

## 6 Zentraleinheit

### 6.1 Ein einfacher Rechner

### 6.2 Die Programmierschnittstelle

### 6.3 Datentypen

### 6.4 "Endian"-Modelle

### 6.5 Befehlssätze

### 6.6 Beispiel: Intels MMX-Erweiterung

### 6.7 Befehlsformate

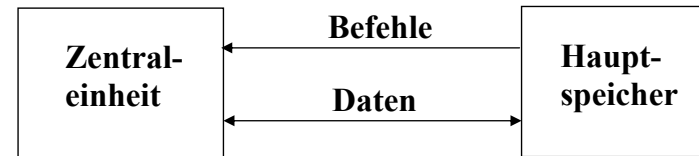
### 6.8 Befehlskodierung

### 6.9 Adressierungsmodi

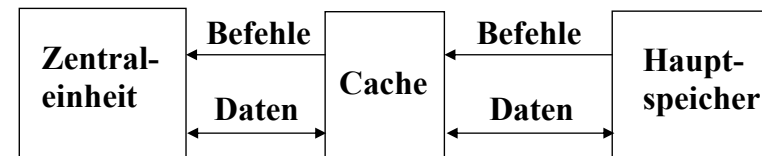
### 6.10 Befehls-Pipelining

### 6.11 Sprungbehandlung

#### Einfacher Rechner:



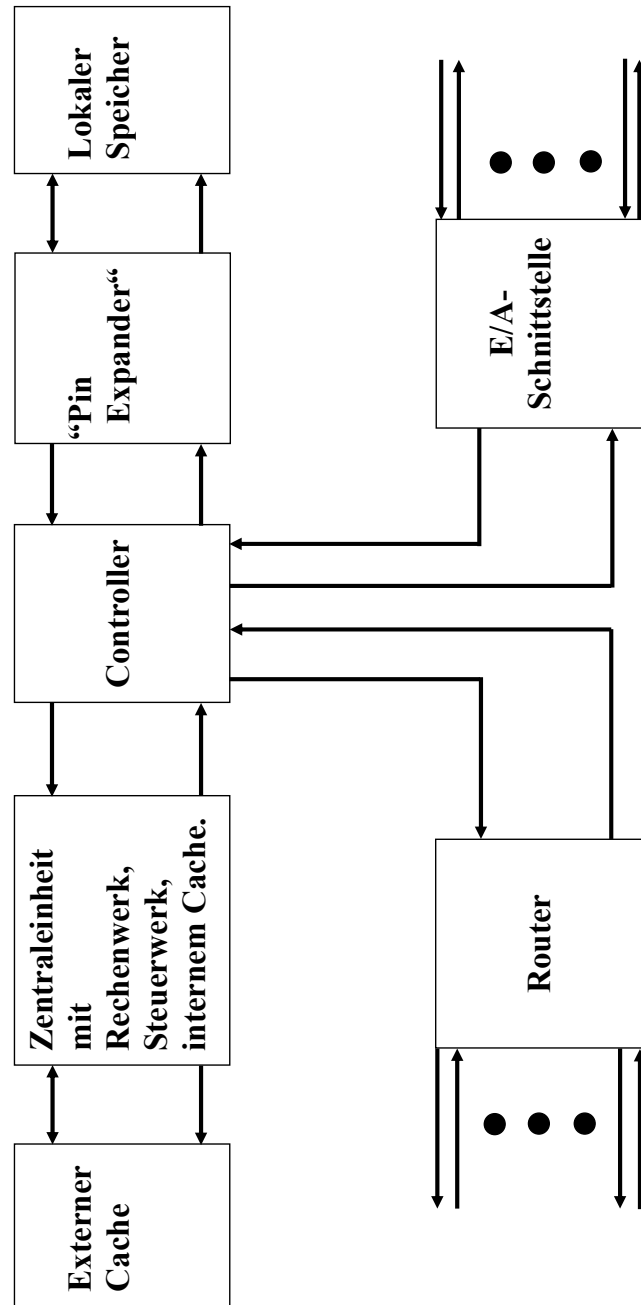
#### Rechner mit Cache:



#### Arbeitsweise der Zentraleinheit:

```
Solange ausführbare Befehle warten
  Hole nächsten Befehl
  Führe Befehl aus
  Falls Unterbrechungen anstehen
    behandle eine Unterbrechung
  Ende
Ende
```

**Knoten eines Parallelrechners:**



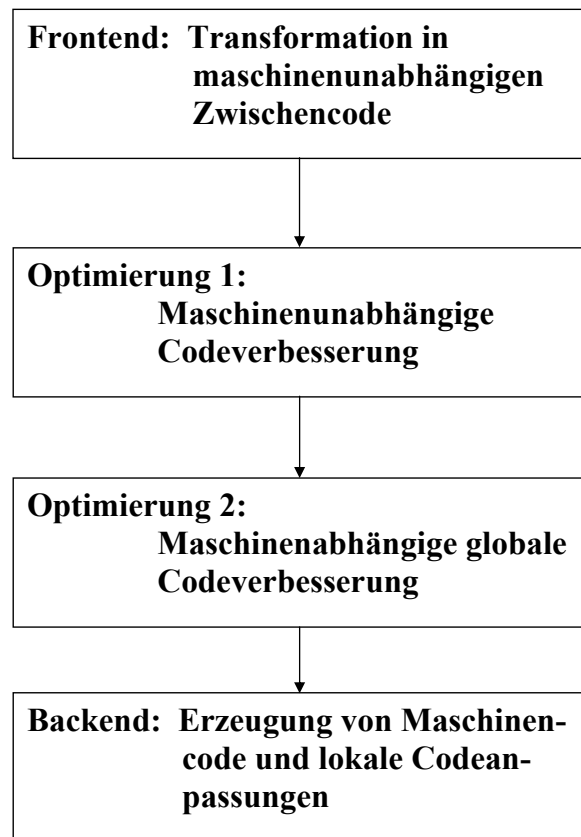
**Beschreibung der Programmierschnittstelle eines Rechners:**

**Ein Rechner wird charakterisiert durch:**

- (i) **Rechnerklasse**  
(Akkumulator-Maschine, Register-Maschine, "load/store"-Architektur, Kellermaschine),
- (ii) **Adressierung der Hauptspeicherzellen,**
- (iii) **Befehlssatz,**
- (iv) **Zahl, Typ und Größe der Datenformate,**
- (v) **Codierung der Befehle,**
- (vi) **Ausrichtung von Befehlen und Daten,**
- (vii) **Unterbrechungsstruktur,**
- (viii) **unterstützende Software.**

## Beispiel zu Merkmal (viii): Compiler

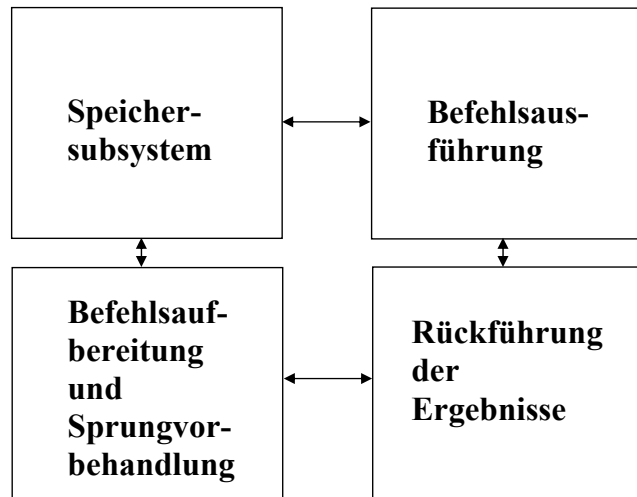
**Ziel:** Rechnerarchitekturen sollten so gewählt werden, daß man mit Standardverfahren Hochsprachenprogramme ohne großen Aufwand in effiziente Maschinenprogramme übersetzen kann.



