

# Rechnerorganisation

Jonas Milkovits

Last Edited: 29. Juli 2020

## Inhaltsverzeichnis

<b>1</b>	<b>Einführung</b>	<b>1</b>
1.1	Begrifflichkeiten und Grundlagen . . . . .	1
1.2	Streifzug durch die Geschichte . . . . .	2
1.3	Ethik in der Informatik . . . . .	3
<b>2</b>	<b>Einführung in die maschinennahe Programmierung</b>	<b>4</b>
2.1	Begrifflichkeiten und Grundlagen . . . . .	4
2.2	Phasen der Übersetzung . . . . .	6
2.3	Ausführung eines Programms im Rechnersystem . . . . .	7
2.4	Befehle eines Rechnersystems . . . . .	8
2.5	Registersatz . . . . .	9
2.6	Adressierung des Speichers, Lesen und Schreiben auf Speicher . . . . .	9
2.7	Kontrollstrukturen in Assembler . . . . .	11
2.8	Nutzung des Hauptspeichers . . . . .	12
2.9	Datenfelder (Arrays) . . . . .	14
2.10	Unterprogramme . . . . .	14
2.11	Stack . . . . .	16
2.12	Rekursion . . . . .	17
2.13	Compilieren, Assemblieren und Linken . . . . .	19
<b>3</b>	<b>Mikroarchitekturen von Rechnersystemen</b>	<b>22</b>
3.1	Begrifflichkeiten und Grundlagen . . . . .	22
3.2	Analyse der Rechenleistung . . . . .	22
3.3	Eintakt-Prozessor . . . . .	23
3.4	Mehrtakt-Prozessor . . . . .	26
3.5	Pipeline-Prozessor . . . . .	29
3.6	Ausnahmebehandlung . . . . .	29

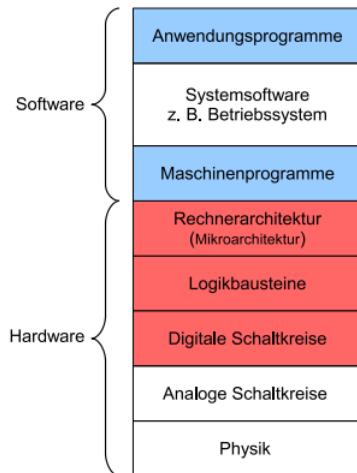
# 1 Einführung

## 1.1 Begrifflichkeiten und Grundlagen

- **Abstraktion**

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

- **Schichtenmodell**



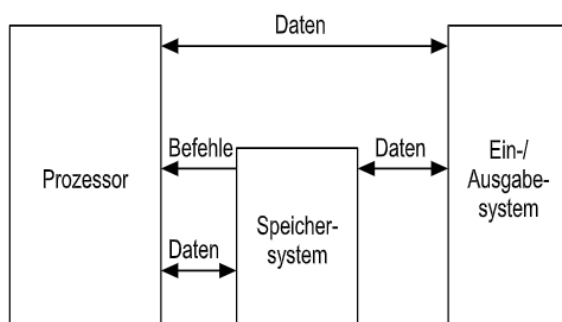
- Untere Schicht erbringt Dienstleistungen für höhere Schicht
- Obere Schicht nutzt Dienste der niedrigeren Schicht
- Eindeutige Schnittstellen zwischen den Schichten
- Vorteile:
  - Austauschbarkeit einzelner Schichten
  - Nur Kenntnis der bearbeitenden Schicht notwendig
- Nachteile:
  - ggf. geringere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

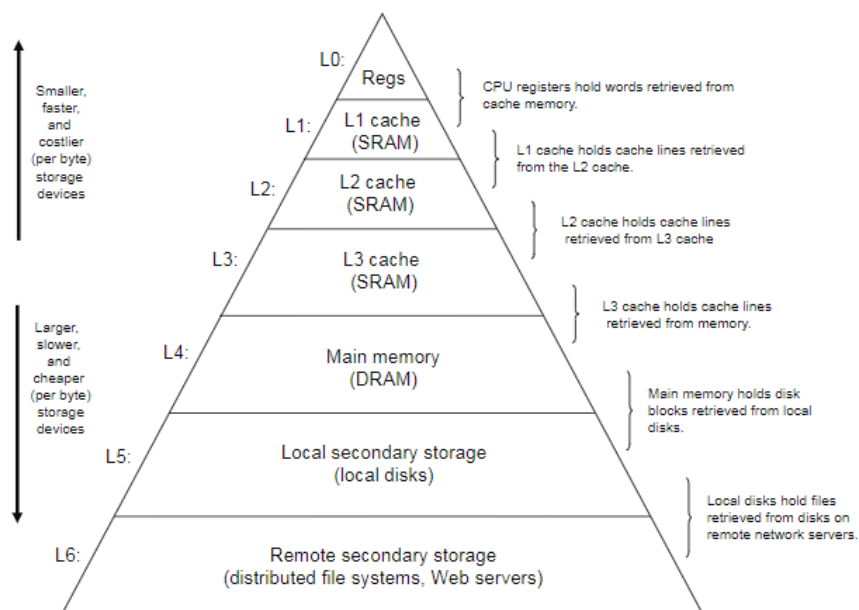
- **Komponenten eines Rechnersystems**

- 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)

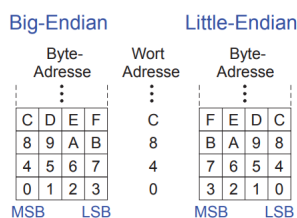


## • Nähere Informationen zum Speicher

- Explizite Nutzung des Speichersystem
  - Internet 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



