Rechnerorganisation
Zusammenfassung SS21

Felix Marx

6. Juli 2021

Inhaltsverzeichnis

1 Vorlesung 1

1.1 Begriffe

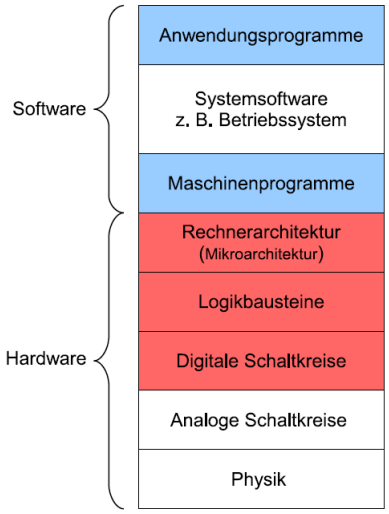

Abbildung 1: Das Schichtenmodell ist eine abstrakte Darstellung eines Computers

- Abstraktion versteckt unnötige Details
- Schichtenmodell teilt Dienstleistungen Schichten zu
  - Schichten verrichten Arbeit für die nächst höhere
  - Vorteile:
    - Austauschbarkeit von Schichten
    - Erfordert i.d.R. nur Kenntnis der aktuellen Schicht
    - z.B. Gerätetreiber erfordern Wissen niedrigerer Schichten
  - Nachteile:
    - ggf. geringere Leistungsfähigkeit

Computer ist ein Synonym für Datenverarbeitungssystem bzw. Rechnersystem:

> "*Ein Datenverarbeitungssystem ist eine Funktionseinheit zur Verarbeitung und Aufbewahrung von Daten. Verarbeitung umfasst die Durchführung* ***mathematischer***, ***umformender***, ***übertragender*** *und* ***speichernder*** *Operationen.*"

Ein Rechnersystem unterscheidet sich von Messgeräten, dadurch dass es über ein ladbares Programm schrittweise eine Funktion ausführt.

Minimale Komponenten eines Rechnersystems:

- Prozessor
- Speicher
- I/O

1.2 Geschichte

Bezeichnung	Technik und Anwendung	Zeit
Abakus, Zahlenstäbchen	mechanische Hilfsmittel zum Rechnen	bis ca. 18. Jahrhundert
mechanische Rechenmaschinen	Apparate zum Rechnen	1623 bis ca. 1960
elektronische Rechenanlagen	Lösen numerischer Probleme	seit 1944
Datenverarbeitungsanlage	Texte und Bilder bearbeiten	seit ca. 1955
Informationsverarbeitungssystem	Bilder und Sprache erkennen (KI)	seit 1968

Zuse Rechner:

- Z1 (1937): mechanische Rechenmaschine, bestehend aus
  - Ein-/Ausgabewerk
  - Rechenwerk
  - Speicherwerk
  - Programmwerk, Programm auf gelochten Filmstreifen
- Z2 (1939): Austausch der mechanischen Schaltglieder durch Relais (ca. 200) 10 Hz
- Z3 (1941): erster funktionsfähiger Digitalrechner weltweit

1.3 Ethik

- Dual Use Problem (Ziviler und militärischer Einsatz)
- Digitale Souveränität bezeichnet die Fähigkeit zu selbstbestimmtem Handeln und Entscheiden im digitalen Raum
  - Selbstbestimmtes Handeln und Entscheiden
  - Digitale Medien souverän zu nutzen (Medienkompetenz)
  - Fähigkeit Vertrauenswürdigkeit und Integrität der Datenübertragung, -speicherung, -verarbeitung durchgängig zu kontrollieren
  - Selbstbestimmung über die Nutzungsbestimmungen für seine Daten
  - eigene Fähigkeiten mit Schlüsseltechnologien eigens Dienste und Plattformen zu betreiben

2 Vorlesung 2

2.1 Speicher

Speicher können danach klassifiziert werden, ob sie für den Programmierer **explizit** oder nur **implizit/transparent** zugreifbar sind.

- explizite Nutzung
  - interner Prozessorspeicher
    - schnell, zur temporären Speicherung von Maschinenbefehlen und Daten
    - direkter Zugriff durch Maschinenbefehle

    - Halbleiter ICs[1]
  - Hauptspeicher
    - relativ groß und schnell, für Programme und Daten während der Ausführung
    - direkter Zugriff durch Maschinenbefehle
    - Halbleiter ICs
  - Sekundärspeicher
    - sehr groß und langsam, für permanente Speicherung von Programmen und Daten
    - indirekter Zugriff über I/O-Programme zur Übertragung in den Hauptspeicher
    - Halbleiter ICs, Magnetplatten, optische Laufwerke, Magnetbänder
- transparente Nutzung
  - bestimmte Register auf dem Prozessor
  - Cache Speicher

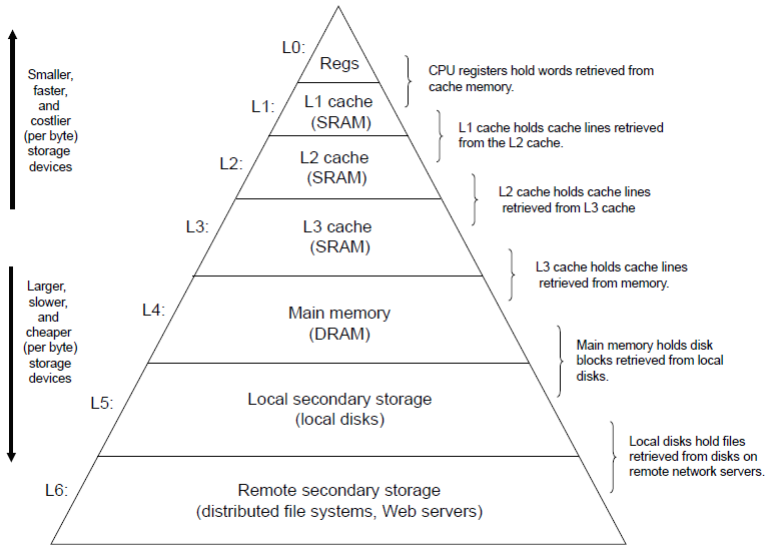

Abbildung 2: Speicherhierarchie, die Größe der Speicher nimmt nach oben ab, die Geschwindigkeit nimmt zu

2.2 Einführung Assembler

Maschinensprache (Assembler) ist ein primitives Paradigma[2].
Das Programmiermodell von Assembler bezeichnet den Registersatz eines Prozessors, bzw. die Register, die durch Programme angesprochen werden können sowie die verfügbaren Befehle (Befehlssatz). Der Instruction Pointer (IP) und Program Counter (PC) zählen nicht zum Registersatz des Prozessors.

Komponenten des Rechnersystems:

[1] IC - Integrated circuit
[2] Paradigma bezeichnet ein übergeordnetes Prinzip, welches sich in Beispielen manifestiert

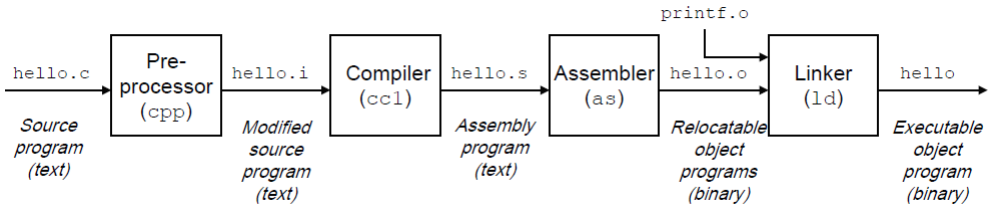

Abbildung 3: Die Phasen der Übersetzung eines C Programmes in binären Maschinencode

- CPU/Prozessor: führt im Hauptspeicher abgelegte Befehle aus
- ALU: Ausführung der Operationen
- PC: Programmzähler, der auf den nächsten Maschinenbefehl im Hauptspeicher zeigt
- Register: Schneller Speicher für Operanden
- Hauptspeicher: Speichert Befehle und Daten
- Bus Interface: Verbinden der einzelnen Komponenten

2.2.1 Übersetzung und Ausführung

1. Preprocessor
   - Aufbereitung durch Ausführung von Direktiven (mit #) Inhalt der referenzierten Datei wird in Programmdatei kopiert
   - Ausgabe: C-Programm mit der Endung .i
2. Compiler
   - Übersetzt das C-Programm name.i in ein Assemblerprogramm name.s
3. Assembler
   - Übersetzt name.s in Maschinensprache in das Objekt-Programm hello.o
4. Linker
   - Zusammenführen verschiedener Module (z.B. printf Funktion)
   - Module werden zu einem ausführbaren Programm kombiniert
   - Ausgabe des Bindeprogramms: name Datei, welche eine ausführbare Objekt-Datei, die in den Speicher geladen und ausgeführt werden kann

Nach dem Übersetzen kann die Objekt-Datei über die Shell ausgeführt werden. Dabei werden zunächst die Zeichen des Kommandos in die Register gelesen und den Inhalt dann in den Hauptspeicher gespeichert.
Anschließend werden Befehle und Daten schrittweise von der Festplatte in den Hauptspeicher kopiert.

gcc programm.c übersetzt das C-Programm
gcc -S programm.c generiert das Assemblerprogramm

Verschiedene Optimierungseinstellungen des Compilers lassen sich mit -O1 und -O2 dazuschalten.
Konstante Ausdrücke werden ggf. zur Compilezeit bereits ausgewertet und in Assembler durch das Ergebnis ersetzt.

