

Rechnerorganisation

Jonas Milkovits

Last Edited: 4. September 2020

Inhaltsverzeichnis

1 Einführung	1
1.1 Begrifflichkeiten und Grundlagen	1
1.2 Streifzug durch die Geschichte	3
1.3 Ethik in der Informatik	3
2 Einführung in die maschinennahe Programmierung	4
2.1 Begrifflichkeiten und Grundlagen	4
2.2 Phasen der Übersetzung	5
2.3 Ausführung eines Programms im Rechnersystem	6
2.4 Befehle eines Rechnersystems	7
2.5 Registersatz	8
2.6 Adressierung des Speichers, Lesen und Schreiben auf Speicher	8
2.7 Kontrollstrukturen in Assembler	10
2.8 Nutzung des Hauptspeichers	12
2.9 Datenfelder (Arrays)	13
2.10 Unterprogramme	14
2.11 Stack	16
2.12 Rekursion	17
2.13 Compilieren, Assemblieren und Linken	19
3 Mikroarchitekturen von Rechnersystemen	22
3.1 Begrifflichkeiten und Grundlagen	22
3.2 Analyse der Rechenleistung	22
3.3 Eintakt-Prozessor	24
3.4 Mehrtakt-Prozessor	27
3.5 Pipeline-Prozessor	30
3.6 Ausnahmebehandlung - Hazards	32
4 Gleitkommazahlen/Gleitkommarechenwerte	34
4.1 Gleitkommazahlen	34
4.2 Code-Analyse	35
4.3 Studium der Datenblätter/Handbücher	36
4.4 SIMD	37
5 Speicher	39
5.1 Speicher	39
5.2 Speicherorganisation	41
5.3 Lokalität	43
5.4 Prinzip des Caches	45
5.5 Cachehierarchie ARM / Intel Core i7	46

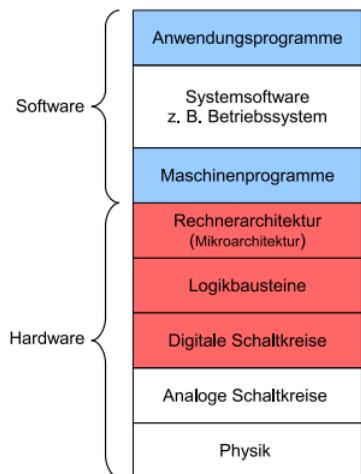
1 Einführung

1.1 Begrifflichkeiten und Grundlagen

- Abstraktion

- Wichtiges und zentrales Konzept der Informatik
- Verstecken unnötiger Details (für spezielle Aufgabe unnötig)

- Schichtenmodell



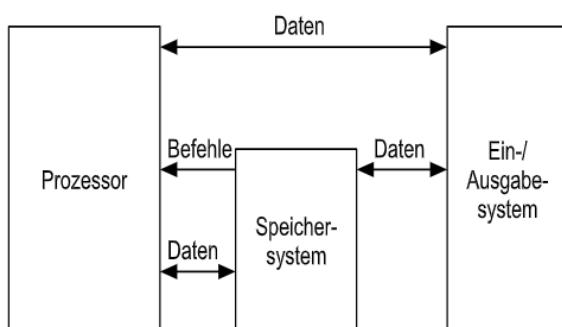
- Untere Schicht erbringt Dienstleistungen für höhere Schicht
- Obere Schicht nutzt Dienste der niedrigeren Schicht
- Eindeutige Schnittstellen zwischen den Schichten
- Vorteile:
 - Austauschbarkeit einzelner Schichten
 - Nur Kenntnis der bearbeitenden Schicht notwendig
- Nachteile:
 - ggf. geringere Leistungsfähigkeit des Systems

- Grundbegriffe

- Computer:
 - Datenverarbeitungssystem
 - Funktionseinheit zur Verarbeitung und Aufbewahrung von Daten
 - Auch Rechner, Informationsverarbeitungssystem, Rechnersystem,...
 - Steuerung eines Rechnersystems folgt über ladbares Programm (Maschinenbefehle)
- Grundfunktionen, die ein Rechner ausführt
 - Verarbeitung von Daten (Rechnen, logische Verknüpfungen,...)
 - Speichern von Daten (Ablegen, Wiederauffinden, Löschen)
 - Umformen von Daten (Sortieren, Packen, Entpacken)
 - Kommunizieren (Mit Benutzer, mit anderen Rechnersystemen)

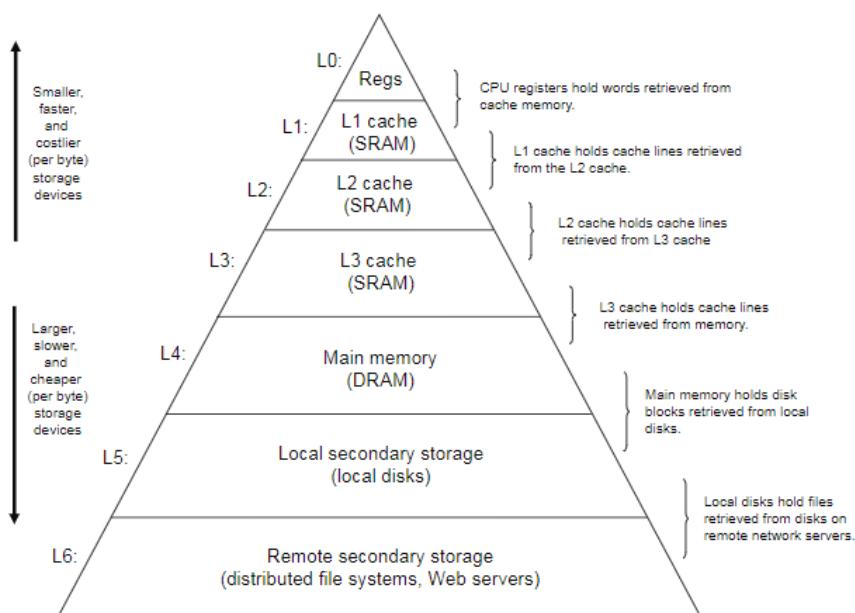
- Komponenten eines Rechnersystems

- Prozessor
 - Zentraleinheit, Central Processing Unit (CPU)
 - Ausführung von Programmen
- Speicher
 - Enthält Programme und Daten (Speichersystem)
- Kommunikation
 - Transfer von Informationen zwischen Speicher und Prozessor
 - Kommunikation mit der Außenwelt (Ein-/Ausgabesystem)



- Nähere Informationen zum Speicher

- Explizite Nutzung des Speichersystem
 - Interne Prozessorspeicher/Register
 - schnelle Register zur temporären Speicherung von Daten/Befehlen
 - direkter Zugriff durch Maschinenbefehle
 - Technologie: Halbleiter ICs
 - Hauptspeicher
 - relativ großer und schneller Speicher für Programme/Daten
 - direkter Zugriff durch Maschinenbefehle
 - Technologie: Halbleiter ICs
 - Sekundärspeicher
 - großer, aber langsamer Speicher für permanente Speicherung
 - indirekter Zugriff über E/A-Programme (Daten → Hauptspeicher)
 - Technologie: Halbleiter ICs, Magnetplatten, optische Laufwerke
 - z.B.: Festplatte
- Implizite (transparente) Nutzung
 - Für das Maschinenprogramm transparent
 - bestimmte Register auf dem Prozessor
 - Cache-Speicher



- Speicherorganisation: Big-Endian und Little-Endian

Big-Endian	Little-Endian	
Byte-Adresse	Wort-Adresse	Byte-Adresse
⋮	⋮	⋮
C D E F	C	F E D C
8 9 A B	8	B A 9 8
4 5 6 7	4	7 6 5 4
0 1 2 3	0	3 2 1 0
MSB LSB	MSB LSB	MSB LSB

- Schemata für Nummerierung von Bytes in einem Wort
- Big-Endian: Bytes werden vom höchstwertigen Ende gezählt
- Little-Endian: Bytes werden vom niederwertigen Ende gezählt

1.2 Streifzug durch die Geschichte

- Übersicht über die geschichtliche Entwicklung mit wichtigsten Meilensteinen

Bezeichnung	Technik und Anwendung	Zeit
Abakus, Zahlenstäbchen	mechanische Hilfsmittel zum Rechnen	bis ca. 18. Jahrhundert
mechanische Rechenmaschinen	mechanische Apparate zum Rechnen	1623 - ca. 1960
elektronische Rechenanlagen	elektronische Rechenanlagen zum Lösen von numerischen Problemen	seit 1944
Datenverarbeitungs- anlage	Rechner kann Texte und Bilder bearbeiten	seit ca. 1955
Informations- verarbeitungssystem	Rechner lernt, Bilder und Sprache zu erkennen (KI)	seit 1968

- Fünf Rechnergenerationen im Überblick:

Generation	Zeitdauer (ca.)	Technologie	Operationen/sec
1	1946 - 1954	Vakuumröhren	40000
2	1955 - 1964	Transistor	200000
3	1965 - 1971	Small und medium scale integration (SSI, MSI)	1000000
4	1972 - 1977	Large scale integration (LSI)	10000000
5	1978 - ????	Very large scale integration (VLSI)	100000000

- Rechner im elektronischen Zeitalter

- 1954: Entwicklung der Programmiersprache Fortran
- 1955: Erster Transistorrechner
- 1957: Entwicklung Magnetplattenspeicher, Erste Betriebssysteme für Großrechner
- 1968: Erster Taschenrechner
- 1971: Erster Mikroprozessor
- 1981: Erster IBM PC, Beginn des PC-Zeitalters

1.3 Ethik in der Informatik

- Ethik in der Informatik
 - Ethik: Bewertung menschlichen Handelns
 - Verbindung zur Informatik: Anwendung von Rechnern für kriegisches Handeln
 - Dual-Use-Problematik:** Verwendbarkeit von Rechnern für zivile als auch militärische Zwecke
- Digitale Souveränität
 - Souveränität: Fähigkeit zur Selbstbestimmung (Eigenständigkeit, Unabhängigkeit)
 - Digitale Souveränität: Souveränität im digitalen Raum

2 Einführung in die maschinennahe Programmierung

2.1 Begrifflichkeiten und Grundlagen

- Allgemein

- Architektur / Programmiermodell
 - Programmiersicht auf Rechnersystem
 - Definiert durch Maschinenbefehle und Operanden
- Mikroarchitektur
 - Hardware-Implementierung der Architektur

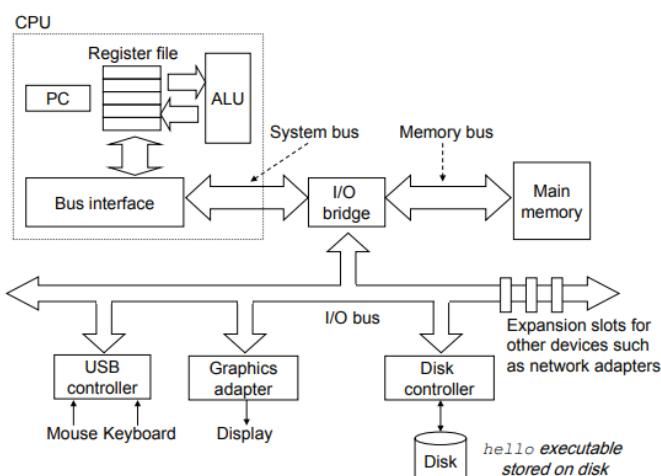
- Programmierparadigmen

- Synonyme: Denkmuster, Musterbeispiel
- Bezeichnet in der Informatik ein übergeordnetes Prinzip
- Dieses Prinzip ist für eine ganze Teildisziplin typisch
- Manifestiert sich an Beispielen, keine konkrete Formulierung
- Maschinensprache (Assembler) ist ein primitives Paradigma

- Programmiermodell

- Bei höheren Programmiersprachen:
 - Grundlegende Eigenschaften einer Programmiersprache
- Bei maschinennaher Programmierung:
 - Bezeichnet dort den **Registersatz** eines Prozessors
 - Registersatz besteht aus:
 - Register, die durch Programme angesprochen werden können
 - Liste aller verfügbaren Befehle (**Befehlssatz**)
 - Register, die prozessorintern verwendet werden (IP/PC) zählen nicht zum Registersatz
 - IC: Instruction Pointer
 - PC: Program Counter

- Verfeinerung des Rechensystems



- CPU/Prozessor:
führt die im Hauptspeicher abgelegten Befehle aus
- ALU/Arithmetic Logical Unit:
Ausführung der Operationen
- PC/Program Counter:
Verweis auf nächsten Maschinenbefehl im Hauptspeicher
- Register:
Schneller Speicher für Operanden
- Hauptspeicher:
Speichert Befehle und Daten
- Bus Interface:
Verbinden der einzelnen Komponenten

- **Assembler**

- Programmieren in der Sprache des Computers
 - Maschinenbefehle: Einzelnes Wort
 - Befehlssatz: Gesamtes Vokabular
- Befehle geben Art der Operation und ihre Operanden an
- Zwei Darstellungen:
 - Assemblersprache: Für Menschen lesbare Schreibweise für Instruktionen
 - Maschinensprache: maschinenlesbares Format (1 und 0)

- **ARM-Architektur - Hier verwendetes Rechnersystem**

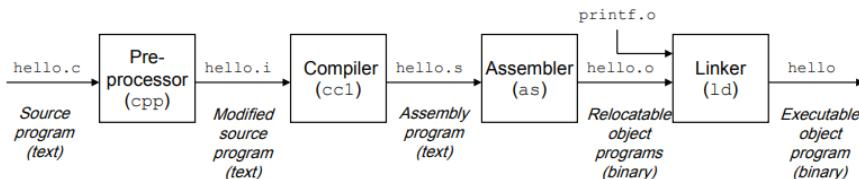
- z.B. verwendet bei Raspberry Pi
- ARM: Acorn RISC Machines / Advanced RISC Machines
- Große Verbreitung heutzutage in Smartphones

2.2 Phasen der Übersetzung

- Beispielhaftes C-Programm:

```
#include <stdio.h> /* Standard Input/Output */ /* Header-Datei*/
int main() {
printf("Hello World\n");
return 0;
}
```

- C-Programm an sich für den Menschen verständlich
- Übersetzung in Maschinenbefehle für Ausführung auf dem Rechnersystem:



- 1. Phase (**Preprocessor**)

- Aufbereitung durch Ausführung von Direktiven (Code mit #)
- z.B.: Bearbeiten von `#include <stdio.h>`
 - Lesen des Inhalts der Datei `stdio.h`
 - Kopieren des Inhalts in die Programmdatei
- Ausgabe: C-Programm mit der Endung .i

- 2. Phase (**Compiler**)

- Übersetzt C-Programm `hello.i` in Assemblerprogramm `hello.s`

- 3. Phase (**Assembler**)

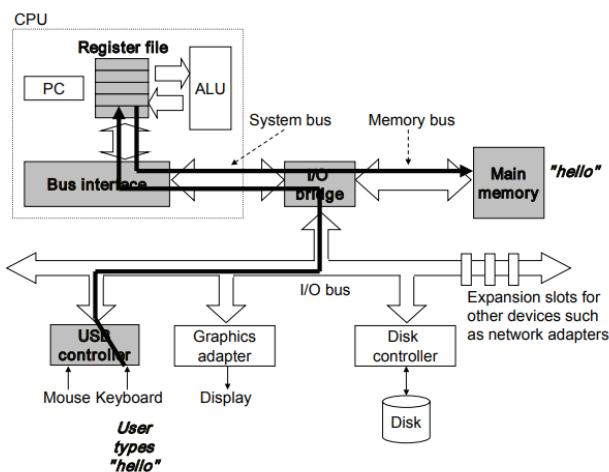
- Übersetzt `hello.s` in Maschinensprache
- Ergebnis ist das Objekt-Programm `hello.o`

- 4. Phase (**Linker**)

