

# RO Reference Sheet

Frederick Wichert

Last Edited: 25. Juli 2020

## Inhaltsverzeichnis

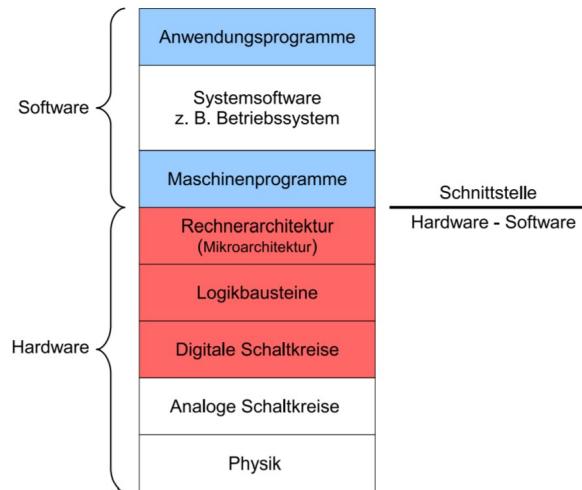
<b>1</b>	<b>Grundlagen der Rechnerorganisation</b>	<b>1</b>
1.1	Abstraktion . . . . .	1
1.2	Definition einiger Grundbegriffe . . . . .	1
1.3	Ein Rechnersystem aus der Praxis . . . . .	1
1.4	Speicherhierarchie . . . . .	2
1.5	Speicherorganisation - Adressierung und Endian . . . . .	3
1.6	Rechnersysteme - Verfeinert . . . . .	4
<b>2</b>	<b>Speicher</b>	<b>5</b>
2.1	Speicherhierarchie . . . . .	5
2.2	Speicher Kategorisierung . . . . .	5
2.3	Speichertechnologien . . . . .	6
2.4	Speicherorganisation . . . . .	7
2.5	Lokalität . . . . .	8
2.6	Lokalität - Beispiele . . . . .	9
2.7	Cache . . . . .	10
<b>3</b>	<b>Maschinennahe Programmierung</b>	<b>12</b>
3.1	Phasen der Übersetzung . . . . .	12
3.2	Ausführung eines Programmes . . . . .	13
3.3	Compilieren, Assemblieren und Linken . . . . .	13
3.4	Kontrolloperationen - Statusbits . . . . .	15
<b>4</b>	<b>Konzepte der maschinennahen Programmierung</b>	<b>16</b>
4.1	Allgemeins . . . . .	16
4.2	Lesen und Schreiben . . . . .	16
4.3	Nutzen des Hauptspeichers . . . . .	16
4.4	Sprünge / Verzweigungen . . . . .	17
4.5	Datenfelder (Arrays) . . . . .	18
4.6	Unterprogramme . . . . .	18
4.7	Stacks . . . . .	20
4.8	Rekursion . . . . .	21
4.9	Zusammenfassung Unterprogrammaufrufe . . . . .	21
<b>5</b>	<b>Gleitkommazahlen</b>	<b>23</b>
5.1	Zahlendarstellung - Visualisierung . . . . .	23
5.2	Gleitkommazahlen nach ANSI/IEEE 754 . . . . .	23
5.3	Verarbeitung reeller Zahlen in ARM . . . . .	24
5.4	Studium der Datenblätter . . . . .	26
5.5	Gleitkommaeinheit . . . . .	26

<b>6</b>	<b>Mikroarchitektur</b>	<b>27</b>
6.1	Terminologie . . . . .	27
6.2	Allgemeines . . . . .	27
6.3	Mikroarchitektur Prozessor . . . . .	28
6.4	Eintakt-Prozessor . . . . .	29
6.5	Eintakt vs Mehrtakt . . . . .	31
6.6	Mehrtakt-Prozessor . . . . .	31
6.7	Pipeline-Prozessor . . . . .	34
6.8	Hazards . . . . .	35
<b>7</b>	<b>Single Instruction, Multiple Data SIMD</b>	<b>38</b>
7.1	Anwendung und Nutzen . . . . .	38
7.2	Klassifikation von Flynn . . . . .	38
7.3	NEON . . . . .	38
7.4	NEON Programmierung . . . . .	40
<b>8</b>	<b>Weiteres</b>	<b>41</b>
8.1	Parallelisierung und OpenMP . . . . .	41
<b>9</b>	<b>Streifzug durch die Geschichte</b>	<b>42</b>
<b>10</b>	<b>Ethik in der Informatik</b>	<b>42</b>

# 1 Grundlagen der Rechnerorganisation

## 1.1 Abstraktion

Durch verstecken unnötiger Details können zentrale Konzepte verständlich gemacht werden



Das Schichtenmodell ist ein Beispiel für Abstraktion zur Simplifikation

## 1.2 Definition einiger Grundbegriffe

Ein Computer

ist ein Datenverarbeitungssystem. Nach DIN 44300: Ein Datenverarbeitungssystem ist eine Funktionseinheit zur Verarbeitung und Aufbewahrung von Daten. Verarbeitung umfaßt die Durchführung mathematischer, umformender, übertragender und speichernder Operationen.

Ein Rechner hat somit vier Grundfunktionen: Verarbeiten, Speichern, Umformen von Daten und Kommunizieren

Ein Paradigma

(Denkmuster, Musterbeispiel) ist ein übergeordnetes Prinzip. Es manifestiert sich in Beispielen und ist für eine ganze Teildisziplin typisch.

Ein Programmiermodell

bezeichnet bei Hochsprachen Grundlegende Eigenschaften einer Programmiersprache und bei der maschinennahen Programmierung den Registersatz eines Prozessors, genauer Register die durch Programme werden können sowie den Befehlssatz

## 1.3 Ein Rechnersystem aus der Praxis

Ein Rechnersystem enthält mindestens:

- ein Prozessor (CPU) - Ausführung der Programme
- ein Speicher - Enthält Programme und Daten (Speichersystem)
- eine Möglichkeit zum Transferieren von Informationen

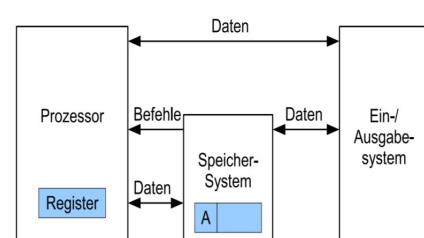
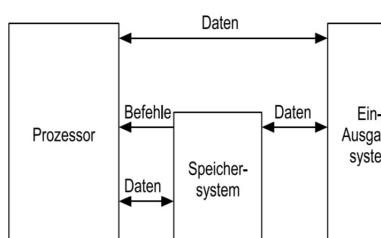
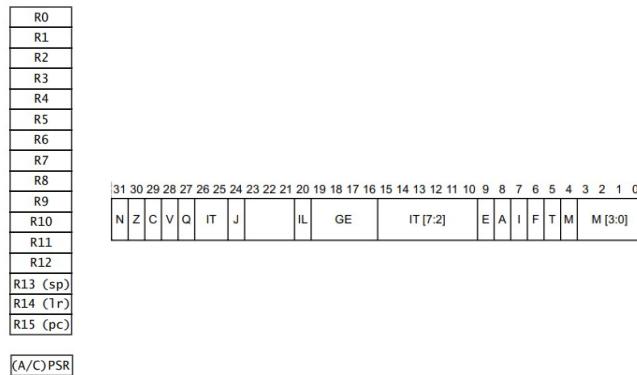


Abbildung: Komponenten eines Rechnersystems (abstrakte Darstellung)

Abbildung: Komponenten eines Rechnersystems (verfeinerte Darstellung)

Der Registersatz ist folgendermaßen Aufgebaut:

- R0
- R1 ... R12
- R13 (sp) - stack pointer (Stapelzeiger)
- R14 (lr) - link register (Rückkehradresse)
- R15 (pc) - program counter (Befehlszähler)
- CPSR - Current Processor Status Register (u.a. Statusflags)



## 1.4 Speicherhierarchie

Transparent nutzbare Speicher werden nicht direkt ansprechbar, sie werden implizit von einem Maschinenprogramm benutzt.

- bestimmte Register auf dem Prozessor
- Cache-Speicher

Explizit nutzbare Speicher können direkt angesprochen werden.

- interner Prozessorspeicher
  - schnelle Register, temporäre Nutzung (Maschinenbefehle, Daten)
  - Direkter Zugriff durch Maschinenbefehle
  - Technologie: Halbleiter ICs
- Hauptspeicher
  - relativ grosser und schneller Speicher, während der Ausführung
  - Direkter Zugriff durch Maschinenbefehle
  - Technologie: Halbleiter ICs
- Sekundärspeicher
  - sehr grosser, sehr langsamer Speicher, permanente Speicherung
  - Indirekter Zugriff über E/A-Programme
  - Technologie: ICs, Magnetplatten, optische Laufwerke, Magnetbänder

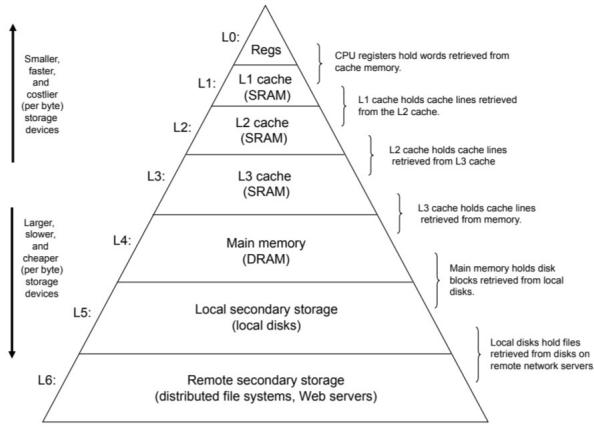
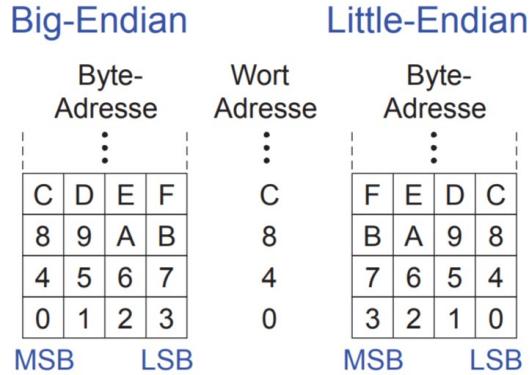


Abbildung: Quelle: [BO10, S. 48]

## 1.5 Speicherorganisation - Adressierung und Endian

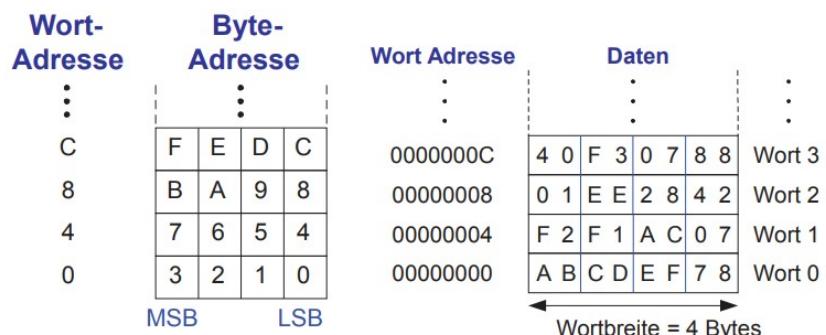
**Endian:**

- Schemata für Nummerierung von Bytes in einem Wort



**Adressierung:** Die Daten passen nicht alle in 15 Register. Daher werden Daten im Hauptspeicher abgelegt, dieser ist allerdings langsamer als Register. Häufig verwendete Daten sind daher in Registern abzulegen. Eine Kombination aus beidem ermöglicht optimale Laufzeit.

- ARM ist byte-adressiert, jedes byte hat eine eindeutige Adresse
- 32.bit Wort = 4 bytes → Eine Wortadresse zeigt auf 4 bytes
- Adressen von Wörtern sind vielfache von 4



## 1.6 Rechnersysteme - Verfeinert

**Maschinenbefehle** Die Menge der Maschinenbefehle die auf einem Rechnersystem zur Verfügung stehen ist durch den Prozessor festgelegt. Man unterscheidet zwischen CISC (Complex Instruction Set Computer), z.B. Intel, und RISC (Reduced Instruction Set Computer), z.B. ARM, Maschinen. Grundsätzlich enthalten alle Rechnersysteme ähnliche Komponenten, aber die genaue Struktur hat Erheblichen Einfluss auf Kosten und Leistung. Einteilung z.B. nach n-Addressmaschinen, 2-Addressmaschinen Intel oder 3-Addressmaschinen ARM.

Wir verwenden eine ARM Architektur.  
ARM = Acorn/Advanced RISC Machines.

Verfeinerung des Rechnersystems:

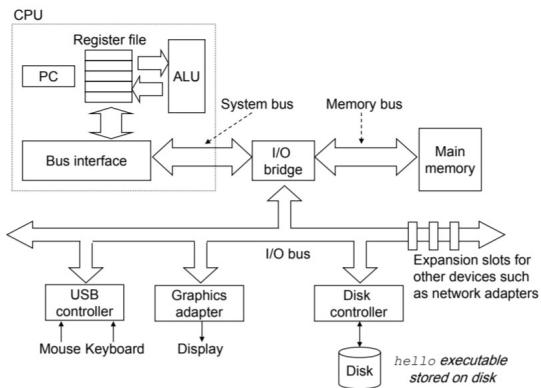


Abbildung: Quelle: [BO10, S. 42]

Komponenten:

- CPU/Prozessor - führt Befehle aus dem Hauptspeicher aus
- ALU - Arithmetic Logic Unit
- PC - Programm Counter, zeigt auf nächsten Maschinenbefehl
- Register file - schneller Speicher für Operanden
- Main memory - Speichert Befehle und Daten
- Bus interface - Verbinden der einzelnen Komponenten

## 2 Speicher

### 2.1 Speicherhierarchie

- Bisher haben wir den Speicher als lineares Feld aus Bytes
- Die CPU kann auf jede Speicherzelle in konstanter Zeit zugreifen
- In der Praxis wird eine Hierarchie mit unterschiedlichen Speichertechnologien verwendet
- Das Lesen und Schreiben dauert gewisse Zeit, i.d.R. relativ lange im Vergleich zur Taktfrequenz
- Je nach Architektur kann das Lesen von Befehlen und Daten Parallel aus zwei Speichern erfolgen (Harvard) oder muss nacheinander aus einem erfolgen (Neumann)

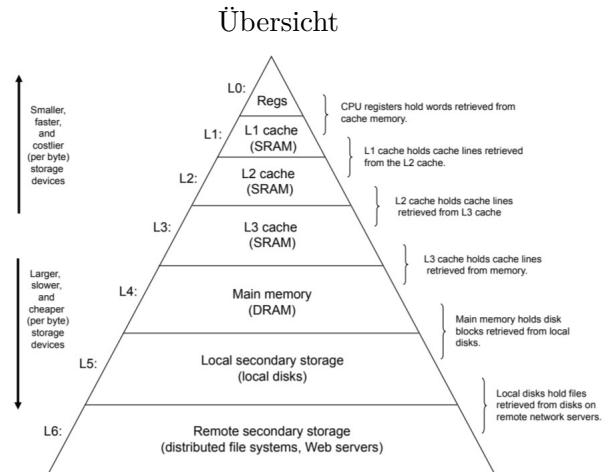


Abbildung: Quelle: [BO10, S. 48]