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

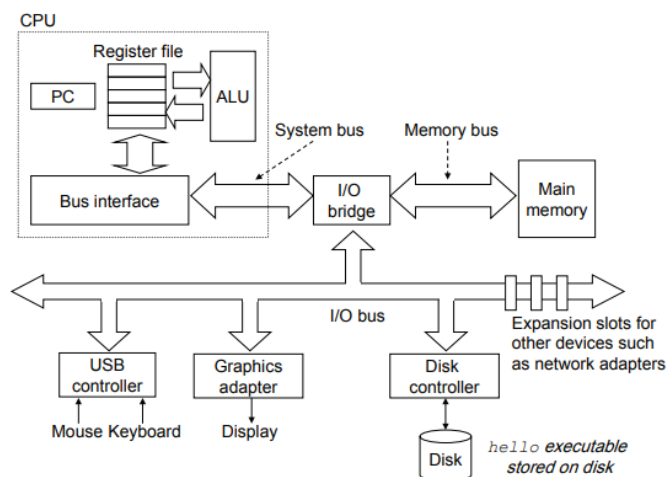
### 1.3 Ethik in der Informatik

- Ethik in der Informatik
  - Ethik: Bewertung menschlichen Handelns
  - Verbindung zur Informatik: Anwendung von Rechnern für kriegsches 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 Rechnersystems



- |                                |   |
|--------------------------------|---|
| • 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                   |

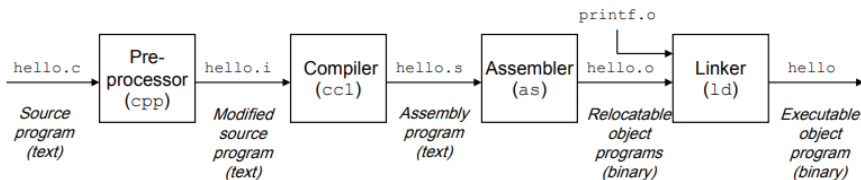
- **Assembler**
  - Programmieren in der Sprache des Computers
    - Maschinenbefehle: Einzelnes Wort
    - Befehlssatz:           Gesamtes Vokabular
  - Befehle geben Art der Operation und ihre Operanden an
  - Zwei Darstellungen:
    - Assemblersprache: Für Menschen lesbare Schreibweise für Instruktionen
    - 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;
}
```

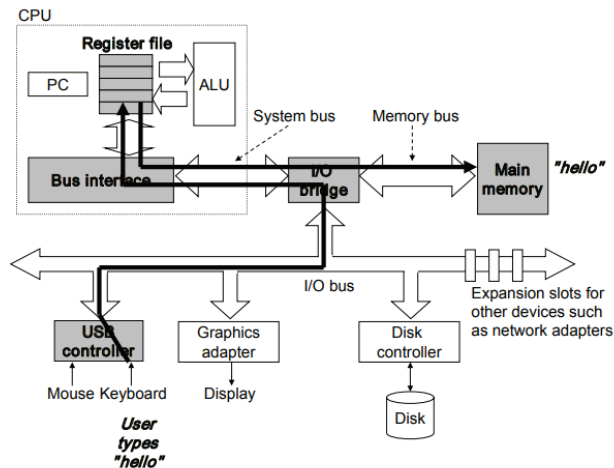
- C-Programm an sich für den Menschen verständlich
- Übersetzung in Maschinenbefehle für Ausführung auf dem Rechner:



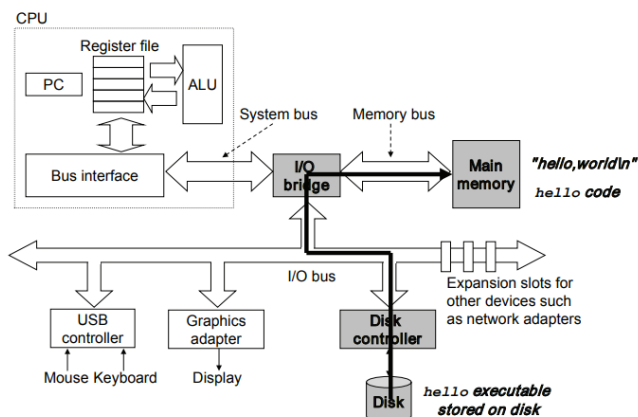
- 1. Phase (**Preprocessor**)
  - Aufbereitung durch Ausführung von Direktiven (Code mit #)
  - z.B.: Bearbeiten von `#include <stdio.h>`
    - Lesen des Inhalts der Datei `stdio.h`
    - Kopieren des Inhalts in die Programmdatei
  - Ausgabe: C-Programm mit der Endung `.i`
- 2. Phase (**Compiler**)
  - Übersetzt C-Programm `hello.i` in Assemblerprogramm `hello.s`
- 3. Phase (**Assembler**)
  - Übersetzt `hello.s` in Maschinsprache
  - Ergebnis ist das Objekt-Programm `hello.o`
- 4. Phase (**Linker**)
  - Zusammenfügen verschiedener Module
    - Code von `printf` existiert bereits als `printf.o`-Datei
  - Linker kombiniert `hello.o` und `printf.o` zu ausführbarem Programm
  - Ausgabe des Bindevorgangs: ausführbare `hello`-Objektdatei

## 2.3 Ausführung eines Programms im Rechnersystem

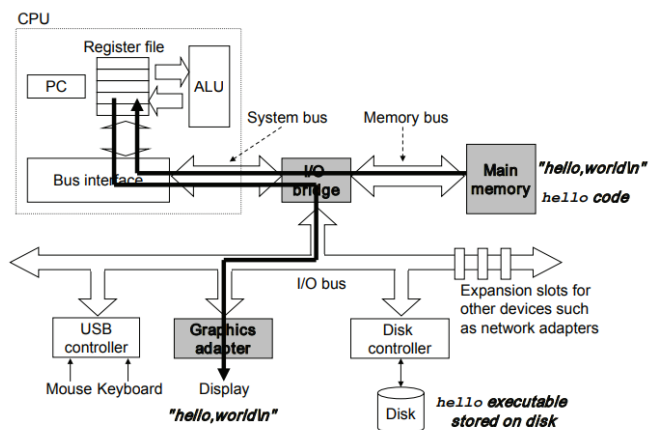
- Ausgangspunkt
  - Ausführbares Objektprogramm `hello` auf der Festplatte
  - Starten der Ausführung des Programms unter Nutzung der Shell
- Ablauf:
  - Shell legt Zeichen des Kommandos ins Register
  - Speichert den Inhalt dann im Hauptspeicher aber



- 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 Pipelining
    - 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 (intermediates)**
  - 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)

(A/C)PSR

- 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)

### • Current Processor Status Register

- Enthält unter anderem die **Statusflags**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0						
N		Z		C		V		Q		IT		J				IL		GE		IT [7:2]				E		A		I		F		T		M		M [3:0]	

- 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

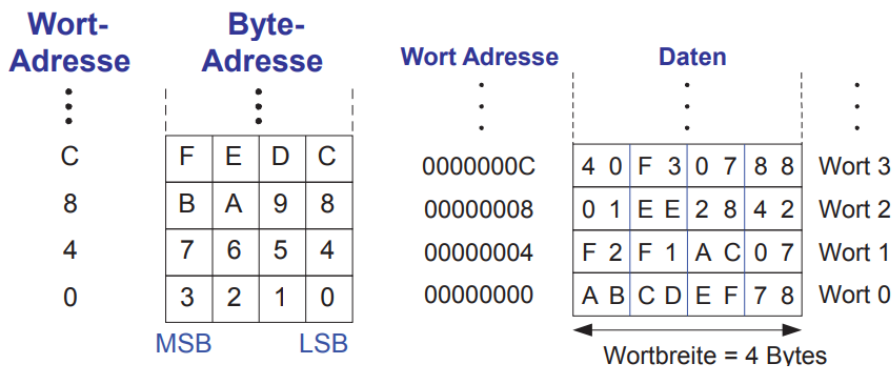
## 2.6 Adressierung des Speichers, Lesen und Schreiben auf Speicher

### • 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 9) 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 */