Wir unterscheiden die Befehle eines Rechnersystems in **CISC** - Complex Instruction Set Computer und **RISC** - Reduced Instruction Set Computer.
CISC besitzen viele komplexe Befehle, wohingegen RISC weitgehend Befehle mit identischer Ausführungszeit hat um effizientes Pipelining zu ermöglichen. RISC werden auch als Load/Store-Architekturen bezeichnet.

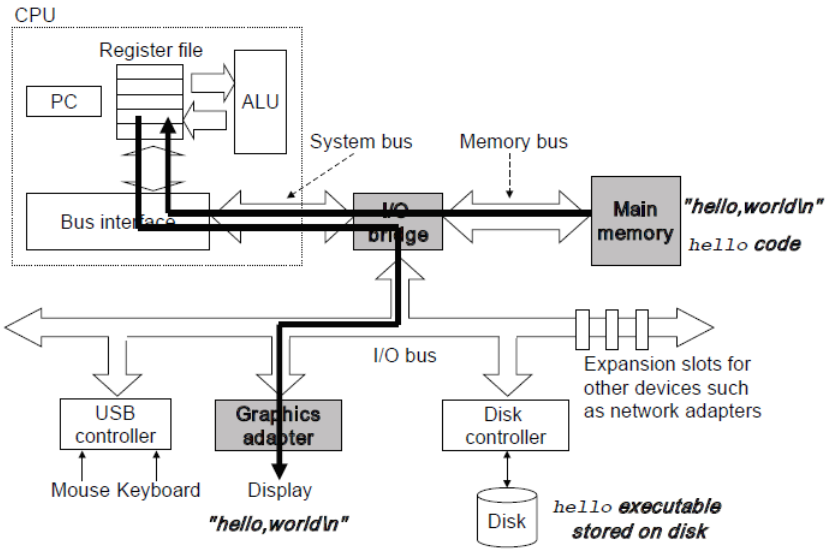

Abbildung 4: Im letzen Schritt der Ausführung werden die Maschinenbefehle des Programmes ausgeführt

Weiterhin beeinflusst die Struktur eines Prozessors die Leistungsfähigkeit und Kosten des Rechnersystems massiv.
Wir Teilen Rechnersysteme nach der Anzahl der Operanden in einem Maschinenbefehl ein (n-Adressmaschinen).

CISC-Maschine: Intel Architektur
2-Adressmaschine: Intel Architektur
RISC-Maschine: ARM Architektur
3-Adressmaschine: ARM Architektur

ARM besitzt 16 Register und ein Current Processor Status Register (CPSR). Dabei sind:

- R0: das Rückgaberegister
- R1 - R12: frei verwendbar
- R13: der Stack Pointer (sp) zeigt auf den Kopf des Stacks
- R14: das Link Register (lr) zeigt auf die Rücksprungadresse
- R15: der Programm Counter (pc) speichert die aktuelle Programmadresse

Wir unterscheiden in der Speicherorganisation: Big-Endian und Little-Endian.
Bei **Big-Endian** werden die Bytes vom höchstwertigen Ende an gezählt.
Bei **Little-Endian** werden die Bytes vom niedrigstwertigen Ende an gezählt.

In der Adressnummerierung hat bei Big-Endian das MSB also die Adresse 0, bei Little-Endian hat das LSB die Adresse 0.

ARM ist byte-adressiert und verwendet i.d.R. Little-Endian (kann aber manuell festgelegt werden). Ein Wort ist dabei 32 Bit = 4 Bytes lang, somit sind Wortadressen immer Vielfache von 4.

3 Vorlesung 3

3.1 Grundlegende Assembler Befehle

Direktwerte, d.h. Werte welche direkt als solche im Assemblercode geschrieben werden, müssen im Zweierkomplement als 12 Bit Wert repräsentierbar sein.

3.1.1 load

Mit dem Ladebefehl ldr (load word) wird ein Datenwort von einer Speicheradresse gelesen und in das angegebene Register geschrieben.

```
1 ldr r1, [r2, #4]
```

Im obigen Beispiel wird ein Datenwort von der Adresse (r2 + 4) gelesen und in das Register r1 geschrieben. Also das zweite[3] Datenwort im Register r2.
Gelesen als Basisadresse r2 plus Offset 4.
Als Basisadresse darf jedes Register verwendet werden.
Der Offset kann auch Hexadezimal angegeben werden (z.B. #0xC für #12).
Auch eine Angabe durch ein Register ist erlaubt.

3.1.2 store

Mit dem Speicherbefehl str (store word) wird ein Datenwort in das zweite Register plus Offset geschrieben.

```
1 str r2, [r3, #0x14]
```

Im obigen Beispiel wird das an der Adresse von r2 gespeicherte Datenwort an die Adresse r3 plus 20 = 0x14 geschrieben. Der Befehl hat auch eine Byte Variante strb, welche das LSB der gegebenen Adresse an das Ziel schreibt.

3.1.3 mov

Schreibt den angegebenen Wert in das Zielregister.

```
1 mov r1, #0
```

Im obigen Beispiel wird der Wert 0 an die Adresse r1 geschrieben.

3.1.4 add

Addiert zwei Werte zusammen und schreibt das Ergebnis an die Ergebnisadresse.

```
1 add r0, r0, #4
```

Im obigen Beispiel wird der Wert an der Adresse r0 um 4 erhöht und dann an die Adresse r0 geschrieben.

[3] Da die Datenworte 4 Byte lang sind

3.1.5 sub

Subtrahiert zwei Werte von einander und schreibt das Ergebnis an die Ergebnisadresse.

```
sub r3, r6, r9
```

Im Beispiel wird der Wert an der Adresse r9 vom Wert an der Adresse r6 subtrahiert und an die Adresse r3 geschrieben.

3.1.6 mul

Multipliziert die beiden Operanden mit einander und speichert die least significant 32 Bit an die Ergebnisadresse.

```
mul r1, r2, r3
```

Im Beispiel wird der Wert an der Adresse r2 mit dem Wert an der Adresse r3 multipliziert und an die Adresse r1 geschrieben.

3.1.7 lsl

Ein logischer Shift nach links[4]. Entspricht einer Multiplikation um die Zahl 2^n. Dabei wird der Wert im zweiten Register um den Wert im dritten Operanden nach links geshifted.

```
lsl r1, r2, r3
lsl r1, r2, #1
```

Wenn der dritte Operand ein Register ist, wird nur das least significant byte beachtet. Ist der Operand ein Direktwert, so darf dieser maximal 32 sein.

3.1.8 lsr

Ein logischer Shift nach rechts. Entspricht einer Division um die Zahl 2^n. Dabei wird der Wert im zweiten Register um den Wert im dritten Operanden nach rechts geshiftet.

```
lsr r1, r2, r3
lsr r1, r2, #1
```

Die selben Einschränkungen wie bei lsl gelten.

3.1.9 ORR/AND

Ein logisches bitweises OR bzw. AND, das Ergebnis wird in das erste Register geschrieben. Der zweite Operand muss ein Register sein, der dritte Operand ist beliebig.

```
AND r1, r2, #0x1234
ORR r1, r2, r3
```

3.2 Flags

Wichtige Flags sind:

- CF Übertragsflag (Carryflag)
- ZF Nullflag (Zeroflag)
- SF Vorzeichenflag (Signflag)

[4]Beim logischen Shift wird der Überfluss abgeschnitten und die leeren Stellen mit 0'en aufgefüllt

- OF Überlaufflag (Overflowflag)

Unterscheidung zwischen CF und OF Flags:
Overflowflag (OF) wird gesetzt, wenn wir z.B. zwei positive Zahlen addiert werden, aber aufgrund eines Speicherüberflusses das Ergebnis negativ ist.

Carryflags (CF) werden gesetzt, wenn wir z.B. eine positive und eine negative Zahl addieren, und das Ergebnis noch korrekt darstellbar ist, aber in der binären Addition ein Bit überläuft. Zum Beispiel so:

```
  0101   5
  1111   -1
  ----
1 0100   4
```

Ist das Ergebnis negativ wird das Signflag (SF) auf 1 gesetzt.