### Eigenschaften von Speichern

- Kosten und Zugriffszeit
- Geschwindigkeit und Kapazität
- Zugriffsverfahren
- Änderbarkeit und Daten
- Permanenz der Daten

### Kosten und Zugriffszeit

- in *Dollar/Bit* oder *Dollar/MByte*
- Kenngröße von Speichern
  - Zugriffszeit: durchschnittliche Zeit, um ein Wort aus Speicher zu lesen
  - Zykluszeit: minimal Zeit zwischen zwei Speicherzugriffen (Busprotokoll spielt eine Rolle)
  - Bandbreite (Datenübertragungsrate): maximale Datenmenge, die pro Sekunde übertragen werden kann in *Byte/Sec*

### 2.2 Speicher Kategorisierung

#### Zugriffsverfahren

- Zwei Möglichkeiten
  - wahlweiser Zugriff (Random Access)
  - serieller Zugriff
- Speicher mit Random Access
  - Register, Cache, Hauptspeicher
- Speicher mit seriellem Zugriff
  - Festplatten, optische Platten, Magnetband

## Änderbarkeit

- Read-only:
  - Speicher kann ausschließlich gelesen werden, Überschreiben ist nicht möglich
  - ROM-Halbleiterspeicher (Read-only Memory) Inhalt wird während Fabrikationsprozess festgelegt
- Read-write:
  - Inhalt kann während des Betriebs geändert werden
  - RAM-Halbleiterspeicher (Random-access Memory) Wird als Cache- Hauptspeicher verwendet, CD-R(W), DVD
- Read-mostly:
  - PROM-Halbleiterspeicher oder (E)EPROM: z.B. für BIOS verwendet

## Permanenz

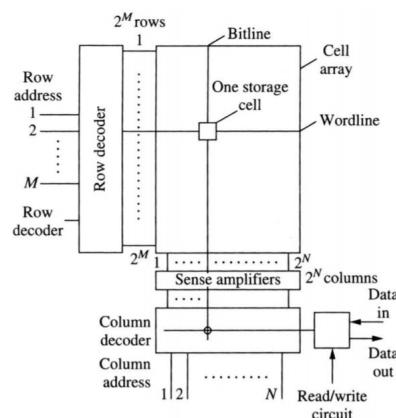
- Flüchtige Speicher (Register Hauptspeicher) vs Nicht flüchtige (ROM, PROM, EPROM Festplatte, optisches Speicherband)
- Flüchtige Speicher werden unterteilt:
  - dynamischer Speicher: Um Informationsverlust zu vermeiden muss die Spannung periodisch erneuert werden (Refreshing)
  - statische Speicher: kein Refreshing notwendig

## 2.3 Speichertechnologien

### Random-Access Memory (RAM)

- Statisches RAM (SRAM) und Dynmaisches RAM (DRAM)
- SRAM ist schneller (deutlich teurer) als DRAM
- SRAM  $\Rightarrow$  Cache
- DRAM  $\Rightarrow$  Hauptspeicher
- Typische (vor 10 Jahren) Verteilung
  - 12 MB SRAM
  - 4 GB DRAM (heute bis zu 1.5 TB)

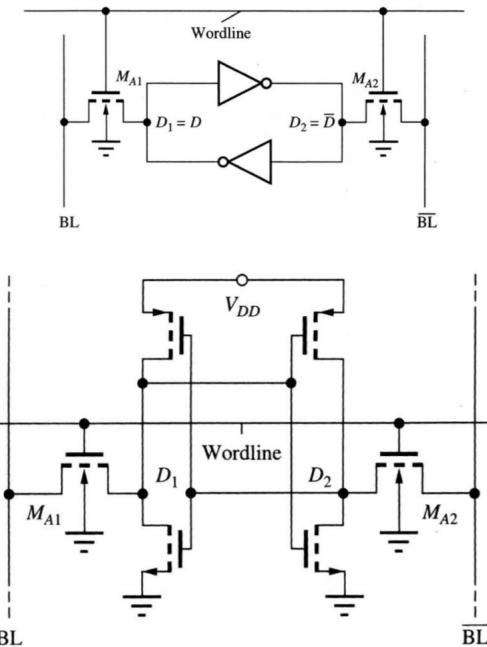
Organisation des Speichers



## Statisches RAM

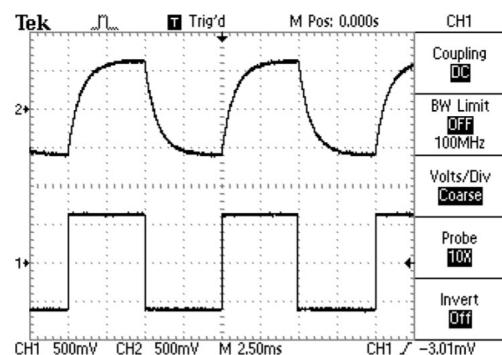
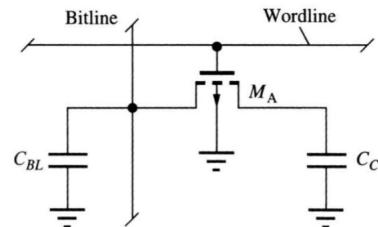
- Statisches (S)RAM speichert Informationen so lange wie die Spannungsversorgung angeschaltet ist
- Die gekoppelten Inverter haben zwei stabile Zustände  $\Rightarrow$  Bistabile Schaltung

Realisierung durch Inverter - 6T Zelle, 6 Transistoren



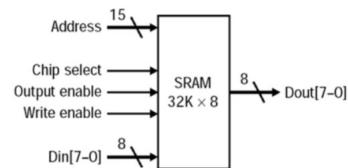
## Dynamisches RAM

- Speichert Informationen in Kondensator
- "1T Zelle"
- Der Kondensator entlädt sich und verliert damit Informationen
- **Refresh-Zyklus** notwendig, um Informationen aufzufrischen



## 2.4 Speicherorganisation

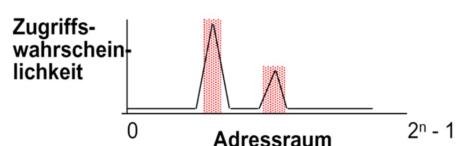
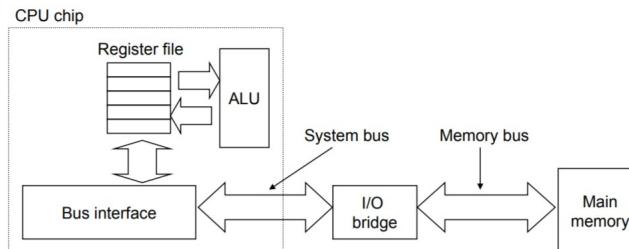
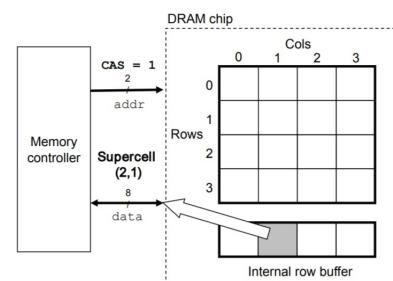
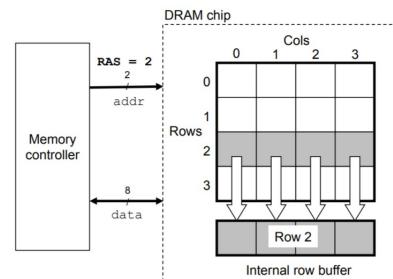
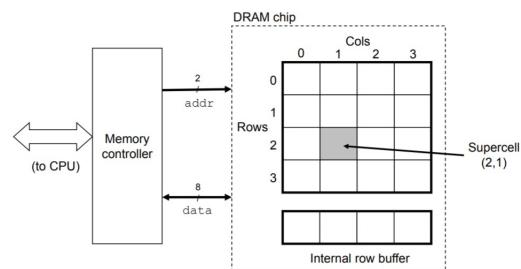
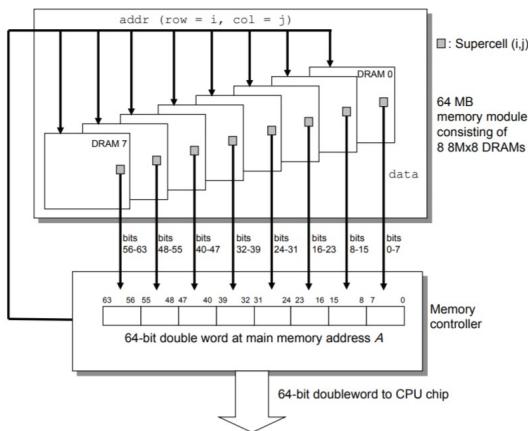
- Eigenschaften eines RAM-Chips sind durch zwei Größen angegeben
  - Anzahl adressierbarer Plätze
  - Breite (in Bit) jedes adressierbaren Platzes
- Beispiel 256K \* 1 SRAM
  - 256K Einträge mit 1 Bit Breite
  - $256K = 2^{18}$ , also 18 Adresseingänge sowie ein 1 Bit Dateneingang/-ausgang
- 32K \* 8 SRAM
  - 32K Einträge mit 8 Bit Breite
  - $32K = 2^{15}$ , also 15 Adresseingänge sowie 8 Bit Dateneingang/-ausgang



## Darstellung: 128 Bit 16 x 8 DRAM Chips

Lesezugriff:

- 1. Schritt Row Address Strobe (RAS)
  - Die ersten zwei Bits wählen eine Zeile (Row) aus, diese wird in den internen Zeilen Speicher (internal row buffer) geladen/gespeichert
- 2. Schritt Column Address Strobe (CAS)
  - Die nächsten zwei Bits wählen aus der gespeicherten Zeile (internal row buffer) die Spalte (column) aus



## 2.5 Lokalität

Das Lokalitätsprinzip stellt fest, dass Programme meistens einen relativ geringen Teil des Addressraums nutzen.

### Formen der Lokalität

- Zeitliche (temporale) Lokalität
  - Nach Zugriff auf einen bestimmten Datensatz wird mit großer Wahrscheinlichkeit bald erneut darauf zugegriffen
  - Beispiel: Schleifen (Indices etc.)
- Räumliche Lokalität
  - Nach dem Zugriff auf einen Datensatz wird mit großer Wahrscheinlichkeit auch auf einen Datensatz zugegriffen, der in unmittelbarer Nähe im Speicher steht
  - Beispiel: sequentielle Instruktionsfolge (ohne Sprünge)
  - Beispiel: Reihungen, Matritzen

## Vorteile

- Gut geschriebene Programme haben eine gute Lokalität
- Die Anwendung des Lokalitätsprinzip hat eine enorme Auswirkung auf die Performance eines Rechner-systes (Hardware/Software)
- Programme mit guter Lokalität laufen schneller, als Programme mit schlechter Lokalität
- Zweifache Unterscheidung:
  - Lokalität der Daten
  - Lokalität der Befehle
- Im Folgenden einige Beispiele für Programme mit guter und schlechter Lokalität

## Lokalität der Befehle

- Wie die Daten sind auch Befehle im Speicher abgelegt
- Auch für Befehle ist also das Lokalitäts Prinzip anwendbar
- Für die for Schleife der Beispiele gilt eine gute räumliche Lokalität
- Da der Schleifenkörper wiederholt ausgeführt wird, gibt es auch eine gute zeitliche Lokalität
- Bei der Programmierung an das Lokalitätsprinzip denken
- Lokalitätsprinzip lässt die Speicherhierarchie funktionieren

## 2.6 Lokalität - Beispiele

### Beispiel 01

- Elemente des Vektors werden sequentiell gelesen
- Beispielhaft Anordnung:
- Lokalität der Berechnungem:
  - schlechte zeitlich Lokalität (da auf jedes Element nur einmal zugegriffen wird)
  - gute räumliche Lokalität (da die Elemente nahe beieinander liegen)
- Lokalität der Schleife:
  - Schleifenindex (i) hat gute Zeitliche und räumliche Lokalität (da immer wieder auf die selbe Variable, und damit auch auf die selbe Stelle zugegriffen wird)

```
1 ...  
2 int sumvec (int v[N])  
3 {  
4     int i ,sum=0;  
5  
6     for ( i=0;i<N; i++)  
7         sum += v[ i];  
8     return sum;  
9 }  
10 ...
```

Beispielhafte Anordnung

Adresse	0	4	8	12	16	20	24	28
Inhalt	$v_0$	$v_1$	$v_2$	$v_3$	$v_4$	$v_5$	$v_6$	$v_7$
Zugriffsreihenfolge	1	2	3	4	5	6	7	8

## Beispiel 02

- Elemente des Vektors werden sequentiell gelesen
- Beispielhaft Anordnung:
- Lokalität der Berechnungem:
  - schlechte zeitlich Lokalität (da auf jedes Element nur einmal zugegriffen wird)
  - schlechte räumliche Lokalität (da die Elemente in falscher Reihenfolge gelesen werden, und "gesprungen" wird)
- Lokalität der Schleife:
  - Schleifenindex ( $i, j$ ) hat gute Zeitliche und räumliche Lokalität (da immer wieder auf die selbe Variable, und damit auch auf die selbe Stelle zugegriffen wird)

```

1 ...
2 int sumarrayrows( int a[M][N])
3 {
4     int i, j, sum = 0;
5
6     for (j=0;j<N; j++)
7         for (i=0; i<M; i++)
8             sum += a[i][j];
9     return sum;
10 }
11 ...

```

Beispielhafte Anordnung

Adresse	0	4	8	12	16	20
Inhalt	$a_{00}$	$a_{01}$	$a_{02}$	$a_{10}$	$a_{11}$	$a_{12}$
Zugriffsreihenfolge	1	3	5	2	4	6

## Beispiel 03

- Vergleich zeigt
  - clientssh-arm: DIM 80000
    - \* Zeilenweise: 38.5 s
    - \* Spaltenweise: 1m48.2 s
  - Raspberry Pi: DIM 10000
    - \* Zeilenweise: 1.5 s
    - \* Spaltenweise: 13.5 s
- Auch beim Schreiben hat eine zeilenweise **Traversierung** des Arrays Vorteile
- Achtung: Verhältnis der Beschleunigung hängt auch von der Problemgröße ab

Schreiben in ein Array, Zeilenweise

```

1 ...
2 #define DIM 22000
3
4 int main()
5 {
6     register long i, j;
7     static long matrix[DIM][DIM];
8
9     for(i= 0; i< DIM; i++)
10        for(j= 0; j< DIM; j++)
11            matrix[i][j]= 1;
12
13 }
14 ...