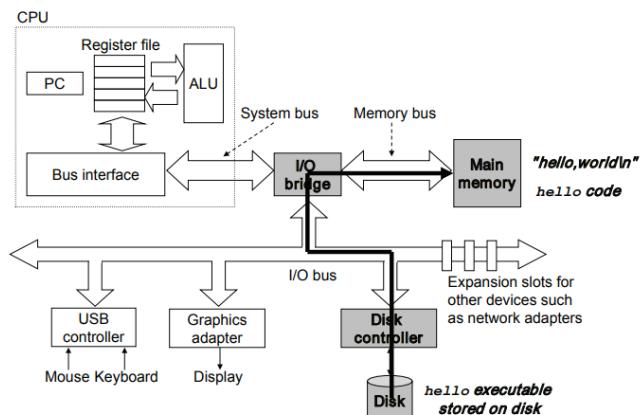
- Zusammenfügen verschiedener Module
 - Code von `printf` existiert bereits als `print.o`-Datei
- Linker kombiniert `hello.o` und `print.o` zu ausführbarem Programm
- Ausgabe des Bindevorgangs: ausführbare `hello`-Objektdatei

2.3 Ausführung eines Programms im Rechnersystem

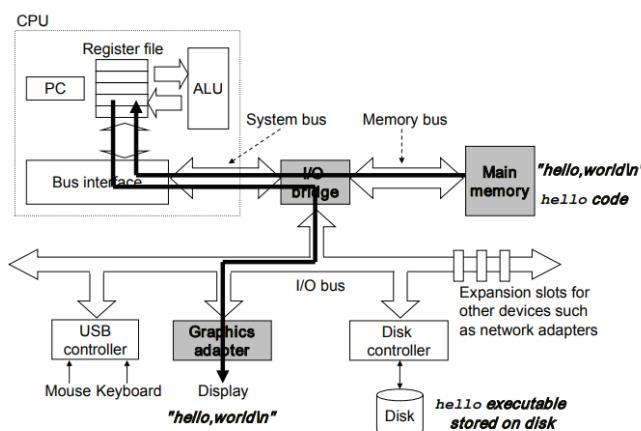
- Ausgangspunkt
 - Ausführbares Objektprogramm `hello` auf der Festplatte
 - Starten der Ausführung des Programms unter Nutzung der Shell
- Ablauf:
 - Shell liegt Zeichen des Kommandos ins Register
 - Speichert den Inhalt dann im Hauptspeicher ab



- Schrittweises Kopieren der Befehle/Daten von Festplatte in Hauptspeicher



- Ausführen der Maschinenbefehle des `hello`-Programms



2.4 Befehle eines Rechnersystems

- Wieviele Befehle und was für Befehle soll ein Rechnersystem haben?
- Viele komplexe Befehle:
 - **CISC-Maschinen** (Complex Instruction Set Computer)
 - Befehlsausführung direkt im Speicher möglich
 - Verwendet von Intel-Architektur
- Weitgehend identische Ausführungszeit der Befehle
 - **RISC-Maschinen** (Reduce Instruction Set Computer)
 - Ermöglicht effizientes Pipeling
 - Werden auch als Load/Store-Architekturen bezeichnet (Nur Ausführung im Register)
 - Verwendet von ARM-Architektur
- Jedoch viele Befehle, die jeder Prozessor hat (AND, OR, NOT,..)
- Unterschiedliche Befehlsformate:
 - Erlauben Flexibilität
 - z.B. `add` und `sub` mit drei Registern als Operanden
 - z.B. `ldr` und `str` verwenden zwei Register und Konstante
 - Anzahl an Formaten sollte jedoch klein sein
 - Hardware weniger aufwendig
 - Erlaubt evtl. höhere Verarbeitungsgeschwindigkeit
- Interner Aufbau eines Rechners hat viele Freiheitsgrade
- Diese Struktur hat erheblichen Einfluss auf die Leistungsfähigkeit eines Rechnersystems
- **n-Adressmaschinen**
 - Einteilung nach der Anzahl der Operanden in einem Maschinenbefehl
 - 2-Adressmaschine (Intel Architektur)
 - 3-Adressmaschine (ARM Architektur)
- **Konstanten in Befehlen (immediates)**
 - Direkt im Befehl untergebracht → Direktwerte
 - Benötigen kein eigenes Register oder Speicherzugriff
 - Direktwert ist Zweierkomplementzahl, die 12 Bit breit ist
 - Bitbreite der Direktwertzahl vom Befehl abhängig
 - Befehle haben immer 32 Bit
 - Registeradressen werden mit 4 Bit kodiert
 - Übrigbleibende Bits für Direktwert

2.5 Registersatz

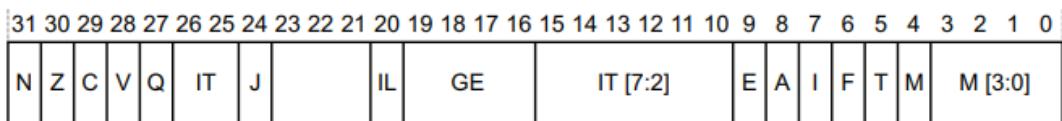
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13 (sp)
R14 (lr)
R15 (pc)

- R0: Verwendet für Rückgabe von Werten an die Shell
- R1-R12: General Purpose Register
- R13: Stack Pointer (sp)
- R14: Link Register (lr)
- R15: Program Counter (pc)
- Current Processor Status Register (CPSR)

(A/C)PSR

- Current Processor Status Register

- Enthält unter anderem die Statusflags



- Werden oft für Vergleiche (`b`, `beq`, ...) verwendet
- N (Negative): Wird verwendet um zu zeigen, dass Ergebnis negativ ist
- Z (Zero): Wird verwendet um zu zeigen, dass Ergebnis 0 ist
- C (Carry): Zeigt, dass Carry-Bit besteht
- V (OverFlow): Zeigt, dass Overflow geschehen ist
- Namen können je nach Prozessor stark variieren

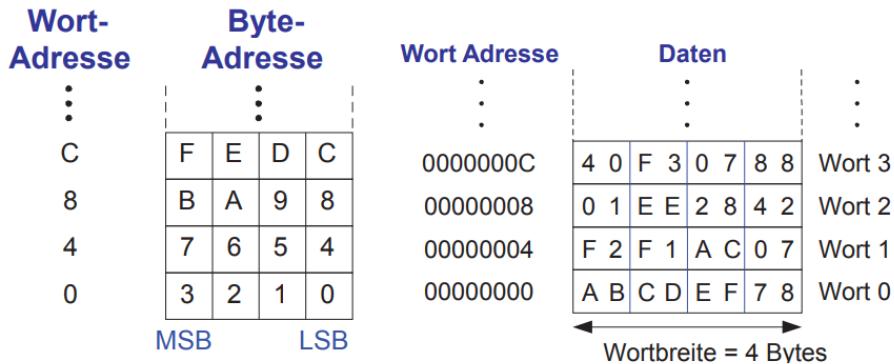
2.6 Adressierung des Speichers, Lesen und Schreiben auf Speicher

- Allgemeine Verwendung von Registerspeicher

- Meist zuviele Daten für die Register
- Kombination des Registers und Hauptspeichers zum Halten von Daten
- Speichern von häufig verwendeten Daten in Registern (Schleifenvariable)

- Wort- und Byte-Adressierung von Daten im Speicher

- Byte-adressiert: (ARM)
 - Jedes Byte hat eine eindeutige Adresse (Zugriff auf jedes Byte möglich)
 - Ein Wort (hier 32Bit) besteht aus 4 Bytes (32 Bits)
 - Wortbreite ist von der Architektur abhängig
 - Wortadressen sind immer Vielfache von 4 (Offset von 4)



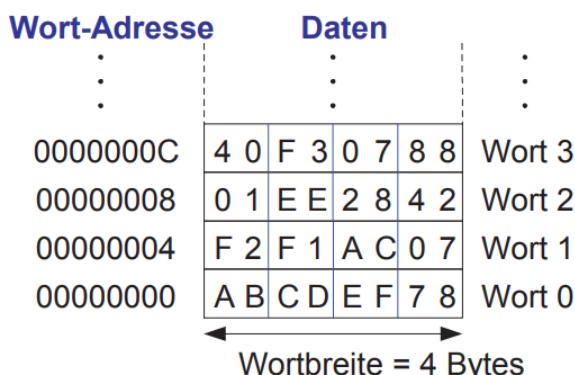
- Rechts wird ein Byte mit zwei Hexawerten dargestellt ($AB : 1011\ 1010$)

- Lesen aus byte-adressiertem Speicher

- Lesen geschieht durch Ladebefehle (Transportbefehl)
- Befehlsname: `load word` (`ldr`)
- Alternative für Bytes statt Wörtern: `ldrb`
- Adressarithmetik:
 - Adressen werden relativ zu Register angegeben
 - Basisadresse (startet bei Wort 0) plus Distanz (offset)
 - Adresse = (`r5 (Basis) + 8 (offset)`)
- Beispiel 1:
 - Lese Datenwort von Speicheradresse (`r5+8`) und schreibe es in Register `r7`

```
mov r5, #0 /* Transportbefehl, schreibt Konstante 0 in r5 */
ldr r7, [r5, #8] /* r7: Zielregister | [r5, #8] Quelle */
```
 - `r7` enthält das Datenwort der Speicheradresse `r5+8`
- Beispiel 2:
 - Lesen Datenwort 3 (Speicheradresse `0xC` (12er Offset)) nach `r7`
 - (Einschub: `0x` sagt dem Compiler, dass das Folgende eine Hexzahl ist)

```
mov r5, #0 /* Schreibt Konstante 0 in r5 */
ldr r7, [r5, #0xC] /* Lädt den Wert (r5+12) in r7 */
```
 - Nach Abarbeiten des Befehls hat `r7` den Wert `0x40F30788`



- Schreiben in byte-adressierten Speicher

- Schreiben geschieht durch Speicherbefehle (Transportbefehl)
- Befehlsname: **store word (str)**
- Alternative für Bytes statt Wörtern: **strb**
- Beispiel:

- Schreibe den Wert aus r9 in Speicherwort 5


```
mov r1,#0 /* Speichert Konstante 0 in r1 */
mov r9,#42 /* Speichert Konstante 42 in r9 */
str r9, [r1,#0x14] /* Schreibt Wert des 5. Wortes von r1 in r9 */
      
```

 - #0x14: $14_{16} = 0001\ 0100_2 = 20_{10}$ (5tes Wort)

2.7 Kontrollstrukturen in Assembler

- Statusbits

- Die Wichtigsten:
 - CF (CarryFlag)
 - ZF (ZeroFlag)
 - SF (SignFlag)
 - OF (OverflowFlag)
- Verwendung:
 - Vergleiche (**cmp**)
 - Gleichheit
- Unterschiede zwischen **Carry** und **Overflow**
 - **Overflow**: Ergebnis passt nicht in maximale darstellbare Werte (z.B. +8 bei 4 Bit im ZK)
 - **Carry**: Ergebnis passt nicht in Bitbreite ($+5 - 1 = +4$)
 - **Sign**: Vorzeichen negativ

- Sprünge / Verzweigungen

- Änderung der Ausführungsreihenfolge von Befehlen
- Unbedingte Sprünge
 - Werden immer ausgeführt
 - **b target** /* Springt von branch zu target */
- Bedingte Sprünge
 - Sprünge abhängig von Bedingung
 - **beq target** /* Ein Beispiel, eq für equal */
- Label
 - Label sind Namen für Adressen im Programm
 - Name muss unterschiedlich von Maschinenbefehlen (Mnemonics) sein
 - Label müssen mit einem Doppelpunkt abgeschlossen werden
 - Werden zur Markierung von Stellen für Sprünge verwendet (target)

- Bedingte Sprünge

```
mov r0,#4      /* r0 = 4 */
add r1,r0,r0  /* r1 = 8 */
cmp r0, r1    /* r0 - r1 = -4: NZCV = 1000 */
               /* StatusBits NZCV */
beq there     /* Kein Sprung: Z != 1 */
               /* Müsste bei Gleichheit (equal) 0 sein */
add r1,r1,#42 /* r1 = r1 + 42 */

there:
add r1,r1,#78 /* r1 = r1 + 78 */
```

- Weitere Bedingungen:

- beq: Equal / Gleichheit
- bne: Not Equal / Ungleichheit
- bge: Greater / Größer
- ble: Less / Kleiner

- if-Anweisung

```

/* r0 = 5; r1 = 10; r2 = f; r3 = i */
cmp r0,r1      /* Vergleicht r0 und r1 */
bne L1          /* Falls Werte ungleich sind, ist hier gegeben */
add r2,r3,#1    /* Wird hier übersprungen */
L1:             /* Hierhin wird gesprungen */
    sub r2,r2,r3

```

- if/else-Anweisung

```

/* r0 = 5; r1 = 10; r2 = f; r3 = i */
cmp r0,r1
bne L1          /* Potentieller Sprung nach L1 */
add r2,r3,#1    /* else-Anweisung (wird übersprungen, falls Bedingung korrekt) */
b L2            /* Überspringen der if-Anweisung, sonst wird beides ausgeführt */
L1:
    sub r2,r2,r3
L2:
    ...

```

- while-Schleifen

```

/* r0 = pow; r1 = x */
mov r0,#1
mov r1,#0
WHILE:           /* Label für Schleife */
    cmp r0,#128   /* Abbruchbedingung: Falls equal Z = 1 */
    beq DONE       /* Sprung aus Schleife */
    lsl r0,r0,#1    /* Linksshift um 1 Bit / Schleifencode */
    add r1,r1,#1    /* x = x + 1 / Schleifencode */
    b WHILE        /* Fortführen der Schleife */
DONE:
    ...

```

- for-Schleifen

```

/* r0 = i; r1 = sum */
mov r1,#0
mov r0,#0
FOR:             /* Label für Schleife */
    cmp r0,#10    /* Abbruchbedingung: Falls i größer als 10 ist */
    bge DONE
    add r1,r1,r0   /* sum = sum + i */
    add r0,r0,#1    /* i = i + 1 */
    b FOR          /* Fortführen der Schleife */
DONE:
    ...

```

2.8 Nutzung des Hauptspeichers

- Erklärung anhand eines Codebeispiels

```

1  /* — speicher_l.s */
2  /* Kommentar */
3
4  .data /* Daten Bereich */
5  var1: .word 5 /* Variable 1 im Speicher, Wert 5 */
6  var2: .word 12 /* Variable 2 im Speicher, Wert 12 */
7
8  .global main /* Definition Einsprungpunkt Hauptprogramm */
9
10 main:          /* Hauptprogramm */
11    ldr r0, adr_var1 /* laedt Adresse von var1 in r0 */
12    ldr r1, adr_var2 /* laedt Adresse von var2 in r1 */
13    ldr r2,[r0] /* Lade Inhalt von Adresse r0 in r2 */
14    ldr r3,[r1] /* Lade Inhalt von Adresse r1 in r3 */
15    add r0, r2, r3
16    bx lr      /* Springe zurueck zum aufrufenden Programm */
17
18 adr_var1: .word var1 /* Adresse von Variable 1 */
19 adr_var2: .word var2 /* Adresse von Variable 2 */

```

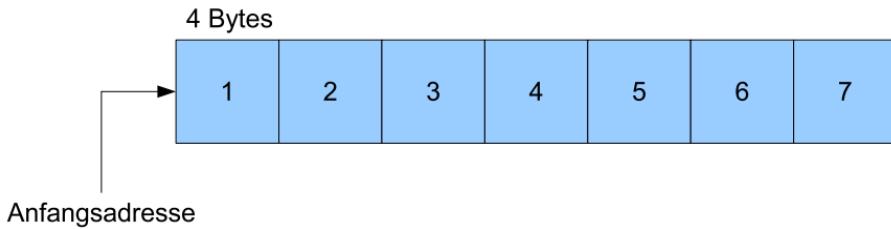
- **.data** (Zeile 4):
 - Variablen, die im Speicher (nicht im Register) abgelegt werden
 - **.word**: Festlegung des Typs (hier 32 Bit)
 - Name **var1**: An sicht frei wählbar
- **.global main** (Zeile 8):
 - Definiert das Label, das als Einsprungspunkt gilt (hier **main**)
- **adr_var1: .word var1** (Zeile 18/19):
 - Hier werden die Adressen der Variablen im Speicher in einer Variable abgespeichert
 - Wichtig: Unterscheidung zwischen Adresse und Wert
- **ldr r0, adr_var1** (Zeile 11):
 - Lädt nun die Adresse unseres Hauptspeicherwertes in ein Register
 - Hierfür nutzen wir die eben erstellte **adr_var1**
- **ldr r2,[r0]** (Zeile 13):
 - Lädt den Inhalt der Adresse in **r0** in **r2**
 - Verwendung von **[]** um dies anzuzeigen
- Variationen:
 - Zeile 13: **ldr r2,[r0,#4]**
 - Hinzufügen eines Offsets beim Laden des Wertes
 - Dies führt dazu, dass der Wert auf **r1** geladen wird (12)
 - Ausgabe des Programms ist damit 24, statt 17
 - **mov r5,#4 | ldr r2,[r0,r5]**
 - In Registern gespeicherte Konstanten auch als Offset möglich
- Zusätzliche Visualisierung:

Adressen	Speicher	Register	Namen
0x00010088	...	0x00010080	r0
0x00010084	12	0x00010084	r1
0x00010080	5	5	r2
0x0001007C	...	12	r3

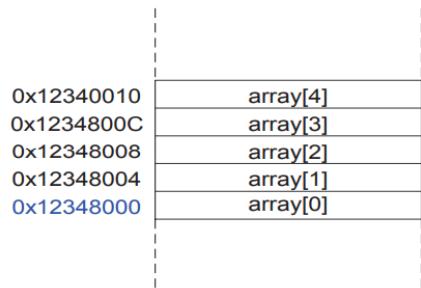
2.9 Datenfelder (Arrays)

- Eigenschaften

- Datenfelder bestehen aus mehreren Worten
- Nützlich um auf eine große Zahl von Daten gleichen Typs zuzugreifen
- Zugriff auf einzelne Elemente über Index



- Verwendung von Arrays



- Array mit 5 Elementen
- Basisadresse: Adresse des ersten Elements (0x1234800)
- Erster Schritt für Zugriff: Lade Basisadresse des Arrays in Register

- Beispiel:

```
/* Umsetzung des folgenden C-Codes in Assembler */
int i;
int scores[200];
for (i = 0; i < 200; i = i + 1)
    scores[i] = scores[i] + 10;

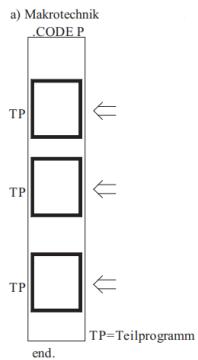
mov r0,#0x14000000 /* Speichern der Basisadresse des Arrays in r0 */
mov r1,#0           /* Verwendung als Zählervariable i */
LOOP:
    cmp r1,#200      /* i < 200 */
    bge L3            /* Falls i > 200, Verlassen des Loops */
    lsl r2,r1,#2       /* r2 = i * 4 -> Aufgrund des Offsets des Arrays von 4 */
    ldr r3,[r0,r2]     /* Laden des Wertes aus Array / r3 = scores[i] */
    add r3,r3,#10      /* r3 = scores[i] + 10 */
    str r3,[r0,r2]     /* Zurückschreiben in Speicher / r3 Quellregister (nicht Ziel) */
    /* scores[i] = scores[i] + 10 */
    add r1,r1,#1        /* i = i + 1 / Hochzählen der Laufvariable */
    b LOOP              /* Wiederholen der Schleife */
L3:
    ...
...
```

2.10 Unterprogramme

- **Einführung**

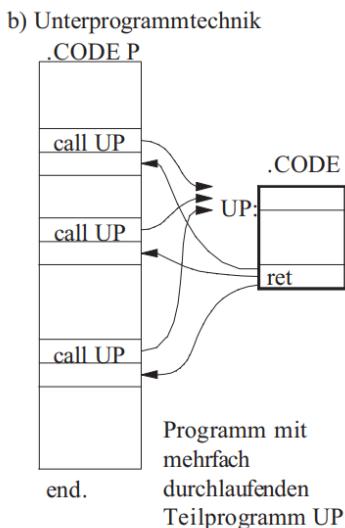
- Unterprogramme helfen bei der strukturierten Programmierung
- Betrachtung Hauptprogramm, in dem ein Teilprogramm an versch. Stellen ausgeführt werden soll
- Zwei Konzepte: Makrotechnik und Unterprogrammtechnik

- **Makrotechnik**



- Teilprogramm wird, an benötigten Stellen, einkopiert
- Zuordnung eines Namens für Teilprogramm (Makroname)
- Nennung des Makronamens an besagter Stelle (Makroaufruf)

- **Unterprogrammtechnik**



- Teilprogramm nur einmal im Code vorhanden
- Kennzeichnung durch Marke (Unterprogrammname)
- Aufruf: Sprungbefehl + Marke
- Rückkehr in aufrufendes Programm nach Ausführung
⇒ durch Sprungbefehl auf Rückkehradresse
- Rückkehradresse wird an anderer Stelle gespeichert
- Beachten der Sichtbarkeit von Variablen (global vs lokal)

- **Funktions- und Prozeduraufruf**

- Aufrufer:

- Ursprung des Funktionsaufrufs
- Übergibt Argumente (aktuelle Parameter) an Aufgerufenen
- Springt Aufgerufenen an

- Aufgerufener:

- Aufgerufene Funktion
- Führt Funktion/Prozedur aus
- Gibt Rückgabewert an Aufrufer zurück
- Darf keine Register oder Speicherstellen überschreiben, die im Aufrufer genutzt werden
 - Beachten mit Sorgfalt und vorhandenem Konzept
 - Genutzte Register sollten gesichert werden, um danach wieder zu überschreiben

- Beispiel:

```

/* Übersetzen des folgenden C-Codes in Assembler */
int main() {
    int y;
    y = diffofsums(14, 3, 4, 5);
}

int diffofsums(int f, int g, int h, int i){ /* 4 formale Parameter */
    int result;
    result = (f + g) - (h + i);
    return result;
}

/* ASSEMBLER */
/* r4 = y */
main:
    mov r0,#14      /* Argument 0 = 14 */
    mov r1,#3       /* Argument 1 = 3 */
    mov r2,#4       /* Argument 2 = 4 */
    mov r3,#5       /* Argument 3 = 5 */
    bl diffofsums   /* Funktionsaufruf / bl: branch and link */
/* Schreibt die Rückkehradresse des folgenden Befehls mov in link register (r14) */
    mov r4,r0       /* y = Rückgabewert */

-----
diffofsums:
    add r8,r0,r1    /* Überschreiben von r8 / Kein Sichern der Werte vorher */
    add r9,r2,r3    /* Selbiges gilt für r9 */
    sub r4,r8,r9
    mov r0,r4       /* Ablegen von Rückgabewert in r0 (return value register) */
    mov pc,lr        /* Übergabe der Rückkehradresse an Program Counter */
/* Program Counter führt dann den nächsten Befehl (mov r4,r0) aus */

```

2.11 Stack

- Eigenschaften des Stacks

- Speicher für temporäres Zwischenspeichern von Werten
- LIFO-Konzept ("last in, first out")
- Dehnt sich aus, falls mehr Daten gespeichert werden müssen
- Zieht sich zusammen, wenn weniger Daten gespeichert werden müssen
- Wächst bei ARM nach unten (von hohen zu niedrigen Adressen)
- Verwendung des Stackpointers sp (r13)
- StackPointer zeigt auf letztes auf dem Stack abgelegtes Element

- Verwendung des Stacks bei Unterprogrammen

- Beispiel diffofsums:

```
diffofsums:
    add r8, r0, r1
    add r9, r2, r3
    sub r4, r8, r9
    mov r0, r4      /* Rueckgabewert in r0 */
    mov pc, lr      /* Ruecksprung zum Aufrufer */
```

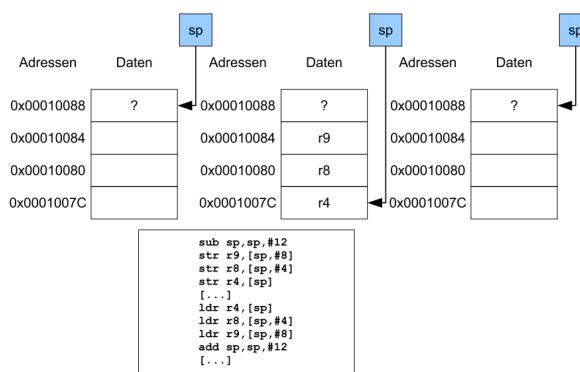
- Problem hier: diffofsums überschreibt r8, r9, r4
- Unterprogramme dürfen aber keine unbeabsichtigten Seiteneffekte haben
- Vorherige Werte in r8, r9 und r4 gehen hierbei aber verloren
- Lösung: Register auf Stack Zwischenspeichern

```
diffofsums:
    sub sp, sp, #12      /* Speicher auf Stack reservieren (3 Adressen "abziehen") */
    str r9, [sp, #8]      /* Speichern an oberster freier Stelle im Stack */
    str r8, [sp, #4]
    str r4, [sp]          /* Speichern an unterster freier Stelle im Stack */
    /* Abspeichern der Werte in Benutzungsreihenfolge hier als Konvention */
    add r8, r0, r1        /* Berechnungen durchführen */
    add r9, r2, r3
    sub r4, r8, r9
    mov r0, r4            /* Rückgabewert in r0 */

    /* Wiederherstellen der Werte nun in umgekehrter Reihenfolge */
    ldr r4, [sp]           /* Wiederherstellen von r4 */
    ldr r8, [sp, #4]       /* Wiederherstellen von r8 */
    ldr r9, [sp, #8]       /* Wiederherstellen von r9 */
    add sp, sp, #12        /* Freigabe von Speicher auf dem Stack */

    mov pc, lr            /* Rücksprung zum Aufrufer */
```

Veränderung des Stacks während diffofsums



- Verwendung des Stacks auch bei Unterprogrammaufrufen
 - Das LinkRegister muss vor Unterprogrammaufrufen gesichert werden

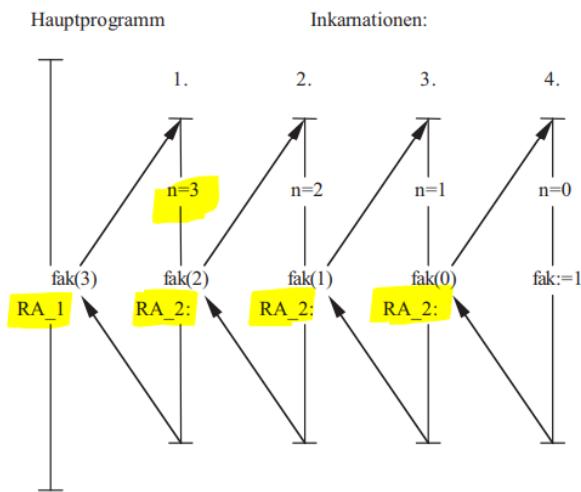
`main:`

```
...  
push {lr}          /* push ist hier nur eine Pseudoinstruktion für str */  
bl diffofsums    /* Das LinkRegister wird beim Programmaufruf verändert */  
...  
pop {lr}          /* pop ist hier die Pseudoinstruktion für ldr */  
bx lr
```

 - push und pop sind auch mit Operandenliste möglich
 - `push {r9, r8, r4}`
 - Allerdings muss hierbei das "poppen" in umgekehrter Reihenfolge beachtet werden

2.12 Rekursion

- Graphische Betrachtung



- Inkarnation: Ablauf eines Unterprogrammes
 - Zwei verschiedene Rückkehradressen:
 - **RA_1**: Adresse im Hauptprogramm
 - **RA_2**: Adresse im Unterprogramm
 - **RA_2** ist immer dieselbe Adresse, da immer selber Code

- Code: Fakultätsberechnung

```

.global main

main:          /* Hauptprogramm */
    push{lr}      /* Sicherung lr */
    mov r0, #3     /* fak von 3 */
    bl fak        /* Aufruf von fak */
RA_1:         mov r4, r0      /* RA_1 ist hier kein sinnvoller Code, nur für Adressen hier */
    mov r0, r4
    pop {lr}
    bx lr

fak:          sub sp, sp, #8   /* Stackspeicher reservieren */
    str r0, [sp, #4]  /* Sichern von r0 */
    str lr, [sp]       /* Sichern lr */
    cmp r0, #1        /* Überprüfung Rekursionsende */
    blt else
    sub r0, r0, #1    /* n = n - 1 */
    bl fak           /* Funktionsaufruf */
RA_2:         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    /* Freigabe Stackspeicher */
    bx lr
else:         mov r0, #1      /* Rekursionsanker */
    b fin

```

- Aufrufer:

- Lege Aufrufparameter in Register oder auf Stack ab
- Sichere notwendige Register auf dem Stack (lr)
- Rufe Unterprogramm auf (bl)
- Stelle gesicherte Register wieder her (lr)
- Verwendung von Rückgabewert

- Aufgerufener:

- Sichere zu erhaltende Register auf dem Stack
- Führe Unterprogrammrechnung aus
- Rückgabewert in r0 legen
- Wiederherstellen der gesicherten Register
- Rücksprung zum Aufrufer

2.13 Compilieren, Assemblieren und Linken

- Optimierungseinstellungen

- Generieren des Assemblercodes mithilfe von `gcc -S code.c` führt zu viel "unnötigem" Code
- z.B. das Speichern von immediate Values auf dem Stack etc
- Optimierungsstufen (`gcc -S -O1 code.c`) erzeugen meist "weniger" Code
- Die Übersetzung eines Programmes kann also viele verschiedene Ergebnisse haben
- Außerdem werden nicht unbedingt alle Elemente einer Hochsprache in Assembler sichtbar

- OpenMP (Einschub)

- Threadparalleles Arbeiten auf Rechnersystemen mit gemeinsamen Adressraum
- Gut geeignet für Multicore-Architekturen
- Programm verzweigt automatisch bei parallel-ausführbarem Code in zusätzliche Threads
- Am Schluss werden diese Threads wieder zu einem einzelnen zusammengeführt
- **Fork-Join-Programmiermodell**
- Verwenden in der Praxis:
 - Einbinden von `#include<omp.h>`
 - Compileraufruf: `gcc -fopenmp name.c`
 - Setzen Umgebungsvariable für Threadanzahl: `OMP_NUM_THREADS=2`
- Programme lassen sich aber eher selten sehr gut parallelisieren

- Assemblerprogramm

- Definition:
 - Programm, das die Aufgabe hat, Assemblerbefehle in Maschinencode zu transformieren
 - symbolischen Namen (Labels) Maschinenadressen zuzuweisen
 - Erzeugung einer oder mehrerer Objektdateien
- Crossassembler
 - Assembler läuft auf Rechnersystem X, generiert aber Maschinencode für Platform Y
 - Verwendung im Bereich der Embedded Systems
- Disassembler
 - Übersetzung von Maschinencode in Assemblersprache
 - Verlust von Kommentaren und symbolischen Namen

- Schrittweiser Assembliervorgang

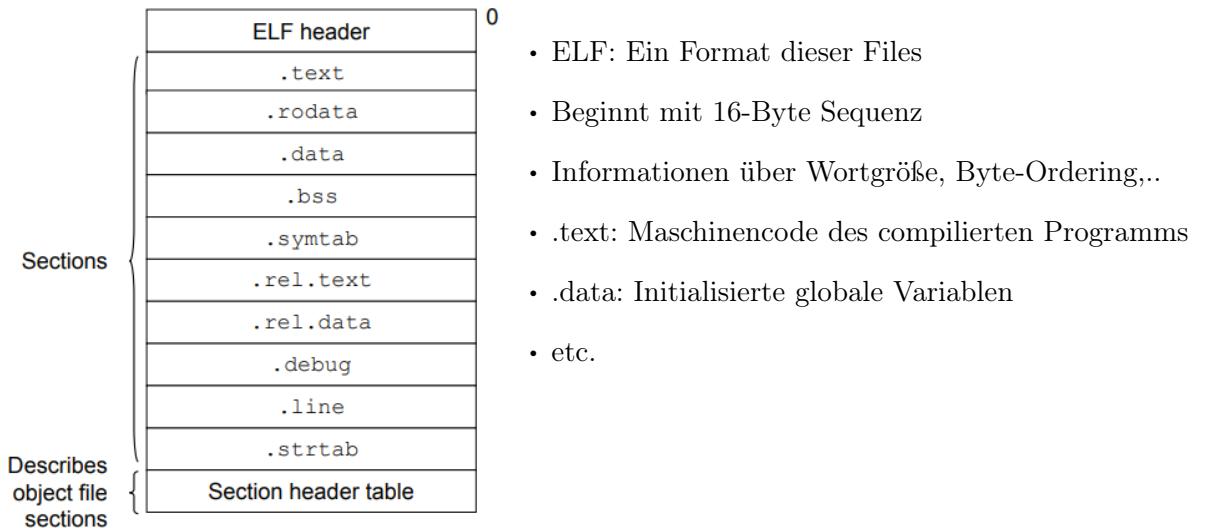
- 1. Schritt:
 - Auffinden von Speicherposition mit Marken (Beziehung zwischen Adresse und Namen bekannt)
 - Übersetzung jedes Assemblerbefehls durch OPCodes, Register und Marken in legale Instruktion
- 2. Schritt
 - Erzeugung einer oder mehrerer Objektdateien
 - Enthalten Maschinencode, Daten, Verwaltungsinformationen
 - Jedoch meist nicht ausführbar (Verweise auf andere Funktionen etc.)
- Probleme beim 1. Schritt
 - Nutzen von Marken, bevor sie definiert sind (Unbekannte Adressen)
 - Lösung: **Two-Pass**
 - Assembler macht 2 Läufe über das Programm
 - 1. Lauf: Zuordnen von Maschinenadressen
 - 2. Lauf: Erzeugen der Codes