## Bemerkungen:

- (i) Die obigen Merkmale eines Rechners faßt man unter dem Begriff der "Instruction Set Architecture" zusammen. Man spricht von der ISA-Ebene eines Computers. Sie stellt die Benutzungsschnittstelle für den Programmierer dar.
- (ii) Eine Grobklassifizierung der Prozessoren erfolgt nach der Größe und Komplexität des Befehlssatzes. Man unterscheidet zwischen RISC- und CISC-Architekturen. Hierbei steht RISC für "reduced instruction set computer" und CISC für "complex instruction set computer".
- (iii) Je nachdem, ob man für Daten und Befehle getrennte Speicherbereiche verwendet oder nicht, unterscheidet man zwischen Princeton- und Harvard-Architekturen. Häufig verwendet man für Daten und Befehle getrennte Cache-Speicher.
- (iv) Die Unterscheidung zwischen RISC- und CISC-Architekturen wird durch die Implementation verwischt, so zeigt der Pentium 4 auf der Benutzungsebene die Charakteristika eines CISC-Prozessors, auf der Implementationsebene ist er aber ein RISC-Prozessor.

Erläuterung zu (iv):



Die vier Grundeinheiten eines Pentium 4

Ein Beispiel:

```
mov    eax, [mema]
imul   eax, 7
add    eax, [memb]
mov    [memc], eax
```

Indem man den Befehl `add eax, [memb]` in zwei Mikrooperationen unterteilt, kann der Speicherzugriff auf die Zelle `memb` zur gleichen Zeit stattfinden wie die Multiplikation mit 7. Die Unterteilung eines ISA-Befehls in mehrere Mikrooperationen ("μops") findet in der Unter-einheit Befehlsaufbereitung statt. Diese Unterteilung ist zu unterscheiden von der Fließbandstruktur der Ausführungs-werke.

Beispiele von Datentypen:

Uninterpretierte Datentypen:

Bit,  
Oktett,  
Halbwort,  
Wort,  
Doppelwort,  
Vierfachwort.

Zahlen:

Ganzzahlen:

Binärzahlen:

vorzeichenlos,  
vorzeichenbehaftet.

Dezimalzahlen:

gepackt,  
ungepackt.

Gleitpunktzahlen:

einfache Genauigkeit,  
doppelte Genauigkeit.

Zeichen:

Zeichenketten:

Zeiger (Adressen):

kurz,  
lang.

**Beispiel: Grundtypen der Pentium-FPU:**

**Ganzzahlformate, Binärform:**

- 16-Bit Zweierkomplement,
- 32-Bit Zweierkomplement,
- 64-Bit Zweierkomplement.

**Ganzzahlformat, BCD-Form:**

18 Dezimalziffern mit Vorzeichen, bcd-codiert.

**Gleitpunktformate:**

**Single Real (32 Bit)**

(1 Bit Vorzeichen, 8 Bit Exponent,  
24 Bit Signifikand, davon 1 Bit virtuell),

**Double Real (64 Bit)**

(1 Bit Vorzeichen, 11 Bit Exponent,  
53 Bit Signifikand, davon 1 Bit virtuell),

**Extended Real (80 Bit)**

(1 Bit Vorzeichen, 15 Bit Exponent,  
64 Bit Signifikand).

**Bemerkung:**

Im Gleitpunktrechenwerk werden alle Daten als Daten der Größe 80 Bit verwaltet. Beim Laden und Speichern der Daten erfolgt eine Konversion. Beispiele sind:

- FLDB** Load Binary Coded Decimal,
- FILD** Load Integer,
- FLD** Load Real,
- FIADD** Add Integer (Speicheroperand),
- FICOMP** Compare Integer (im Speicher).

Bemerkung: FPU = Floating Point Unit

Aus Intel-Manual 243190: Codierung von BCD-Zahlen für Intels FPU:

Zahl	Vorzeichen	18-ziffriger Betrag									
		Ziffer	Ziffer	Ziffer	...	Ziffer	Ziffer	...	Ziffer	Ziffer	
positiv: größte	0	0000000	1001	1001	1001	1001	...	1001	...	1001	
kleinste Null	.	.	0000000	0000	0000	0000	...	0000	...	0000	
	0	0	0000000	0000	0000	0000	...	0000	...	0000	
negativ: Null	1	0000000	0000	0000	0000	...	0000	...	0000	0000	
kleinste	1	0000000	0000	0000	0000	...	0000	...	0000	0000	
größte	.	.	1001	1001	1001	1001	...	1001	...	1001	
	1	1	1111111	1111	1111	1111	UUUU	UUUU	...	UUUU	
indefinit	1	1111111	1111	1111	1111	UUUU	UUUU	...	UUUU	UUUU	
		1 Byte								9 Byte	

Bemerkung: U steht für undefiniert (beliebig).

## Verteilung großer Daten auf kleine Speicherzellen:

Benötigt ein Datum mehrere zusammenhängende Speicherzellen, dann ergeben sich viele Möglichkeiten der Speicherung. Betrachten wir zum Beispiel die Verteilung einer 32-Bit-Ganzzahl auf vier 8-Bit-Byte. Man hat in diesem Fall 4! mögliche Anordnungen. Von diesen werden nur vier genutzt.

Beispiel:      Sedezimalzahl = 0x12345678  
                                  = 305.419.896<sub>10</sub>

### Anordnung 1, "Little Endian":

Adresse+0	Adresse+1	Adresse+2	Adresse+3
78	56	34	12

### Anordnung 2, "Big Endian":

Adresse+0	Adresse+1	Adresse+2	Adresse+3
12	34	56	78

### Anordnung 3, "Mixed Endian":

Adresse+0	Adresse+1	Adresse+2	Adresse+3
34	12	78	56

### Anordnung 4:

Adresse+0	Adresse+1	Adresse+2	Adresse+3
56	78	12	34

## Bemerkungen:

- (i) Im Big-Endian-Modell werden die höherwertigen Teile eines Datums an den niedrigen Adressen gespeichert.
- (ii) Im Little-Endian-Modell werden die niederwertigen Teile eines Datums an den niedrigen Adressen gespeichert.
- (iii) Der "Endian-Begriff" für Zeichenketten ist belanglos, falls eine Speicherzelle genau ein Zeichen aufnimmt. Für die Codierung der Unicode-Zeichen im UTF-16 Format kennt man die beiden Varianten UTF-16BE und UTF-16LE, jeweils eingeleitet durch das Zeichen U+FEFF oder das Nichtzeichen U+FFFE.
- (iv) Neben der Byte-Ordnung existiert auch eine Bit-Ordnung. Diese ist wichtig für eine korrekte rechnerunabhängige Interpretation von Adressen und Größenangaben in Netzen.
- (v) Die Anordnung der Elemente kommt beim Datenaustausch zwischen autonomen Rechnern zum Tragen, so sind GIF-Dateien "little endian" und JPEG-Dateien "big endian".
- (vi) Die Vorzüge des einen Speichermodells gegenüber dem anderen sind gering, so lassen sich Hex-Dumps von "Big Endian"-Maschinen etwas leichter lesen als solche von "Little Endian"-Maschinen, andererseits lassen sich Ganzzahl-Arithmetiken bei "Little-Endian"-Speicherung besonders einfach erweitern.
- (vii) Daß das Byte-Ordnungsproblem hohe praktische Bedeutung hat, zeigt, daß Intel ab dem Rechner 486 einen Byte-Swap-Befehl zur schnellen Datenkonversion zur Verfügung stellt.

## Befehlssätze:

### Elemente eines Befehls:

Befehlscode,  
Operandenbeschreibungen,  
Angabe über Folgebefehl.