```

Schreiben in ein Array, Spaltenweise

```

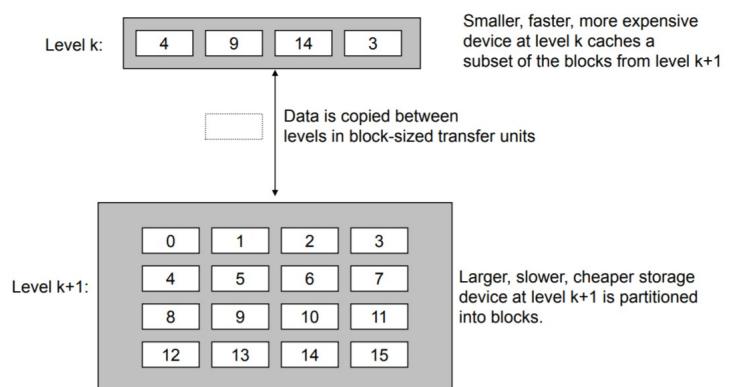
1 ...
2 #define DIM 22000
3
4 int main()
5 {
6     register long i, j;
7     static long matrix[DIM][DIM];
8
9     for(j= 0; j< DIM; j++)
10        for(i= 0; i< DIM; i++)
11            matrix[i][j]= 1;
12
13 }
14 ...

```

## 2.7 Cache

### Prinzip des Caches

- Cache ist ein kleiner, schneller SRAM
- *caching* = Benutzung eines Caches
- Zentrale Idee der Speicherhierarchie
- Ein schnellerer und kleinerer Speicher auf dem Level  $k$  fungiert als Cache für einen langsameren und größeren Speicher auf  $k+1$  ( $k+1$  bezeichnet hierbei eine niedrigere Ebene als  $k$ )
- Vergleich mit Speicherhierarchie zeigt, dass das Mehrfach angewendet wird
- Deshalb ist das Lokalitätsprinzip für eine sinnvolle Nutzung der Speicherhierarchie notwendig



## Cache Hit

- Wenn ein Programm Daten von einem Objekt  $d$  braucht, wird geschaut, ob  $d$  in einem der Blöcke auf Level  $k$  gespeichert ist
- Wenn  $d$  in dem Cache auf Level  $k$  gefunden wird, nennt man das einen **Cache Hit**
- Das Programm liest dann  $d$  direkt aus dem Level  $k$
- Da Level  $k$  schneller ist als  $k + 1$  ergibt sich dadurch ein Geschwindigkeitsvorteil
- Bemerkung: auch wenn L1, L2 und L3 als Cache alle als SRAM ausgeführt sind, gibt es trotzdem Geschwindigkeitsunterschiede

## Cache Miss

- Wenn ein Programm Daten von einem Objekt  $d$  braucht, wird geschaut, ob  $d$  in einem der Blöcke auf dem Level  $k$  gespeichert ist
- Wenn  $d$  in dem Cache auf Level  $k$  **nicht** gefunden wird, nennt man das einen **Cache Miss**
- Bei einem Cache Miss holt der Cache auf dem Level  $k$  den Block mit  $d$  von dem Cache auf Level  $k + 1$
- Möglicherweise muss dann ein existierender Block (wenn Cache voll) überschrieben werden
- Das s.g. *Ersetzen* kann unterschiedlich erfolgen:
  - Zufallsersetzung
  - Least-recently used (LRU) Ersetzung
- Bedeutung der Lokalität beachten

## Speicherhierarchie

- Caching in Modernen Rechnersystemen (siehe Abbildung)
- Die Blöcke werden größer
- Die Verzögerung (Latenz) wird größer

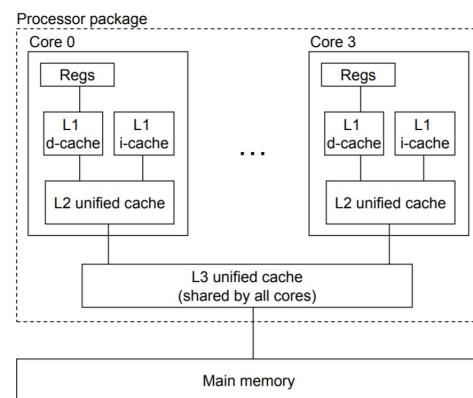
Typ	What cached	Latenz (cycles)
CPU Register	4 Byte oder 8 Byte	0
L1 Cache	64 Byte Block	1
L2 Cache	64 Byte Block	10
L3 Cache	64 Byte Block	30
...	...	...
Platten Cache	Disk Sektors	100000

## Cachehierarchie ARM/Intel Core i7

- Von-Neumann und Harvard differenzierung nicht komplett möglich
- L1 Cache Harvard, L2 Von-Neumann

Cache Typ	Zugriffszeit (cycles)	Cache Größe
L1 i-Cache	4	32 KB
L1 d-Cache	4	32 KB
L2 unified cache	11	256 KB
L3 unified cache	30-40	8 MB

Charakteristische Werte - Intel



## ARM

- Getrennte Caches für Daten (d-cache) und Instruktionen (i-cache) auf Level 1
- ARM Cortex - A53 MPCore Processor, Technical Reference Manual

## Intel Core i7

- Getrennte Caches für Daten (d-cache) und Instruktionen (i-cache) auf Level 1
- Alle SRAM Cache Speicher sind auf dem CPU Chip (on die)

### 3 Maschinennahe Programmierung

#### 3.1 Phasen der Übersetzung

Phasen der Übersetzung

- Das C Programm ist für den Menschen verständlich
- Zum Ausführen auf Rechnersystem muss es in Maschinenbefehle übersetzt werden
- Beispiel: gcc -o hello hello.c

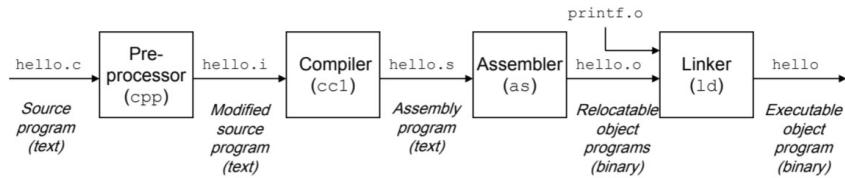


Abbildung: Quelle: [BO10, S. 39]

##### Phase Eins - Preprocessor

- hello.c → hello.i
- Aufarbeiten von Direktiven (Befehle mit # )
- Vorbereitung zur assemblierung

##### Phase Zwei - Compiler

- hello.i → hello.s
- C-Programm in Assemblerprogramm

##### Phase Drei - Assembler

- hello.s → hello.o (Maschinensprache)
- Ergebnis ist ein Objektprogramm

##### Pashe Vier - Linker/Binder

- hello.o → hello (Ausführbares Programm)
- Zusammenfügen verschiedener Module
- Zum Beispiel externe Funktionen die importiert werden (printf aus stdio.h)

Bei der Ausführung des Programms wird in der Shell der Befehl `./hello` ausgeführt. Die Shell ist ein Kommandozeileninterpreter der dann das Programm `hello` auf der Festplatte ausführt.

### 3.2 Ausführung eines Programmes

Wir betrachten die Ausführung eines Programmes, als Beispiel `hello` mittels der Shell an einer Beispielhaftendarstellung eines Rechnersystems.

Shell liest zunächst die Zeichen des Kommandos in die Register und speichert den Inhalt dann im Hauptspeicher ab.

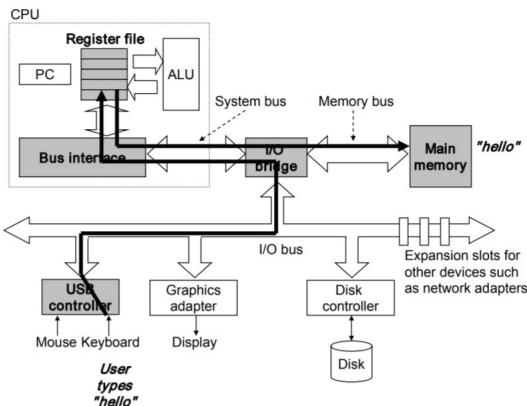


Abbildung: Quelle: [BO10, S. 45]

Schrittweises Kopieren der Befehle und Daten von Festplatte in den Hauptspeicher (hier wird Direct Memory Access (DMA) benutzt).

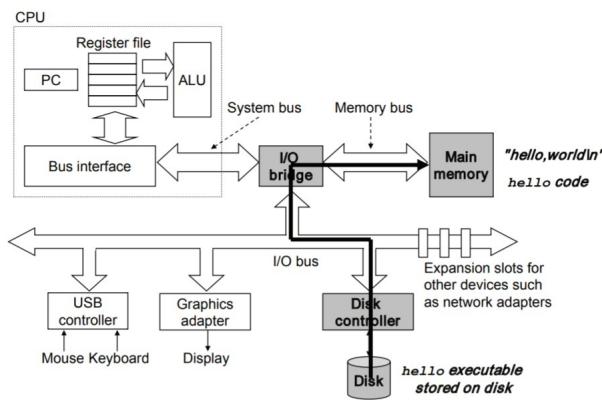


Abbildung: Quelle: [BO10, S. 45]

Ausführen der Maschinenbefehle des hello Programms

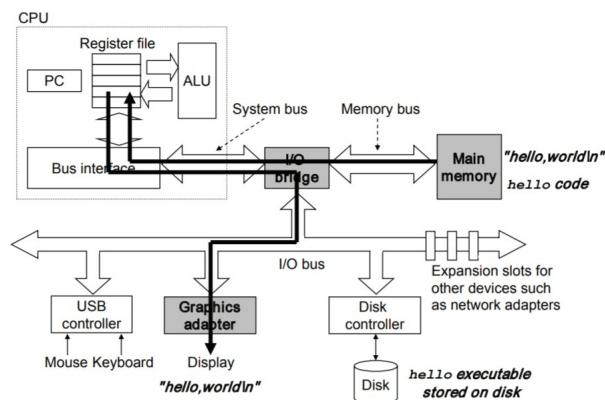


Abbildung: Quelle: [BO10, S. 46]

### 3.3 Compilieren, Assemblieren und Linken

**Compilieren** Diskussion der Assemblerprogramme

- Die Übersetzung in den verschiedenen Phasen ist nicht eindeutig, ein Programm kann zu mehreren Übersetzungen führen
- Nicht alle Elemente werden in Assembler abgebildet, manche, z.B. triviale, Dinge können "weggelassen" werden
- Beispiel:
  - Nicht jede lokale Variable wird gespeichert, manche werden nur in Register realisiert
  - Manches wird direkt bei der Übersetzung berechnet
  - ...

**Assemblieren** ist der Prozess des Übersetzens von Assemblerprogrammen (Mnemonics) in Maschinensprache:

Ein Assembler ist ein Programm, das die Aufgabe hat, Assemblerbefehl in Maschinecode zu transformieren und dabei:

- Symbolische Namen (z.B. Labels bei Rücksprungmarken) Maschinenadressen zu zuweisen
- und eine oder mehrere Objektdatei(en) zu erzeugen.
- **Crossassembler:** Assembler läuft aus System X, generiert Maschinencode für Plattform Y
- **Disassembler:** Übersetzung von Maschinensprache in Assemblersprache, üblicherweise Verlust von Kommentaren und symbolischen Namen

Arbeitsweise und Probleme:

- 1. Schritt:
  - Auffinden von Speicherpositionen mit Marken. Beziehungen zwischen symbolischen Namen und Adressen werden bekannt
  - Übersetzung des Assemblerprogrammbefehls in legale Instruktion (mittels Opcodes etc.)
- Probleme beim 1. Schritt, Beispiel:  
Das Programm wird Zeile für Zeile abgearbeitet, bei Vorwärtsprüngen kann der Zielmarker also noch unbekannt sein. Daher wird das Programm zwei mal durchlaufen, beim ersten mal werden alle Maschinenadressen zugeordnet, beim zweiten mal wird der Code erzeugt.
- 2. Schritt: Objektdateien werden erzeugt. Enthält Maschinencode, Daten und Verwaltungsinformation  
Ist im allgemeinen nicht ausführbar, da sie auf andere Dateien verweist (nächster Schritt: Linker)
- Probleme beim 2. Schritt:
  1. Fall:
    - Assembler verwendet absolute Adressen
    - Laden ist unmittelbar möglich, aber:
    - Verschieben des Programms im Speicher ist nicht möglich
  2. Fall:
    - Assembler verwendet relative Adressen
    - Es werden mehr, oder genau eine Objektdatei erzeugt, aber:
    - weitere Transformationsschritte sind notwendig ⇒ Linker

## Linker/Binder

- **Definiton Linker**

Der Linker verbindet oder linked verschiedene Objektfiles und macht aus ihnen ausführbare Objektprogramme indem noch offene Referenzen aufgelöst werden

- Das Objektprogramm kann durch einen Lader ausgeführt werden

- **Definition Lader**

Ein loader (Lader) Ist ein Systemprogramm, das Objektprogramme in den Speicher lädt und ggf. ihre Ausführung initialisiert

- Dazu wird das Objektprogramm in den Speicher kopiert

Arbeitsweise eines Laders:

- Ein Programmmodul (Lademodul) wird beginnend mit bei einer Startadresse in den Hauptschpeicher geladen
- Varianten:
  - absolutes laden (absolute loading)
  - relatives laden (relocatable loading)
  - dynamisches laden zur Laufzeit (dynamic run time loading)

## 3.4 Kontolloperationen - Statusbits

### Carry und Overflow

Statusflags (die wichtigsten):

- C - Carryflag (Übertragsflag)
- Z - Zeroflag (Nullflag)
- N - Negativflag (Vorzeichenflag)
- V - Overflowflag (Überlaufflag)

Verwendungszweck:

- Z: Hochsprache  $\rightarrow t == 0$
- N: Hochsprache  $\rightarrow t < 0$

Ein Carrybit tritt auf, wenn bei einer Rechnung Bits überlaufen, also über die Wortlänge heraus, die Rechnung aber korrekt ist. Z.B.:

0101 5  
1111 -1  
E: 1 0100 4

0101 5  
0011 3

Ein Overflowbit tritt auf, wenn bei Berechnung durch Übertrag ein falsches Ergebnis entsteht: E: 1000 8

Ein Negativflag tritt auf, wenn die Zahl negativ ist, also in 2K Darstellung das höchstwertige bit 1 ist.

## 4 Konzepte der maschinennahen Programmierung

### 4.1 Allgemeins

Operanden (immediates) können bei manchen Befehlen verwendet werden.  
Sie werden durch ein # vorangegangen. Ein Direktwert ist eine 12-Bit breite Zahl im Zweierkomplement.

