gemeinsame Nutzung, Auswahl und Aggregation von Ressourcen (Rechenknoten, Speicher, Platten …) zur Verfügung.

Beim *Grid-Computing* wird also ein Supercomputer auf dem virtuellen Zusammenschluss (lose gekoppelter) einfacherer Rechnerressourcen emuliert.

Die einzelnen Rechnereinheiten stehen entweder räumlich konzentriert in einem Rechenzentrum. Oder es können auch die Arbeitsplatzrechner einer Firma zu einem Grid zusammengeschaltet werden. Die Rechenknoten müssen bei einem Grid nicht mehr gleich sein. Sie können sogar aus unterschiedlichen administrativen Bereichen stammen.

Die Zuordnung zu einer Anwendung geschieht automatisch und hängt von der aktuellen Verfügbarkeit, Kapazität und Leistung der einzelnen Komponenten ab.

→ heterogene und räumlich verteilte Rechenknoten
unter einer einheitlichen Oberfläche
mit standardisierten Schnittstellen und Protokollen
Nutzerverwaltung mit
- geschützter Nutzung der Ressourcen
- häufig nutzungsabhängige Verrechnung der in Anspruch genommenen Ressourcen

→ erlaubt Bereitstellung großer Rechenleistung bzw.
flexible Auslastung eines großen Netzwerks von Recheneinheiten

Cloud Computing

Cloud Computing abstrahiert noch stärker von der unterlagerten Hardware. Durch die global verfügbare Bereitstellung bzw. Nutzung von Diensten ist es für den Anwender völlig transparent, wo seine Anwendung tatsächlich ausgeführt wird bzw. wo seine Daten liegen.

→ **Service-orientierte Verarbeitung**

Ressourcen werden als nutzbare Dienste zur Verfügung gestellt und tatsächlich dort ausgeführt, wo gerade die angefragten Ressourcen verfügbar sind. Das kann sogar in anderen Ländern sein.

→ vollständige **Virtualisierung**

Vorteile:

- Virtualisiertes Rechenzentrum mit anscheinend unendlichen Ressourcen
- Der Nutzer bezahlt nur die Kosten für die tatsächlich beanspruchte Leistung → Kostenvorteile
- Der Betreiber eines Rechnerpools kann seine Infrastruktur wegen der flexiblen Zuweisung von Diensten optimal ausnutzen → Kostenvorteile
- Der Nutzer braucht sich nicht um die Anwenderprogramme kümmern (z.B. Updates, …)
- Der Nutzer braucht sich nicht um Backups kümmern

Probleme:

- Abhängigkeit von einem anonymen Rechnerpool-Betreiber
- eingeschränkte Privatsphäre durch Ausführung und Speicherung an einem unbekannten Ort
- Transfer über Netzwerk bietet viel Angriffspotential

Stand aktueller CPU-Technik (2011)

Die Fortschritte in der Mikroelektronik erlauben heute

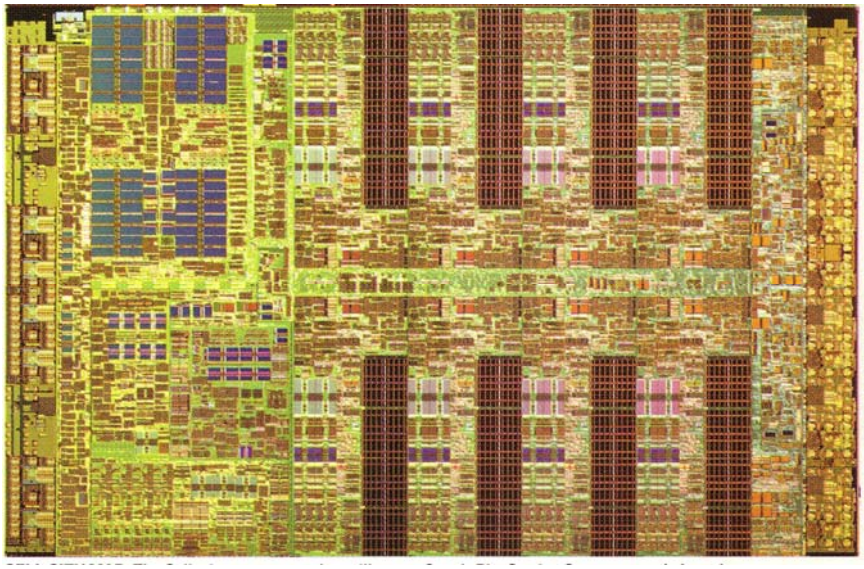
- mehrere Milliarden Transistoren auf einem Chip, z.B. Intel Paulson Itanium Prozessor 3,1 mia. T. auf einem 18*30 mm^2 großen Chip (**8** *x86*-Cores mit 4 Shared Caches und 128 GBytes/s processor-to-processor-Kommunikation)
- mehrere Cores auf einem Chip, z.B.
  - Intel Xeon in 32nm-CMOS-Technologie **10** *x86*-Cores (mit individueller Steuerung der Leistungsaufnahme)

    → *Multi-Core*
  - **64** nicht-*x86*-Cores auf einem Chip der Tilera Corp.

    → *Many-Core*

Neben diesen Prozessoren mit mehreren gleichen Cores und einer Hochgeschwindigkeitsvernetzung auf einem Chip werden auch zunehmend heterogene Recheneinheiten auf einem Chip untergebracht, vorzugsweise für die *datenparallele* Verarbeitung (Grafik, Multimedia, …).

Beispiel: Cell-Mikroprozessor mit einem 64 Bit-Hochleistungsmikroprozessor (oben links) und acht Multimedia-Cores mit je 4 Daten parallel in einer Instruktion



(aus: IEEE Spectrum, Jan. 2006)