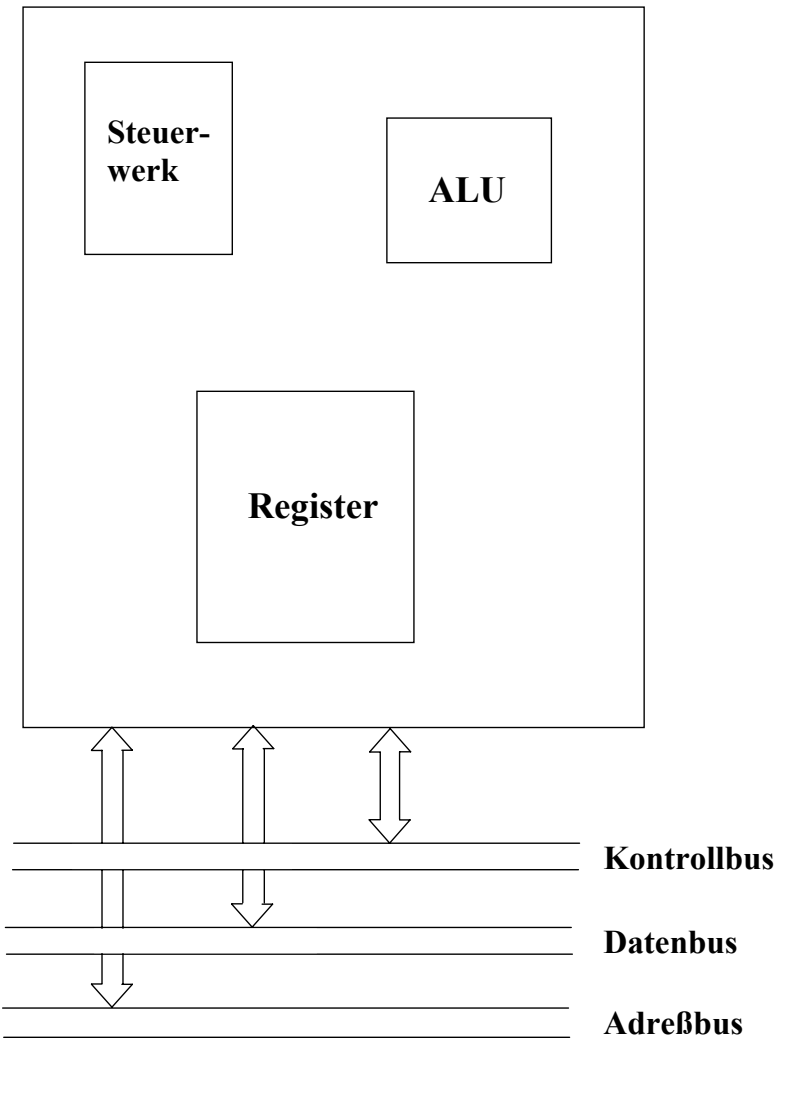
### Grobklassifizierung der Befehle:

Arithmetische Operationen,  
Logische Operationen,  
Interne Datenbewegungen,  
Kontrolltransfers,  
E/A-Operationen,  
Verwaltungsoperationen.

### Wünschenswerte Eigenschaften eines Befehlssatzes:

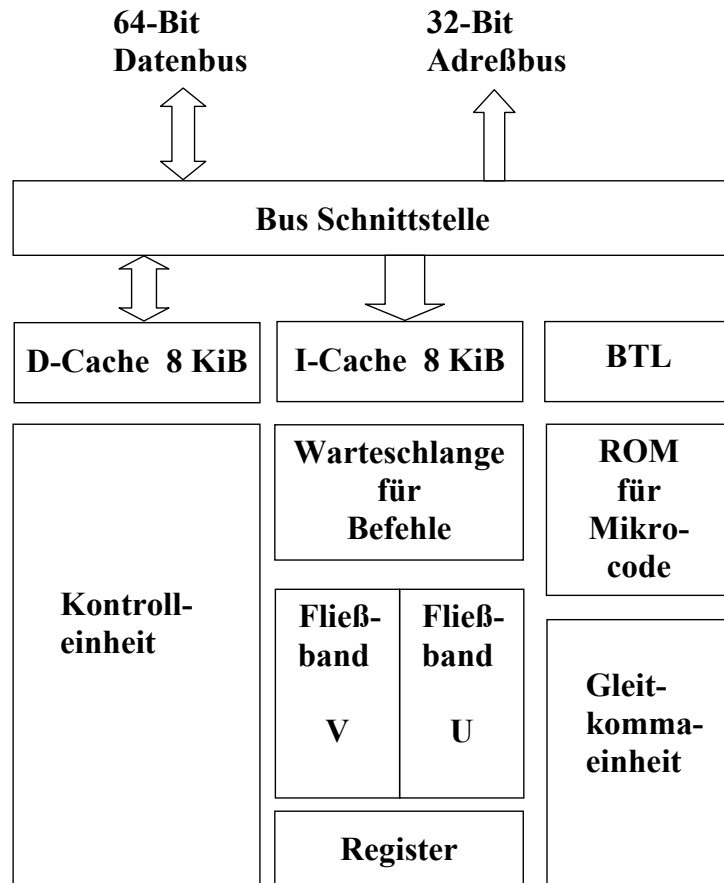
- Vollständigkeit:** Jede berechenbare Funktion soll programmiert werden können.
- Regularität:** Die Menge der angebotenen Operationen soll abgeschlossen sein.
- Effizienz:** Häufig benutzte Funktionen sind einfach darzustellen.
- Orthogonalität:** Befehle, Datentypen und Adressierungsmodi sind unabhängig voneinander.
- Kompatibilität:** Die Befehlsstruktur berücksichtigt das bisher Geschaffene.

## Bild einer Zentraleinheit:



Bemerkung: ALU = Arithmetic and Logical Unit

**Vereinfachtes Pentium Blockdiagramm:**



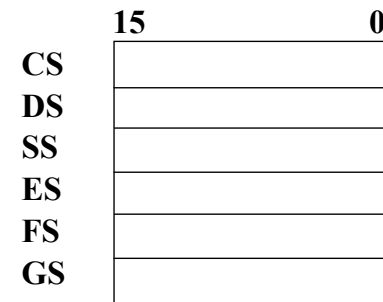
**Bemerkung:** BTL = Branch Trace Logic;  
der Datencache nutzt ein MESI-Protokoll.

**Register der Anwendungsebene des Pentium:**

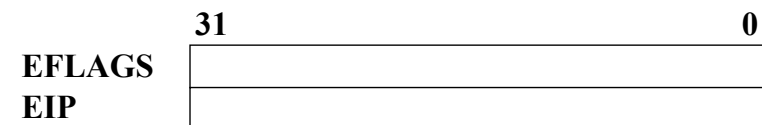
**Acht allgemeine Register, 32 Bit breit**



**Sechs Segmentregister, 16 Bit breit**



**Zwei Status-und-Kontroll-Register, 32 Bit breit**



**Befehlsarten (Auswahl):**

**Datentransfer (Auswahl):**

**Move**  
**Store**  
**Load**  
**Exchange**  
**Clear**  
**Set**  
**Push**  
**Pop**

**Arithmetik (Auswahl):**

**Add**  
**Subtract**  
**Multiply**  
**Divide**  
**Absolute**  
**Negate**  
**Increment**  
**Decrement**

**Logik (Auswahl):**

**And**  
**Or**  
**Not**  
**Exclusive Or**  
**Test**  
**Compare**  
**Shift**  
**Rotate**

**Transfer of Control (Auswahl):**

**Jump**  
**Jump Conditional**  
**Jump to Subroutine**  
**Return**  
**Execute**  
**Skip**  
**Skip Conditional**  
**Wait**  
**No Operation**

**Ein-/Ausgabe (Auswahl):**

**Read**  
**Write**  
**Start I/O**  
**Test I/O**

**System- und Verwaltungsbefehle (Auswahl):**

**Read Machine Status**  
**Write Machine Status**  
**Load Descriptor Table**  
**Store Descriptor Table**  
**Load Access Rights**  
**Store Access Rights**  
**Change Privilege Level**  
**Flush Cache**  
**Invalidate Cache**  
**Halt Processor**