Maschinenbefehle (Mnemonics)

### 4.2 Lesen und Schreiben

Befehle:

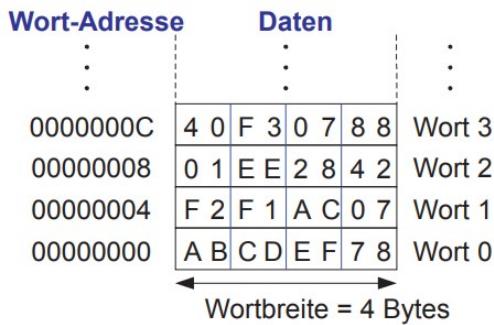
- mov: Move (r, r/# )
- ldr: Load (r, [Adresse] ) - Adresse: r, #
- str: Store (r, [Adresse] ) - Adresse: r, #
- Können auch auf bytes arbeiten: ldrb, strb

Befehle:

- mov r5, # 0
- ldr r7, [r5, + 0xC]

Adressarithmetik:

- Basisadresse (r5) plus Distanz (offset) (0xC)
- Nach Abarbeitung r7 == 0x40F30788



Das Schreiben funktioniert auf analoge weise.

### 4.3 Nutzen des Hauptspeichers

- Bisher wurden Daten direkt in Register geschrieben (z.B. mov)
- Man kann Werte auch direkt durch Angabe der Variablennamen laden
- Voraussetzung:

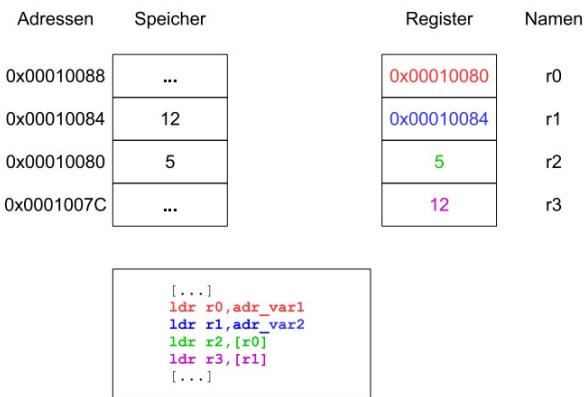
```
.data /* Daten Bereich */
var1: .word 5 /* Var1 im Speicher, Wert 5 */
... main: ...
adr_var1: .word var1 /* Adresse von Var1 */
```
- Dies geht direkt:  
`ldr r0, var1`

- Oder indirekt über die Adressen:

```
ldr r0, adr_var1 /* laedt Adresse von var1 in r0 */
ldr r1, [r0] /* Lade inhalt von Adresse r0 in r1 */
```

Eine vereinfachte Syntax stellt dabei =var1 dar:

```
ldr r0, =var1 /* laedt Adresse von var1 in r0 */
ldr r1, [r0] /* Lade inhalt von Adresse r0 in r1 */
```



## 4.4 Sprünge / Verzweigungen

Befehle:

- b target: branch target - springe zu target
- beq target: branch equals - bedingter Sprung
- cmp: Compare - vergleicht zwei register, für beq oder bne

Bei Sprüngen wird immer zu einem target/label gesprungen. Dies sind Namen für Stellen (Adressen) im Programm. Sie müssen anders als Mnemonics heißen Labels müssen mit Doppelpunkt abgeschlossen werden. Optional kann durch einen Vergleich (cmp) ein Bedingter Sprung ausgeführt werden. Dabei setzt cmp die zu vergleichenden Register fest, und der branch befehl handelt abhängig von dem Vergleich.  
Bei äquivalenten ausdrücken in Hochsprachen wird meist das gegenteil abgefragt.

Unbedingt

```
1 ...
2 /* ARM – Assemblersprache */
3 add r1,r2,#17 /* r1 = r2 + 17 */
4 b target /* branch zu target */
5 orr r1,r1,r3 /* nicht ausgefuehrt */
6 and r3,r1,#0xFF /* nicht ausgefuehrt */
7
8 target: /* Positionsmarkierung (label) */
9 sub r1,r1,#78 /* r1 = r1 - 78 */
10 ...
```

Bedingt

```
1 ...
2 /* ARM – Assemblersprache */
3 mov r0,#4 /* r0 = 4 */
4 add r1,r0,r0 /* r1 = r0 + r0 = 8 */
5 cmp r0,r1 /* setze Flags auf Basis der */
6 /* Rechnung r0 – r1 = -4 */
7 /* N=1, da Ergebnis negativ */
8 /* Statusflags: N=1, Z=0, C=0, V=0 */
9 beq there /* kein Sprung (Z != 1) */
10 orr r1,r1,#1 /* r1 = r1 or 1 = 9 */
11
12 there:
13 add r1,r1,#78 /* r1 = r1 + 78 = 87 */
14 ...
```

Beispiele für Anweisungen:

```

1 /* Hochsprache */
2 if (apples == oranges)
3   f = i + 1;
4 else
5   f = f - i;

1 /* ARM – Assemblersprache */
2 /* r0=apples; r1=oranges, r2=f, r3=i */
3 cmp r0,r1
4 bne L1
5 add r2,r3,#1
6 b L2
7 L1:
8 sub r2,r2,r3
9 L2:

1 /* Hochsprache // addiere Zahlen von 0 bis 9 */
2 int sum = 0; int i;
3 for (i=0; i < 10; i=i+1){
4   sum = sum + i;}

1 /* ARM – Assemblersprache r0 = i, r1 = sum */
2 mov r1,#0
3 mov r0,#0
4 FOR:
5 cmp r0,#10
6 bge DONE
7 add r1,r1,r0 /* sum = sum + i */
8 add r0,r0,#1 /* i=i+1 */
9 b FOR
10 DONE:

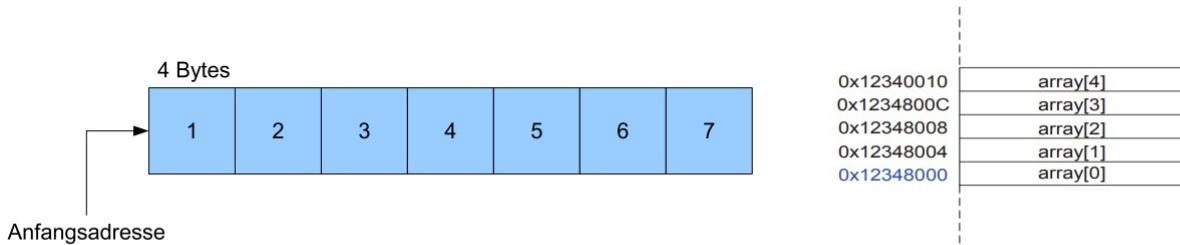
```

## 4.5 Datenfelder (Arrays)

Prinzip:

Ein Array besteht aus mehreren Worten, dies ist nützlich um aus einer großen Zahl von Daten gleichen Typs zuzugreifen

Auf die Elemente wird über einen Index zugegriffen. Konkret bedeutet das, dass auf die Basisadresse immer ein Offset addiert wird, genau eine Wortlänge, um zum nächsten Element zu kommen:



Zugriff:

Der Zugriff auf das Array verläuft mittels Adressarithmetik. Dabei wird auf die Basisadresse immer das  $i$ -fache eines Offsetes addiert:

```

1 mov r0, #0x1400000000 /* Basisadresse in r0 */
2 mov r1, #0 /* r1 = 0, Laufvariable i */
3 lsl r2, r1, #2 /* r2 = i * 4, leftshift logic */
4 ldr r3, [r0, r2] /* r3 = array[i] */

```

## 4.6 Unterprogramme

### Einführung

- Helfen bei strukturierter Programmierung
- Aus einem Programm (Hauptprogramm) wird eine Teilprogramm ausgeführt
- Makrotechnik vs Unterprogrammtechnik

Zu beachten ist:

- Wie erfolgt die Parameterübergabe und wie werden Ergebnisse ausgetauscht
- Sichtbarkeit von Variablen, global vs lokal
- Jede Variable besitzt während ihrer Lebensdauer einen Speicherplatz

Regeln:

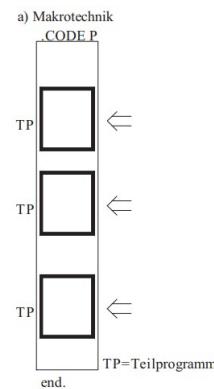
- Aufrufer:

- Übergibt Argumente (aktuelle Parameter) und Aufrufenen
- Springt Aufrufenen an
- Aufgerufener:
  - Führt Funktion / Prozedur aus
  - Gibt Ergebnis (Rückgabewert) und Aufrufer zurück (für Funktion)
  - **Wichtig:** Darf keine Register oder Speicherstellen überschreiben die im Aufrufer genutzt werden  
⇒ Stacks als Lösung

## Makro- vs Unterprogrammtechnik

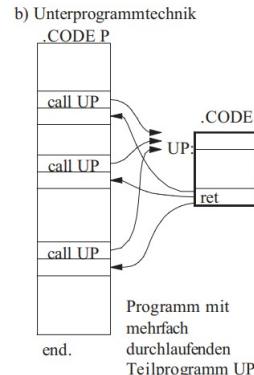
Makrotechnik:

- Das Teilprogramm wird an benötigten Stellen einkopiert
- Dazu wird dem TP ein, dem s.g. Makro, ein Name zugeordnet (Makroname)
- Das passiert durch die s.g. Makrodefinition
- An den zu einkopierenden Stellen wird dann der Makroname genannt
- ⇒ Makroaufruf



Unterprogrammtechnik:

- Hier ist das Unterprogramm nur einmal im Code vorhanden
- An den Stellen an denen das Unterprogramm ausgeführt werden soll erfolgt der Aufruf durch einen Sprungbefehl
- Am Ende erfolgt die Rückkehr in das aufrufende Programm
- Rückkehradresse wird gespeichert



## Beispiel in Assembler

```

1 /* r4 = y */
2 main:
3 mov r0,#14 /* Argument 0 = 14 */
4 mov r1,#3 /* Argument 1 = 3 */
5 mov r2,#4 /* Argument 2 = 4 */
6 mov r3,#5 /* Argument 3 = 5 */
7 bl diffosums /* Funktionsaufruf */
8 mov r4, r0 /* y = Rueckgabewert */
9
10 diffosums:
11 add r8,r0,r1
12 add r9,r2,r3
13 sub r4,r8,r9
14 mov r0,r4 /* Lege Rueckgabewert in r0 ab */
15 mov pc,lr /* Ruecksprung zum Aufrufer */

```

- Nach Abarbeitung der Zeilen 3 - 6 sind die zu übergebenden Argumente in den Registern r0 - r3 abgelegt.
- Zeile 7 ruft das Unterprogramm mit dem Befehl bl (branch & link) auf. Dieser Befehl sorgt dafür, dass die Rückkehradresse 8 in das Register r14 (lr) geschrieben wird.
- Das Unterprogramm beginnt an der Adresse 10. Nach Durchführung der arithmetischen Operationen (Zeilen 11 - 13) wird das Ergebnis in das Register r0 geschrieben (Zeile 14). r0 wird auch *return value* Register genannt.
- In Zeile 15 wird die zuvor gespeicherte Rückkehradresse (8) aus dem Register r14 (lr) in das Register r15 (pc) kopiert. Der nächste auszuführende Befehl ist an der Adresse 8 zu finden und lautet mov r4, r0

## 4.7 Stacks

### Einleitung

- Stacks (Stapel- Kellerspeicher) sind Speicher zur temporären Nutzung
- Bekannt aus AuD - last in first out LIFO
- Dynamische Speicherzuweisung
- Wächst bei ARM nach unten (von hohen zu niedrigen Speicheradressen)
- Stapelzeiger ("stack pointer"): sp (r13)
- zeigt auf zuletzt auf dem Stapel abgelegtes Datenelement

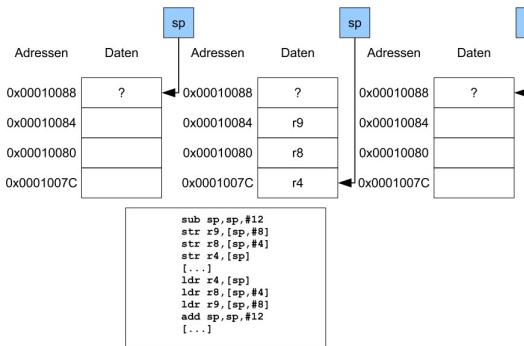
### Implementierung in Teilprogramme

- TPs dürfen keine Seiteneffekte haben
- Problem: doffosums überschreibt die drei Register r8, r9, r4

```

1 /* r4 = y */
2 diffofsums:
3 add r8,r0,r1
4 add r9,r2,r3
5 sub r4,r8,r9
6 mov r0,r4 /* Lege Rueckgabewert in r0 ab */
7 mov pc,lr /* Ruecksprung zum Aufrufer */

```



```

1 diffofsums:
2 sub sp,sp,#12 /* Speicher auf Stack reservieren */
3 str r9,[sp,#8] /* sichern von r9 auf Stack */
4 str r8,[sp,#4] /* sichern von r8 auf Stack */
5 str r4,[sp] /* sichern von r4 auf Stack */
6
7 add r8,r0,r1 /* Rechnung durchfuehren */
8 add r9,r2,r3
9 sub r4,r8,r9
10 mov r0,r4 /* Lege Rueckgabewert in r0 ab */
11
12 ldr r4,[sp] /* herstellen von r4 */
13 ldr r8,[sp,#4] /* herstellen von r8 */
14 ldr r9,[sp,#8] /* herstellen von r9 */
15 add sp,sp,#12 /* Speicher auf Stack freigeben */
16
17 mov pc,lr /* Ruecksprung zum Aufrufer */

```

Vor Aufruf des Unterprogramms muss die Rücksprungadresse gespeichert werden, z.B.:

- Mittels mov:
 

```

mov r10, lr /* sichern der Rücksprungadresse in r10 */
bl diffofsums /* Funktionsaufruf */
mov r4, r0 /* y = Rückgabewert */
mov lr, r10 /* herstellen der Rücksprungadresse */

```
- Mittels pop und push als Pseudoinstruktionen:
 

```

push {lr} /* sichern der Rücksprungadresse in r10 */
bl diffofsums /* Funktionsaufruf */
mov r4, r0 /* y = Rückgabewert */
pop {lr} /* herstellen der Rücksprungadresse */

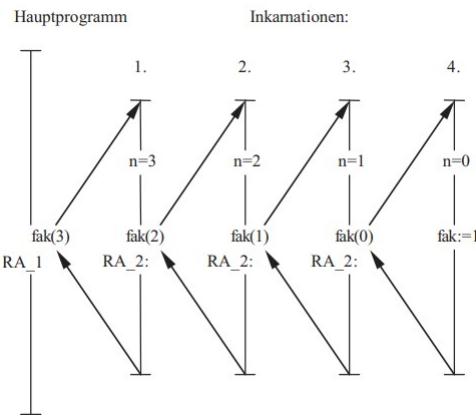
```

## 4.8 Rekursion

### Prinzip

Das Prinzip hinter der Rekursion, ist das wiederholte aufrufen von Funktionen in sich selber. Konkret wird die Funktion im Hauptprogramm aufgerufen und ruft sich unter bestimmten Bedingungen selbst wieder auf. Tritt der Fall ein, dass sie sich nicht selbst aufruft, wird das Ergebnis an die Aufrufende Funktion zurückgegeben. Diese kann ihr Ergebnis wiederum zurückgeben usw.