- Probleme beim 2. Schritt (Erzeugen des Objektdatei)

- 1. Fall:
 - Assembler verwendet **absolute** Adressen und eine Objektdatei
 - Laden unmittelbar möglich, Speicherort muss jedoch vorher bekannt sein
 - Nachteil: Verschieben des Programms nicht möglich
- 2. Fall:
 - Assembler verwendet **relative** Adressen und ggf. mehrere Programm-Segmente als Eingabe
 - Assembler Ausgabe: ≥ 1 Objekt-Dateien
 - Adressen werden relativ zu Objektdateien vergeben
 - Deswegen sind weitere Transformationsschritte notwendig (Binder/Linker/Lader)

- **Aufbau eines Objekt-Programms**

- Verschiedene Arten von Objekt-Programmen:
 - Relocatable (verschiebbare) Object Files:
Enthalten binären Code und Daten in einer Form, die mit anderen verschiebbaren Objekt-Files zu einem ausführbaren Objekt-File zusammengeführt werden können. Diese Files werden in der Regel generiert.
 - Executable Object Files:
Enthalten binären Code und Daten in einer Form, die direkt in den Speicher kopiert und ausgeführt werden kann.
 - Shared Object Files:
Spezieller Typ von Relocatable Object Files, welche in den Speicher geladen werden können und dynamisch mit anderen Object-Files zusammengeführt werden können.
- Aufbau eines ELF relocatable object files



- Beispiel einer ELF-Header-File (16 Bytes)
 - `as -o prog.o prog.s` (Übersetzung eines C-Programms)
 - `readelf -h prog.o` (Lesen ELF-Header / -h für Header)

```

ELF Header:
  Class: ELF32
  Data: 2's complement little endian
  Version: 1 (current)
  OS/ABI: UNIX - System V
  ABI Version: 0
  Type: REL (Relocatable file)
  Machine: ARM
  Version: 0x1
  Entry point address: 0x0
  Start of program headers: 0 (bytes into file)
  Start of section headers: 348 (bytes into file)

```

- `readelf -a prog.i` (Übersicht über wichtigsten Einträge)
- `objdump -S prog.o` (Rückgabe des Maschinencodes)

```

Schleife.o:      file format elf32-littlearm
Disassembly of section .text:
00000000 <main>:
    0: e3a00001  mov r0, #1
    4: e3a01000  mov r1, #0
00000008 <WHILE>:
    8: e3500c01  cmp r0, #256 ; 0x100
    c: 0a000002  beq 1c <DONE>
   10: e1a00080  lsl r0, r0, #1
   14: e2811001  add r1, r1, #1
   18: eaaffffa  b 8 <WHILE>
0000001c <DONE>:
   1c: e1a00001  mov r0, r1
   20: e12fff1e  bx lr

```

- Links ist der Maschinencode mit 8 Byte (32 Bit) in Hexa zu sehen

- **Binder/Linker und Lader**

- Definition Binder/Linker
 - Erzeugung eines ausführbaren Objektprogramms aus einzelnen verschiebbaren Objekt-Files
 - Hierzu auflösen der offenen externen Referenzen
- Definition Lader
 - Systemprogramm, das die Objektprogramme in den Speicher lädt und die Ausführung anstößt
 - Kopieren des Objektprogrammes in den Speicher
 - Verschiedene Arten des Ladevorgangs:
 - absolutes Laden (absolute loading)
 - relatives Laden (relocatable loading)
 - dynamisches Laden zur Laufzeit (dynamic run-time loading)

- **Laufzeitanalyse von C-Programmen**

- Erinnerung: Operationen auf den Registern sind schneller als Operationen auf Hauptspeicher
- Möglichkeit des Programm Profiling hier:
 - Hauptprogramm, das zwei Funktionen (eine iterativ, andere rekursiv) aufruft
 - gcc kann hier bei der Bestimmung der Laufzeit weiterhelfen
 - `gcc -pg -o function_fak function_fak.c` (-pg ist ein run-time flag)
 - Danach Aufruf des Programms (dauert etwas länger)
 - `gprof function_fak gmon.out > analysis.txt` (Wertet die Profile-Datei aus)
 - Diese splittet die Zeit der Unterprogramme auf und zeigt die Laufzeiten an

3 Mikroarchitekturen von Rechnersystemen

3.1 Begrifflichkeiten und Grundlagen

- Drei Phasen der Befehlsausführung

- Befehle, als auch Daten stehen im Speicher
- *Befehlsholphase*: Prozessor liest die Befehle aus dem Speicher
- *Befehlsdekodierung*: Dekodierung des Befehls, nachdem dieser in ein Register geholt wurde
- *Befehlsausführung*: Ausführung des Befehls, danach Holen des nächsten Befehls

- Takt/Taktfrequenz

- Gemeinsame Zeitbasis der Komponenten eines Rechnersystems: Takt
- Beim Takt handelt es sich um ein Rechtecksignal
- Dient der Synchronisation der Komponenten eines Rechnersystems
- Taktfrequenz: $f = \frac{1}{T}$
- Je höher Taktfrequenz, desto schneller werden Daten verarbeitet
- Leistungssteigerung durch Erhöhen der Taktfrequenz (CMOS: $P = U^2 \cdot f \cdot C_l$)

- Terminologie

- *ISA*: instruction set architecture (Menge der verfügbaren Befehle)
- *RISC*: reduced instruction set computer (kleine ISA)
- *CISC*: complex instruction set computer (aufwendige ISA)
- *SIMD*: single instruction multiple data (paralleles Arbeiten)
- *VLIW*: very long instruction word (static multi-issue)
- *μarch*: microarchitecture (Hardware, die ISA abarbeitet - ISA Implementierung)
 - *IPC*: Anzahl der Befehle pro Zyklus
 - *ILP*: Pipelining für Parallelismus
 - Sprungvorhersagen,... etc

3.2 Analyse der Rechenleistung

- Mikroarchitektur

- *Mikroarchitektur*: Hardware-Implementierung einer Architektur
- *Datenpfad*: Verbindet funktionale Blöcke (Speicher/Prozessor)
- *Kontrollpfad*: Steuersignale/Steuerwerk
- *Eintakt-Implementierung*: Jeder Befehl wird in einem Takt ausgeführt
- *Mehrtakt-Implementierung*: Jeder Befehl wird in Teilschritte zerlegt
- *Pipeline-Implementierung*: Teilschritte + Parallele Ausführung der Teilschritte

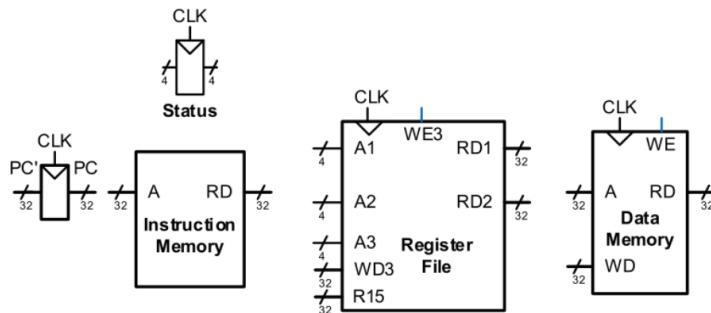
- Rechenleistung eines Prozessors

- Ausführungszeit eines Programms
 - ⇒ $Ausfuehrungszeit = (\#Instruktionen) \cdot \left(\frac{Takte}{Instruktion}\right) \cdot \left(\frac{Sekunden}{Takt}\right)$
- CPI: Takte/Instruktion
- Taktperiode: Sekunden/Takt
- IPC: 1/CPI = Instruktionen/Takt

- Mikroarchitektur ARM

- Befehlsmenge: (ldr, add, sub etc)
- Architekturzustand: Sichtbare Daten auf Ebene der Architektur
- Sichtbare Daten bestimmen den Zustand (Program Counter, 16 Register, etc)

- Elemente des Architekturzustands



- Register File

- A1,A2,... - Registeradresse (4 Bit → 16 Möglichkeiten)
- RD1,RD2,... - Register Data (Ausgabedaten)
- WD3 - Write Data 3 (Eingabedaten)

- Status

- Gibt Flags an mit 4 Bits (Sign, Zero, Overflow, Carry)

- Program Counter (PC)

- Instruction Memory

- A - Adressen
- RD - Read Data (Instruktionen)

- Data Memory

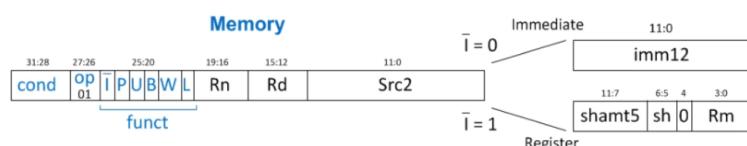
- WE - Write Enable (Benötigt für Schreibprozesse - Steuersignal)

- Von-Neumann-/Harvard-Architektur

- Von-Neumann: gemeinsamer Speicher für Befehle und Daten
- Harvard-Architektur: Befehlsspeicher und Datenspeicher sind getrennt
- Verhalten des Speichers: Asynchrones Lesen möglich, jedoch nur Synchrones Schreiben

- Vorgehensweise und Bitfelder eines Befehls

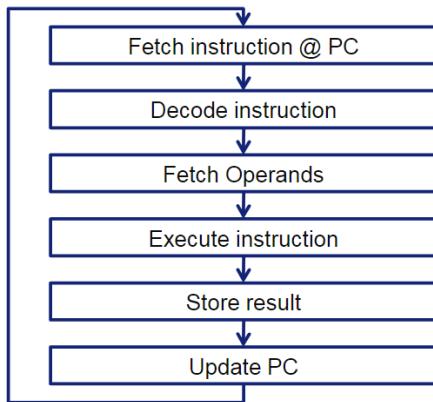
- Hier: Befehl ldr
- Allgemein: ldr Rd, [Rn, imm12] (imm12: immediate value - 12 Bit)



- 32 Bit Länge -> z.B.: E13A0110 (In Hexa)
- **Rd**: Adresse register destination
- Ganz hinten entweder Register oder Direktwert (festgelegt durch **I**)
- Vorne: Conditions für die Ausführung (möglich für jeden Befehl)

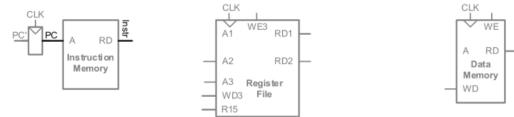
3.3 Eintakt-Prozessor

- Phasen der Befehlsausführung

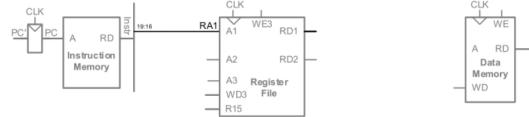


- Ablauf der Befehlausführung anhand eines Eintaktprozessors

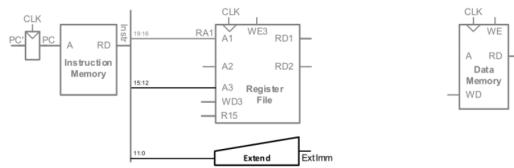
1. Befehl holen



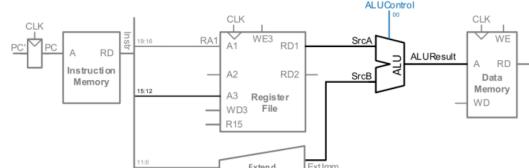
2. Lesen der Quelloperanden vom Registerfeld



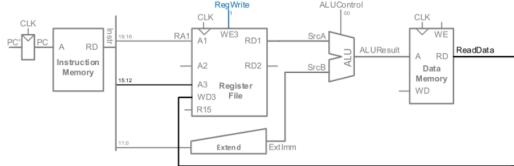
3. Erweiterung Direktwert (32 Bit)



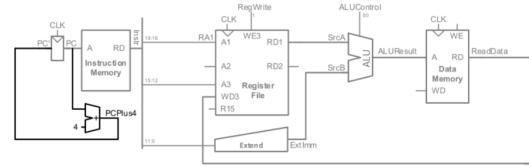
4. Berechnung der Speicheradresse



5. Lesen aus Speicher und Schreiber in Register

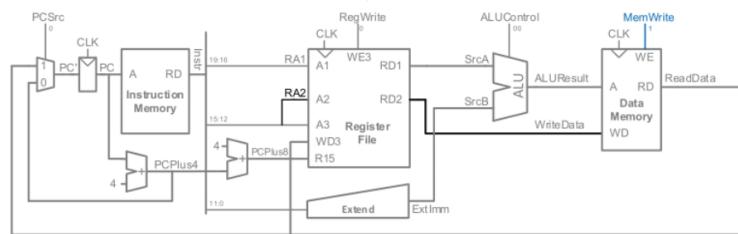


6. Berechnung der nächsten Befehladresse



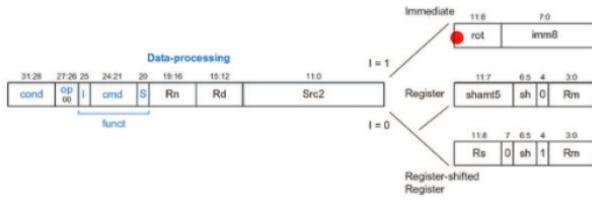
- Befehl str

- STR Rd, [Rn, imm12] (Rd hier Quellregister nicht Ziel - Rd nach Speicher schreiben)
- Erweiterung des Datenpfades zur Realisierung von **str**
- Schreibe DATUM vom registerfeld in den Datenspeicher



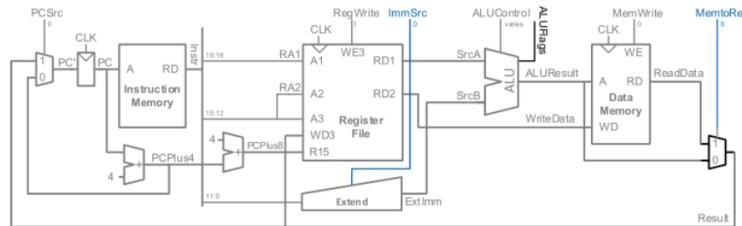
- Arithmetische und logische Befehle

- ADD Rd, Rn, imm8



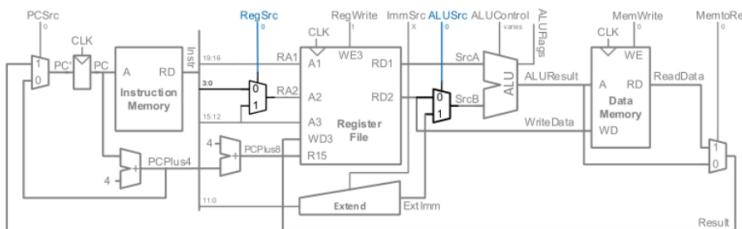
- immediate Src2* (Src2 hier als Direktwert)