**Konversionen (Auswahl):**  
Translate  
Convert

**Beispiel: Translate-Befehl der S/360-Architektur**

**TR R1, R2, Length**

**Inhalt(R1) = Adresse der umzusetzenden Byte-Kette**

**Inhalt(R2) = Adresse der Umsetztabelle**

**Length = Länge der Byte-Kette**

**Bemerkung:**

**Die x86-Architektur kennt Befehlsvorsätze, diese sind REP (Repeat while ECX is not zero), REPE, REPZ, REPNE, REPNZ. Ein Programmsegment zum Vergleich zweier Speicherblöcke könnte so aussehen:**

```
cld                ; Setze Richtungsangabe
mov esi, block1    ; Adresse von Block 1
mov edi, block2    ; Adresse von Block 2
mov ecx, size      ; Größe in Byte
repe cmpsb        ; Wiederhole solange
                  ; Anzeige Z gesetzt ist
                  ; und ecx > 0 ist.
je gleich          ; Falls Z gesetzt ist, dann
                  ; sind Blöcke gleich.
...               ; Code, falls Block 1 ≠ Block 2
jmp weiter
gleich: ...        ; Code, falls Block 1 = Block 2
weiter:
```

**Multimedia-Erweiterungen der Befehlssätze von Rechenanlagen, hier am Beispiel des Intel-Pentiums und seiner Nachfolger.**

- 1997:** Einführung von 57 neuen Befehlen, die die Gleitpunktregister des Pentiums nutzen. Es werden gleichzeitig bis zu acht Datenelemente bearbeitet.
- 1999:** (Pentium III, SSE). Es werden weitere 70 Befehle der Pentium-Architektur hinzugefügt, zusammengefaßt unter dem Begriff Streaming SIMD Extensions. Diesmal werden acht neue Register der Breite 128 Bit bereitgestellt, so daß vier Gleitpunktoperationen einfacher Genauigkeit parallel ausgeführt werden können.
- 2001:** (Pentium 4, SSE2). Es werden 144 neue Instruktionen eingeführt, die eine parallele Verarbeitung zweier doppelt genauer Gleitpunktoperationen gestatten.
- 2004:** (Prescott, SSE3). Es werden weitere 13 Instruktionen bereitgestellt.
- 2006:** (Supplemental SSE3). Der SSE-Befehlssatz wird um weitere 32 (= 2 \* 16) Befehle ergänzt.
- 2007:** (Penryn, Nehalem, SSE4). Es werden weitere 54 Instruktionen definiert, unterteilt in zwei Gruppen.
- 2008:** Für das Jahr 2010 wird eine Advanced Vector Extension angekündigt, hierbei werden die SSE-Register von 128 auf 256 Bit erweitert.

**Beispiel: Intels MMX-Befehle (ohne SSE-Erweiterungen)**  
**(Bemerkung: MMX = Multimedia Extensions)**

**Charakteristika:**

**3 neue Datentypen:**

**8 gepackte 8-Bit-Ganzzahlen,  
4 gepackte 16-Bit-Ganzzahlen,  
2 gepackte 32-Bit-Ganzzahlen.**

**47 neue Befehle aus den Gruppen**

**Datentransfer,  
Arithmetik,  
Vergleiche,  
Wandlungen,  
Packen,  
Schieben,  
Logik,  
Zustandsänderung.**

**3 Arten der Arithmetik:**

**Modulo-Arithmetik,  
Grenzarithmetik für vorzeichenlose Ganzzahlen,  
Grenzarithmetik für vorzeichentragende Ganzzahlen.**

**Zahlbereiche:**

**$-128 \leq$  vorzeichenbehaftete 8-Bit-Zahl  $\leq$  127  
 $-32768 \leq$  vorzeichenbehaftete 16-Bit-Zahl  $\leq$  32767  
 $0 \leq$  vorzeichenlose 8-Bit-Zahl  $\leq$  255  
 $0 \leq$  vorzeichenlose 16-Bit-Zahl  $\leq$  65535**

**Liste der MMX-Befehle:**

**PADDB, PADDW, PADDD,  
PADDSB, PADDSW,  
PADDUSB, PADDUSW,**

**PSUBB, PSUBW, PSUBD,  
PSUBSB, PSUBSW,  
PSUBUSB, PSUBUSW,**

**PMULLW, PMULHW, PMULHUW,**

**PMADDWD,**

**PCMPEQB, PCMPEQW, PCMPEQD,**

**PCMPGTPB, PCMPGTPW, PCMPGTPD,**

**PACKSSWB, PACKSSDW, PACKUSWB,**

**PUNCKHBW, PUNCKHWD, PUNPCKHDQ,  
PUNCKLBW, PUNCKLWD, PUNPCKLDQ,**

**PAND, PANDN, POR, PXOR,**

**PSLLW, PSLLD, PSLLQ,  
PSRLW, PSRLD, PSRLQ,  
PSRAW, PSRAD,**

**MOVD, MOVQ,**

**EMMS ; Empty MMX State.**

**Beispiel: Add Packed Byte Integers**

**PADDB mm0, mm1**

**mm0:**

11111111	00000000	01101001	10111111	00101010	01101010	10101111	10111101
----------	----------	----------	----------	----------	----------	----------	----------

**+**

**mm1:**

11111110	11111111	00001111	10101010	11111111	00010101	11010101	00101010
----------	----------	----------	----------	----------	----------	----------	----------

**=**

**mm0:**

11111101	11111111	01111000	01101001	00101001	01111111	10001000	11100111
----------	----------	----------	----------	----------	----------	----------	----------

**Bemerkung: Die einzelnen Summen werden modulo 256 berechnet.**

**Beispiel: Compare Packed Bytes for Equal**

**PCMPEQB mm0, mm1**

**mm0:**

11111111	00000000	00000000	10101010	00101010	01101010	10101111	10111101
----------	----------	----------	----------	----------	----------	----------	----------

**?**

**mm1:**

11111111	11111111	00000000	10101010	00101011	01101010	11010101	00101010
----------	----------	----------	----------	----------	----------	----------	----------

**=**

**mm0:**

11111111	00000000	11111111	11111111	00000000	11111111	00000000	00000000
----------	----------	----------	----------	----------	----------	----------	----------

**true false true true false true false false**

**Beispiel: Bedingte Zuweisung in 64-Bit-Kette**

**PAND mm0, mm1 ; Logisches Und für 64-Bit-Ketten**

**mm0:**

11111111	00000000	11111111	11111111	11111111	00000000	11111111	00000000	00000000
----------	----------	----------	----------	----------	----------	----------	----------	----------

**and**

**mm1:**

10110011	11001100	00001111	10101010	111110001	00010101	11010101	00101010
----------	----------	----------	----------	-----------	----------	----------	----------

**=**

**mm0:**

10110011	00000000	00001111	10101010	00000000	00010101	00000000	00000000
----------	----------	----------	----------	----------	----------	----------	----------

**Beispiel: Mittelwertbildung (Eingeführt mit SSE-Erweiterung)**

**PAVGB mm0, mm1 ; (Byte1 + Byte2 + 1) >> 1  
; existiert für 64-Bit- und 128-Bit-Werte**

**mm0:**

00000000	11111111	11111111	11111111	00000010	00000111	00000001	00000000
----------	----------	----------	----------	----------	----------	----------	----------

**pavgb**

**mm1:**

00000000	11111101	11111110	11111111	00001111	00000111	00000001	00000000
----------	----------	----------	----------	----------	----------	----------	----------

**=**

**mm0:**

00000000	11111110	11111111	11111111	00001001	00000111	00000001	00000000
----------	----------	----------	----------	----------	----------	----------	----------