- Die Abbildung der Inkarnation zeigt, dass es zwie verschiedene Rücksprungadressen gibt
- RA\_1 = Adresse im Hauptprogramm
- RA\_2 = Adresse im Unterprogramm (für alle Rekursionstufen gleich)



Beispielprogramm mit Rekursion:

```

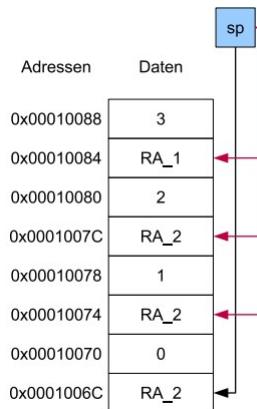
push {lr}
mov r0, #3 /* Fak von 3 */
bl fak
mov r1, r0 /* Ergebnis von fak in r1
...
fak:
sub sp, sp, #8 /* Speicher Stack reservieren */
str r0, [sp, #4] /* sichern von r0 auf stack (3) */
str lr, [sp] /* sichern Rücksprungadresse */
cmp r0, #1 /* Rekursionsende prüfen */
blt else
sub r0, r0, #1 /* n = n - 1 */
bl fak /* Funktionsaufruf */
ldr r1, [sp, #4] /* laden von n */
mul r0, r1, r0 /* fak (n-1)*n */
fin:
ldr lr, [sp] /* laden Rückkehradresse
add sp, sp, #8 /* Speicher Stack freigeben */
bx lr /* Sprung zum Aufrufer */
else:
mov r0, #1 /* Rekursionsanker */
b fin

```

```

1 #include<stdio.h>
2 int fak (int n)
3 {
4     if (n>=1) {
5         return fak (n-1) * n;
6     else {return 1; } /* 0! = 1 */
7 }
8 int main (void)
9 {
10 int n = 3; /* 3! = 6 */
11 int z; /* lokale Variable */
12 z = fak (n);
13 printf("Fakultaet %d\n", z);
14 return 0;
}

```



## 4.9 Zusammenfassung Unterprogrammaufrufe

- Aufrufer
  - Lege Aufrufparameter in Register oder Stack ab
  - Sichere notwendige Register auf Stack (z.B. lr)
  - bl Aufgerufener
  - Stelle gesicherte Register wieder her
  - Hole evtl. Rückgabewert aus r0 (bei Funktionen)
- Aufgerufener
  - Sichere zu erhaltende verwendete Register auf Stack

- Führe Berechnung des Unterprograms aus
  - Lege Rückgabewert in r0
  - Stelle gesicherte Register wieder her
  - Rücksprung zum Aufrufer: mov pc, lr
- Unter Aufrufkonvention versteht man die Methode, mit der einem Unterprogramm Daten übergeben werden

## 5 Gleitkommazahlen

### 5.1 Zahlendarstellung - Visualisierung

#### Darstellung Reeller Zahlen

- Zwei Arten für reelle Zahlen
  - Festkommadarstellung
  - Gleitkommazahlen
- Zur Festkommadarstellung
  - Man lässt bei der internen Darstellung das Komma weg und merkt sich wo es stehen müsste
  - Zum Programmbeginn wird die Kommastelle der entsprechenden Variable definiert und bleibt dann während des Programmablaufs fest

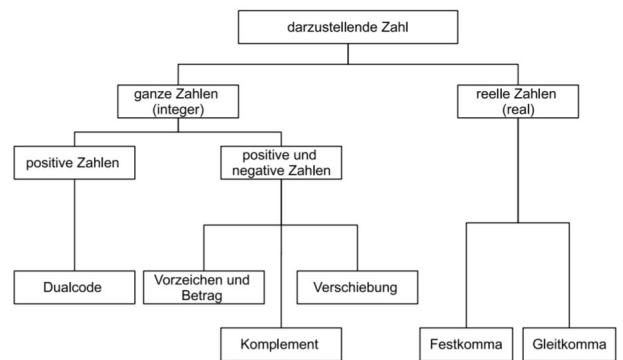
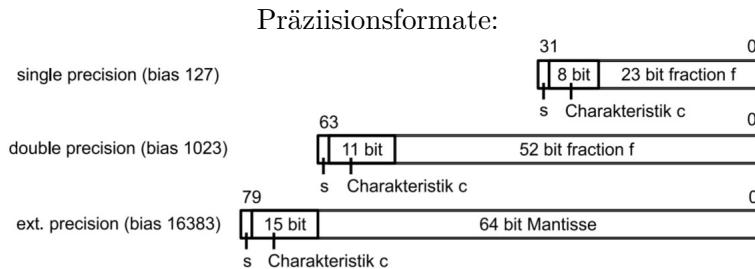


Abbildung: Zahlendarstellung in Rechnersystemen

### 5.2 Gleitkommazahlen nach ANSI/IEEE 754

- Bei Gleitkommadarstellung (halbarithmetisch Darstellung) ist die Kommastelle Bestand der Zahl und kann sich im Laufe der Rechnung verändern
- Durch standardisierte Schreibweise/Darstellung braucht man Komma nicht explizit angeben
- Floatingpoint Darstellung ist standardisiert nach ANSI/IEEE 754
- Hat sich weltweit durchgesetzt und dient als Grundlage
- 1985 verabschiedet, 2008 gab es eine Revision



- Besonderheiten: Die Null kann (durch Normalisierung) nicht dargestellt werden
- Unendliche Zahlen, oder keine Zahlen, sollen auch dargestellt werden

Normalisiert	$\pm$	$0 < \text{Charakteristik} < \max$	Beliebiges Bitmuster
Nicht normalisiert	$\pm$	0	Beliebiges Bitmuster $\neq 0$
Null	$\pm$	0	0
Unendlich	$\pm$	111...1 (max)	0
Keine Zahl <sup>2</sup>	$\pm$	111...1 (max)	Beliebiges Bitmuster $\neq 0$

#### Beispiel Gleitkommazahl

Eine Gleitkommazahl  $z$  wird als  $z = \pm m * b^{\pm e}$  dargestellt, wobei  $m$  die Mantisse,  $e$  der Exponent und  $b$  die Basis des Exponenten ist.

Zur Darstellung einer Gleitkommazahl wird folgendes benötigt:

- Vorzeichen der Mantisse
- Betrag der Mantisse mit Komma
- Basis (ganze Zahl)
- Vorzeichen des Exponenten
- Betrag des Exponenten (ganze Zahl)

Um dies zu vereinfachen gibt es den ANSI/IEEE Standard:

- 1. Schritt - Zahl wird normalisiert:

$$654,321 \rightarrow 6,54321 * 10^2$$

$$0,006543 \rightarrow 6,543 * 10^{-2}$$

$$1001,011 \rightarrow 1,001011 * 2^3$$

$$0,011011 \rightarrow 1,1011 * 10^{-2}$$

Definition: Eine Gleitkommazahl heißt normalisiert, falls  $b > |m| \geq 1$

- 2. Schritt - Charakteristik

Die Charakteristik, oder auch biased Exponent, ergibt sich durch die Addierung des Exponenten und des bias:  $\text{Charakteristik} = \text{biasedExponent} = \text{Exponent} + \text{bias}$

- 3. Schritt - Zusammensetzen

Nun kann man die Zahl mit allen gegebenen Informationen Zusammensetzen

#### • Beispiele:

- Beispiel: für einen Exponenten mit  $n = 8$ :

- ▶ → Charakteristik: 1 bis 254 (0 und 255 haben spezielle Bedeutung)
- ▶ → Exponent: -126 bis +127
- ▶ → Bias: 127

$$\begin{aligned} x &= 3,5 = 1,75 \cdot 2^1 = 1,75 \cdot 2^{128-127} \\ &= 1,110\ 0000\ 0000\ 0000\ 0000\ 0000 \cdot 2^{128-127} \\ &= (0|128|0,75) \\ &= (0|1000\ 0000|110\ 0000\ 0000\ 0000\ 0000) \end{aligned}$$

- Beispiele:

- ▶ Exponent = 45 → Charakteristik =  $45 + 127 = 172$
- ▶ Exponent = -28 → Charakteristik =  $-28 + 127 = 99$

$$\begin{aligned} x &= -11,375 = -1011,011 \cdot 2^0 = -1,011011 \cdot 2^3 \\ &= (1|130|0,421875) \\ &= (1|1000\ 0010|011\ 0110\ 0000\ 0000\ 0000) \end{aligned}$$

#### • Vorzeichen – Exponent – Mantisse

### 5.3 Verarbeitung reeller Zahlen in ARM

- Bemerkung:

Bzgl. der Nutzung von Gleitkommazahlen/Gleitkommaoperationen in Eingebetteten Systemen gibt es auch die Auffassung, dass man diese aufgrund von Geschwindigkeitsgründen vermeiden sollte. Ausnahme bildet das Vorhandensein einer Hardware unterstützte Einheit für Gleitkommazahlen. Hat man keine Hardware unterstützte Gleitkommaoperationen Einheit, kann man dies auch mit Integer Register simulieren.

### Code Analyse

Addition zweier Gleitkommazahlen - float

```

1 /* Addition */
2 #include <stdio.h>
3
4 int main(){
5
6 float p = 5.4;
7 float q = 12.6;
8 float result = p + q;
9 printf("result ist %f\n", result)
10 return 0;
11 }

```

```

1    main:   push    {fp, lr}
2        add     fp, sp, #4
3        sub     sp, sp, #16
4        ldr     r3, .L3
5        str     r3, [fp, #-8]    @ float
6        ldr     r3, .L3+4
7        str     r3, [fp, #-12]   @ float
8        vldr.32 s14, [fp, #-8]
9        vldr.32 s15, [fp, #-12]
10       vadd.f32      s15, s14, s15
11       vstr.32 s15, [fp, #-16]
12       vldr.32 s15, [fp, #-16]
13       vcvt.f64.f32    d7, s15
14       vmov    r2, r3, d7
15       ldr     r0, .L3+8
16       bl      printf
17       mov     r3, #0
18       mov     r0, r3
19       sub     sp, fp, #4
20       @ sp needed
21       pop    {fp, pc}
22 .L4:    .align 2
23 .L3:    .word 1085066445
24 .L3:    .word 1095342490

```

- Speicherung der Konstanten im Dezimalsystem (Binärzahl nach ANSI)
  - $p = 5.4 = -1085066445$
  - $q = 12.6 = -1095342490$
  - 0100 0000 1010 1100 1100 1100 1101
- Neue Register und Befehle
  - vldr.32 (load 32 bit?), vadd.32 (add 32 bit?),...
  - s14 (single precision?), d7 (double precision?)

### Addition zweier Gleitkommazahlen - double

```

1 /* Addition */
2 #include <stdio.h>
3
4 int main(){
5
6 double p = 5.4;
7 double q = 12.6;
8 double result = p + q;
9 printf("result ist %f\n", result);
10 return 0;
11 }

```

```

1    main:   push    {fp, lr}
2        add     fp, sp, #4
3        sub     sp, sp, #24
4        ldr     r2, .L3
5        ldr     r3, .L3+4
6        strd   r2, [fp, #-12]
7        ldr     r2, .L3+8
8        ldr     r3, .L3+12
9        strd   r2, [fp, #-20]
10       vldr.64 d6, [fp, #-12]
11       vldr.64 d7, [fp, #-20]
12       vadd.f64      d7, d6, d7
13       vstr.64 d7, [fp, #-28]
14       ldrd   r2, [fp, #-28]
15       ldr     r0, .L3+16
16       bl      printf
17       ...
18 .L3:    .word  -1717986918
19 .L3:    .word  1075157401
20 .L3:    .word  858993459
21 .L3:    .word  1076441907

```

- Speicherung der Konstanten im Dezimalsystem (Binärzahl nach ANSI)

Beachte: Dabei werden zwei 32 Bit Register verwendet um ein 64 Bit Register darzustellen

  - $p = 5.4 = -17179869181075157401$
  - $q = 12.6 = -8589934591076441907$
- Neue Befehle und Register
  - Ähnliche Befehle wie bei float Berechnungen, angepasst auf double Werte

## 5.4 Studium der Datenblätter

- Zwei Möglichkeiten Gleitkommazahlen zu verarbeiten:
  - eine Funktionseinheit NEON
  - eine Funktionseinheit VFP
- Zusätzliche Register zur behandlung von Gleitkommazahlen
  - s0 - s31 (Single Precision Register)
  - d0 - d31 (Double Precision Register)
  - q0 - q15 (128 Bit (2\*Double) Precision Register)
- Beide Funktionseinheiten benutzen die gleichen Register

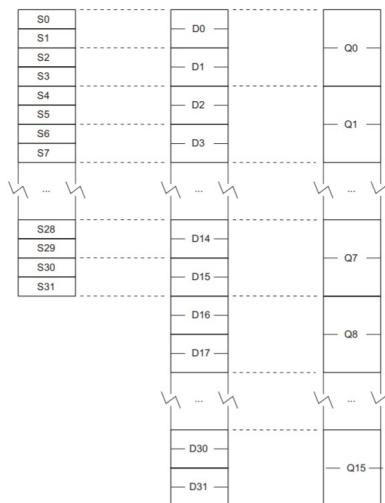
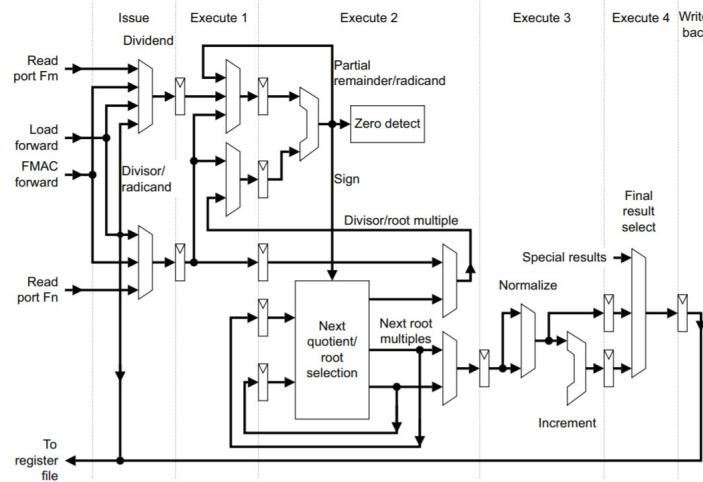
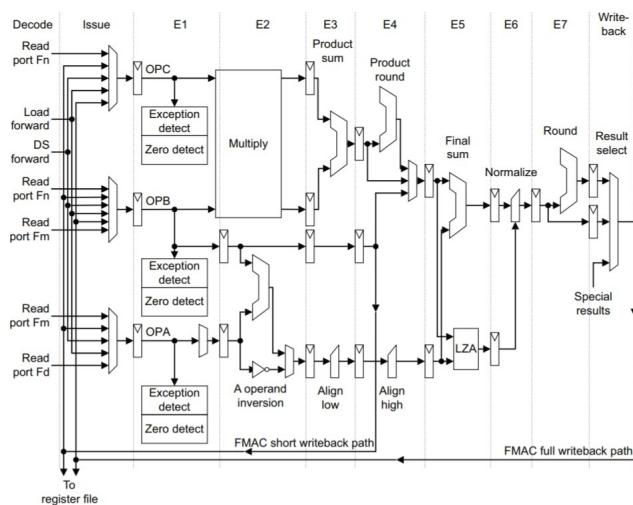