3.3 Assembler Sprungbefehle

3.3.1 Unbedingte Sprünge

Unbedingte Sprünge (b) werden immer ausgeführt sobald sie im Programmablauf erreicht werden.
Die Sprungadressen werden mit Labels im Programmcode festgelegt und mit einem Doppelpunkt abgeschlossen.

```
mov r1, #4
b label
sub r1, r1, #2
label:
...
```

3.3.2 beq

Der Sprung wird nur ausgeführt falls das Zeroflag (ZF) gesetzt ist.
Dabei wird i.d.R. vor dem Sprungbefehl der Vergleichsbefehl cmp aufgerufen, welcher zwei Werte durch Subtraktion vergleicht. Der Befehl cmp muss als ersten Parameter immer ein Register bekommen, der zweite Parameter ist beliebig.

```
mov r0, #4
cmp r0, #8 /* setzte Flags auf Basis von r0 - 8 = -4 ⇒ NZCV = 1000 */
beq label /* Hier kein Sprung, da Z != 1 */
...
label:
```

3.4 If-Verzweigungen

In Hochsprache:

```
if (r0 == r1)
     r2 = r3 + 1;
r2 = r2 - r3;
```

In Assembler negieren wir die Bedingung, sodass falls diese erfüllt ist wir den konditionalen Block überspringen können.

```
cmp r0, r1
bne L1 /* Springt falls Werte ungleich ⇔ Z = 0 */
add r2, r3, #1
L1:
sub r2, r2, r3
```

3.5 If-Else-Verzweigungen

In Hochsprache:

```
if (r0 == r1)
     r2 = r3 + 1;
else
     r2 = r2 - r3;
```

In Assembler negieren wir wieder die Bedingung, um somit direkt in den else Case zu springen, hängen ans Ende des if-Blockes noch einen unbedingten Sprungbefehl, um den else-Block zu überspringen.

```
cmp r0, r1
bne L1
add r2, r3, #1
b L2
L1:
sub r2, r2, r3
L2:
```

3.6 while-Schleifen

In Hochsprache:

```
int pow = 1;
int x = 0;
while (pow != 128) {
     pow = pow * 2;
     x = x + 1;
}
```

In Assembler überprüfen wir nach unserer Einstiegsmarke, ob die negierte Fortsetzungsbedingung erfüllt ist und beenden die Ausführung im positiven Fall. Sonst wird der Schleifenkörper ausgeführt und wir springen unbedingt zur Einstiegsmarke zurück.

```
mov r0, #1
mov r1, #0
WHILE:
cmp r0, #128
beq DONE
lsl r0, r0, #1 /* pow = pow * 2 */
add r1, r1, #1 /* x = x + 1 */
b WHILE
DONE:
```

3.7 for Schleifen

In Hochsprache:

```
int sum = 0;
int i;
for (i = 0; i < 10; i = i + 1) {
```

```
4       sum = sum + i;
5   }
```

In Assembler prüfen wir direkt nach der Einstiegsmarke, ob die negierte Fortsetzungsbedingung erfüllt ist und beenden im positiven Fall. Sonst wird der Schleifenkörper ausgeführt und im Anschluss Zähler entsprechend angepasst. Dann springen wir unbedingt zur Einstiegsmarke zurück.

```
1 mov r1, #0
2 mov r0, #0
3 FOR:
4 cmp r0, #10
5 bge DONE /* Springe falls der gilt r0 > 10 ⇔ N != 0 && Z = 0 */
6 add r1, r1, r0 /* sum = sum + i */
7 add r0,r0, #1 /* i = i +1 */
8 b FOR
9 DONE:
```

4 Vorlesung 4

4.1 Nutzen des Hauptspeichers

Anstatt Werte als Direktwerte zu benutzen, können sie auch als Variablen im Datenbereich des Programms definiert werden.

```
1  .data /* Datenbereich */
2  var1: .word 5 /* Variable 1 im Speicher mit Wert 5 */
3  var2: .word 12
4  var3: .word 15
5
6  .global main /* Definition Einsprungpunkt Hauptprogramm */
7
8  main: /* Hauptprogramm */
9       ldr r0, var1 /* laedt Wert von var1 in r0 */
10
11      ldr r1, adr_var2 /* laedt Adresse von var2 in r1 */
12      ldr r2, [r1] /* Laedt den Inhalt von Adresse r1 in r2 */
13
14      ldr r3, =var3 /* Laedt die Adresse von var3 direkt in r3 */
15      ldr r4, [r3] /* Laedt den Inhalt von Adresse r3 in r4 */
16
17      add r0, r0, r2
18      bx lr /* Springe zurueck zum aufrufenden Programm */
19
20 adr_var2: .word var2 /* Adresse von Variable 2 */
```

Die Daten in var1 werden direkt adressiert, die Daten in var2 werden indirekt adressiert. Die Abkürzung für die Adresse einer im Datenbereich definierten Variable ist: =variablenname, wie bei var3 dargestellt.

Die einzelnen Registerfelder liegen direkt übereinander, sodass wenn wir Zeile 15 durch:

```
1 ldr r4, [r3, #8]
```

ersetzen, die Adresse (r3 + 8) = (r2 + 4) = r1 erhalten und von dieser Adresse den Wert 12 an r4 schreiben.

4.2 Arrays

Arrays oder auch Datenfelder können als konsekutive Datenwörter aufgefasst werden welche übereinander im Speicher liegen, sodass der Index eines einzelnen Feldes dem Offset von der Basisadresse entspricht.
Dabei lädt man i.d.R. die Basisadresse des Arrays in ein Register und den Index in ein weiteres. Um nun auf einzelne Datenwörter zuzugreifen, muss der Index um die Wortbreite (4 Bytes) multipliziert werden.

```
1 mov r0, #0x11111 /* Basisadresse des Arrays */
2 mov r1, #0 /* Index i */
3 ...
4 lsl r2, r1, #2 /* r2 = i * 4 (Offset) */
5 ldr r3, [r0, r2] /* Laedt das indizierte Arrayelement */
```

4.3 Unterprogramme

Unterprogramme helfen bei der strukturierten Programmierung, indem Teile des Hauptprogrammes in Teilprogramme (TP) ausgelagert werden.
Man unterscheidet zwischen Makrotechnik und Unterprogrammtechnik. Bei der **Makrotechnik** wird das Programm an den benötigten Stellen einkopiert, dabei wird dem TP, dem sogenannten Makro, ein Name zugeordnet (Makroname). An den Stellen wo das Makro einkopiert werden soll, wird der Makroname genannt.
Bei der **Unterprogrammtechnik** ist das Unterprogramm (UP) nur einmal im Code vorhanden. Es wird durch eine Marke (Unterprogrammname) gekennzeichnet. Soll das UP aufgerufen werden, erfolgt ein Sprungbefehl mit der Marke als Operand. Am Ende des UPs erfolgt die Rückkehr in das aufrufende Programm mittels speziellem Sprungbefehl (z.B. bx). Die Rückkehradresse wird in Registern oder auf dem Stack gespeichert.

Lokale Variablen werden i.d.R. über den Stack realisiert. Als Regel für Prozeduraufrufe gilt:

- Der Aufrufer übergibt Argumente an den Aufgerufenen und springt zum Aufgerufenen
- Der Aufgerufene führt eine Funktion/Prozedur aus und gibt das Ergebnis an den Aufrufer zurück (r0)
- Die Rücksprungadresse liegt direkt hinter der Aufrufstelle
- Der Aufgerufene darf keine Register oder Speicherstellen überschreiben, die im Aufrufer genutzt werden
- Aufgerufene Unterprogramme dürfen keine unbeabsichtigten Seiteneffekte haben

```
1  main:
2  mov r0, #4 /* Argument 0 ist 4 */
3  mov r1, #30 /* Argument 1 ist 30 */
4  bl sum /* Funktionsaufruf, bl legt Rueckkehradresse in r14 (lr) */
5  mov r2, r0 /* r0 ist der Rueckgabewert */
6
7  sum: /* Unterprogrammname */
8  add r2, r0, r1 /* Fuehrt seine Funktion aus */
9  mov r0, r2 /* Legt Rueckgabewert in r0 ab */
10 mov pc, lr /* Ruecksprung zum Aufrufer */
```

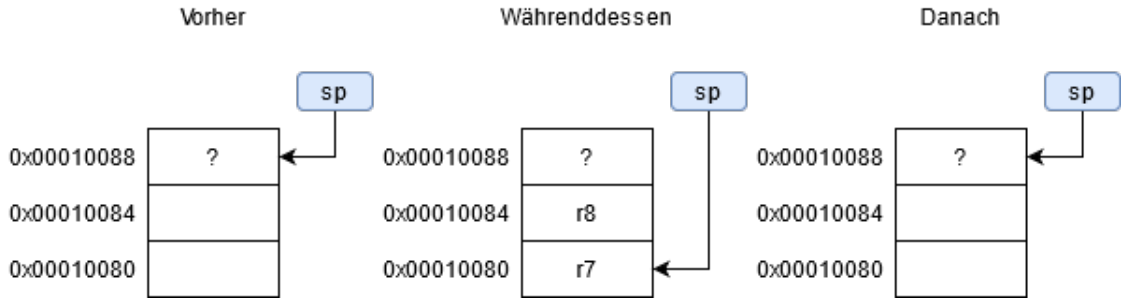

Abbildung 5: Repräsentation des Stacks vor, während und nach der Programmausführung

5 Vorlesung 5

5.1 Stack

Der Stack wächst bei ARM nach unten, d.h. von hohen zu niedrigen Speicheradressen. Der Stackpointer: r13 (sp) zeigt auf das oberste Element des Stacks.

```
sub sp, sp, #8 /* reserviert Speicher fuer 2 Woerter auf dem Stack */
str r8, [sp, #4] /* sichert den Wert von r8 auf dem Stack */
str r7, [sp]
...
ldr r7, [sp] /* stellt den Wert von r7 wieder her */
ldr r8, [sp, #8] /* stellt den Wert von r8 wieder her */
add sp, sp, #8 /* gibt den Speicher auf dem Stack wieder frei */
```

Werden weiter Unterprogramme aufgerufen, muss auch die Rücksprungadresse lr auf dem Stack gesichert werden, sonst geht diese beim Prozeduraufruf verloren.