### Beispiele aus SSE3:

#### Packed Single-FP Horizontal Add:

##### HADDPS A B

A (128 Bit, 4 Datenelemente):  $3_a, 2_a, 1_a, 0_a$

B (128 Bit, 4 Datenelemente):  $3_b, 2_b, 1_b, 0_b$

Ergebnis (in A):  $3_b+2_b, 1_b+0_b, 3_a+2_a, 1_a+0_a$

#### Packed Single-FP Horizontal Subtract:

##### HSUBPS A B

A (128 Bit, 4 Datenelemente):  $3_a, 2_a, 1_a, 0_a$

B (128 Bit, 4 Datenelemente):  $3_b, 2_b, 1_b, 0_b$

Ergebnis (in A):  $2_b-3_b, 0_b-1_b, 2_a-3_a, 0_a-1_a$

#### Move Packed Single-FP High and Duplicate:

##### MOVSHDUP A B

A (128 Bit, 4 Datenelemente):  $3_a, 2_a, 1_a, 0_a$

B (128 Bit, 4 Datenelemente):  $3_b, 2_b, 1_b, 0_b$

Ergebnis (in A):  $3_b, 3_b, 1_b, 1_b$

#### Move Packed Single-FP Low and Duplicate:

##### MOVSLDUP A B

A (128 Bit, 4 Datenelemente):  $3_a, 2_a, 1_a, 0_a$

B (128 Bit, 4 Datenelemente):  $3_b, 2_b, 1_b, 0_b$

Ergebnis (in A):  $2_b, 2_b, 0_b, 0_b$

#### Packed Single-FP Add/Subtract:

##### ADDSUBPS A B

A (128 Bit, 4 Datenelemente):  $3_a, 2_a, 1_a, 0_a$

B (128 Bit, 4 Datenelemente):  $3_b, 2_b, 1_b, 0_b$

Ergebnis (in A):  $3_a+3_b, 2_a-2_b, 1_a+1_b, 0_a-0_b$

### Befehlsformate:

#### Beispiel zum Vergleich der Befehlsformate:

Beispiel:  $Z := (A - B) / (C + D * E)$

#### Code im Drei-Adreß-Format:

```
SUB    Z, A, B
MUL    H, D, E    ; H = Hilfszelle
ADD    H, H, C
DIV    Z, Z, H
```

#### Code im Zwei-Adreß-Format:

```
MOV    Z, A
SUB    Z, B
MOV    H, D    ; H = Hilfszelle
MUL    H, E
ADD    H, C
DIV    Z, H
```

#### Code im Ein-Adreß-Format:

```
LOAD  D
MUL  E
ADD  C
STO  Z
LOAD  A
SUB  B
DIV  Z
STO  Z
```

### Code im Null-Adreß-Format:

Hier: Nutzung eines Kellers:

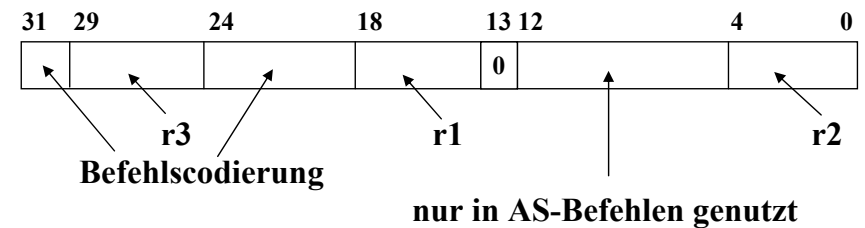
PUSH A  
PUSH B  
SUB  
PUSH C  
PUSH D  
PUSH E  
MUL  
ADD  
DIV  
POP Z

### Bemerkungen:

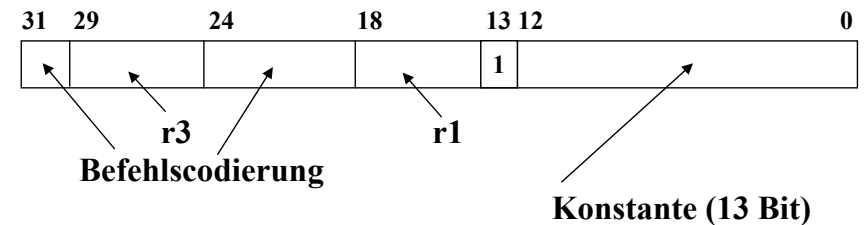
- (i) Je weniger Befehlsadressen man benutzt, desto mehr Befehle benötigt man zur Beschreibung einer Rechnung; dies bedeutet aber nicht, daß die befehlsreicheren Programme bezüglich der Maße Speicherplatz und Laufzeit im Nachteil sind.
- (ii) Auch auf einer reinen Kellermaschine benötigt man Datentransportbefehle, also Ein-Adreß-Befehle. Die Kellerkonventionen für nichtkommutative Operationen bedürfen besonderer Beachtung.

### Befehlskodierung für die Sparc V8:

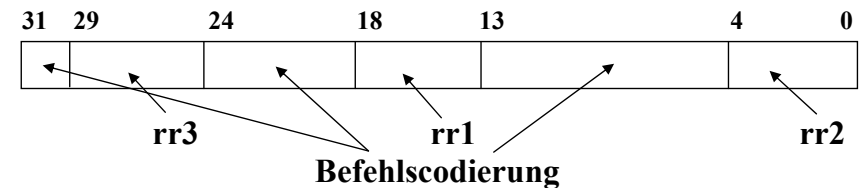
(1) Befehlscode r1, r2, r3



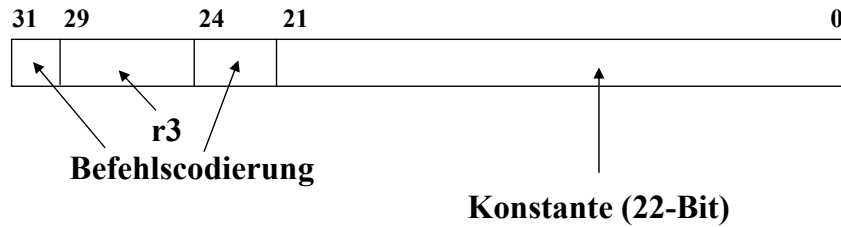
(2) Befehlscode r1, 13-Bit-Konstante, r3



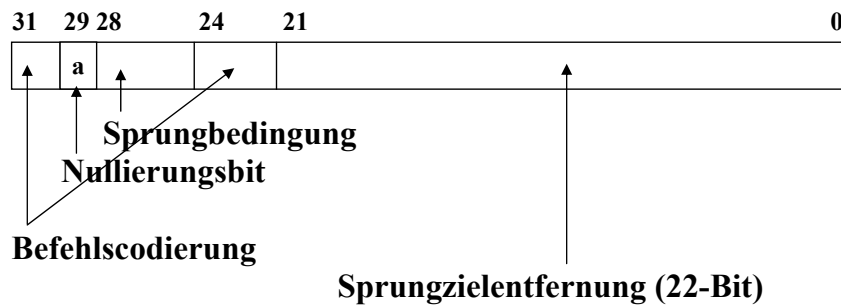
(3) Befehlscode rr1, rr2, rr3



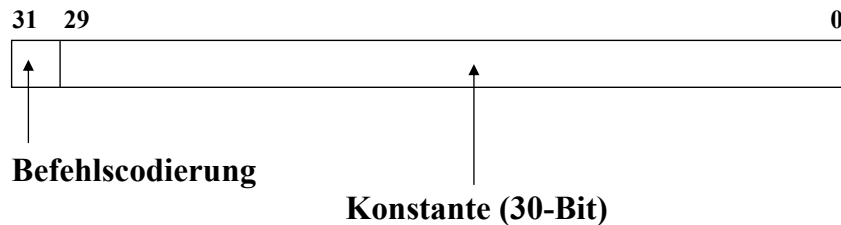
**(4) Befehlscode      22-Bit-Konstante, r3**



**(5) Sprungbefehl      Bedingung, a      Sprungziel**



**(6) Unterprogrammaufruf**



**Adressierungsmodi:**

**Ziel:**      **Bequemes und effizientes Ansprechen der von einem Rechner zur Verfügung gestellten Daten-Quantitäten.**

**Weitverbreitete Modi:**

**Immediatmodus:**

**Operand im Befehl,  
Datum konstant während der Laufzeit,  
kein zusätzlicher Speicherzugriff,  
Wertebereich beschränkt durch Größe  
des Adressierungsfeldes.**

**Direktmodus:**

**Adresse des Operanden im Befehl,  
keine Adreßberechnung,  
ein zusätzlicher Speicherzugriff,  
Adresse konstant, Datum variabel,  
Adreßbereich beschränkt.**

**Indirektmodus:**

**Adresse im Befehlsfeld spezifiziert eine  
Speicherzelle, die die Endadresse enthält,  
zwei Speicherzugriffe,  
hohe Flexibilität,  
Adreßbereich wird durch Wortbreite bestimmt.**

**Registermodus:**

Wie Direktmodus, aber Angabe eines Operandenregisters, kein zusätzlicher Speicherzugriff.