Table 5-1 Location of NEON instructions

Mnemonic	Brief description	Page
VABA, VABD	Absolute difference, Absolute difference and Accumulate	page 5-51
VABS	Absolute value	page 5-52
VACGE	Absolute Compare Greater than or Equal	page 5-29
VACGT	Absolute Compare Greater Than	page 5-29
VACLE	Absolute Compare Less than or Equal (pseudo-instruction)	page 5-79
VACLT	Absolute Compare Less than (pseudo-instruction)	page 5-79
VADD	Add	page 5-53
VADDHN	Add, select High half	page 5-54
VAND	Bitwise AND	page 5-25

## 5.5 Gleitkommaeinheit

- Die Berechnung von Gleitkommazahlen wird seit langem in s.g. Gleitkomma-Coprozessoren ausgelagert
- Gleitkomma-Coprozessoren besitzen häufig separate Instruction-Pipelines z.B.
  - Multiplikations und Additions (FMAC) Pipelines
  - Divisions und Quadratwurzel (DS) Pipelines
- Diese Pipelines können auch unabhängig voneinander operieren



## 6 Mirkoarchitektur

### 6.1 Terminologie

- ISA - instruction set architecture (Menge der Verfügbaren Instruktionen)
- RISC - reduced instruction set computer (kleine und schnell ausführbare ISA), z.B. ARM
- CISC - complex instruction set computer (schwergewichtige ISA), z.B. Intel
- SIMD - single instruction multiple data (vector processing), z.B. NEON
- VLIW - very long instruction word (static multi-issue), superscalar processor
- $\mu$ arch - microarchitecture (ISA Implementierung)
  - instructions per cycle (IPC)
  - instruction level parallelism (ILP) - RO07
  - branch prediction (Sprungvorhersage)
- Architekturzustand - dem Programmierer sichtbare Elemente

### 6.2 Allgemeines

Im allgemeinen ist die Mirkoarchitektur eine Hardware Implementierung einer Architektur. Dabei wird zwischen Datenpfaden, die funktionale Blöcke verbinden, und Kontrollpfaden die Steursignale übermitteln, unterschieden.

#### Bearbeitung eines Befehls

Drei Phasen der Befehlausführung:

- Aufgabe des Prozessors bzw. des Steuerstacks ist das lesen der Befehle aus dem Speicher zu lesen - Befehlsholphase (instruction fetch)
- Nach dem holen des Befehls muss er dekodiert werden - Befehlsfekodierung (instruction decode)
- Nun wird der Befehl ausgeführt und der nächste geladen - Befehlausführung (instruction execute)

#### Takt und Taktfrequenz

Die Prozesse eines Rechnersystems haben eine gemeinsame Zeitbasis damit das Zusammenspiel funktioniert. Diese nennt man Takt. Die Taktfrequenz ergibt sich aus  $f = \frac{1}{T}$  wobei  $T$  die Periodendauer von einer steigenden Taktflanke zur nächsten angibt.  $\Rightarrow$  Je höher die Taktfrequenz, desto schneller wird verarbeitet.

- Bei CMOS Technologie steigt der Leistungsumsatz mit  $P = U^2 * f * C_L$

Implementierung:

- | Eintakt-Implementierung                      | Mehrtakt-Implementierung                    | Pipelines-Implementierung  |
|--|---|--|
| • Jeder Befehl wird in einem Takt ausgeführt | • Jeder Befehl wird in Teilschritte zerlegt | • Jeder Befehl wird in parallel ausgeführte Teilschritte zerlegt |

## Rechnleistung eines Prozessors

Ausführung eines Programms:

- $\text{Ausführungszeit} = (\# \text{Instruktionen}) * (\frac{\text{Takte}}{\text{Instruktionen}}) * (\frac{\text{Sekunden}}{\text{Takte}})$

Definitonen:

- $\frac{\text{Takte}}{\text{Instruktionen}} = CPI$
- $\frac{\text{Sekunden}}{\text{Takte}} = \text{Taktperiode}$
- $\frac{1}{CPI} = \frac{\text{Instruktionen}}{\text{Takt}} = IPC$

## 6.3 Mikroarchitektur Prozessor

### Neumann / Harvard

Von-Neumann-Architektur

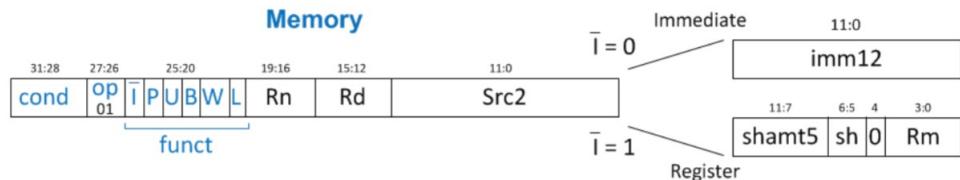
- gemeinsamer Speicher für Maschinenbefehle und Daten

Harvard-Architektur

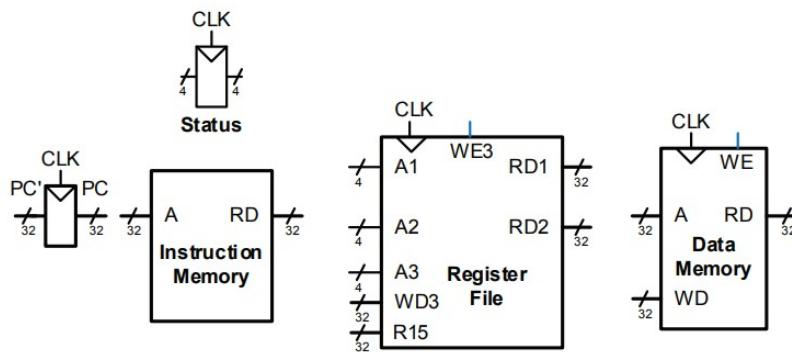
- Befehlsspeicher und Datenspeicher sind getrennt

Speicher können asynchron gelesen werden, aber nur synchron geschrieben werden.

### Elemente der Architektur/Prozessor (Eintakt)



- Begin der Entwickeling des Datenpfades mit dem Befehl 1dr
- 1dr Rd, [Rn, imm12]  
 1dr = Befehl, Rd = Register destination, Rn = Register name, imm12 = 12Bit immediate



## Register File:

- A1 .. A3 sind Registeradressen (4 Bits = 16 Register)  
RD1 (Register Data 1) Gelesene Daten aus A1
- A3 ist das Zielregister  
WD3 (Write Data 3) Geschriebene Daten in A3
- WE3 (Write Enable 3) steuert ob auf A3 geschrieben werden soll oder nicht

## Instruction Memory:

- A
- RD

## Status:

- Hier werden Statusflags zur Verwendung in Operationen gespeichert

## Programm Counter (PC):

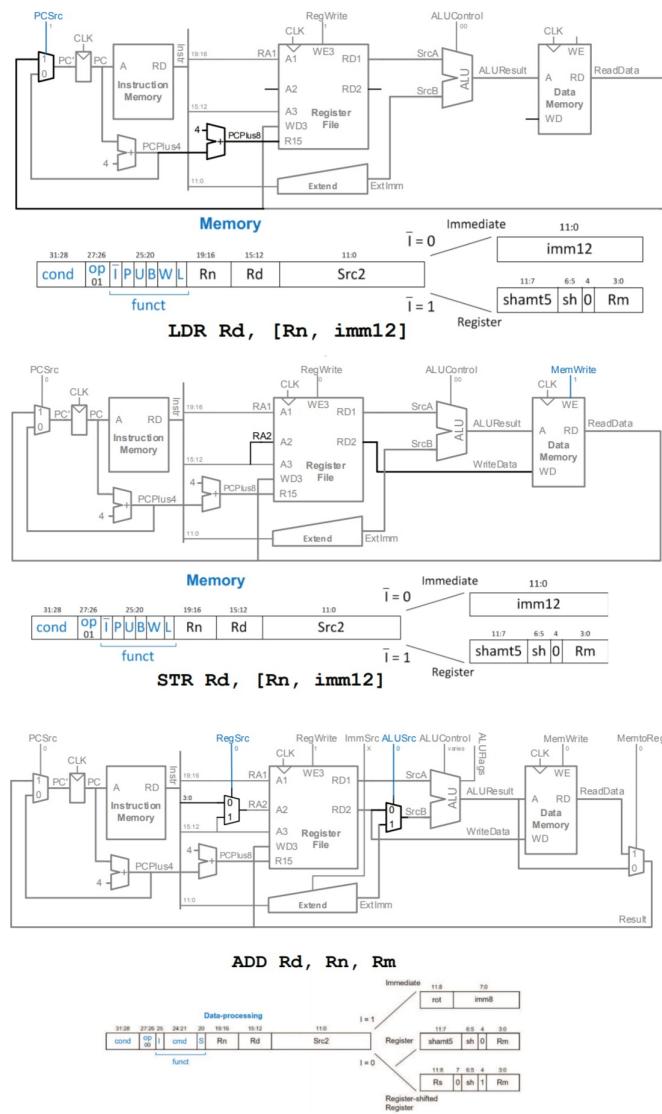
- Unser wie bisher bekannte Programm Counter

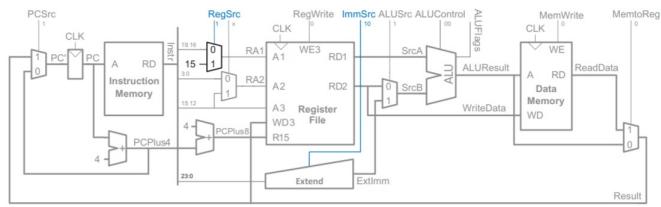
## Data Memory:

•

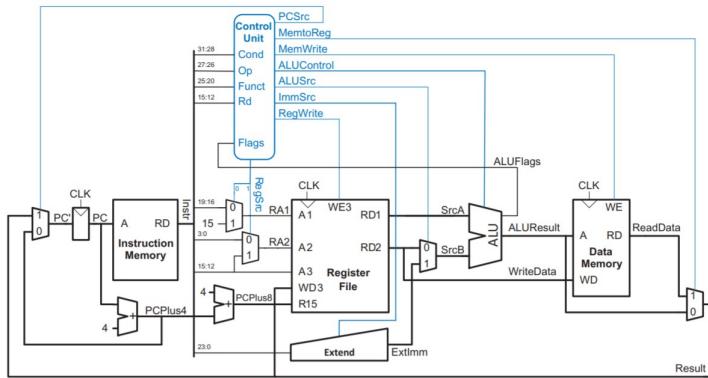
## 6.4 Eintakt-Prozessor

### Bitfelder und Prozessor Darstellung





Branch-Befehle



Prozessor mit Controll-Unit

### Phasen der Befehlsausführung: allgemein

Drei Phasen der Befehlsausführung:

- Befehlsholphase
- Befehlsdekodierung
- Befehlsausführung

### Phasen der Befehlsausführung: ldr

- 1. Befehl holen

Der **Programm Counter** lädt den Befehl aus dem **Instruction Memory**

- 2. Lesen der Quell-Operanden vom Registerfeld

Rn Register aus dem **Bitfeld** des Befehls wird an A1 übergeben

- 3. Erweitern des Offsets

Eventueller Offset muss von seinen 12 Bit auf 32 Erweitert werden mittels **Extend** (mit 0 aufgefüllt)  
Rd Register aus dem **Bitfeld**, das Zielregister, wird an A3 angelegt

- 4. Berechnen der Speicheradresse

Die Werte von Rn und imm12 müssen addiert werden, also die Werte aus RD1 des **Register Files** und imm12 von **Extend** werden in eine ALU geleitet

- 5. Lesen der Daten aus dem Speicher und schreiben in das Registerfeld

Die Berechnete Adresse wird an **Data Memory** übergeben, die gelesenen Daten werden an WD3 angelegt

- 6. Berechnen der Adresse des nächsten Befehls

Der **Programm Counter** wird um 4 (ggf. auch um 8 aus "historischen Gründen") erhöht  
Natürlich kann PC auch als Zielregister an WD3 genutzt werden

### Phasen der Befehlsausführung: arithmetisch

- ImmSrc entscheidet ob der immediate um 20 oder um 24 Bit erweitert werden muss
- MemtoReg entscheidet ob das Ergebnis der ALU als Adresse oder als arithmetisches Ergebnis verwendet werden soll
- RegSrc und AluSrc entscheiden ob ein arithmetischer Befehl mit zwei Registeradressen, oder mit Adresse und Direktwert aufgerufen wird

## Phasen der Befehlsausführung: branch

- Sprungbefehl berechnet den Programmcounter
- Berechnen des Sprungbefehls:  
 $BTA = (\text{ExtImm}) + (\text{PC} + 8)$
- ExtImm (Sprungdistanz) = Imm24 (als Direktwert)  $\ll 2$  ( $\text{Imm24} * 4$ )
- RegSrc legt r15 an falls ein Sprungbefehl ausgeführt wird

## 6.5 Eintakt vs Mehrtakt

Eintakt:

- + einfach
- - Taktfrequenz wird durch langsamste Instruktion bestimmt
- - drei Addierer / ALUs und zwei Speicher

Mehrtakt:

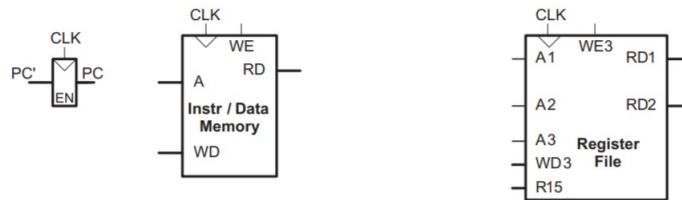
- + höhere Taktfrequenz
- + einfache Instruktionen laufen schneller
- + effizienter
- - aufwendigere Ablaufsteuerung

Gleiche Grundkomponenten:

- Datenpfad: verbindet funktionale Blöcke
- Kontrollpfad: Steuersignale / Steuerwerk