Kurzschreibweisen (Pseudoinstruktionen) für das reservieren/freigeben von Speicher und dem ablegen/wiederherstellen von Wörtern sind die Instruktionen:

```
push {lr} /* Ablegen der Ruecksprungadresse*/
...
pop {lr} /* Wiederherstellen der Ruecksprungadresse */
```

5.2 Rekursion

Bei einer rekursiven Programmausführung repräsentiert die Anzahl an Rücksprungadressen auf dem Stack im wie vielen Aufruf wir uns befinden. Der Ablauf eines Unterprogramms wird auch als Inkarnation (wiederholter Unterprogrammaufruf) bezeichnet.
Auszug einer rekursiven Funktion:

```
.text /* markiert den Beginn eines Code Segments */
fak:      push {r0, lr}
          cmp r0, #1 /* Rekursionsende pruefen */
          blt else /* branch less ⇔ N = 1 */
          sub r0, r0, #1 /* n-1 */
          bl fak /* rekursiver Funktionsaufruf */
          /* Ruecksprungadresse beim zurueckgehen */

RA_2:     ldr r1, [sp, #4] /* laden von n */
          mul r0, r1, r0 /* fak(n-1) * n */
fin:      pop {lr} /* laden der Rueckkehradresse */
          add sp, sp, #4 /* Freigeben des Stackspeichers */
          bx lr     /* Sprung zum Aufrufer */
```

```
14
15 else:     mov r, #1 /* Rekursionsanker */
16          b fin /* Rekursionsende, abbauen des Stacks */
```

6 Vorlesung 6

6.1 OpenMP

OpenMP ist eine Sammlung von Compiler-Anweisungen (Direktiven) und Bibliotheksfunktionen für die Thread Programmierung.
Es ist eine Kombination von C, C++ und Fortran. Dadurch wird threadparalleles Arbeiten ermöglicht.
OpenMP startet als einzelner Thread und forkt für einzelne parallele Programmabschnitte zusätzliche Threads. Nach Abschluss joinen die Threads wieder zusammen. Dies wird auch **Fork-Join-Programmiermodell** bezeichnet.

```
1 #include<omp.h> /* OpenMP Library */
2 ...
3 #pragma omp parallel for private(x) reduction(+:sum)
4 for( i = 0; i < num_steps; i++) {
5      x = (i + 0.5) * step;
6      sum += 4.0 / (1.0 + x * x)
7 }
8 ...
```

6.2 Assembler

Der Assembler ist ein Programm, dass die Aufgabe hat, Assemblerbefehle in Maschinencode zu transformieren und dabei symbolische Namen Maschinenadressen zuweist und eine oder mehrere Objektdatei(en) erzeugt.
Varianten sind Crossassembler, welche Maschinencode für eine andere Plattform erzeugen als sie gerade laufen, sowie Disassembler welche Maschinensprache in Assemblersprache übersetzen.

Die Übersetzung erfolgt in zwei Schritten:
1. Schritt:

- Auffinden von Marken um Beziehungen zwischen symbolischen Namen und Adressen zu kennen (Syntax- und Kontextanalyse)
- Übersetzen jedes Assemblerbefehls durch Kombination der Opcodes[5], Bezeichner und Marken zu legalen Instruktionen (Codegen)

2. Schritt: Erzeugen einer oder mehrerer Objektdateien. Diese ist meist nicht ausführbar, da sie auf Funktionalitäten anderer Dateien angewiesen ist.

Probleme bei den Schritten:

- Bei Schritt 1 sind zukünftige Marken nicht bekannt → Aufteilen in Syntax- und Kontextanalyse (2 Läufe)
- Bei Schritt 2 werden in der Objektdatei absolute Adressen verwendet

[5]Nummer des entsprechenden Maschinenbefehls

  - Programm kann direkt ausgeführt werden, aber der Speicherort muss vorher bekannt sein
  - Nachträgliches verschieben des Programms ist nicht möglich
- Bei Schritt 2 werden relative Adressen verwendet, und die aktuelle Adresse bei Programmbeginn übergeben
  - Erzeugt ggf. mehrere Objektdateien
  - Adressen werden relativ zu den einzelnen Objektdateien vergeben
  - Vor Ausführung sind weitere Transformationsschritte nötig, Aufgabe des Binders/Linkers und Laders

6.3 Objekt-Programme

Wir unterscheiden:

- Relocatable object files: Enthält binären Code und Daten, sodass diese mit anderen relocatable object files zu einem ausführbaren Objektfile zusammengefügt werden können
- Executable object files: Enthält binären Code und Daten, welche direkt in den Speicher kopiert und ausgeführt werden können
- Shared object files: Spezialfall der relocatable object files, welche in den Speicher geladen werden können und dynamisch mit anderen Objekt-Files zusammengeführt werden können

In der Regel generieren Compiler und Assembler relocatable object files.

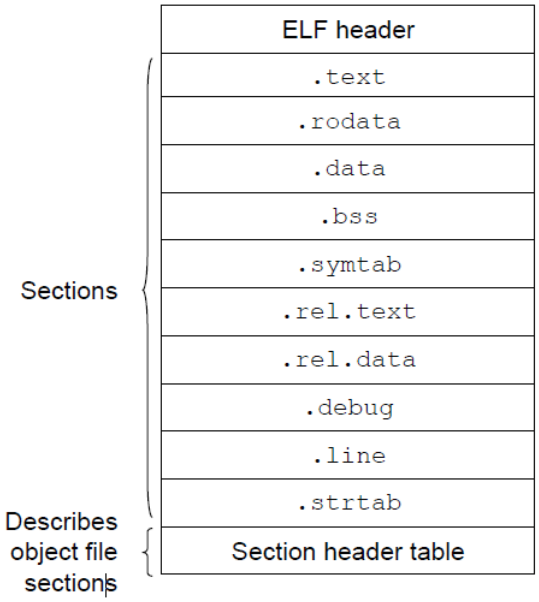

Abbildung 6: Das Format eines typischen ELF relocatable object files

Ein ELF[6] relocatable object file beginnt mit einer 16-Byte Sequenz:

- Wort-Größe
- Byte-Ordering

[6] ELF - Executable and Linkable Format

- Weitere Infos für den Binder/Linker z.B. Maschinentyp

Die einzelnen Segmente des files haben folgende Bedeutung:

- .text Maschinencode des compilierten/assemblierten Programms
- .rodata Daten, welche nur gelesen werden müssen z.B. Formatierungsstrings oder Sprungtabellen für switch
- .data Initialisierte globale Variablen
- .bss[7] Uninitialisierte globale Variablen
- .symtab Symboltabelle mit Infos über Funktionen und globale Variablen
- .rel.text Liste an Stellen, welche beim Linken modifiziert werden müssen, z.B. Kombination mit anderen files
- .rel.data Relocations Informationen für globale Variablen
- .debug Debugging Symboltabelle (Nur erzeugt, falls C-Compiler mit -g aufgerufen wurde)
- .line Zuordnung der C-Anweisung zu Maschinencode (Nur falls -g gesetzt)
- .strtab Zeichentabelle für Symboltabelle udn Debugging Symboltabelle

Diese Datei kann mit dem Befehl **readelf -h filename.o** geöffnet werden.
Mit **readelf -a filename.o** erhält man eine Übersicht über die wichtigsten Einträge.
Mit **objdump -S filename.o** erhält man den Maschinencode.

Analysewerkzeuge sind:

- IDA, ein kommerzieller Disassembler
- Ghidra: Reverse-Engineering-Werkzeug der NSA

6.4 Binder und Lader

Der Binder (linker) hat die Aufgabe aus mehreren einzelnen verschiebbaren Objekt files ein ausführbares Objektprogramm zu erzeugen, indem die noch offenen externen Referenzen aufgelöst werden.
Das Objektprogramm kann dann durch einen Lader zur Ausführung gebracht werden.
Ein Lader (loader) ist ein Systemprogramm, welches Objektprogramme in den Speicher lädt und ggf. deren Ausführung anstößt.
Der Lader lädt ein Programmmodul (Lademodul) beginnend mit einer vom Betriebssystem vorgegebenen Startadresse in den Hauptspeicher.
Varianten sind:

- absolutes Laden
- relatives Laden
- dynamisches Laden zu Laufzeit

[7] Block Storage Start, Better Save Space

6.5 Laufzeit

Laufzeitmessungen von Programmen auf Assemblerebene können mit sogenanntem Programm Profiling durchgeführt werden.
Mit **gcc -pg -o functionname programmname.c** kann eine Objektdatei erzeugt werden, welche ein solches Profiling durchführt. Die Datei wird mit **./functionname** ausgeführt, dann lässt man sich mit **gprof functionname** die Profile Datei ausgeben.
Dies gibt Aufschluss über eventuelle Bottlenecks und zeigt Informationen wie z.B. die Zeit die das Programm in einzelnen Subroutinen verbracht hat. Aufbauend darauf kann dann z.B. Loop-Unrolling oder Thread Programmierung durchgeführt werden.

7 Vorlesung 7

7.1 Microarchitekturen

Die Befehlsausführung funktioniert grob in mehreren Schritten:

1. Befehlsholphase (instruction fetch): Das Steuerwerk liest die Befehle in den Speicher (Register)
2. Befehlsdekodierung (instruction decode): Dekodiert den Befehl
3. Befehlsausführung (instruction execute): Befehl wird ausgeführt, der nächste Befehl wird geholt

Damit die Komponenten eines Rechnersystems kommunizieren können muss i.d.R. ein gemeinsamer Takt existieren.
Dieser wird als $f = \frac{1}{T}$, mit T als Periodendauer bestimmt. Je höher der Takt, desto schneller werden Daten verarbeitet.
Der Leistungsumsatz ist $P \approx U^2 \cdot f \cdot C_L$.