**Register-Indirekt-Modus:**

Wie Indirektmodus, aber Angabe eines Registers, das die Speicheradresse des Operanden enthält.

**Versatz-Modus:**

Die Adresse wird in zwei Teilen angegeben, einem Versatz im Befehl und einem weiteren Addenden, dieser kann ein Register oder auch der Befehlszähler sein, die effektive Adresse wird berechnet gemäß:

$$\text{Effektive Adresse} = \text{Versatz} + \text{I(REG)}$$

**Indexmodus:**

Ähnlich dem Versatzmodus, hier wird aber der Versatz als Basisadresse betrachtet, Skalierung möglich.

**Bemerkung:** Es existieren bedeutend elaboratere Adressierungsmodi, so kann man Mehrfachindizierung mit Mehrfachindirektheit und automatischer Adreßmodifizierung verbinden. Hierbei sind noch die Modi zur Speichersegmentierung zu berücksichtigen.

**Beispiel: Indexadressierung für Pentium II:**

Basis		Index		Skalierung		Versatz
$\left( \begin{array}{c} \text{EAX} \\ \text{EBX} \\ \text{ECX} \\ \text{EDX} \\ \text{ESP} \\ \text{EBP} \\ \text{ESI} \\ \text{EDI} \end{array} \right)$	+	$\left( \begin{array}{c} \text{EAX} \\ \text{EBX} \\ \text{ECX} \\ \text{EDX} \\ \text{EBP} \\ \text{ESI} \\ \text{EDI} \end{array} \right)$	*	$\left( \begin{array}{c} 1 \\ 2 \\ 4 \\ 8 \end{array} \right)$	+	$\left( \begin{array}{c} \text{kein} \\ 8\text{-bit} \\ 16\text{-bit} \\ 32\text{-bit} \end{array} \right)$

**Berechnung der segmentrelativen Adresse:**

$$\text{Inhalt (Basis)} + \text{Inhalt (Index)} * \text{Skalierung} + \text{Versatz}$$

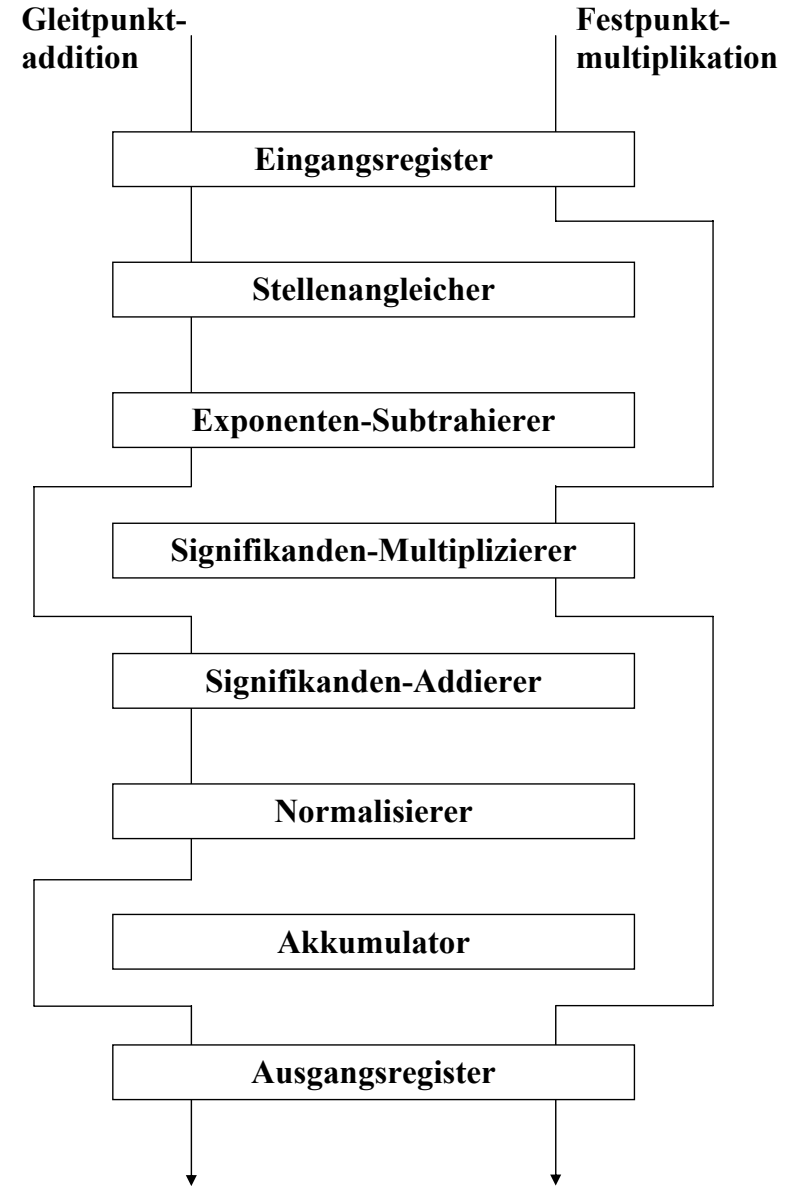
### Illustration des Fließbandprinzips:

Eine Operation teilt man in mehrere Schritte auf, Operation = S1, S2, . . . , Sm, die von verschiedenen Stationen ausgeführt werden; im Beispiel werden 4 Stationen benutzt.

Zeit	Station 1	Station 2	Station 3	Station 4	
1	A01-T1				
2	A02-T1	A01-T2			
3	A03-T1	A02-T2	A01-T3		
4	A04-T1	A03-T2	A02-T3	A01-T4	
5	A05-T1	A04-T2	A03-T3	A02-T4	A01
6	A06-T1	A05-T2	A04-T3	A03-T4	A02
7	A07-T1	A06-T2	A05-T3	A04-T4	A03
8	A08-T1	A07-T2	A06-T3	A05-T4	A04
9	A09-T1	A08-T2	A07-T3	A06-T4	A05
10	A10-T1	A09-T2	A08-T3	A07-T4	A06
11	A11-T1	A10-T2	A09-T3	A08-T4	A07
12	A12-T1	A11-T2	A10-T3	A09-T4	A08
13	A13-T1	A12-T2	A11-T3	A10-T4	A09
14	A14-T1	A13-T2	A12-T3	A11-T4	A10
15					

**Bemerkung:** Die einzelnen Arbeitsschritte müssen im Mittel gleich lange dauern. Die Anzahl der isolierbaren Arbeitsschritte bestimmt den Geschwindigkeitsgewinn.

### Ein achtstufiges Pipeline-Rechenwerk:



### Ein möglicher Befehlsausführungszyklus:

#### Wiederhole bis *Programmende*

1. Bestimme die Adresse des nächsten Befehls.
2. Lade den nächsten Befehl aus dem Speicher.
3. Decodiere den Befehl.
4. Für jeden Operanden führe aus:  
Bestimme seine Adresse und hole ihn.
5. Führe den Befehl aus.
6. Bestimme die Adresse des Ergebnisses.
7. Speichere das Ergebnis.
8. Bediene anstehende Unterbrechungen.