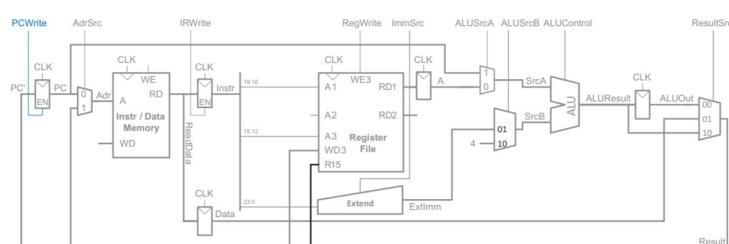
## 6.6 Mehrtakt-Prozessor

### Elemente eines Mehrtakt Prozessors

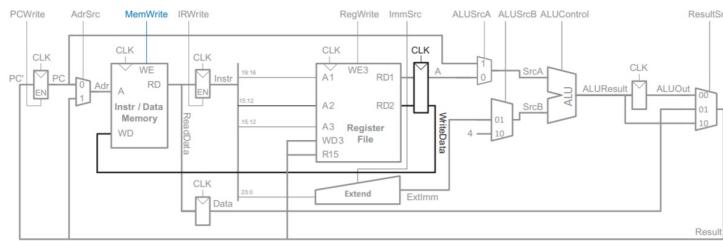


### Datenpfade

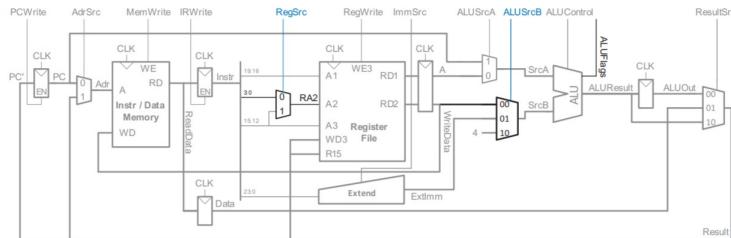
Datenpfad: ldr



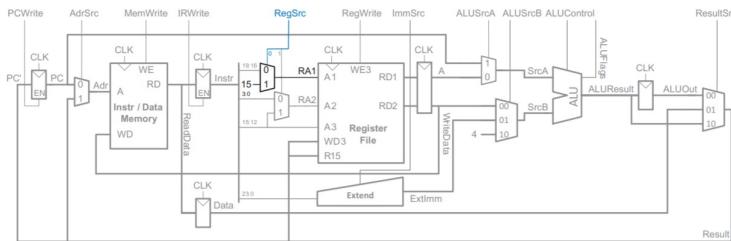
## Datenpfad: str



## Datenpfad: arithmetisch



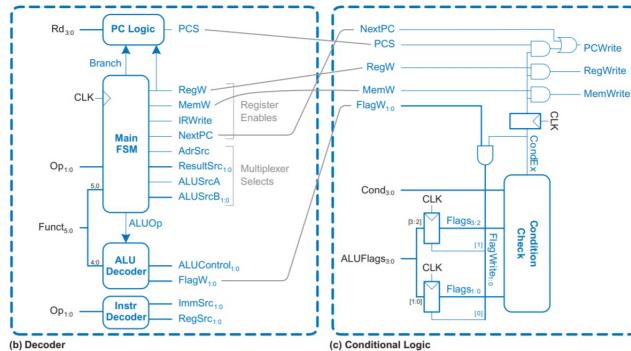
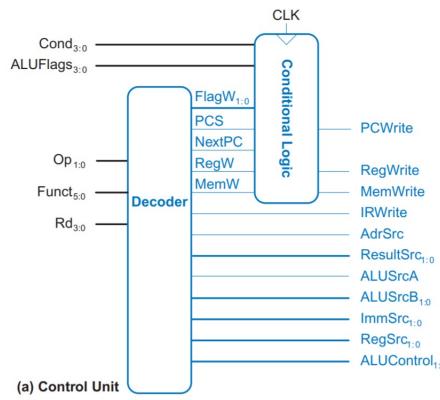
## Datenpfad: branch



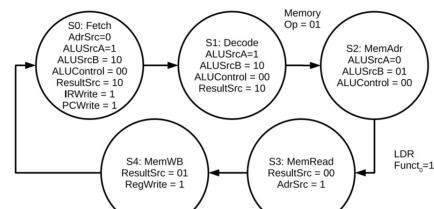
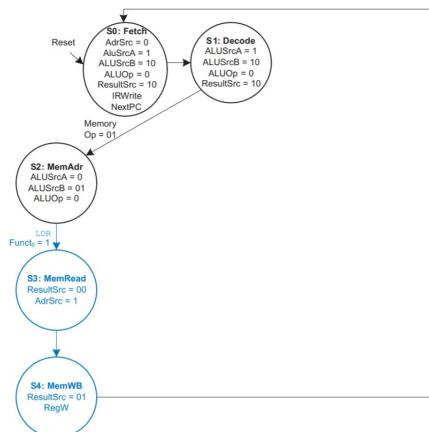
## Phasen der Befehlsausführung: ldr

- 1. Hole Instruktion  
Adresse aus PC wird in **Instr/Data Memory** angelegt, und der Inhalt wird in ein Instruction Register geschrieben (nicht für Programmierer sichtbar, EN Box z.B.)
- 2. Lese Quelloperanden aus Registerfeld und werte Direktwerte aus
- 3. Berechne die Speicheradresse (Basisadresse + Offset)  
Berechnung läuft über eine ALU
- 4. Lese Daten aus Speicher  
Multiplexer kontrolliert welches Signal an Memory anlegt, damit unter anderem auch das Ergebnis der ALU als Adresse angelegt werden kann
- 5. Schreibe die Daten ins passende Register  
Nach Laden des Registers werden die Daten gespeichert. Dafür gibt ResultSrc an welches Signal benutzt werden soll. Das berechnete, oder das eben aus dem Memory gelesene
- 6. Berechne Adresse des nächsten Befehls  
Berechnung läuft nicht über zusätzliche ALU, ALUSrcA und ALUSrcB kontrollieren ob die ALU eine neue Befehlsadresse berechnen soll
- 7. Behandlung von r15 (pc)  
Programm Counter muss aktualisiert werden

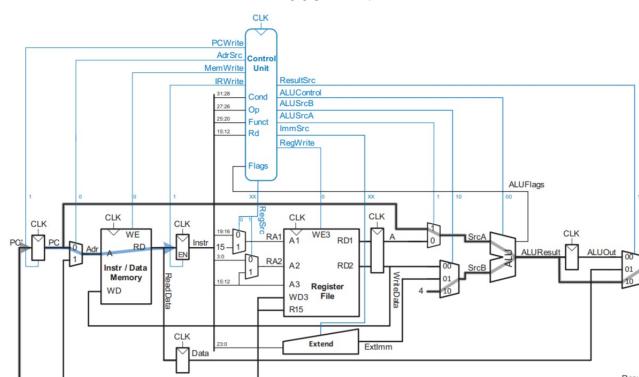
## Control Unit



## Entwicklung des Steuerpfads: ldr

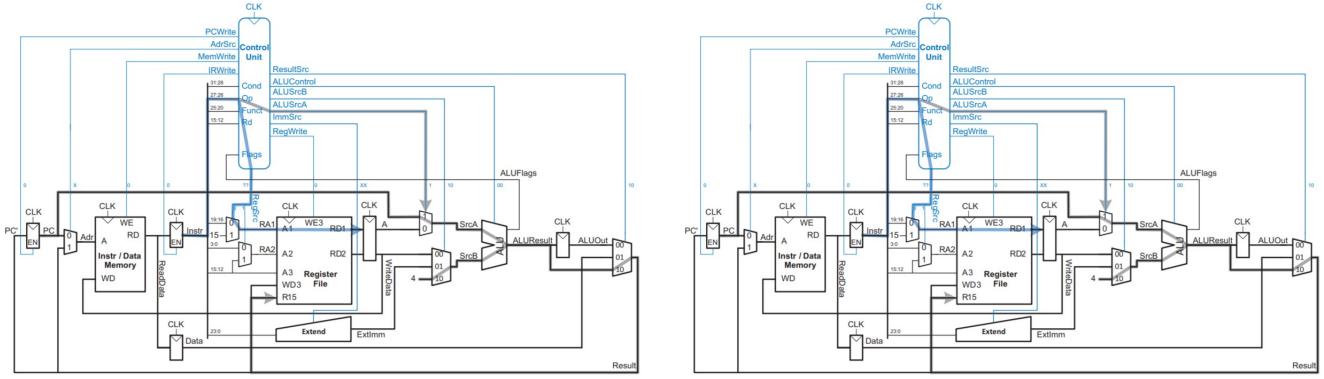


## Fetch ldr



## Decode ldr

## Execute ldr



## 6.7 Pipeline-Prozessor

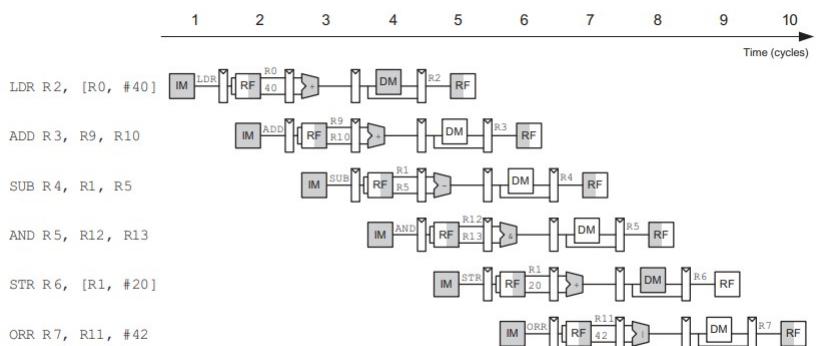
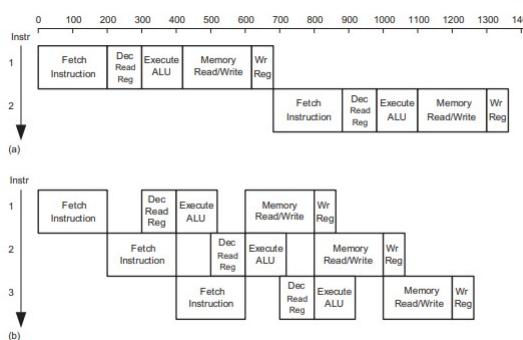
### Phasen der Befehlsausführung

Feinere Einteilung in Ausführungsphasen für Pipeline-Prozessoren

- Instruction Fetch
- Instruction Decode, Read Register
- Execute ALU
- Memory Read/Write
- Write Register

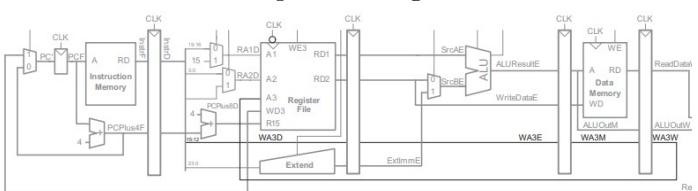
### Pipeline Prinzip

Beim Pipelining werden Aufgaben in Teilaufgaben zerlegt. Dies hat zur Folge, dass sich die Bearbeitungszeiten der einzelnen Befehle überlappen, sie also Teilweise parallel ausgeführt werden.

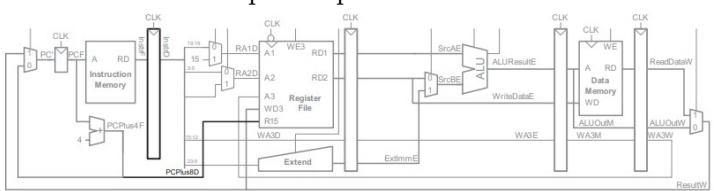


### Datenpfade

#### Datenpfad-Korrigiert



#### Datenpfad-Optimiert



## 6.8 Hazards

Treten auf wenn eine

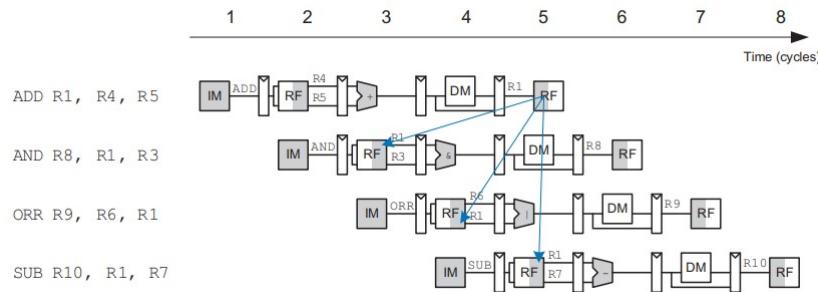
- Instruktion vom Ergebnis einer vorhergehenden abhängt und diese noch kein Ergebnis geliefert hat

Arten von Hazards:

- Data Hazard: z.B. Neuer Wert von Register noch nicht in Registerfeld geschrieben
- Control Hazard: Unklar welche Instruktion als nächstes ausgeführt werden muss (z.B. bei Verzweigungen)

## Data Hazard und Lösungen

Auftreten



Möglichkeiten

- Plane Wartezeit von Anfang an ein
  - Füge nops zur Compile-Zeit ein (no operation)
  - scheduling (Ablaufplanung)
- Stelle Maschinencode zur Compile-Zeit um
  - scheduling / reordering
- Leite Daten zur Laufzeit schneller über Abkürzung weiter
  - bypassing / forwarding
- Halte Prozess zur Laufzeit an bis Daten da sind
  - stalling

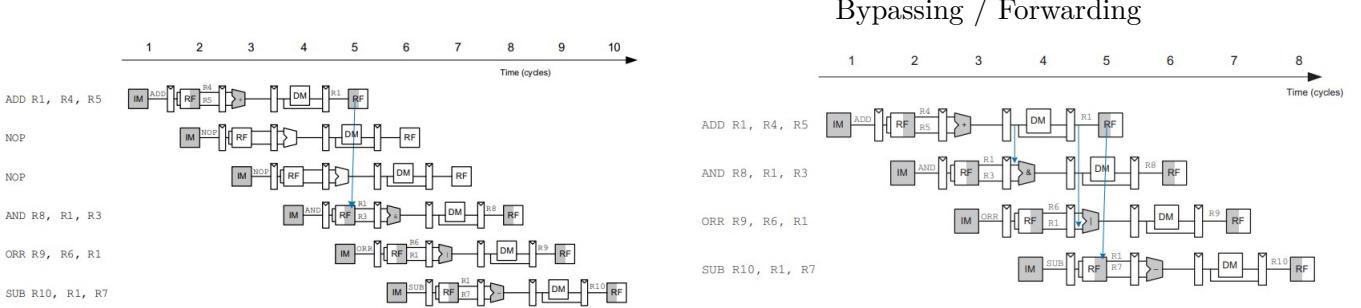
## NOPS und Bypassing / Forwarding

NOPS, no operations, sind leere Operationen die eingefügt werden, damit die folgende Operation mit den richtigen Registern, bzw. ihren Ergebnissen rechnen kann.