```

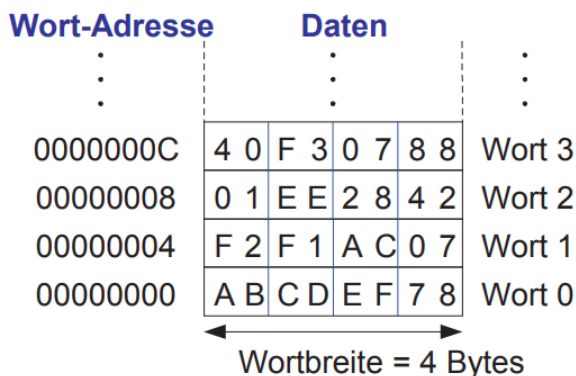
  - r7 enthält das Datenwort der Speicheradresse r5+8
- Beispiel 2:
  - Lesen Datenwort 3 (Speicheradresse 0xC (12er Offset)) nach r7
  - (Einschub: 0x sagt dem Compiler, dass das Folgende eine Hexzahl ist)

```

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:  $14_{16} = 0001\ 0100_2 = 20_{10}$  (5.tes 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 */

```

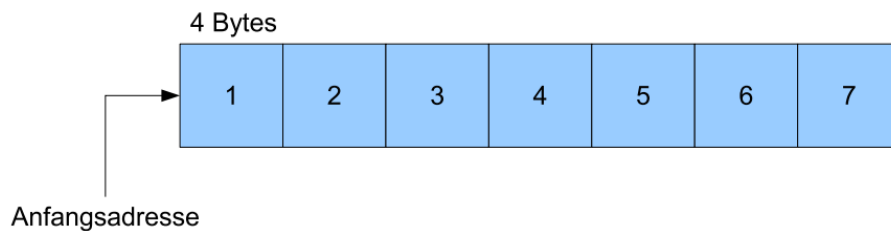
- `.data` (Zeile 4):
  - Variablen, die im Speicher (nicht im Register) abgelegt werden
  - `.word`: Festlegung des Typs (hier 32 Bit)
  - Name `var1`: An sicht frei wählbar
- `.global main` (Zeile 8):
  - Definiert das Label, das als Einsprungpunkt gilt (hier `main`)
- `adr_var1: .word var1` (Zeile 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

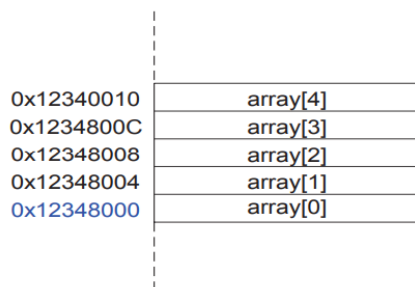
## 2.9 Datenfelder (Arrays)

- **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;

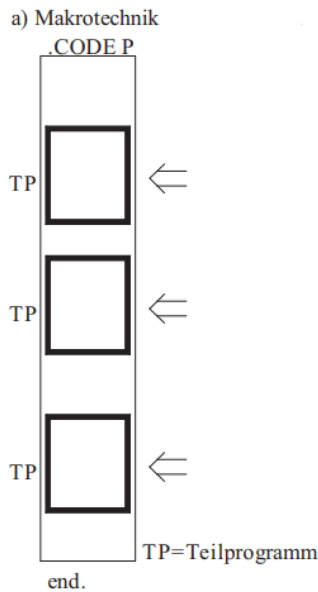
mov r0, #0x14000000 /* Speichern der Basisadresse des Arrays in r0 */
mov r1, #0          /* Verwendung als Zählervariable i */
LOOP:
cmp r1, #200        /* i < 200 */
bge L3              /* Falls i > 200, Verlassen des Loops */
lsl r2, r1, #2       /* r2 = i * 4 -> Aufgrund des Offsets des Arrays von 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**