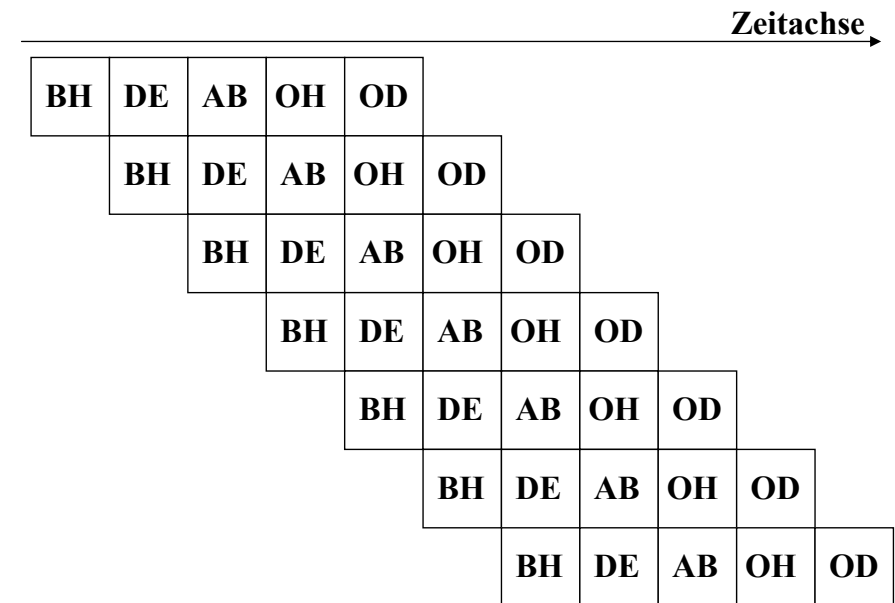
**Bemerkung:** Die Behandlung einer Unterbrechung erfordert die Rettung des Systemzustandes.

### Befehls-Pipelining:

Bei der Ausführung eines Befehls kann man verschiedene, abgeschlossene Tätigkeiten unterscheiden, z. B.

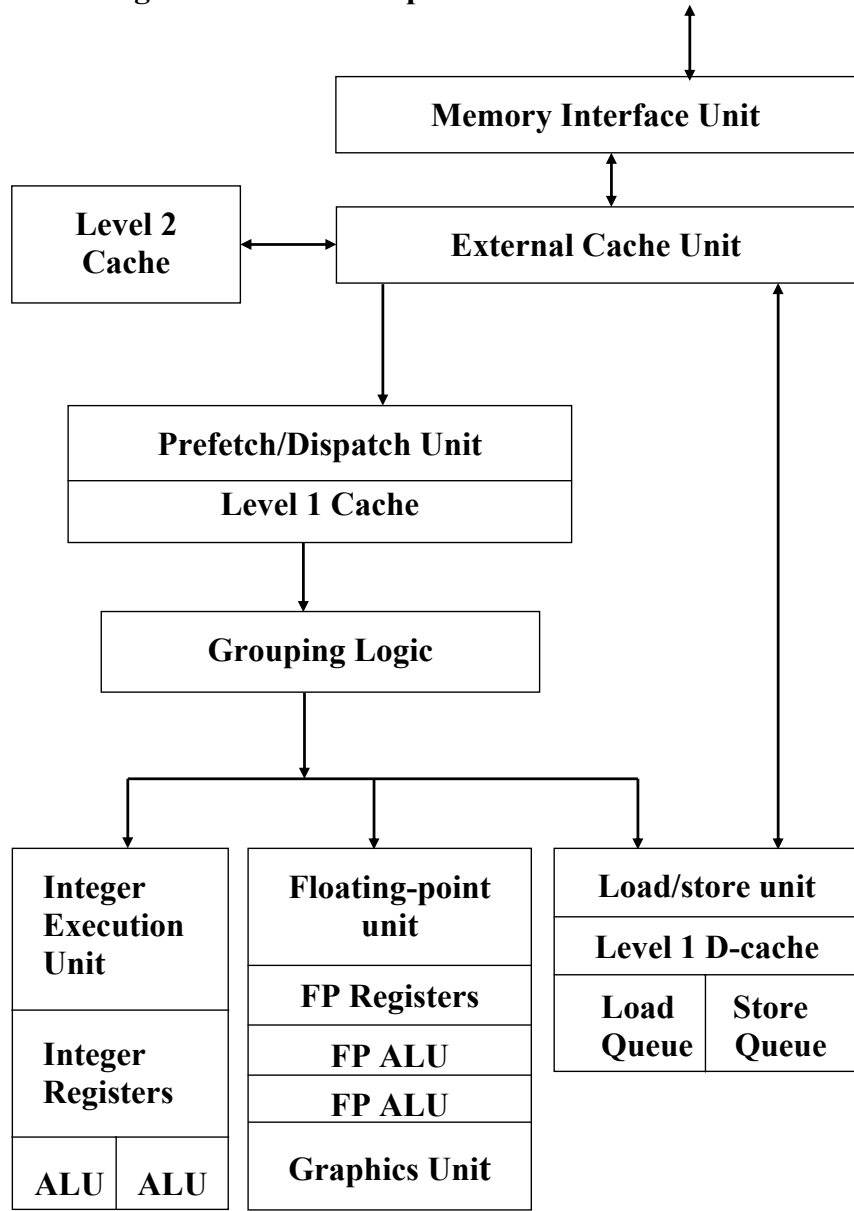
- BH** = Befehl holen,
- DE** = Befehl decodieren,
- AB** = Adressen bestimmen,
- OH** = Operanden holen,
- OD** = Operation durchführen.

Benötigen die einzelnen Befehlsschritte die gleiche Ausführungsdauer, dann lassen sie sich problemlos überlappt ausführen, wie das folgende Bild zeigt.

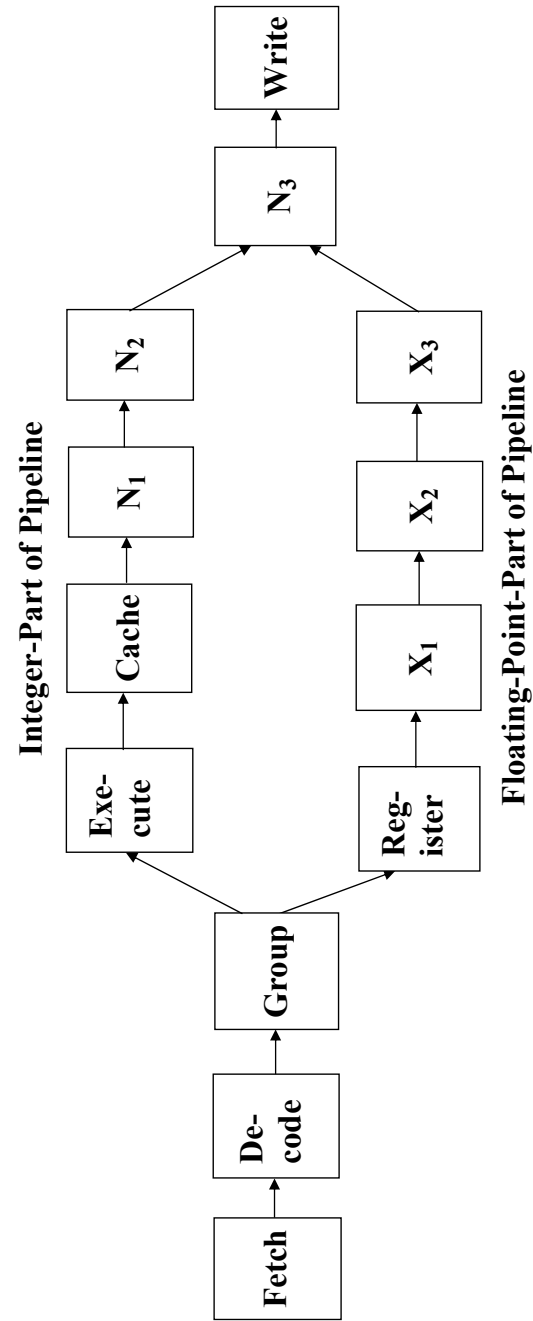




**Blockdiagramm der Ultra-Sparc II:**



**Pipeline der Ultra-Sparc II:**



**Bemerkung:** Die Ultra-Sparc II ist ein Beispiel für eine superskalare Architektur. In der Gruppenphase werden bis zu 4 Befehle angestoßen. In der Fetch-Phase wird auch die Sprungvorhersage getätigt. N<sub>2</sub> ist eine reine Wartephase, es wird auf die Beendigung von X<sub>3</sub> gewartet. N<sub>3</sub> dient der Ausnahmebehandlung.