Terminologie:

- ISA: instruction set architecture (Menge der verfügbaren Befehle)
- RISC: reduced instruction set computer (kleine und schnell ausführbare ISA) z.B. ARM
- CISC: complec instruction set computer (schwergewichtige ISA) z.B. Intel
- SIMD: single instruction multiple data (vector processing) z.B. NEON
- VLIW: very long instruction word (static multi-issue), superscalar processor
- μarch microarchitecture (ISA Implementierung):
  - Anzahl der Befehle pro Zyklus (IPC)
  - Pipelining für Instruktionslevel Parallelismus (ILP)
  - Sprungvorhersage
  - out-of-order execution von Befehlen
  - multi-issue systems (startet mehrere Befehle pro Zyklus)

Wir unterscheiden verschiedene Architekturen:

- Eintakt-Implementierung/-Prozessor: Jeder Befehl wird in einem Takt ausgeführt

- Mehrtakt-Implementierung: Jeder Befehl wird in Teilschritte zerlegt
- Pipelined-Implementierung: Jeder Befehl wird in Teilschritte zerlegt und diese parallel ausgeführt

Die Ausführungszeit eines Programmes ergibt sich als:
Ausführungszeit = (#Instruktionen) · (Takte pro Instruktion) · (Sekunden pro Takt)

Takte pro Instruktion := CPI[8]
Sekunden pro Takt := Taktperiode
$\frac{1}{CPI}$ = Instruktionen pro Takt = IPC[9]

7.2 Eintaktprozessor

Mit dem Architekturzustand bezeichnen wir die für den Programmierer zugänglichen Daten. Dazu zählen:

- PC - Programm Counter
- 16 Register
- Speicher

Wir unterscheiden zwei Speichersysteme:

- Von-Neumann-Architektur
  - gemeinsamer Speicher für Maschinenbefehle und Daten
- Harvard-Architektur
  - Befehlsspeicher und Datenspeicher sind getrennt

Dabei kann der Speicher bezogen auf den Takt asynchron gelesen werden, aber nur synchron geschrieben werden.

7.2.1 Entwicklung des Datenpfades am Eintaktprozessor

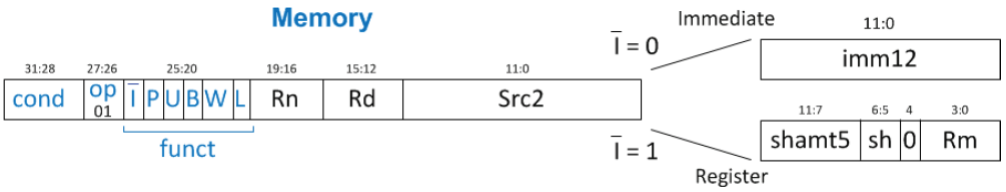

Abbildung 7: Aufbau des ldr Befehls, n:m bezeichnet, dass der Bereich Bit m bis n umfasst

Wir beginnen mit der Entwicklung des Datenpfades für den LDR (ldr Rd, [Rn, imm12]) Befehl:
Im ersten hell-orange markierten Schritt wird der Befehl im neuen Takt vom Instruction Memory geholt.
Im zweiten dunkel grün markierten Schritt wird der Quell-Operand Rn vom Register file gelesen.
Im dritten cyan blau markierten Schritt wird der Offset ausgewertet und zur weiteren

[8] cycles per instruction
[9] instructions per cycle

Benutzung auf 32 Bit extended. Außerdem legen wir die Zieladresse an den passenden Port A3 an.
Im vierten rot markierten Schritt wird der erweiterte Immendiate mit der Basisadresse RD1, welche vom Register file aus der an A1 anliegenden Registernummer ermittelt wird, addiert um die tatsächliche Adresse zu erhalten.
Im fünften magenta blau markierten Schritt gibt der Data Memory Block die an der Adresse liegenden Daten an WD3 (Write data 3) weiter. Außerdem wird das RegWrite Steuersignal auf 1 gesetzt um die Daten aus WD3 an die Zieladresse A3 zu schreiben.
Im sechsten dunkel cyan markierten Schritt wird die Adresse des nächsten Befehls (PC + 4 Byte) berechnet und angelegt.
Im siebten mittel gelb markierten Schritt wird der Prozessor um die Möglichkeit erweitert die PC Adresse zu verwenden (Sowohl Quelle als auch Ziel). Da per Definition in der ARM Architektur das Lesen am Register R15 den PC + 8 zurückgibt, muss hierfür die nächste Befehlsadresse um 4 Byte inkrementiert werden und an der entsprechenden Stelle am Register file angelegt werden. Da PC aber auch geschrieben werden kann, muss noch der entsprechende Pfad vom Data Memory Block zur Auswahl an einen Multiplexer angelegt werden.

Um dieses Rechnersystem nun um den Befehl store (str) zu erweitern, welcher den Wert vom ersten Operanden Rd an die ermittelte Adresse aus dem zweiten Rn und dritten Operanden imm12 schreibt, muss einer Verbindung zwischen Rd und A2 hergestellt werden. Außerdem muss dann die Adresse von RD2 nun an den Write Data WD Port angelegt werden. Dies geschieht im hell magenta markierten Schritt.

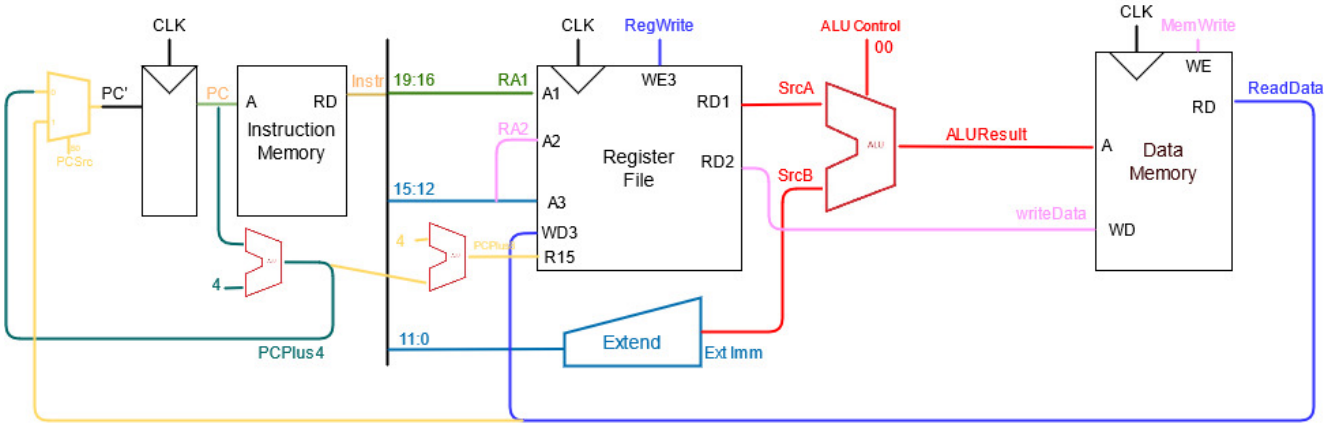

Abbildung 8: Eintaktprozessor für die Befehle ldr und str

Die Operation welche die ALU ausführt, wird mit folgenden über ALUControl gesteuerten Flags festgelegt:

$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

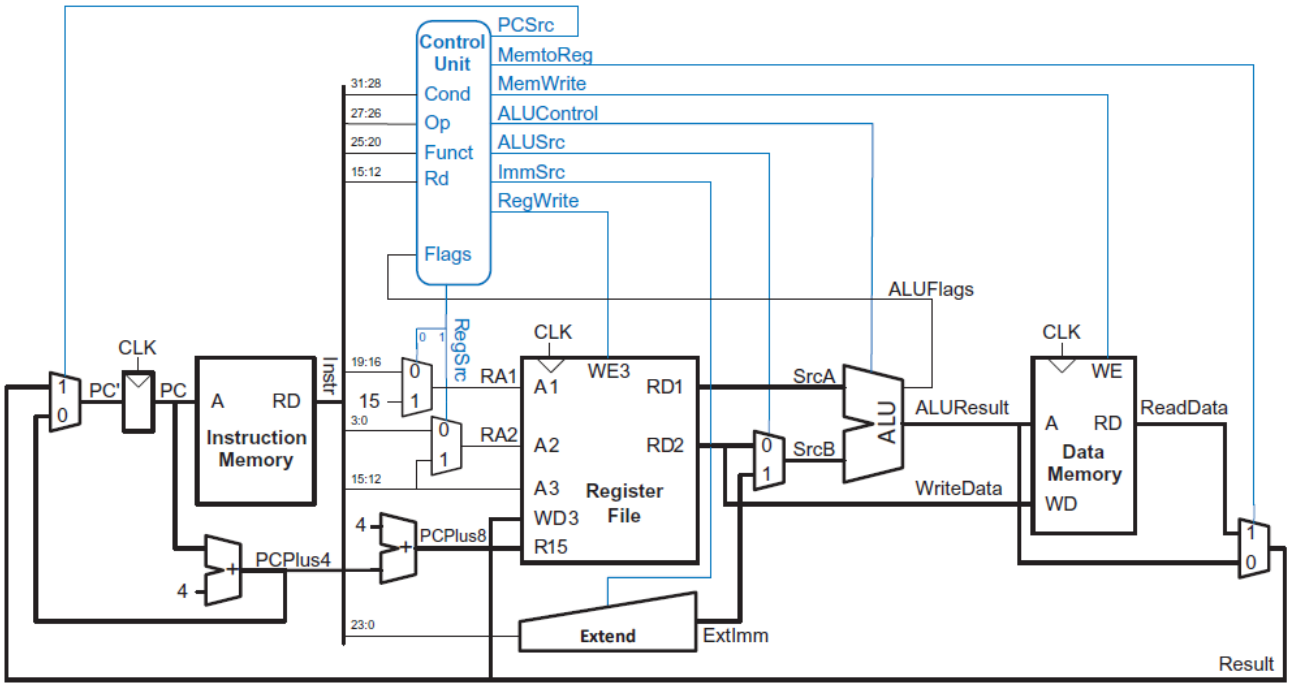

Abbildung 9: Der vollständige ARM Eintaktprozessor mit Kontrolleinheit

Die Flags der Kontrolleinheit haben folgende Bedeutungen:

Flag	Belegung	Bedeutung
PCSrc	0	PC soll auf den im Programm nachfolgenden Befehl zeigen
	1	PC soll auf eine berechnete Stelle im Programm zeigen
MemtoReg	0	Die Ergebnisdaten stammen aus einer Berechnung der ALU (z.B. ADD)
	1	Die Ergebnisdaten stammen aus einem Register (z.B. LDR)
MemWrite	0	Gebe Daten über RD weiter an die Register File
	1	Schreibe die Daten welche an der Adresse WD liegen in die Adresse A
ALUSrc	0	Verwende einen Wert aus einem Register für die Berechnung
	1	Verwende den erweiterten Direktwert für die Berechnung
ImmSrc	00	Erweitere um 24 0-Bits bei arithmetischen/logischen Befehlen ($\{24'b0, Instr_{7:0}\}$)
	01	Erweitere um 20 0-Bits bei ldr/str Befehlen ($\{20'b0, Instr_{11:0}\}$)
	10	Erweitere um 6 Vorzeichenbits bei Branch ($\{6\{Instr_{23}, Instr_{23:0}\}\}$)
RegWrite	0	Es soll nicht ins Zielregister geschrieben werden
	1	Das an WD3 anliegende Ergebnis soll in das Zielregister A3 geschrieben werden
RegSrc	0	Es existiert ein zweiter Operand, welcher vom Zielregister verschieden ist (z.B. ADD)
	1	Es sollen Werte aus dem Zielregister ausgelesen werden (z.B. STR), bzw. es soll ein Sprung ausgeführt werden, d.h. nehme Wert aus R15 (PC), dann gleichzeitig ImmSrc = 10, MemtoReg = 0 und PCSrc = 1 um das Ergebnis zu übernehmen

Sprungbefehle:
Bei Sprungbefehlen wird der nächste PC durch eine Berechnung ermittelt. Dabei wird der zukünftige PC auch als BTA - Branch Target Address bezeichnet.
Diese wird als $BTA = (ExtImm) + (PC + 8)$ berechnet, wobei $ExtImm = Imm24 << 2$ (ImmSrc = 10) ist.

7.2.2 Kontrolleinheit

Die Kontrolleinheit setzt sich aus zwei Teilen zusammen, der Conditional Logic und dem Decoder.
Die Kontrolleinheit berechnet die Steuersignale entsprechend der cond, op und funct Felder im Befehl (Bit $Instr_{31:20}$), sowie der von der ALU generierten ALUFlags und dem Fakt ob der PC das Zielregister ist.
Die Conditional Logic speichert die ALUFlags und veranlasst nur eine Änderung des Ar-

chitekturzustands, wenn die Instruktion nur bedingt ausgeführt werden soll.
Der Decoder generiert Steuersignale basierend auf der Instruktion.

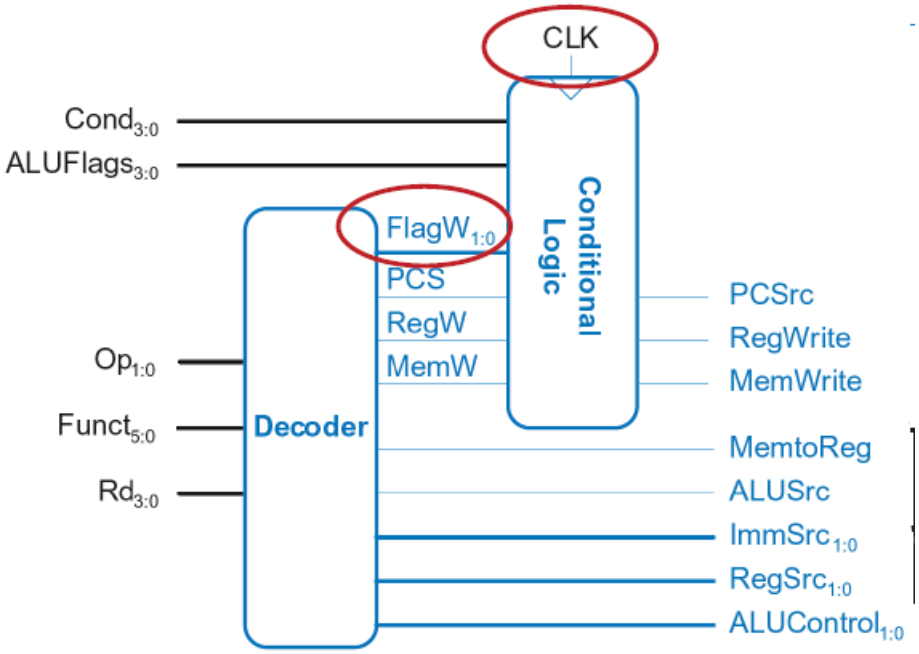

Abbildung 10: Die Kontrolleinheit des Eintaktprozessors

Die $FlagW_{1:0}$ Information gibt an, welche ALUFlags (NZCV) gespeichert werden sollen. Die ADD, SUB Befehle updaten alle Flags, wohingegen AND, ORR nur NZ verändern können. Daraus ergibt sich folgende Interpretation:
$FlagW_1 = 1$ NZ $(ALUFlags_{3:2})$ wird gespeichert
$FlagW_0 = 1$ CV $(ALUFlags_{1:0})$ wird gespeichert

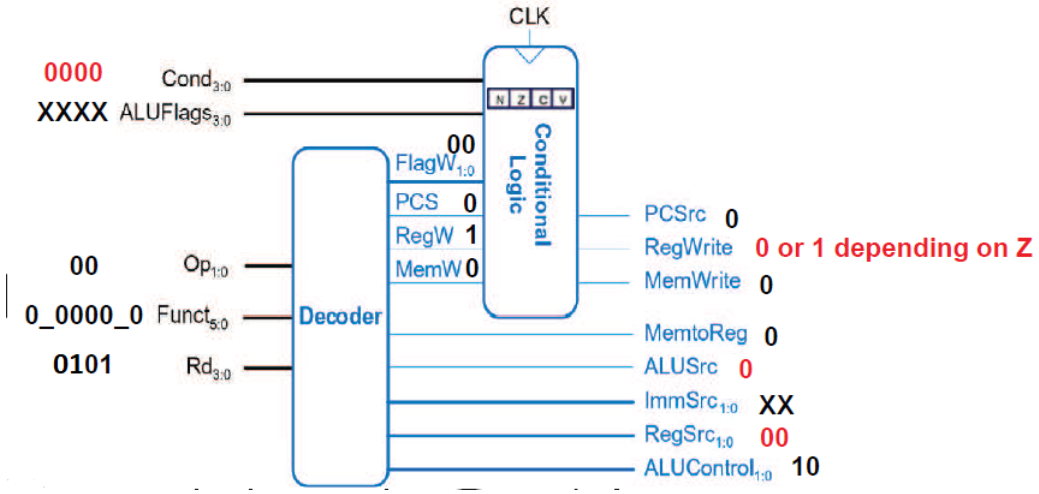

Abbildung 11: Beispielbelegung der Kontrollsignale für den Befehl: ANDEQ r5, r6, r7

Die Belegung von Cond, Op, Funct ergibt sich aus den Datenblättern.

8 Vorlesung 8

8.1 Mehrtaktprozessor

Im folgenden Mehrtaktprozessor wird keine Harvard-Architektur, sondern eine Von-Neumann-Architektur[10] verwendet und ist auch heute weiter verbreitet. Trotzdessen könnte ein Mehrtaktprozessor auch eine Harvard-Architektur sein.
Ein Eintaktprozessor kann hingegen keine Von-Neumann-Architektur sein, da so im selben Takt auf zwei verschiedene Adressen zugegriffen werden müsste.