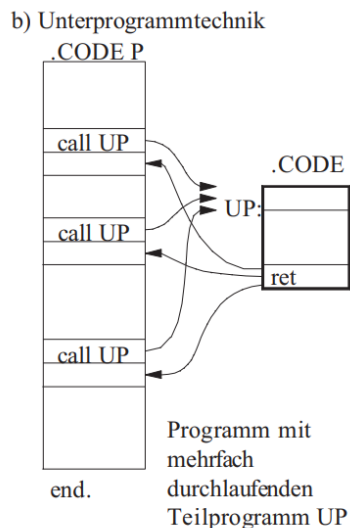
- Unterprogramme helfen bei der strukturierten Programmierung
- Betrachtung Hauptprogramm, in dem ein Teilprogramm an versch. Stellen ausgeführt werden soll
- Zwei Konzepte: Makrotechnik und Unterprogrammtechnik

- **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

```

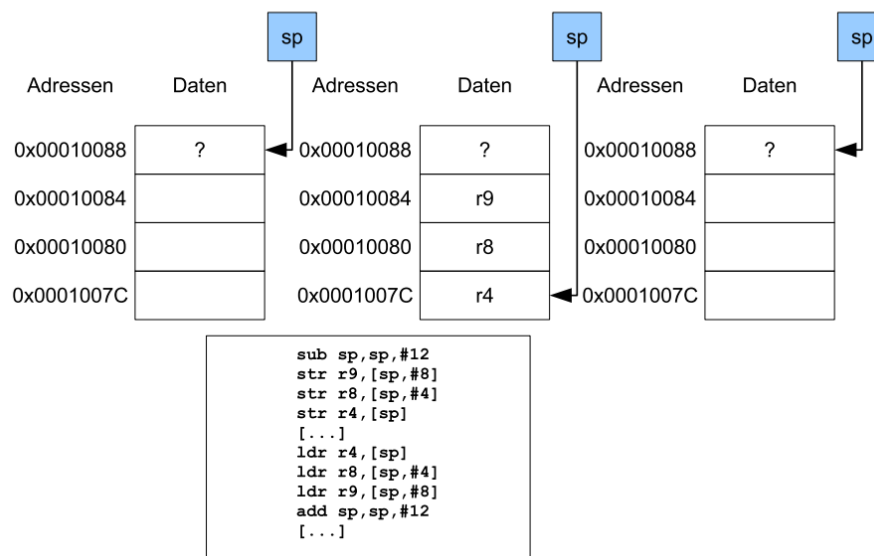
diffofsums:
sub sp, sp, #12      /* Speicher auf Stack reservieren (3 Adressen "abziehen") */
str r9, [sp, #8]     /* Speichern an oberster freier Stelle im Stack */
str r8, [sp, #4]
str r4, [sp]         /* Speichern an unterster freier Stelle im Stack */
/* Abspeichern der Werte in Benutzungsreihenfolge 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