### Verzögerte Ausführung von Sprüngen:

Betrachten wir zwei Arten von Befehlen, einfache arithmetische Befehle mit zwei Ausführungsabschnitten, **H D**, Befehl holen und Befehl durchführen, und Speicherbefehle mit drei Ausführungsabschnitten, **H A S**, Befehl holen, Adresse berechnen und Datentransfer vom/zum Speicher.

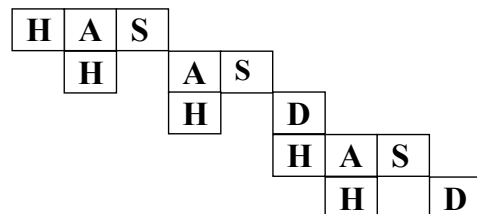
#### Programmsegment:

Load	R1, X	<b>H A S</b>
Load	R2, Y	<b>H A S</b>
Add	R1, R2, R3	<b>H D</b>
Store	R3, Z	<b>H A S</b>
Branch	M1	<b>H D</b>

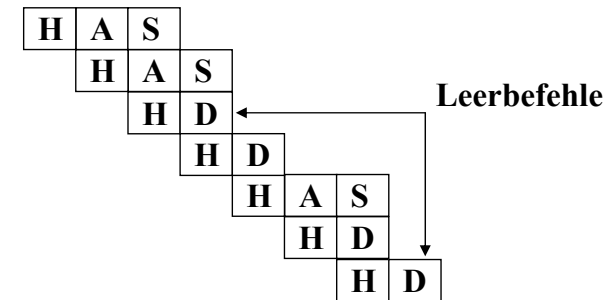
#### Sequentielle Ausführung: 13 Zeiteinheiten



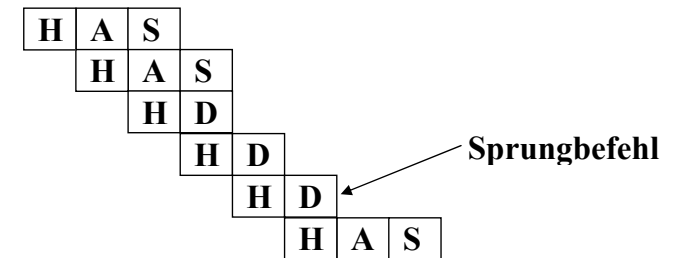
#### Teilweise überlappte Ausführung: 9 Zeiteinheiten



Die Leerzeiten im Zeitdiagramm beruhen auf der Annahme, daß der Speicher nur einen Zugang hat. Verwendet man zweiportigen Speicher und setzt man NOP-Befehle als Füllsel ein, dann erhält man eine einfachere überlappte Befehlsausführung.



Die Nulloperation nach einem Sprungbefehl kann man einsparen, falls man die Ausführung des Sprungbefehls um "einen Befehl verzögert." Im Beispiel wird man den Sprungbefehl vor dem Speicherbefehl plazieren.



Nach der Holephase des Speicherbefehls ist der Befehlszähler mit der Sprungadresse aktualisiert. Die Sparc verwendet zwei Befehlszähler, PC und Next-PC.

## Vorhersage bei bedingten Sprüngen:

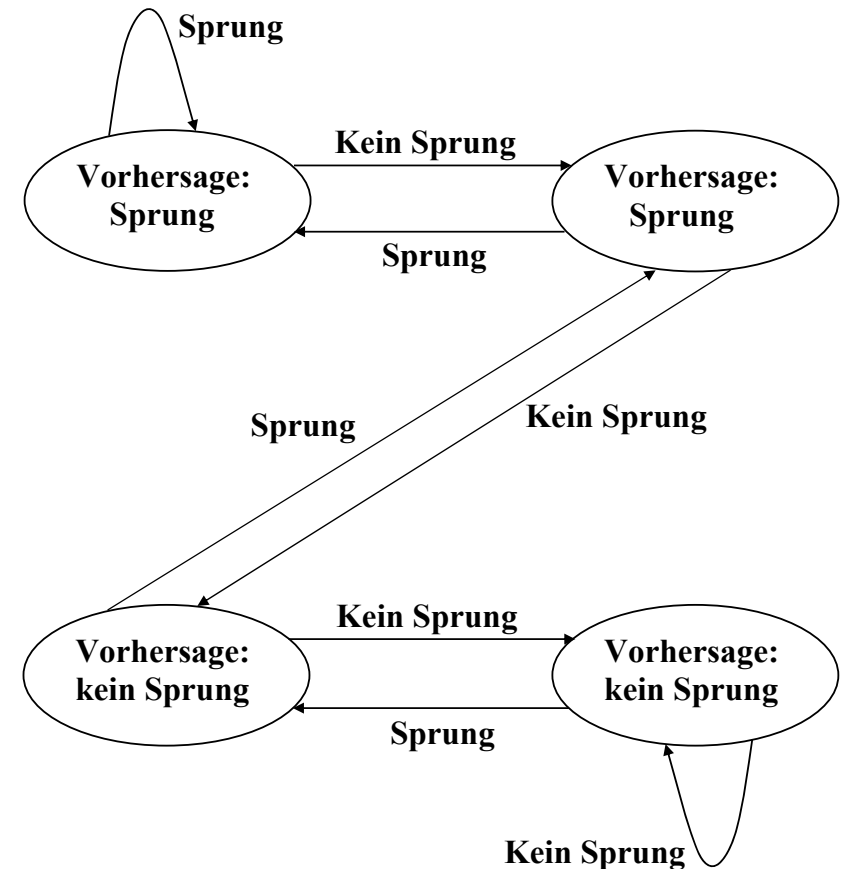
### Statische Vorhersage:

- (i) Sprung wird nicht ausgeführt.
- (ii) Sprung wird ausgeführt.
- (iii) Rückwärtssprung wird ausgeführt.
- (iv) Vorwärtssprung wird ausgeführt.
- (v) Sprungvorhersage in Abhängigkeit vom Befehlscode.

### Dynamische Vorhersage:

- (i) Aufzeichnung der individuellen Sprunghistorie in einem n-Bit-Zähler, genutzt werden 1-Bit- und 2-Bit-Zähler.
- (ii) Aufzeichnung der globalen Sprunghistorie in einem n-Bit-Zähler ( $n \geq 5$ ).
- (iii) Mischformen von (i) und (ii).
- (iv) Statt einzelner Zähler kann man auch die Zielbefehle für jeden Sprung in einem schnellen Puffer halten. Man spart dann den Zugriff auf den Hauptspeicher.

## Übergangsdiagramm zur Sprungvorhersage:



Den Zustand des Übergangsgraphen merkt man sich für jeden Sprung in einem 2-Bit-Zähler. Häufig trifft man Speicher für 4096 Sprünge an; Programmabschnitte, die mehr als 4000 bedingte Sprünge enthalten, sind selten.