⇒ Steuersignal *ImmSrc* regelt um wieviele Bits erweitert wird
 ⇒ 0: Erweiterung um 24 Bit (ALU-Befehle) | 1: Erweiterung um 20 Bit (ldr,str Befehle)
 ⇒ Erweiterung um Multiplexer
 ⇒ Schreibe ALUResult in Registerfeld statt Speicher (je nach Multiplexer)

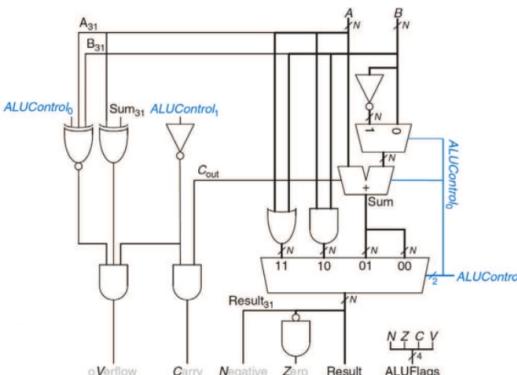


- Register Src2* (Src2 hier als Register)

⇒ Multiplexer vor Register File und nach Extend



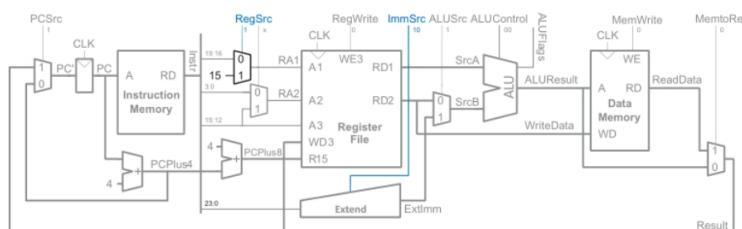
- Arithmetisch Logische Einheit



ALUControl _{1:0}	Function
00	Add
01	Subtract
10	AND
11	OR

- Sprungbefehl b

- Berechnen der Sprungadresse
- BTA = (ExtImm) + (PC + 8)
- ExtImm = Imm24 « 2 inkl. Vorzeichenerweiterung +



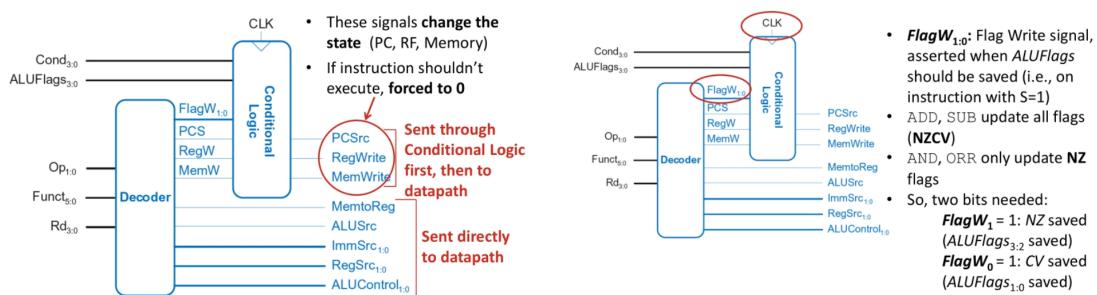
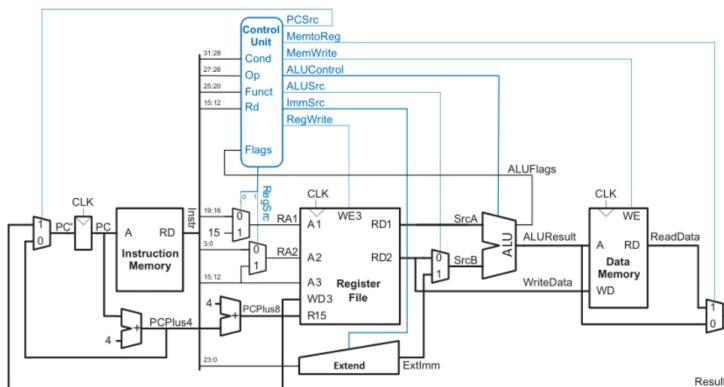
- ExtImm

- Unterschiedliche Funktionen werden benötigt
- Erweiterung der Werte auf 32 Bit wird benötigt

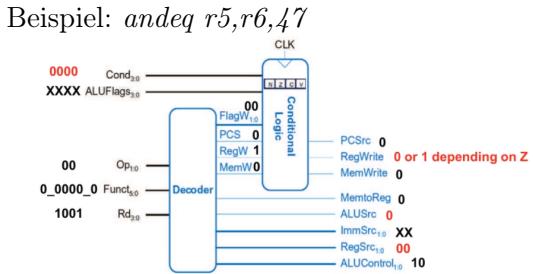
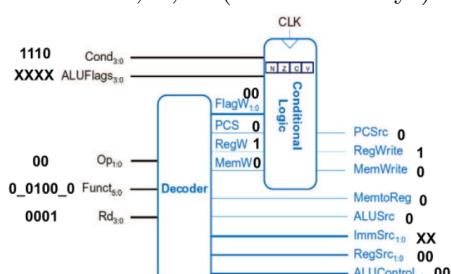
ImmSrc _{1:0}	ExtImm	Description
00	{24'b0, Instr _{7:0} }	Zero-extended imm8
01	{20'b0, Instr _{11:0} }	Zero-extended imm12
10	{6{Instr ₂₃ }, Instr _{23:0} }	Sign-extended imm24

6{Instr₂₃} → Erstes Bit 6x (Sprungbefehl)

- Kontrolleinheit



Beispiel: add r1,r2,r3 (1110 → always)



- Condition Field

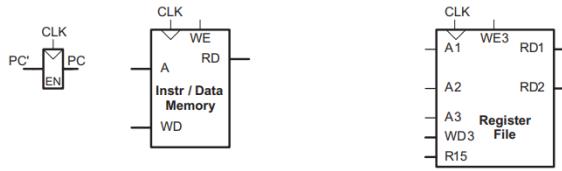
- Verwendung dieser Codes für jeden Befehl als Condition möglich

Code	Suffix	Flags	Meaning
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear or Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equals V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

3.4 Mehrtakt-Prozessor

- Zustandselemente im Mehrtakt-Prozessor

- Ersetze getrennte Instruktions- und Datenspeicher (Harvard-Architektur)
- durch einen gemeinsamen Speicher (Von-Neumann-Architektur)

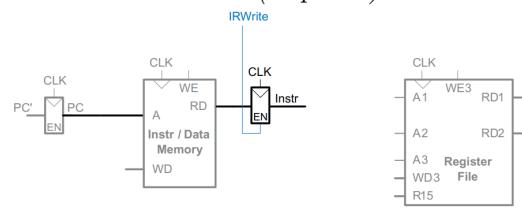


- Ablauf der Befehlausführung anhand eines Mehrtaktprozessors

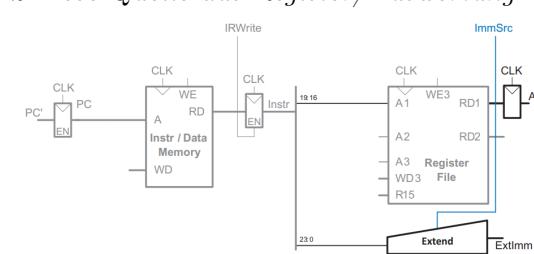
- Register werden hier zum Zwischenspeichern von Werten genutzt

⇒ Damit Wert aufgrund von Takt nicht verloren geht

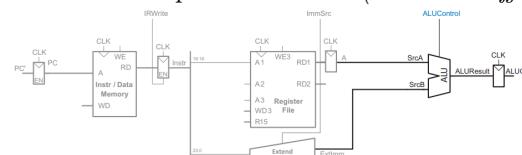
- Hole Instruktion (Bsp. ldr)



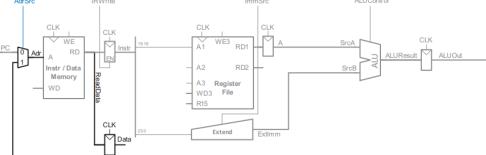
- Lese Quelle aus Register/Auswertung Direktwert



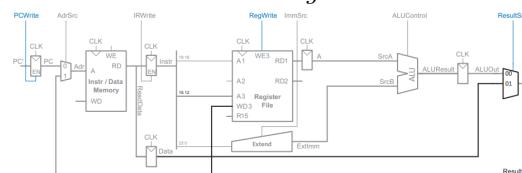
- Berechne Speicheradresse (Basis + Offset)



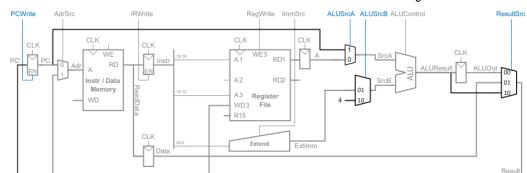
- Lese Daten aus Speicher



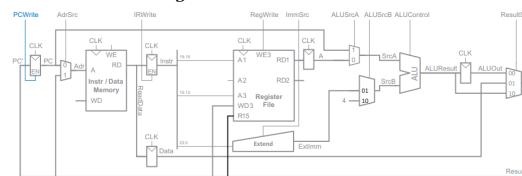
- Schreibe Daten in Register



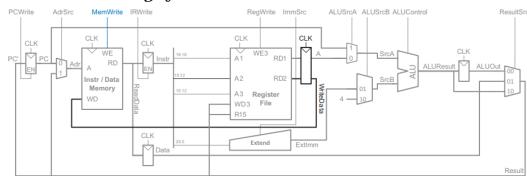
- Berechne Adresse nächster Befehl



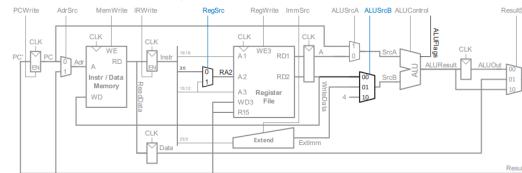
- Behandlung von r15



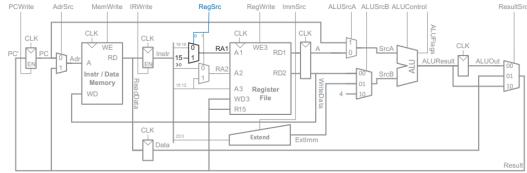
- Erweiterung für str



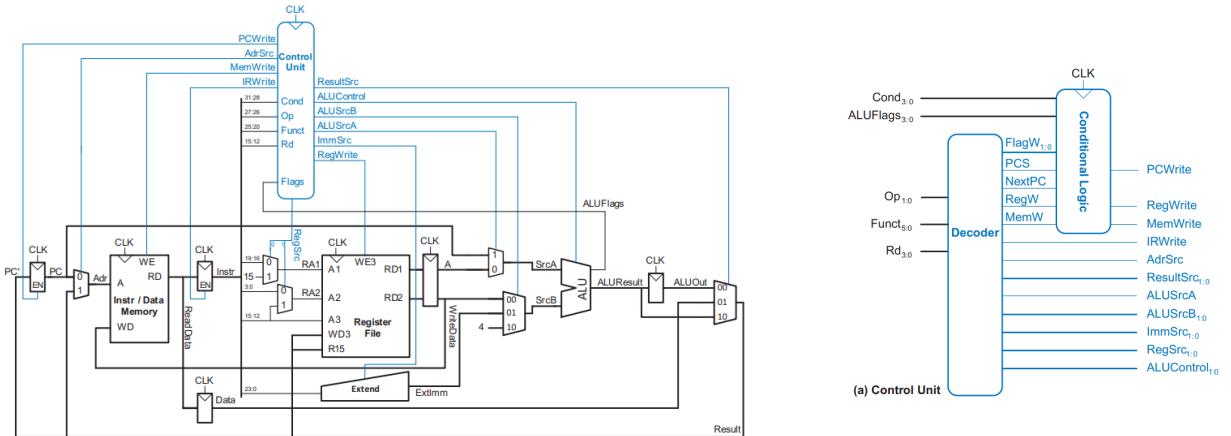
- Arithmetische/logische Befehle



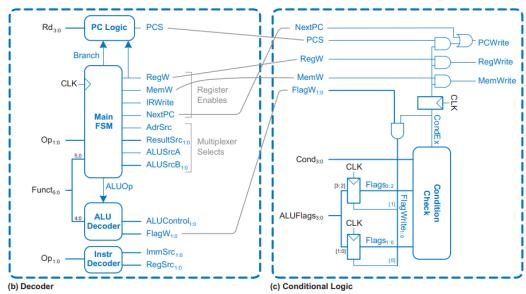
- Sprungbefehl b



- Datenpfad und Kontrolleinheit



Kontrolleinheit - Detail



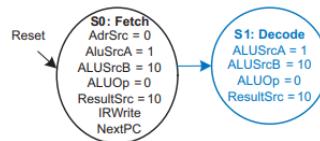
- Entwicklung des Steuerwerks

- Setzen gewisser Steuersignale beim Holen eines Befehls ist notwendig
- Steuersignale sind solange 1, wie sie in Zuständen dediziert auf 1 gesetzt sind

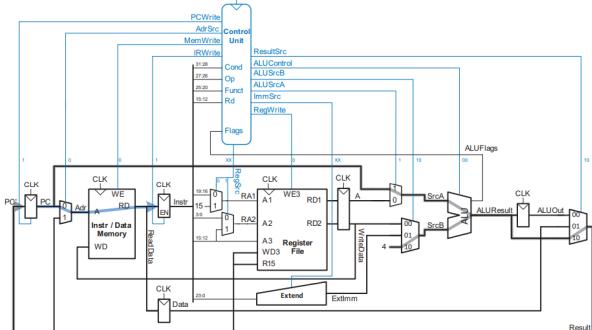


Table 7.6 Instr Decoder logic for RegSrc and ImmSrc

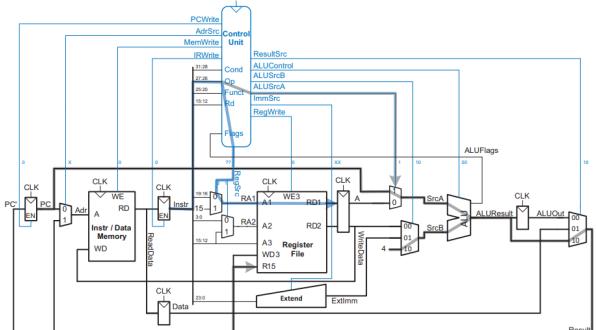
Instruction	Op	Funct ₅	Funct ₄	RegSrc ₁	RegSrc ₀	ImmSrc ₁
LDR	01	X	1	X	0	01
STR	01	X	0	1	0	01
DP immediate	00	1	X	X	0	00
DP register	00	0	X	0	0	00
B	10	X	X	X	1	10