```

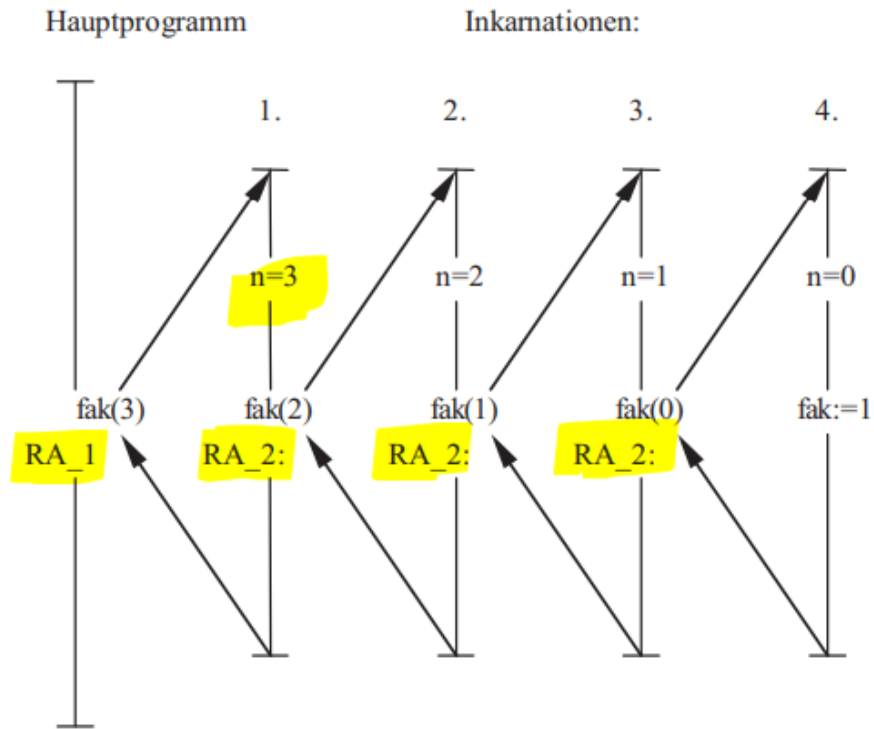
main:
...
push {lr}          /* push ist hier nur eine Pseudoinstruktion für str */
bl diffofsums      /* Das LinkRegister wird beim Programmaufruf verändert */
...
pop {lr}           /* pop ist hier die Pseudoinstruktion für ldr */
bx lr

```

- 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 diesselbe 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]                     /* Sicherung von r0 */
    str lr, [sp]                          /* Sicherung 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 Intermediate 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 Plattform Y
  - Verwendung im Bereich der Embedded Systems
- Disassembler
  - Übersetzung von Maschinencode in Assemblersprache
  - Verlust von Kommentaren und symbolischen Namen

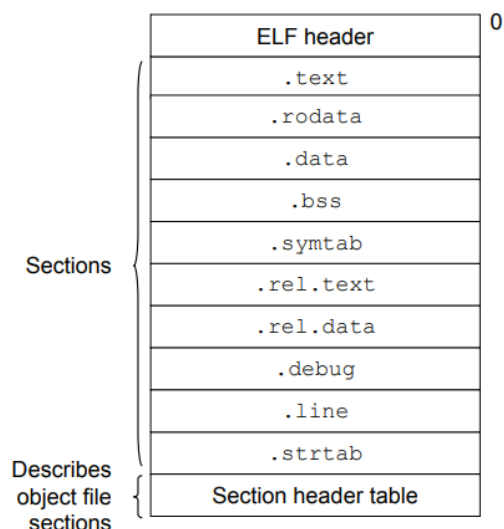
### • Schrittweiser Assembliervorgang

- 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:  $\geq 1$  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 zusammengefügt 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



- ELF: Ein Format dieser Files
- Beginnt mit 16-Byte Sequenz
- Informationen über Wortgröße, Byte-Ordering,...
- **.text**: Maschinencode des compilierten Programms
- **.data**: Initialisierte globale Variablen
- etc.

- 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: eaffffff  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

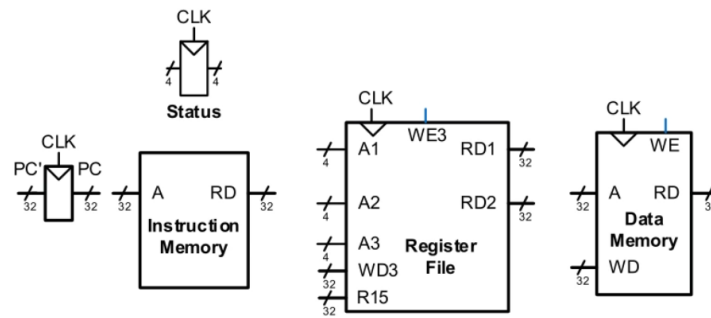