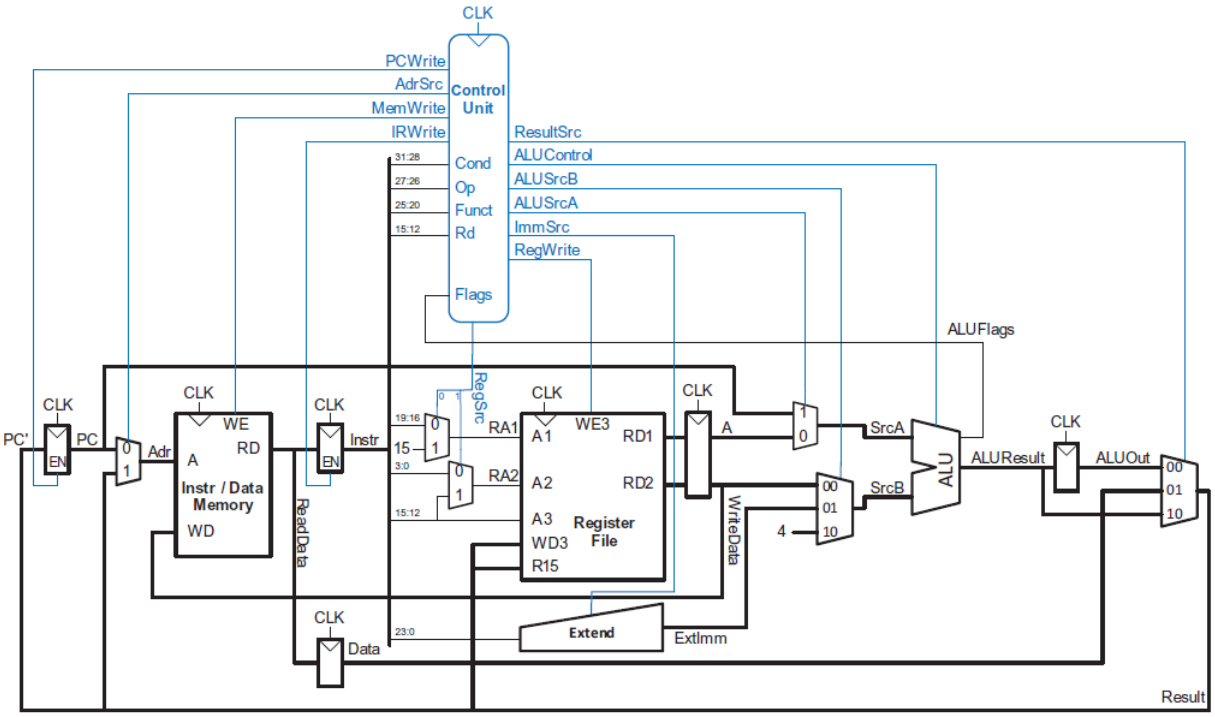

Abbildung 12: Ein Mehrtakt-Prozessor mit Kontrolleinheit

Die Kontrolleinheit des Prozessors basiert auf einer FSM, welche die einzelnen Phasen der Befehlsausführung modelliert und die Steuersignale entsprechend der Phase vorgibt. Jede Phase wird durch einen eigenen Zustand beschrieben, sodass der Prozessor die entsprechenden Aktionen ausführt.

Ein Mehrtaktprozessor hat in der Regel eine höhere Taktfrequenz und einfache Instruktionen laufen schneller. Außerdem findet eine bessere Wiederverwendung von Hardware in verschiedenen Taktet statt, hat dafür jedoch eine aufwendigere Ablaufsteuerung.

Sie besitzen jedoch die selben Grundkomponenten.

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

[10] D.h. gemeinsamer Instruktions- und Datenspeicher

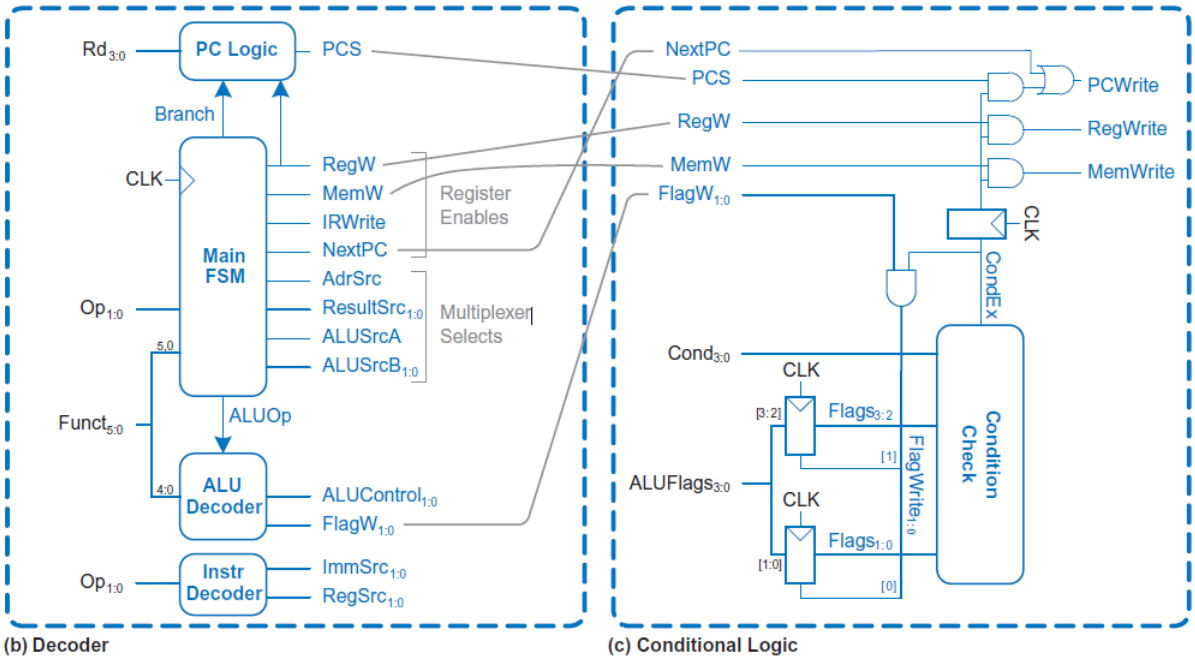

Abbildung 13: Aufgeschlüsselte Kontrolleinheit des Mehrtaktprozessors

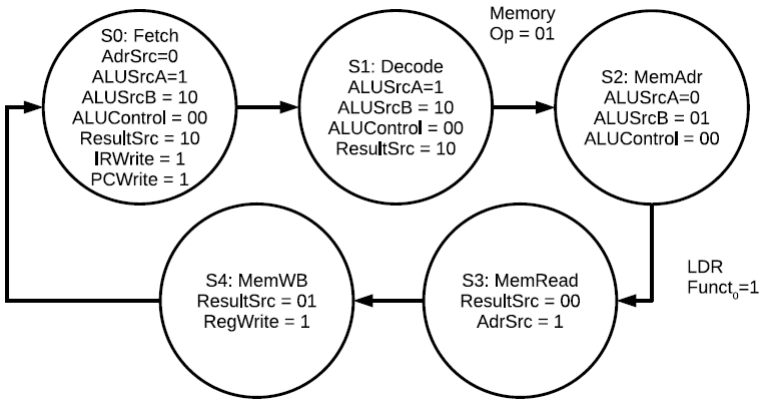

Abbildung 14: Das Steuerwerk für die Ausführung des Befehls ldr, welches das Verhalten der Kontrolleinheit in den einzelnen Ausführungsphasen beschreibt.

9 Vorlesung 9

9.1 Pipeline-Prozessor

Bei ARM wird die Befehlsausführung in fünf Schritte unterteilt:

1. Instruction Fetch
2. Instruction Decode, Read Register
3. Execute ALU
4. Memory Read/Write
5. Write Register

Wir verwenden folgende Abkürzungen:

- IM - Instruction Memory (Befehlsspeicher)
- RF - Register Field (Registerfeld)

- DM - Data Memory (Datenspeicher)

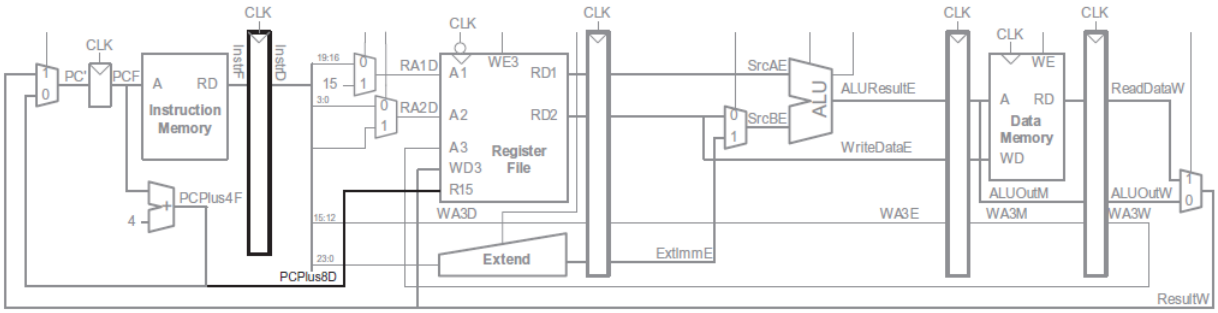

Abbildung 15: Ein optimierter Pipeline Prozessor, jede Pipelinestufe ist eine Stufe der Befehlsausführung.

Damit das richtige Writeback Register verwendet wird, wird dieses als WA3D durch die Stufen gereicht. Damit vom PC/R15 Register gelesen werden kann, könnten in die Fetch und Decode Pipelinestufen zwei Addierer den +8 Offset berechnen, oder wie hier den um +4 erhöhten PC des nächsten Befehls verwenden, und so einen Addierer sparen.