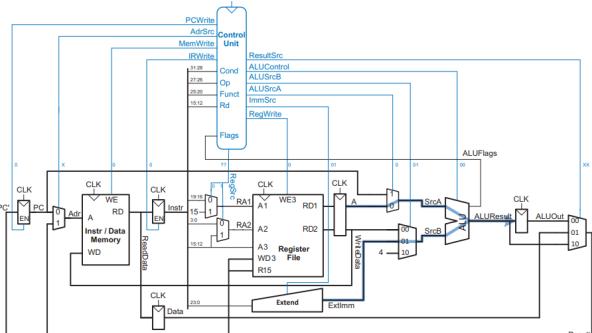
1. Fetch

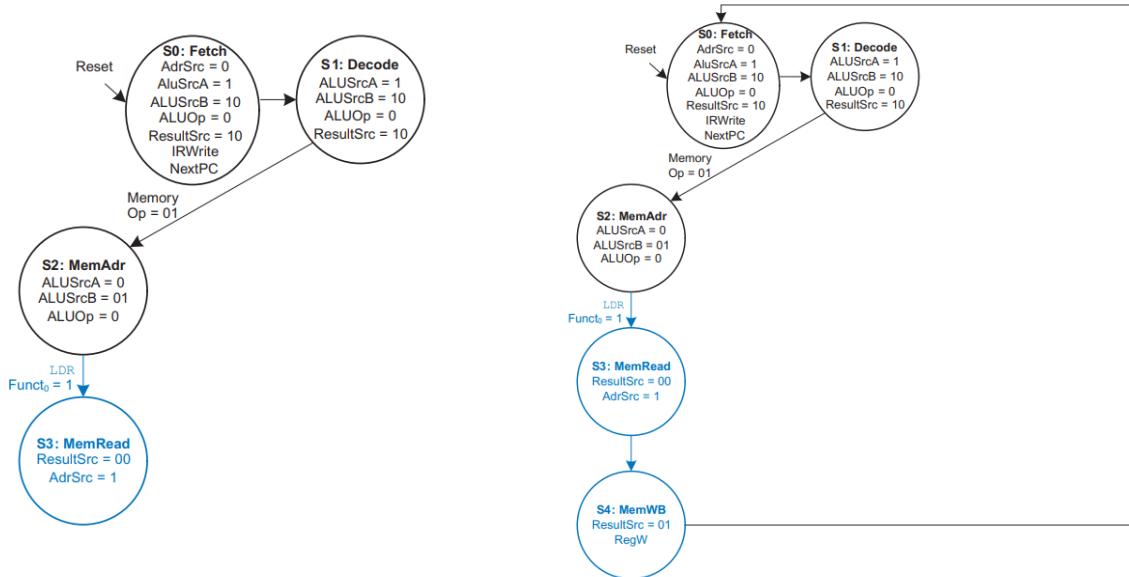


2. Decode ldr



3. Execute





- *Fetch* und *Decode* sind für alle Befehle gleich
- *S4:MemWB* schreibt das Ergebnis, danach wird zum nächsten Befehl übergegangen
- 5 versch. Phasen (können z.B. 5 Takte sein → Überlagern mit Pipelining)

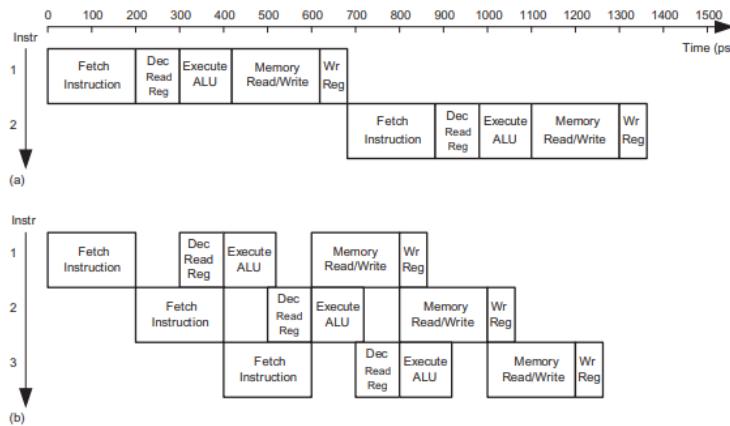
- **Eintakt- vs Mehrtaktprozessor**

- *Gemeinsamkeiten*
 - **Datenpfad**: verbindet funktionale Blöcke
 - **Kontrollpfad**: Steuersignale/Steuerwerk
- *Eintakt-Prozessor*
 - + einfach
 - Taktfrequenz wird durch langsamste Instruktion bestimmt
 - Drei ALUs und zwei Speicher
- *Mehrtakt-Prozessor*
 - + höhere Taktfrequenz
 - + einfache Instruktionen laufen schneller
 - + bessere Wiederverwendung von Hardware in versch. Taktten
 - aufwendigere Ablaufsteuerung

3.5 Pipeline-Prozessor

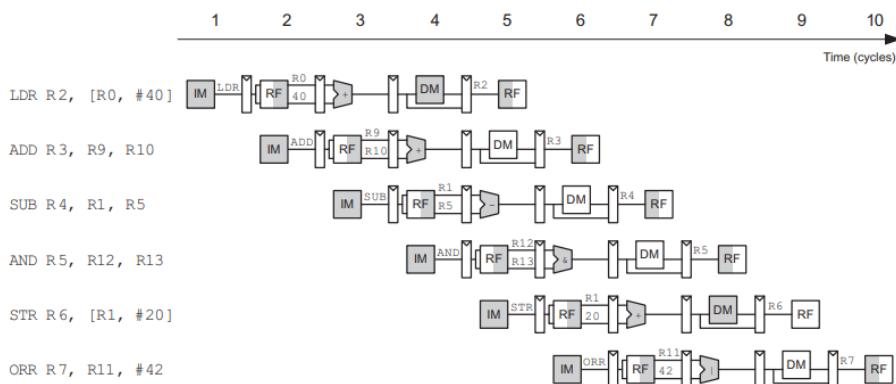
- Befehlausführung

- 5 Phasen bei ARM
 - Instruction Fetch
 - Instruction Decode, Read Register
 - Execute ALU
 - Memory Read/Write
 - Write Register
- Idee: Pipelining dieser Befehlausführung (siehe unten)



- Abstrakte Darstellung

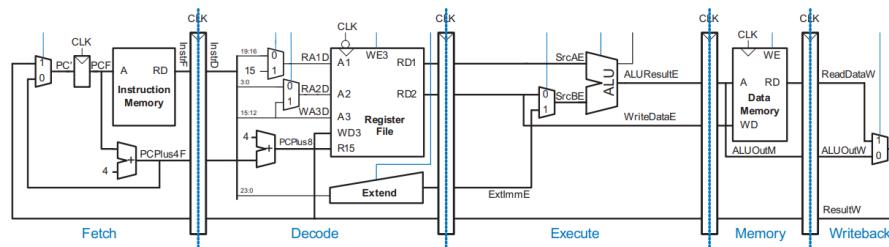
- Vereinfachte Darstellung des Datenpfads der Mikroarchitektur



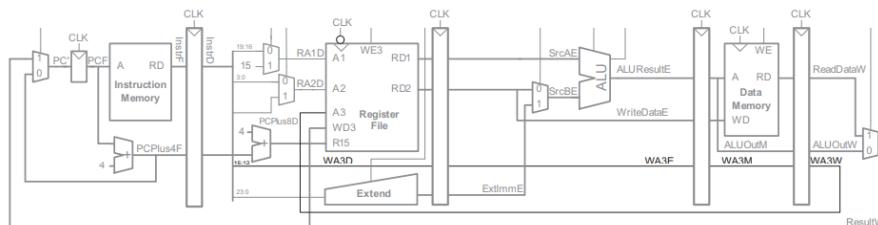
- *IM*: Instruction Memory
- *RF*: Register Field
- *DM*: Data Memory
- Hinterer Teil grau: lesen / vorderer Teil grau: schreiben

- Datenpfad

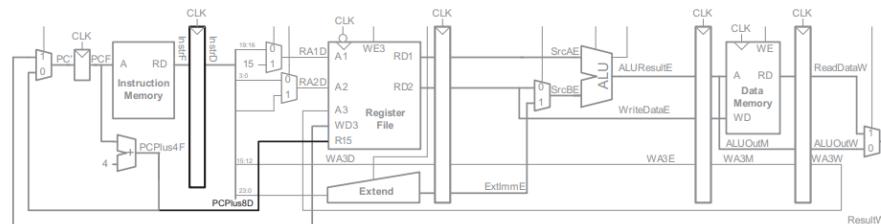
- Einführung von Registern → Führt zur Überlappung der Befehlsphasen



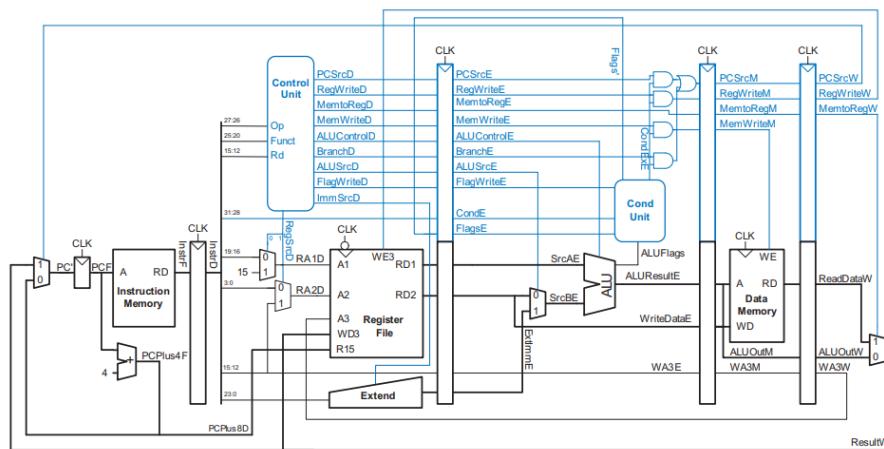
- Führung der Zieladresse über Register (da diese sonst verloren geht)



- Optimierung der Program Counter Logik



- Kontrolleinheit (blau: Kontrollpfad)



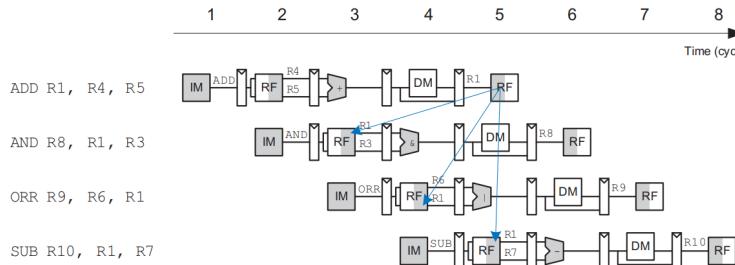
3.6 Ausnahmebehandlung - Hazards

- Information

- Treten auf wenn Instruktion auf Ergebnis vorhergehender wartet, dieses aber noch nicht vorhanden ist
- Data Hazard*: z.B. neuer Wert von Register noch nicht in Registerfeld eingetragen
- Control Hazard*: Unklar, welche Instruktion als nächstes ausgeführt wird (Sprünge)

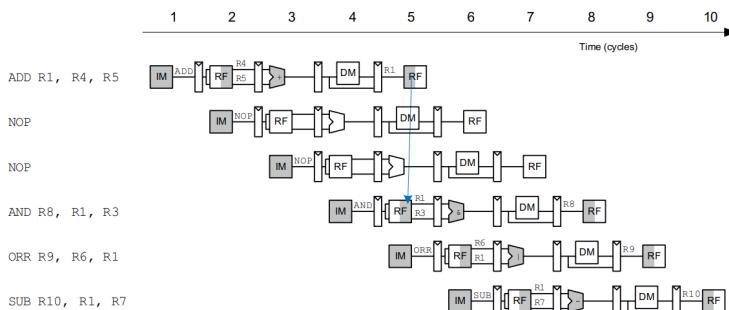
- Data Hazard

- Hier: *Read-After-Write Hazard (Raw)* ($r1$ muss vor Lesen geschrieben werden)



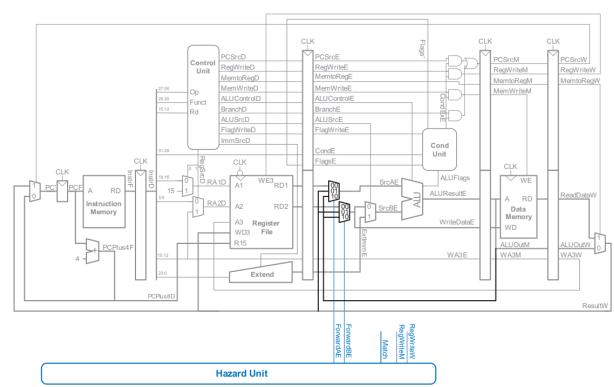
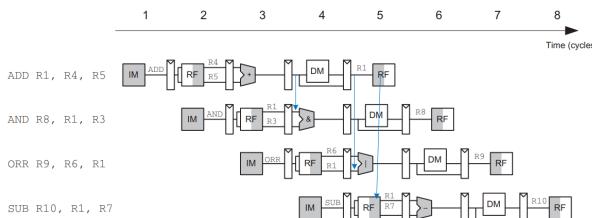
- Möglichkeiten:

- Plane Wartezeiten von Anfang an ein (Einfügen von *nops* (no operations) zur Compilezeit)
- Stelle Maschinencode zur Compile-Zeit um (*scheduling/reordering*)
- Leite Daten zur Laufzeit schneller über Abkürzungen weiter (*bypassing/forwarding*)
- Halte Prozessor zur Laufzeit an bis Daten da sind (*stalling*)
- Einfügen von *nops*



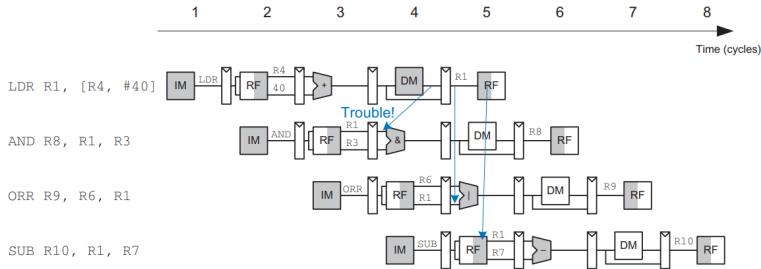
- Bypassing/Forwarding

- Früheres Einfügen in anderen Pipelinevorgang
- Benötigt Erweitern der Datenpfade

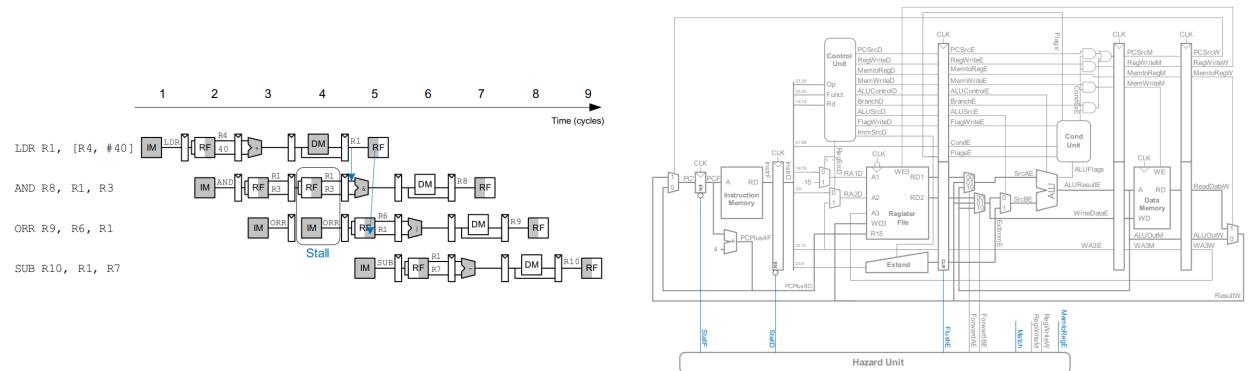


- Stall

- Wird z.B. bei 1dr benötigt

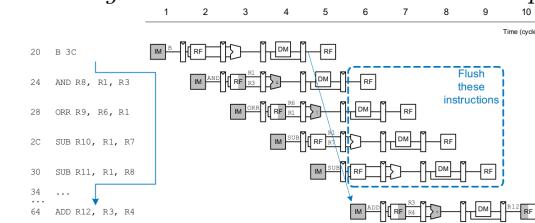


- Lösung: Pipeline-Stall
- Nachteil: Mehr Cycles / Benötigt auch Anpassung der Datenpfade

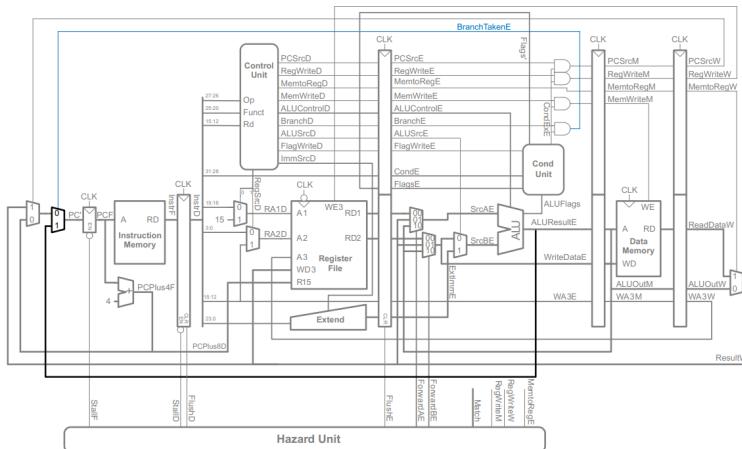


• Control Hazards

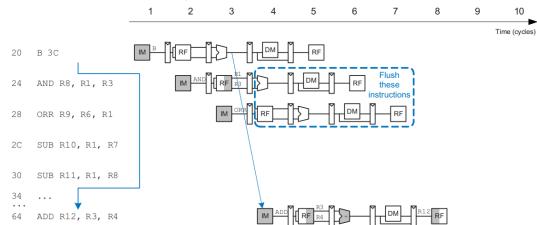
- "Leerung" der Instruktionen bei Sprungbefehl
Leerung der anderen Instruktionen bei Sprung