- **Drei Phasen der Befehlsausführung**
  - Befehle, als auch Daten stehen im Speicher
  - *Befehlsholphase*: Prozessor liest die Befehle aus dem Speicher
  - *Befehlsdekodierung*: Dekodierung des Befehls, nachdem dieser in ein Register geholt wurde
  - *Befehlsausführung*: Ausführung des Befehls, danach 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*: verly 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  
 $\Rightarrow \text{Ausführungszeit} = (\# \text{Instruktionen}) \cdot \left(\frac{\text{Takte}}{\text{Instruktion}}\right) \cdot \left(\frac{\text{Sekunden}}{\text{Takt}}\right)$
  - CPI: Takte/Instruktion
  - Taktperiode: Sekunden/Takt
  - IPC:  $1/\text{CPI} = \text{Instruktionen/Takt}$
- **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**



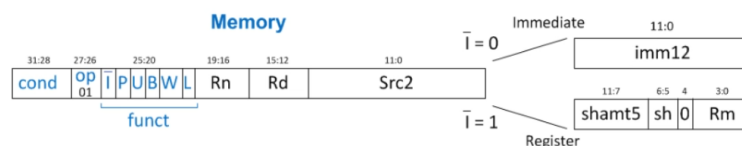
- *Register File*
  - A1,A2,... - *Registeradresse* (4 Bit → 16 Möglichkeiten)
  - RD1,RD2,... - *Register Data* (Ausgabedaten)
  - WD3 - *Write Data 3* (Eingabedaten)
- *Status*
  - Gibt Flags an mit 4 Bits (Sign, Zero, Overflow, Carry)
- *Program Counter (PC)*
- *Instruction Memory*
  - A - *Adressen*
  - 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: intermediate 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

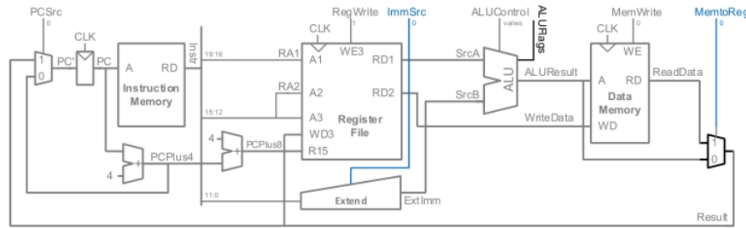
- **Phasen der Befehlsausführung**





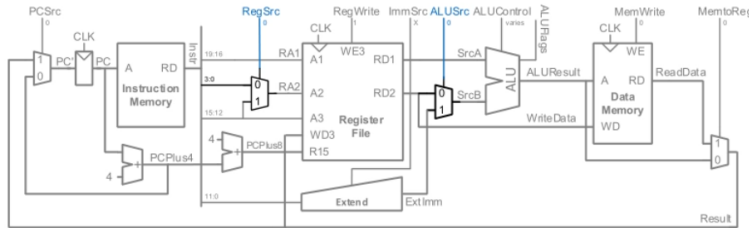
⇒ Erweiterung um Multiplexer

⇒ Schreibe ALUResult in Registerfeld statt Speicher (je nach Multiplexer)