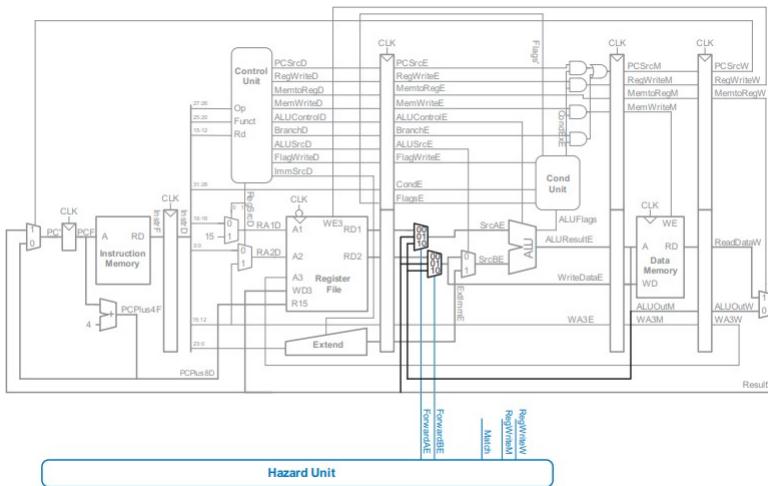
Nachteil: In der Zeit werden keine Berechnungen ausgeführt!

Bypassing ist der Prozess des abkürzens des Dateipfades. Wenn bekannt ist, dass ein Ergebnis in der nächsten Operation benutzt wird, kann das Ergebnis, schon bevor es gespeichert und oder in Register geschrieben wird,

an die nächste Operation, heist an vorangegangene Einheit, weitergegeben werden.  
NOPs

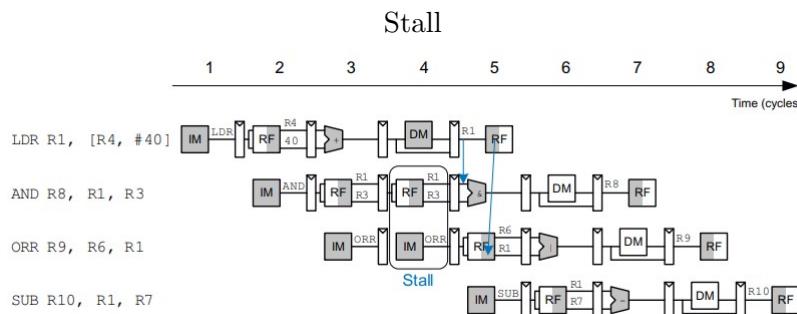


Datenpfad, Kontrolleinheit, Hazard-Unit mit Bypassing

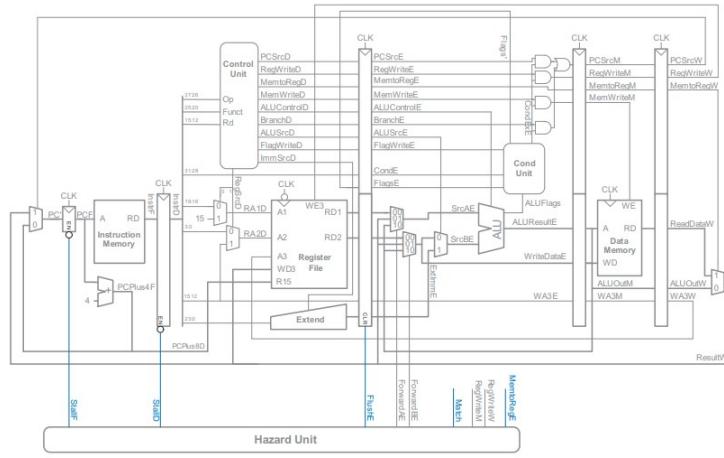


## Stall

Stall, also das anhalten des Prozessors, kann genutzt werden um Hazards zu umgehen. Ist Bypassing keine Option, weil das benötigte Ergebnis erst nach der anstehenden Rechnung zur Verfügung stehen würde, kann man die kommenden Befehle, bzw FlipFlops stallen und um beliebig viele Takte verzögern. Z.B. solange bis Bypassing wieder zeitlich betrachtet wieder eine Option wird.  
Konkret heißt das, dass der Instruction Fetch und der Instruction Decode gestallt werden. Optional müssen falsche Ergebnisse geflusht (gespült) werden.

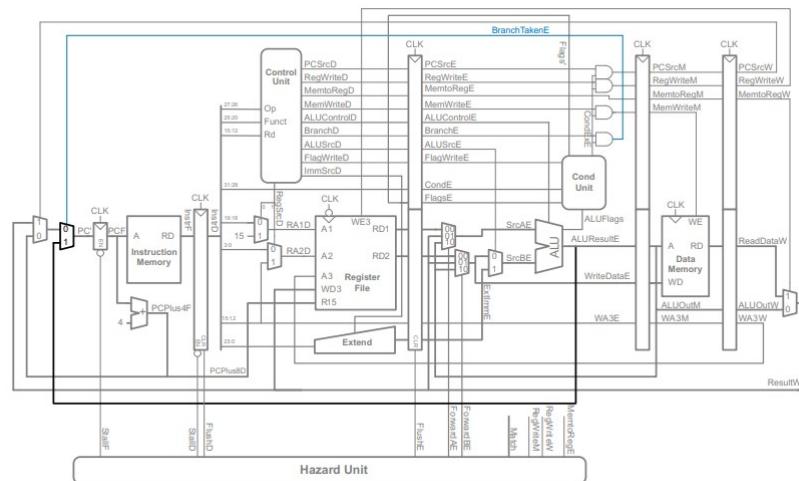
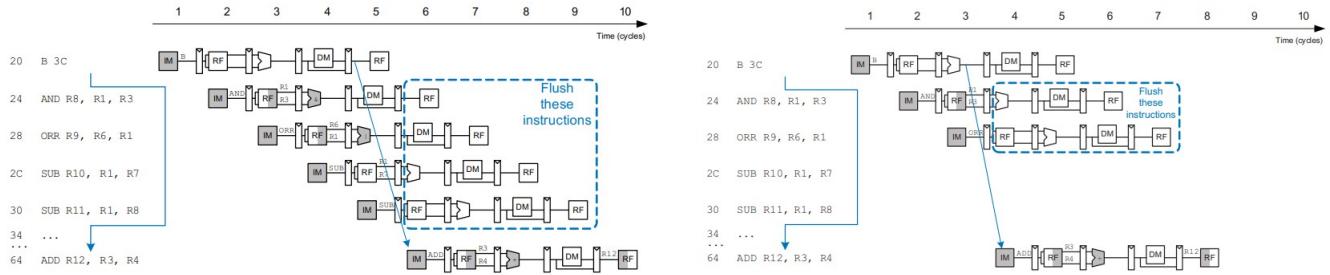


Datenpfad, Kontrolleinheit, Hazard-Unit mit stall Option



## Control Hazard

Damit wir mit allen Befehlen umgehen können, muss noch der Branch Befehl betrachtet werden. Besonders hier können viele Flushes benötigt werden. Wichtig ist hierbei ein neuer Datenpfad und ein weiterer Multiplexer um Bypassing auch bei Branches zu ermöglichen. Bis zum Sprung werden allerdings neue Befehle berechnet. Diese müssen nach dem Sprung alle geflusht (gespült) werden.



## 7 Single Instruction, Multiple Data SIMD

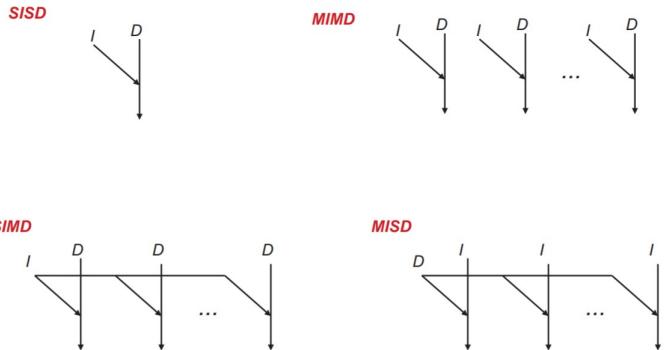
### 7.1 Anwendung und Nutzen

- Zur Beschleunigung von Berechnungen (Ganzahlen und Gleitkommazahlen) werden dedizierte Funktionseinheiten in Prozessoren integriert
- Bei ARM heißt diese Funktionseinheit NEON
- In zahlreiche Prozessoren werden diese integriert, unter anderen Namen
- Häufige Anwendung: Verarbeitung von Multimediadaten/Bildern

### 7.2 Klassifikation von Flynn

- Aufteilung in s.g. *Instruction Streams* und *Data Streams*
- Instruction Stream
  - SI - Single Instruction
  - MI - Multiple Instruction (Mehrere Befehle zu einem Zeitpunkt)
- Data Stream
  - SD - Single Data
  - MD - Multiple Data

- Ergibt vier Kombinationen
  - SISD - Von Neumann Rechner
  - SIMD - Feldrechner, Vektorrechner
  - MIMD - Multiprozessorsysteme
  - MISD - ?
- grobe Klassifikation
- verschiedene Aspekte werden nicht oder grob erfaßt, wie
  - Pipelining
  - Wortbreite
  - Verbindungsstrukturen
  - Speicherorganisation
    - \* lokale/globale Speicher, Virtuelle Speicher, Caches
    - \* Trennung von Programm- und Datenspeicher
- die Begriffe haben sich etabliert

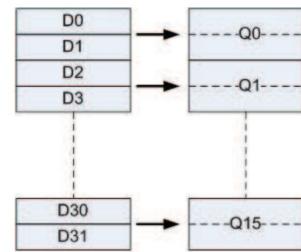


### 7.3 NEON

- NEON ist eine SIMD-Einheit in ARM Cortex-A Serie
- Die SIMD-Einheit beschleunigt zahlreiche Anwendungen wie Signalverarbeitung, Video Encodierung/-Decodierung, Bildverarbeitung usw.
- NEON Befehle führen Packed SIMDäus
  - Register werden als Vektoren von Elementen desselben Datentyps betrachtet
  - Verschiedene Datentypen sind möglich
  - Die Ausführung der Befehle in s.g. lanes erfolgt gleichzeitig
- Es gibt eine separate Register-Bank mit 32 \* 128 bit Registern
- Bis zu 16 \* 8 bit Operationen pro Befehl möglich

## NEON Register

- AArch32 hat  $32 * 64$  bit NEON Register (D0-D31)
- Diese Register können auch als  $16 * 128$  bit Register (Q0-Q15) gesehen werden



## NEON Befehle

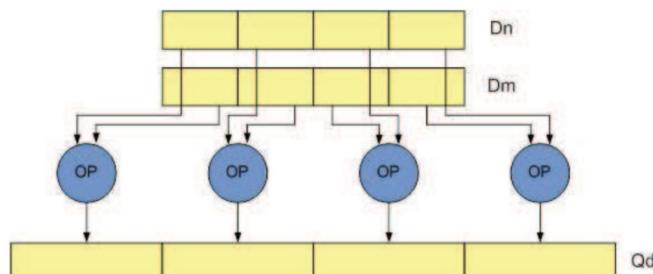
- Befehle haben folgendes Format:  
 $V\{<\text{mod}>\}<\text{op}>\{<\text{shape}>\}\{<\text{cond}>\}\{.\<\text{dt}>\} \{<\text{dest}>\}, \text{src1}, \text{src2}$
- Dabei sind folgende Modifier (mod) möglich
  - Q: Der Befehl nutzt Saturations-Arithmetik
  - ...

## Saturations-Arithmetik

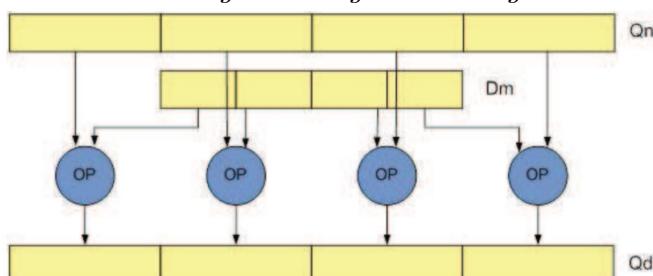
- Die Resultate von Operationen sind auf einen Bereich begrenzt
- Das Resultat einer Operation, welches GröSSer als das Maximum ist, wird auf das erlaubte Maximum gesetzt. Analog selbiges Verfahren mit kleineren Ergebnissen

## Beispiele

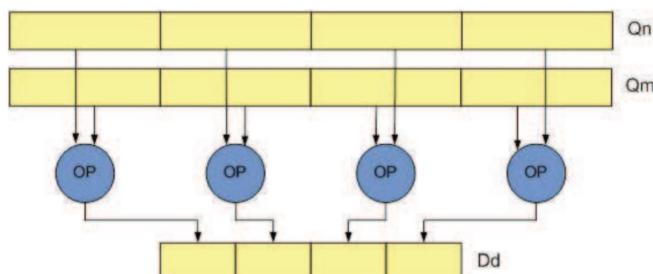
$32\text{bitReg} \times 32\text{bitReg} \rightarrow 64\text{bitReg}$



$64\text{bitReg} \times 32\text{bitReg} \rightarrow 64\text{bitReg}$



$64\text{bitReg} \times 64\text{bitReg} \rightarrow 32\text{bitReg}$



## 7.4 NEON Programmierung

- Es gibt viele Wege, NEON zu nutzen
  - NEIN optimized libraries (OpenMax DL, Ne10)
  - Vectroizin compilers (gcc)
  - NEON intrinsics
  - NEON assembly
- NEON Programmers Guide

## 8 Weiteres

### 8.1 Parallelisierung und OpenMP

- Ein Thread ist ein Leichtprozess, ein Betriebssystemkonzept zur Ausführung von Programmen
- Durch Parallelisierung bei Algorithmen kann die Laufzeit verbessert werden, aber aufwendig zu implementieren. Software zur Vereinfachung und Automatisierung, OpenMP:
  - Sammlung von Direktiven (Compiler-Anweisungen) und Bibliotheksfunktionen
  - C, C++, Fortran

Grundkonzept:

- OpenMP startet als eigener Prozess
- Bei Ausführung des Codes werden Codeabschnitte die parallel ausgeführt werden können parallelisiert. Dabei forkt (verzweigt) sich das Programm in zusätzliche Threads
- Diese so genannten parallelen Regionen werden am Ende zu einem einzelnen Thread zusammengeführt (join)  
⇒ **Fork-Join-Programmiermodell**

Praxis:

- Einbindung von `#include<omp.h>`
- Compileraufruf: `gcc -fopenmp pi_omp.c`
- Setzen der Umgebungsvariablen für die Anzahl der Threads: `export OMP_NUM_THREADS=2`

Bei ideal parallelisierbaren Programmen sollte sich die Laufzeit um den Faktor  $\frac{1}{C_{\text{Core}}}$  verbessern. In der Praxis lässt sich aber nicht jedes Programm gleich gut parallelisieren. Bei den meisten Programmen nimmt die Effektivität mit wachsender Kernzahl zu, hat einen Maximalwert, und nimmt dann wieder ab.

## **9 Streifzug durch die Geschichte**

TODO

## **10 Ethik in der Informatik**

TODO