- Anpassung der Datenpfade



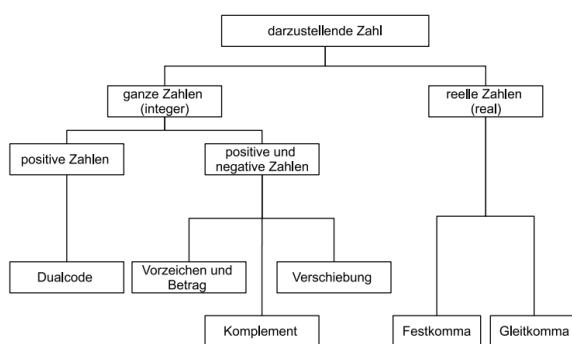
Optimierung (frühere Feststellung des Sprungs)



4 Gleitkommazahlen/Gleitkommarechenwerte

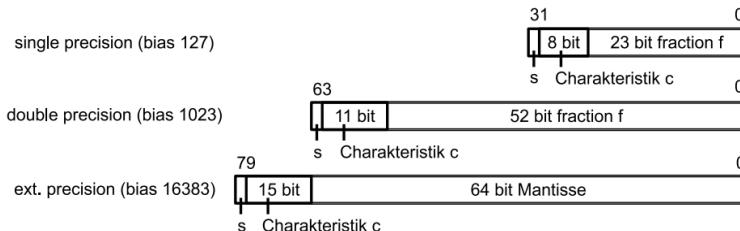
4.1 Gleitkommazahlen

- Zahlendarstellung in Rechnersystemen
 - Unterscheidung zwischen ganzen Zahlen und reellen Zahlen



- Darstellung reeller Zahlen

- *Festkommadarstellung*
 - Weglassen des Kommas bei der internen Zahlenbildung
 - Definierung zu Programmbeginn, bleibt daraufhin fest
- *Gleitkommadarstellung*
 - Kommastelle ist Bestandteil der Zahl und kann sich verändern
 - Standardisiert nach ANSI/IEEE 754 (Keine Angabe des Kommas notwendig)
 - Weltweit durchgesetzt - Grundlage für weltweiten Datenaustausch/Rechenwerke
- *Formate Gleitkommadarstellung*
 - Hinten: Mantisse | s: Vorzeichenbit



- Besonderheiten: Null kann durch Normalisierung nicht 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 ²	\pm	111...1 (max)	Beliebiges Bitmuster $\neq 0$

4.2 Code-Analyse

- Beispiel 1:

```

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:
23 .L3:
24 .word 1085066445
25 .word 1095342490
26 
```

- *s-Register*: Single Register (Precision) (32 Bit Register)
- *d-Register*: Double Register (Precision) (64 Bit Register)
- Speicherung der Konstanten im Dezimalsystem:

$p = 5.4 \rightarrow 1085066445 \rightarrow 0100\ 0000\ 1010\ 1100\ 1100\ 1100\ 1101$ (Umrechnung)

- Neue Befehle und Register:

vldr.32, vadd.f32, s14, s15, d7,...

- Beispiel 2:

```

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:
19 .word -1717986918
20 .word 1075157401
21 .word 858993459
22 .word 1076441907
23 
```

- Speicherung der Konstanten im Dezimalsystem - je 2 32 Bit pro 64 Bit Zahl:

$p = 5.4 \rightarrow -1717986918_1075157401$

$q = 12.6 \rightarrow 858993459_1076441907$

- Neue Befehle und Register:

strd, vldr.64, vadd.f64, d6, d7, ...

- CPU-Informationen unter Linux

- cat /proc/cpuinfo
- features: vfp, neon, vfpv3, vfpv4 für Gleitkommazahlen

```

1 processor : 0
2 model name : ARMv7 Processor rev 4 (v7l)
3 BogoMIPS : 38.40
4 Features : half thumb fastmult vfp edsp neon vfpv3 tls
5           vfpv4 idiva idivt vfpd32 lpaes evtstrm crc32
6 CPU implementer : 0x41
7 CPU architecture: 7
8 CPU variant : 0x0
9 CPU part : 0xd03
10 CPU revision : 4
11 ...
12 ...
13
14 Hardware : BCM2835
15 Revision : a020d3
16 Serial : 0000000068dbc12b

```

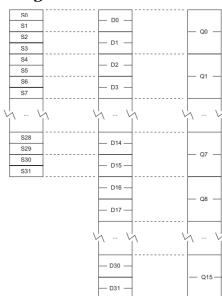
4.3 Studium der Datenblätter/Handbücher

- Suche

- Suche im Internet nach Datenblättern
- Fund: Datenblätter/Handbücher zum Programmiermodell
- Fund: Datenblätter/Handbücher zur Hardware-Architektur

- Handbuch zum Programmiermodell

Registersatz



Befehle

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

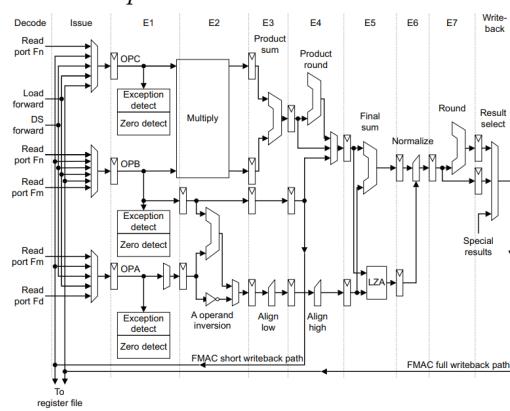
Bemerkungen

- Zwei Möglichkeiten Gleitkommazahlen zu verarbeiten
- *Funktionseinheit NEON*
- *Funktionseinheit VFP*
- Beide Funktionseinheiten verwenden selbe Register

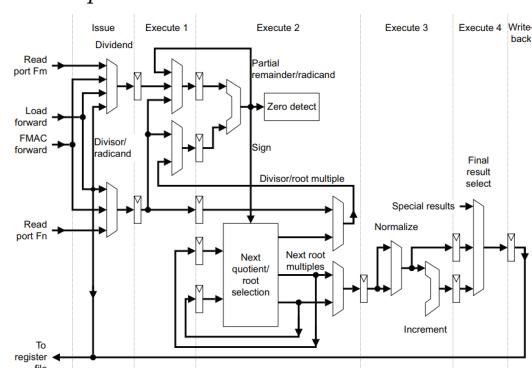
- Mögliche Realisierung einer Gleitkommaeinheit

- Auslagerung in sog. Gleitkomma-Coprozessoren (heutzutage in einem Chip)
- Diese besitzen iofot separate Instruktions-Pipelines
 - Multiplikations und Additions Pipeline (FMAC)
 - Divisions und Quadratwurzel Pipeline (DS)
- Pipelines können auch unabhängig voneinander operieren

FMAC-Pipeline



DS-Pipeline



- Analyse der Assembler-Köpfe

- Wahl welcher Übersetzung (NEON: gcc -mfpu=neon xyz.c)

```

1 .ascii "result_ist.%f_\012\000"
2 .text
3 .align 2
4 .global main
5 .syntax unified
6 .arm
7 .fpu vfp
8 .type main, %function
9
10 .ascii "result_ist.%f_\012\000"
11 .text
12 .align 2
13 .global main
14 .syntax unified
15 .arm
16 .fpu neon
17 .type main, %function

```

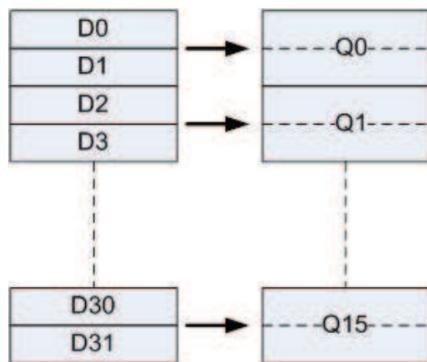
4.4 SIMD

- **Informationen**

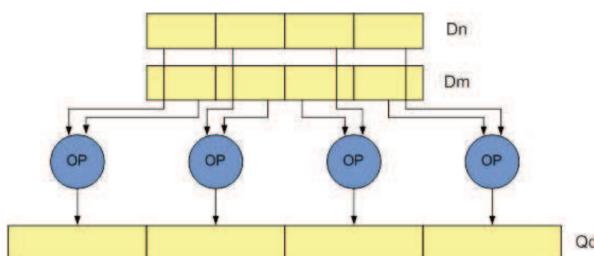
- SIMD: Single Instruction Multiple Data
- Integration dedizierter Funktionseinheiten in Prozessor zur Beschleunigung von Berechnungen
- Für Ganzzahl und Gleitkommazahlen
- Beschleunigt z.B. Signalverarbeitung, Video Encodierung, Bildverarbeitung,..
- Bei ARM: NEON

- **NEON**

- SIMD-Einheit in den ARM Cortex-A Serie Prozessoren
- NEON führt "Packed SIMD" aus
 - Register werden als Vektoren von Elementen desselben Datentyps betrachtet
 - Verschiedene Datentypen sind jedoch möglich
 - gleichzeitige Ausführung der Befehle in sog. "lanes"
- *NEON Register*
 - AArch32 hat 32 x 64 Bit NEON Register (D0 - D31)
 - Diese können auch als 16 x 128 Bit Register gesehen werden (Q0 - Q15)

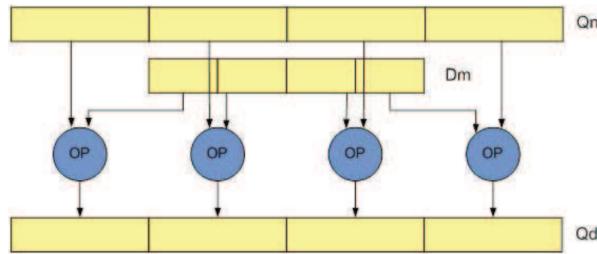


- *NEON Befehle*
 - Format: $V\{<\text{mod}>\}<\text{op}>\{<\text{shape}>\}\{<\text{cond}>\}\{.<\text{dt}>\}\{<\text{dest}>\}$, src1, src2
 - Modifier (mod): z.B. Q: der Befehl nutzt Saturations-Arithmetik
- *Saturations-Arithmetik*
 - Resultate werden auf einen festen Bereich zwischen minimalen und maximalen Wert begrenzt
 - Kleiner als Minimum \rightarrow Minimum / Größer als Maximum \rightarrow Maximum
- *Beispiel I*



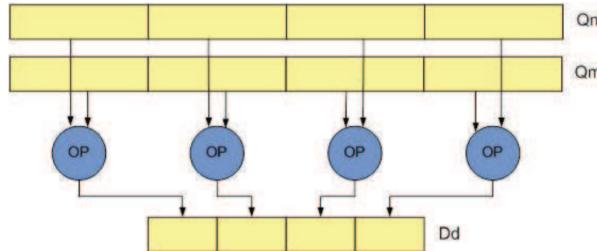
Beide Ausgangsregister 64 Bit, Zielregister 128 Bit (z.B. Multiplikation)

- Beispiel II



Ein Ausgangsregister 128 Bit, das Andere 64 Bit, Zielregister 128 Bit

- Beispiel III



Beide Ausgangsregister, Zielregister 64 Bit (z.B. Multiplikation)

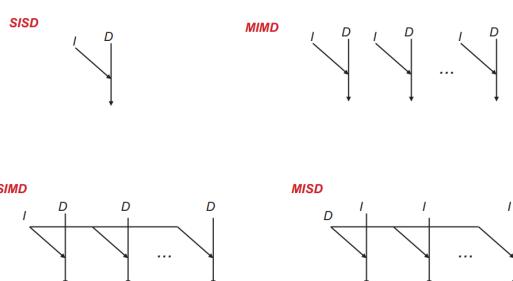
- Beispielcode

- VMULL.S16 Q2, D8, D9
- NEON-Programmierung

- NEON optimized libraries
- Vectorizing compilers (gcc)
- NEON intrinsics
- NEON assembly

- Klassifikation von Flynn

- Aufteilung in sog. *Instruction Streams* und *Data Streams*
- Grobe Klassifikation - Aber stark etablierte Begriffe
- Verschiedene Aspekte werden nicht oder nur groß erfasst (Pipelining, Wortbreite, Speicher,...)
- *Instruction Stream*
 - SI - Single Instruction
 - MI - Multiple Instruction
- *Data Stream*
 - SD - Single Data
 - MD - Multiple Data
- *Kombinationen*
 - SISD - Von-Neumann-Rechner
 - SIMD - Feldrechner, Vektorrechner
 - MIMD - Multiprozessorsysteme
 - MISD - Keine Anwendung
- *Veranschaulichung*



5 Speicher

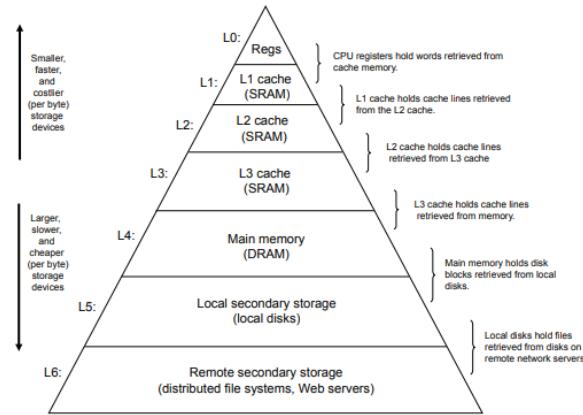
5.1 Speicher

- Bisheriges Modell

- Drei Phasen der Befehlsausführung
- Prozessor mit Registern
- Speicher (var1: .word 5)
- Hier besteht der Speicher nur aus einem linearen Feld aus Bytes
- Die CPU kann auf jede Speicherzelle in konstanter Zeit zugreifen
⇒ Einfaches Modell, welches die Realität nicht abbildet

- Speicherhierarchie

- Lesen und Schreiben in einen Speicher dauert gewisse Zeit
⇒ Relativ lang im Vergleich zur Taktfrequenz
- Von-Neumann: Daten, Befehle etc müssen alle nacheinander aus dem Speicher geholt werden
- Harvard: Obiges kann auch parallel ausgeführt werden
- Einführung von Pipelinestufen möglich
- *Speicherhierarchie-Pyramide:*



- Eigenschaften von Speichern

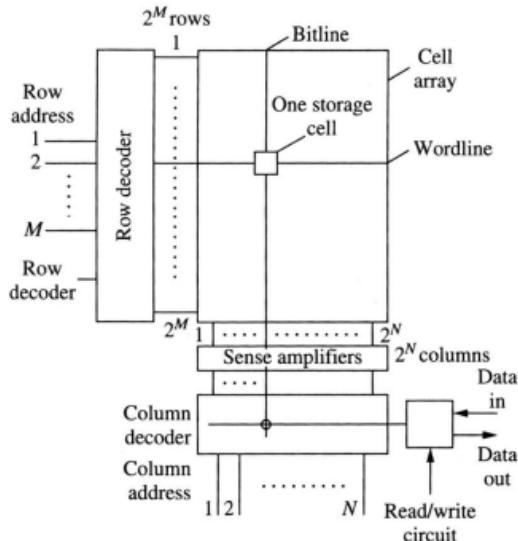
- *Kosten und Zugriffszeit*
 - *Kosten:* Messung in Dollar/Bit oder Dollar/MByte
 - *Zugriffszeit:* durchschn. Zeit um Wort aus Speicher zu lesen
 - *Zykluszeit:* minimale Zeit zwischen zwei Speicherzugriffen
 - Das Busprotokoll ist hierfür auch von Relevanz
 - *Bandbreite:* Maximale Datenmenge, die pro Sekunde übertragen werden kann (Byte/sec)
- *Geschwindigkeit und Kapazität*
- *Zugriffsverfahren*
 - Zwei verschiedene Zugriffsverfahren
 - *Random Access Zugriff*
 - Register
 - Cache-Speicher
 - Hauptspeicher
 - *Serieller Zugriff*
 - Festplatte
 - Optische Daten
 - Magnetband