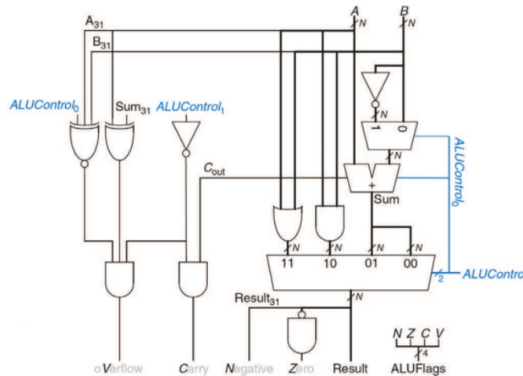


- **Register Src2 (Src2 hier als Register)**

⇒ Multiplexer vor Register File und nach Extend



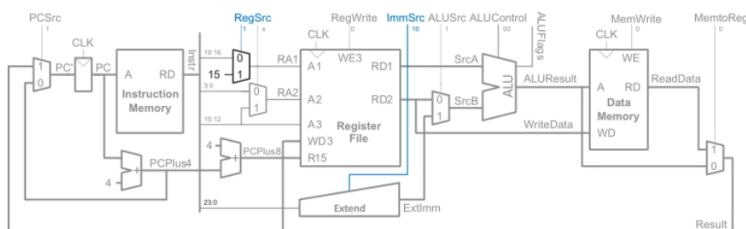
- **Arithmetisch Logische Einheit**



ALUControl <sub>1:0</sub>	Function
00	Add
01	Subtract
10	AND
11	OR

- **Sprungbefehl b**

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



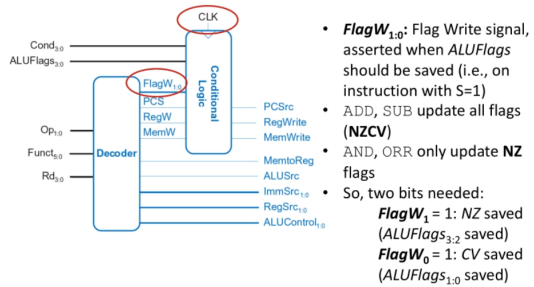
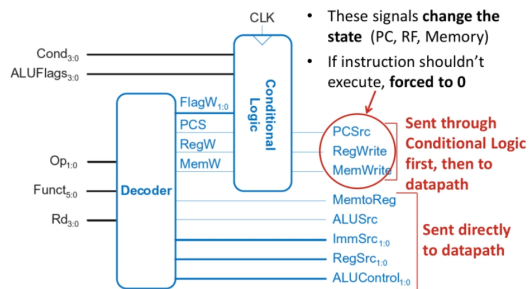
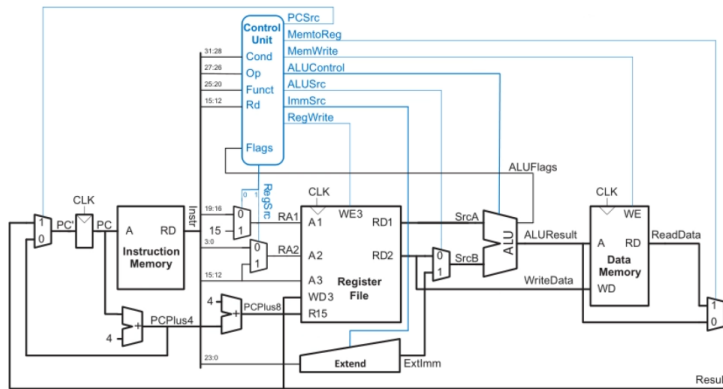
- **ExtImm**

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

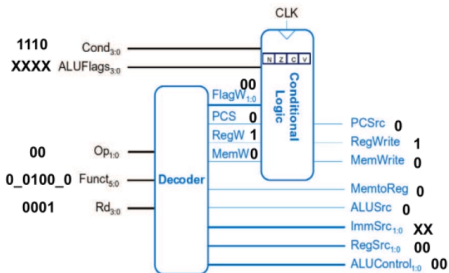
ImmSrc <sub>1:0</sub>	ExtImm	Description
00	{24'b0, Instr <sub>7:0</sub> }	Zero-extended imm8
01	{20'b0, Instr <sub>11:0</sub> }	Zero-extended imm12
10	{6{Instr <sub>23</sub> }, Instr <sub>23:0</sub> }	Sign-extended imm24

6{Instr<sub>23</sub>} → Erstes Bit 6x (Sprungbefehl)

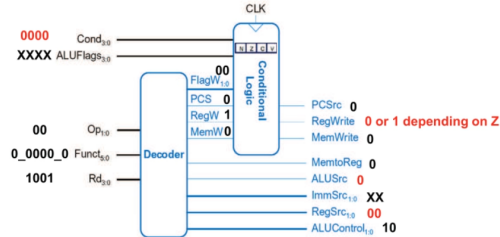
## • Kontrolleinheit



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



Beispiel: *andeq r5,r6,47*



## • Condition Field

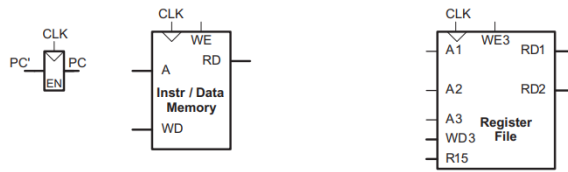
- Verwendung dieser Codes für jeden Befehl als Condition möglich

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)



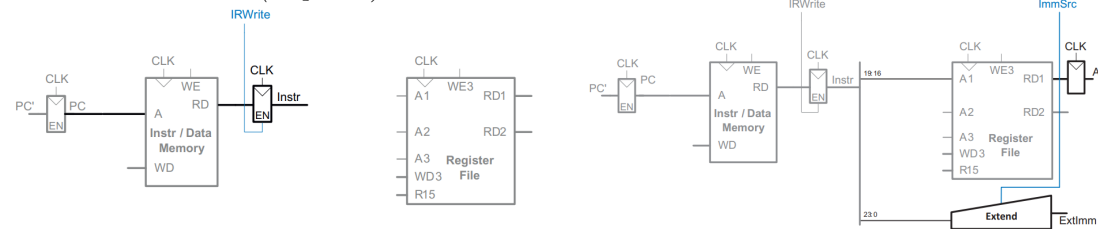
## • Ablauf der Befehlsausführung anhand eines Mehrtaktprozessors

- Register werden hier zum Zwischenspeichern von Werten genutzt

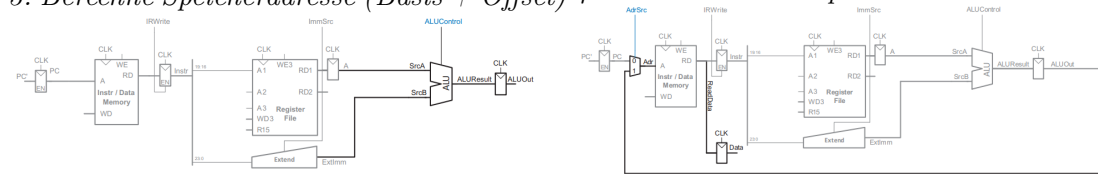