- Änderbarkeit

- *Read-Only*: Nur Lesen möglich, kein Überschreiben (ROM)
- *Read-Write*: Inhalt kann während des Betriebs geändert werden (RAM)
- *Read-Mostly*: (PROM-Halbleiterspeicher, z.B. für BIOS)
- Permanenz
- *Flüchtige Speicher*: Inhalt geht bei Stromausfall verloren (Register, Hauptspeicher)
 - dynamische Speicher: Periodische Erneuerung der Spannung gegen Informationsverlust
 - statischer Speicher: Kein Refreshing wie beim dynamischen notwendig
- *Nicht flüchtige Speicher*: Inhalt bleibt erhalten (ROM, PROM, Festplatte)

- Speichertechnologien - Random-Access-Memory

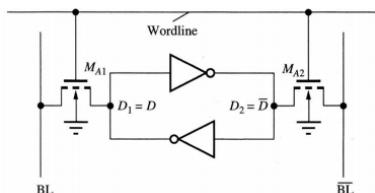
- *Statisches RAM* (SRAM)
- *Dynamisches RAM* (DRAM)
- SRAM schneller und deutlich teurer als DRAM
- SRAM \Rightarrow Cache / DRAM \Rightarrow Hauptspeicher
- Verteilung meist: 12MB SRAM / 8GB+ DRAM
- *Organisation des Speichers*



- Cellarray mit Rows und Columns
- Wordline: 32 Bit Wort z.B.
- Storage Cell: speichert 1 Bit (0/1)
- Auswahl über Zeilenadresse + Bitline
- Amplifier: Verstärkung geringer Spannungen
- Ausgabe: Diskrete 0 oder 1

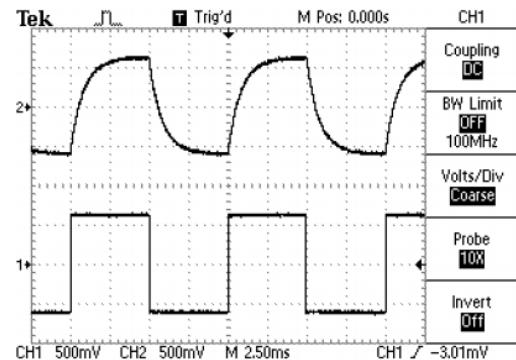
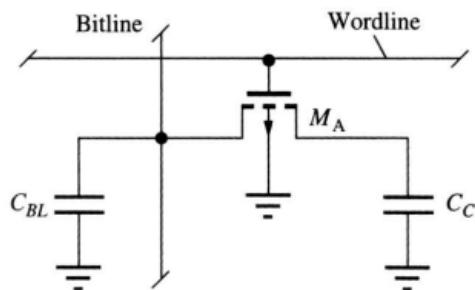
- Statisches RAM

- Speichert Informationen solange, wie Spannungsversorgung angeschaltet ist
- Bistabile Schaltung (zwei stabile Zustände) (wird nicht so in der Klausur abgefragt)
- 6T Zelle wegen 6 Transistoren



- Dynamisches RAM

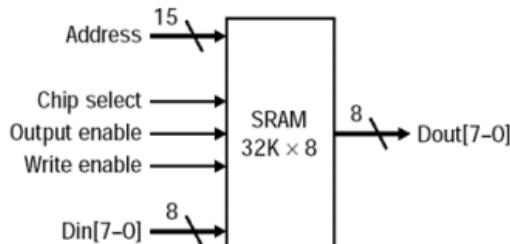
- Speichert Informationen in einem Kondensator
- Bezeichnung als 1T-Zelle
- Refresh-Zyklus notwendig, da Kondensator sich entleert



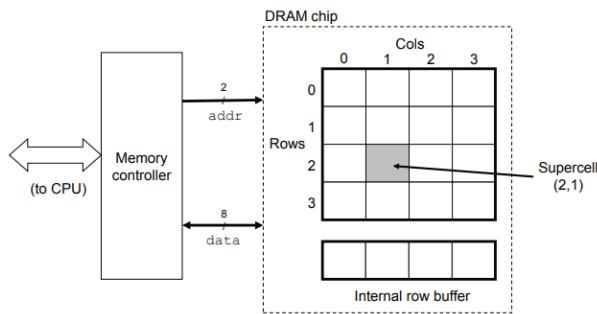
5.2 Speicherorganisation

- Bezeichnungen

- Eigenschaften eines RAM-Chips werden durch zwei Größen gegeben
 - Anzahl adressierbarer Plätze
 - Breite in Bit jedes adressierbaren Platzes
- Bsp: 256K x 1 SRAM
 - 256K Einträge mit 1 Bit Breite
 - $256K = 2^{18} \rightarrow 18$ Adresseingänge + 1 Bit Dateneingang/ausgang
- Bsp: 32K x 8 SRAM
 - 32K Einträge mit 8 Bit Breite
 - $32K = 2^{15} \rightarrow 15$ Adresseingänge + 8 Bit Dateneingang/ausgang



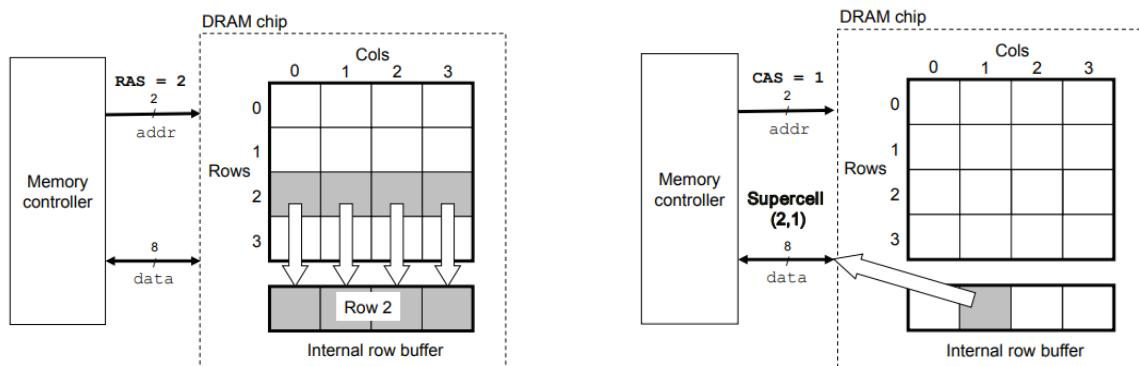
- Abstrakte Darstellung eines 128 Bit 16x8 DRAM Chips



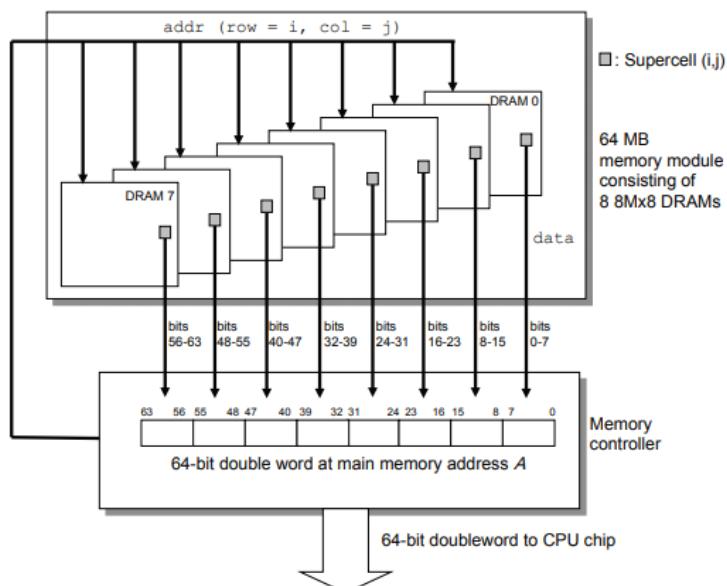
16 Felder, 8 Bit Dateneingang/ausgang

Auswahl der Reihe - **RAS (Row Adress Strobe)**

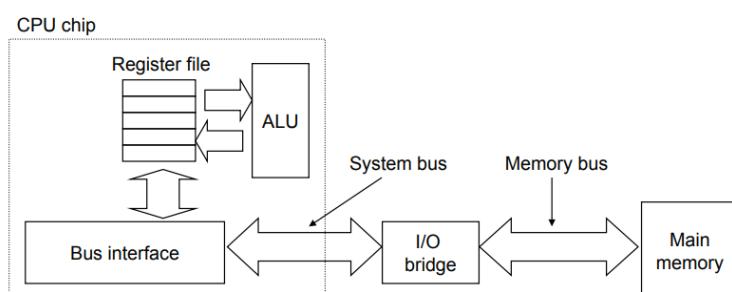
Auswahl der Spalte - **CAS (Column Adress Strobe)**



- Verschaltung mehrerer Speichermodule



- Verbindung von CPU und Speicher

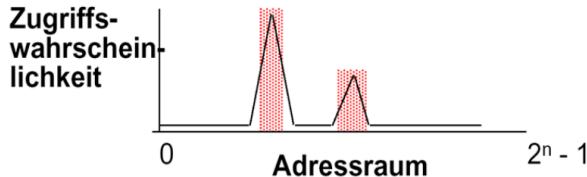


5.3 Lokalität

- **Informationen**

- Lokalität ist wichtig für eine funktionierende Speicherhierarchie
- *Lokalitätsprinzip:*

Bei Programmausführung wird nur auf einen geringen Teil des Adressraumes zugegriffen



- **Formen der Lokalität**

- *zeitliche (temporale) Lokalität*
 - Nach Zugriff auf Daten wird mit großer Wahrscheinlichkeit bald wieder darauf zugegriffen
 - Bsp.: Schleifen
- *räumliche Lokalität*
 - Sukzessiver Zugriff auf Daten, die in der Nähe liegen
 - Bsp.: sequentielle Instruktionsfolgen (ohne Sprünge)
 - Bsp.: Reihungen, Matrizen

- **Vorteile**

- Gut geschriebene Programme haben eine gute Lokalität
- Hat hohe Auswirkung auf die Performanz (bessere Lokalität → schneller)
- Lokalität der Daten / Lokalität der Befehle

- **Beispiel 1 - Summe eines Vektors**

```
int sumvec (int v[N])
{
    ...
    int i, sum = 0;

    for (i = 0; i < N; i++)
        sum += v[i];
    return sum;
    ...
}
```

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

- Gute räumliche Lokalität
- Schlechte zeitliche Lokalität (1x Zugriff)
- (i allerdings gute zeitliche Lokalität)

- **Beispiel 2 - Summe eines Arrays**

```
int sumvec (int v[N])
{
    ...
    int sumarrayrows(int a[M][N])
    {
        int i,j,sum = 0;
        for (i = 0; i < M; i++)
            for (j = 0; j < N; j++)
                sum += a[i][j];
        return sum;
    }
    ...
}
```

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

- Gute räumliche Lokalität
- Schlechte zeitliche Lokalität (1x Zugriff)
- (i allerdings gute zeitliche Lokalität)

- Beispiel 3 - Summe eines Arrays

```

int sumvec (int v[N])
{
    ...
int sumarrayrows(int a[M][N])
{
    int i,j,sum = 0;
    for (i = 0; i < N; i++)
        for (j = 0; j < M; j++)
            sum += a[i][j];
    return sum;
}
...
}

```

	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

- Schlechte räumliche Lokalität (Springen im Speicher)
- Schlechte zeitliche Lokalität (1x Zugriff)
- (i allerdings gute zeitliche Lokalität)

- Lokalität der Befehle

- Befehle sind auch im Speicher abgelegt, hier auch gute Lokalität möglich
- Für die for Schleifen der Beispiele gilt gute räumliche Lokalität
- Schleifenkörper wiederholt ausgeführt → gute zeitliche Lokalität

- Beispiel 4/5 - Zeilenweises/Spaltenweises Schreiben in ein Array

Zeilenweises Schreiben eines Arrays

```

int main()
{
    register long i,j;
    static long matrix[DIM][DIM];

    for (i = 0; i < DIM; i++)
        for (j = 0; j < DIM; j++)
            matrix[i][j] = 0;

    return 0;
}

```

Spaltenweises Schreiben eines Arrays

```

int main()
{
    register long i,j;
    static long matrix[DIM][DIM];

    for (j = 0; j < DIM; j++)
        for (i = 0; i < DIM; i++)
            matrix[i][j] = 0;

    return 0;
}

```

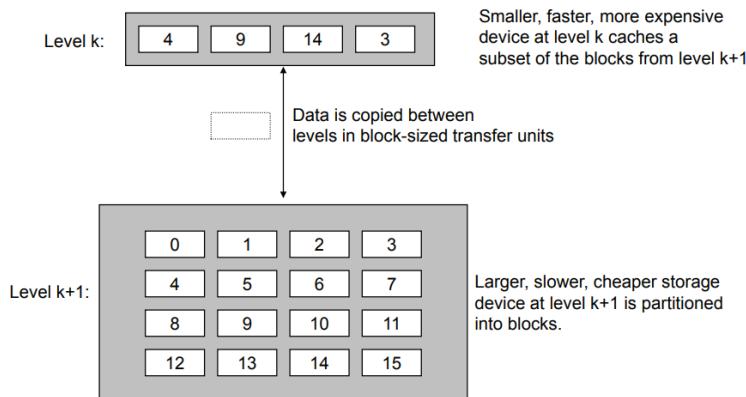
⇒ Zeilenweises Schreiben um Einiges schneller als spaltenweises

5.4 Prinzip des Caches

- **Informationen**

- Kleiner, schneller Speicher (SRAM)
- Prozess einen Cache zu benutzen: *caching*
- Schneller, kleinerer Speicher auf Level k fungiert als Cache für langsameren, schnelleren Speicher auf Level $k + 1$
- Wird mehrfach angewendet (siehe Speicherhierarchie)

- **Prinzip des Cachings**



- Laden ganzer Wordlines → Lokalität sehr wichtig

- **Cache Hits**

- Programm benötigt Daten vom Objekt d
⇒ Überprüfen, ob d in einem der Blöcke auf Level k gespeichert ist
- Wenn d gefunden wird ⇒ **Cache Hit**
- k schneller als $k + 1$ ⇒ Geschwindigkeitsvorteil
- (Auch Geschwindigkeitsunterschiede zwischen L1, L2, L3 (Speicherpyramide))

- **Cache Misses**

- Programm benötigt Daten vom Objekt d
⇒ Überprüfen, ob d in einem der Blöcke auf Level k gespeichert ist
- Wenn d nicht gefunden wird ⇒ **Cache Miss**
- Cache auf Level k holt dann Block mit d von Cache auf Level $k + 1$
- Falls Cache auf Level k bereits voll ⇒ Überschreiben (Ersetzung)
- Unterschiedliche Strategien für Blockersetzung (Zufall, Least-Recently-Used (LRU))

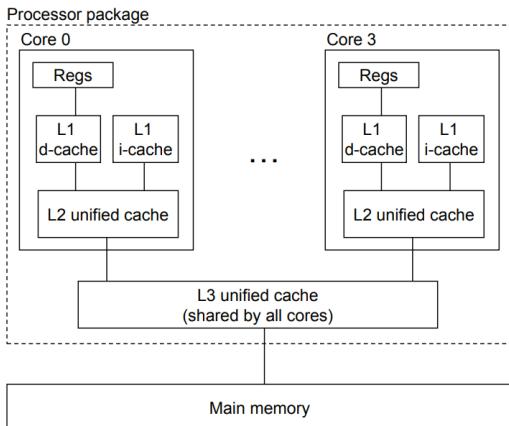
- **Blick auf Speicherhierarchie**

Blöcke und Latenz werden nach unten hin 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

5.5 Cachehierarchie ARM / Intel Core i7

Speicheraufbau



- Jeder Kern hat eigenen Registersatz
- d-Cache: Datencache
- i-Cache: Instruktionscache