⇒ Damit Wert aufgrund von Takt nicht verloren geht

1. Hole Instruktion (Bsp. ldr)

2. Lese Quelle aus Register/Auswertung Direktwert

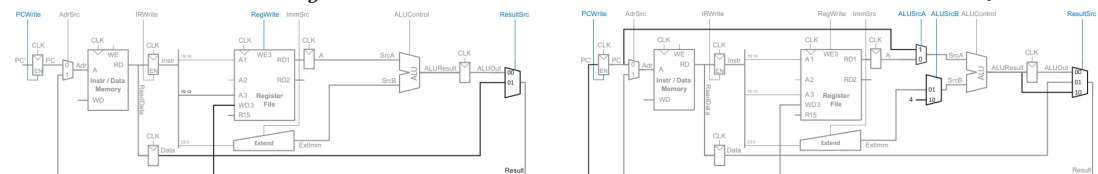


3. Berechne Speicheradresse (Basis + Offset) 4. Lese Daten aus Speicher



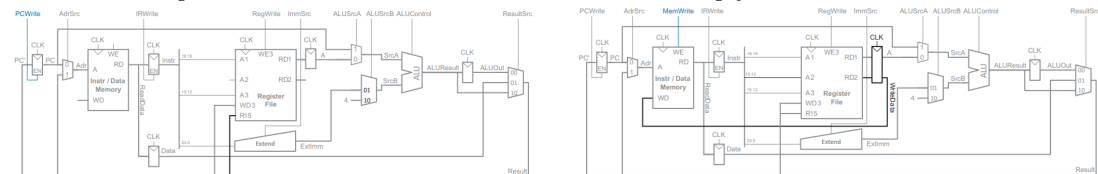
5. Schreibe Daten in Register

6. Berechne Adresse nächster Befehl



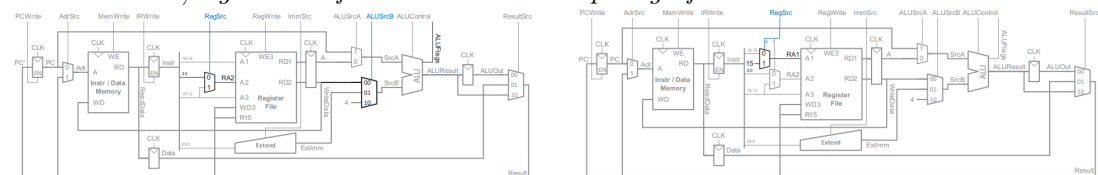
7. Behandlung von r15

Erweiterung für str

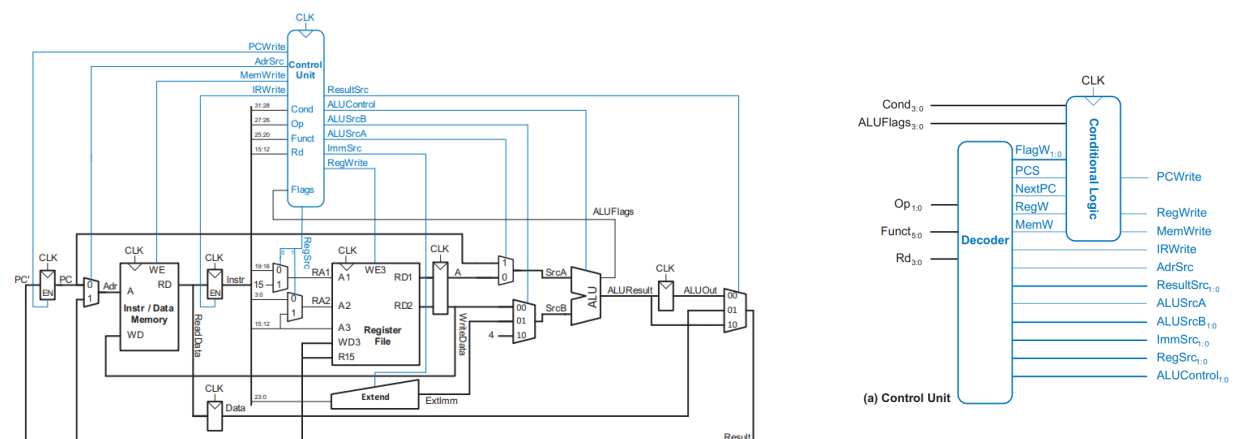


Arithmetische/logische Befehle

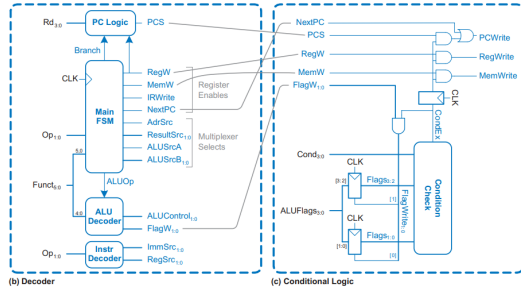
Sprungbefehl b



## • Datenpfad und Kontrollleinheit



## Kontrollinheit - 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

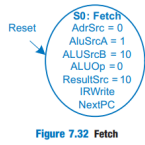
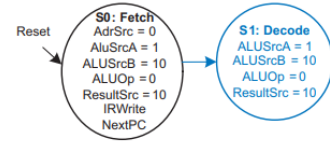
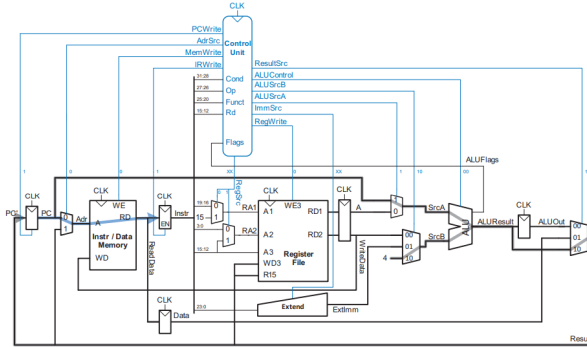


Table 7.6 Instr Decoder logic for RegSrc and ImmSrc

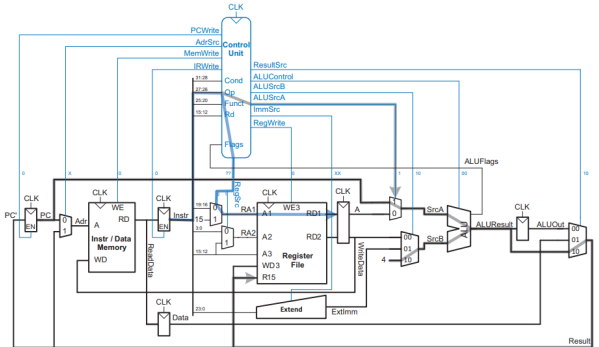
Instruction	Op	Func <sub>5</sub>	Func <sub>0</sub>	RegSrc <sub>5</sub>	RegSrc <sub>0</sub>	ImmSrc <sub>5:0</sub>
LDR	01	X	1	X	0	01
STR	01	X	0	1	0	01
DP immediate	00	1	X	X	0	00
DP register	00	0	X	0	0	00
B	10	X	X	X	1	10



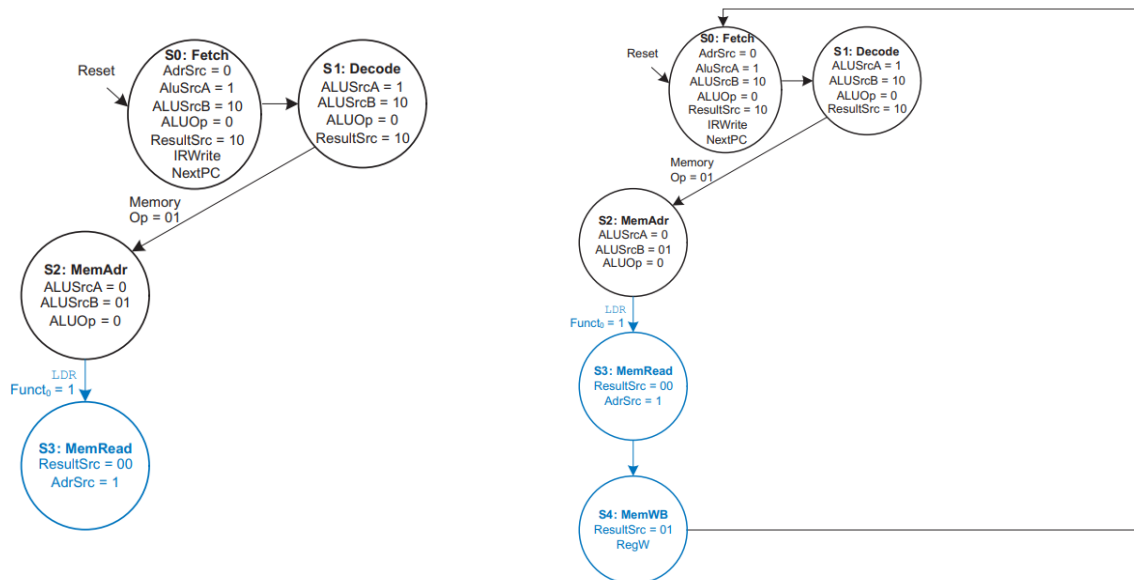
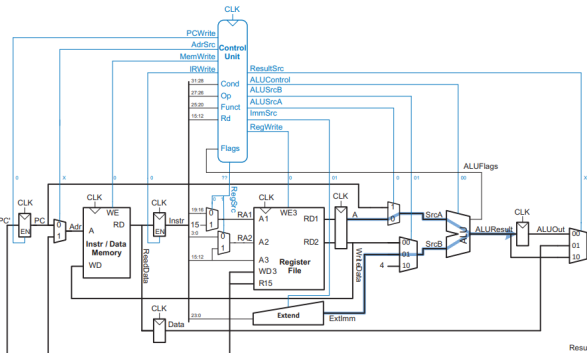
#### 1. Fetch



#### 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 (könnten 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. Takten
  - aufwendigere Ablaufsteuerung

### 3.5 Pipeline-Prozessor

### 3.6 Ausnahmebehandlung