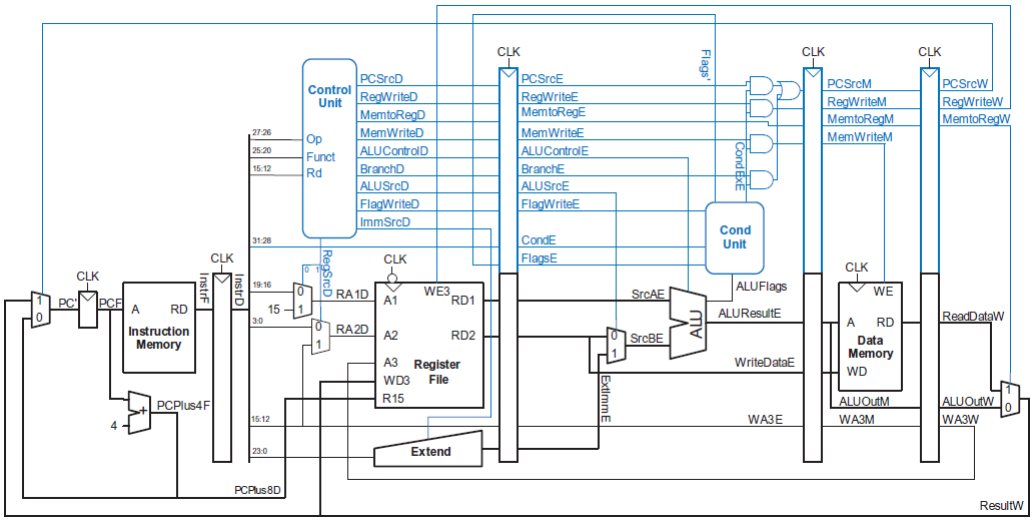

Abbildung 16: Der vollständige Pipeline Prozessor mit wie in 13 dargestellt, aufgeteilter Kontrolleinheit.

9.2 Hazards

Hazards können beim Pipeline Prozessor auftreten, wenn eine Instruktion vom Ergebnis einer vorherigen abhängt, dieser aber noch kein Ergebnis geliefert hat.
Wir unterscheiden **Data Hazards** - Neuer Wert von Register noch nicht in Registerfeld eingetragen, und **Control Hazards** - Unklar welche Instruktion als nächstes ausgeführt werden muss.

9.2.1 Data Hazard

Ein Read-after-write Hazard (RAW) tritt auf, wenn eine vorherige Instruktion den neuen Registerwert noch nicht geschrieben hat, aber eine nachfolgende auf dieses Registerfeld zugreift.

Möglichkeiten um diesen Hazard zu umgehen sind:

- Wartezeiten einplanen, d.h. nops (no operations) zur Compile-Zeit einplanen und Ablaufplanung
- Maschinencode zur Compile-Zeit umstellen (reordering)
- Daten zur Laufzeit umleiten (bypassing)
- Prozessor zur Laufzeit anhalten, bis Daten da sind (stalling)

Bei nops werden leere Instruktionen durch den Compiler in den Programmcode eingefügt, sodass benötigte Daten einen Takt weiter verarbeitet werden, und dann entsprechend aus dem Register gelesen werden können.

Beim Bypassing werden Ergebnisse aus anderen Pipelinestufen zu den benötigten Stellen weitergeleitet, um diese Ergebnisse dort zu verwenden, auch wenn diese noch nicht ins Register geschrieben wurden.

Beim stalling wird ein Schritt der Befehlsausführung, z.B. das Instruction Decode wiederholt um so einen zusätzlichen Takt für die Ausführung zu benötigen. Dadurch kann auf das Ergebnis eines Lade Befehls gewartet werden.

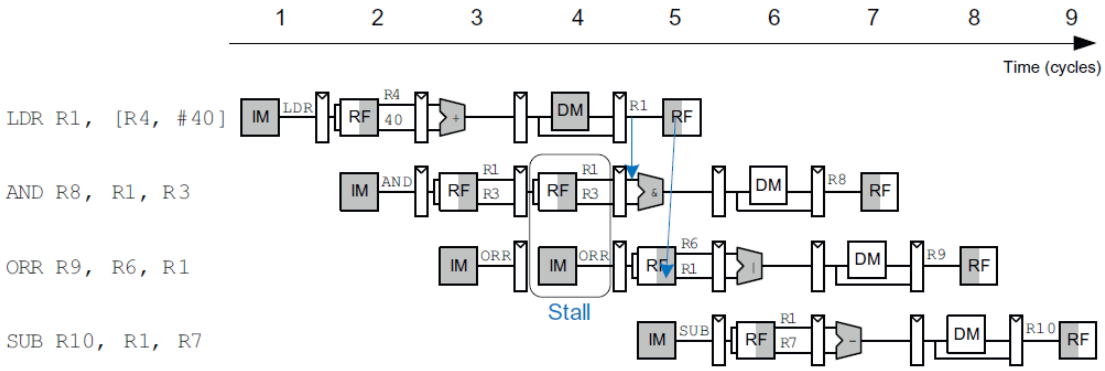

Abbildung 17: Stalling zur Vermeidung eines Daten Hazards

9.2.2 Control Hazard

Bei einem Sprungbefehl werden die direkt nachfolgenden Instruktionen aufgrund der Natur des Prozessors geladen, aber ggf. nicht benötigt. Diese Instruktionen müssen nun aus dem Prozessor geflushed werden, also deren Ausführung unterbunden werden.
Bis das Ergebnis des Sprunges allerdings durch alle Pipelinestufen propagiert ist, dauert es mehrere Takte, obwohl das Ergebnis bereits früher vorliegt. Dies kann durch eine direkte Verbindung vom Addierer welche den PC mit dem Offset addiert, mit dem Register des nächsten PCs erreicht werden.

Damit dies möglich ist wird eine weitere Kontrolleinheit - die Hazard Unit benötigt.

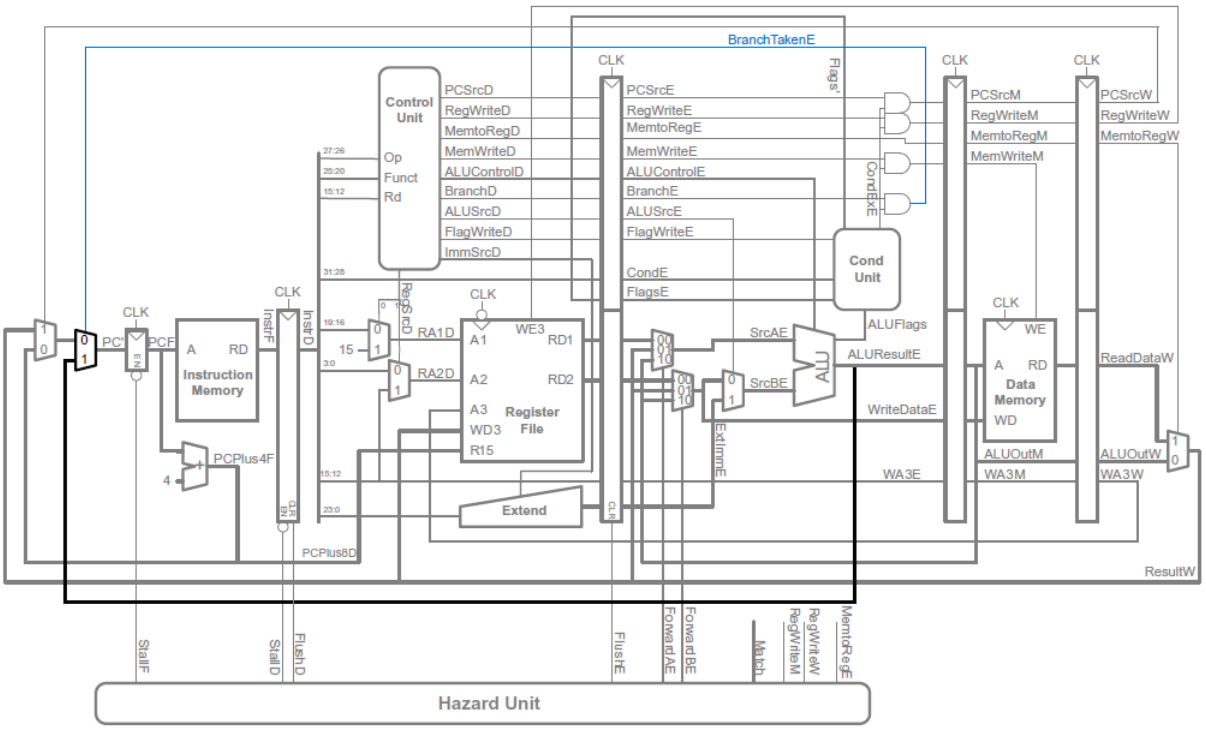

Abbildung 18: Die Hazard Unit steuert die für die Hazard Bewältigung benötigten Signale

10 Vorlesung 10

10.1 Leistungsbewertung 1

Leistungsfähigkeit sowohl von Hardware, als auch OS abhängig.
Einige Leistungskriterien für Rechnersysteme sind:

- Taktfrequenz $f = \frac{1}{T}$, T Periodendauer
- Anzahl der Prozessoren
- Größe und Art des Speichers
- Antwortzeiten: Abhängig vom Aufgabentyp
- Durchsatz: Relevanz im Rechenzentrum
- Ausführungszeit: Reine CPU Zeit ohne Ein-/Ausgabe

Ausführungszeit bestimmt als:

- system CPU time: CPU-Zeit für Betriebssystemaufgaben
- user CPU time: CPU-Zeit die zur Ausführung eines Programms benötigt wird

Leistungsumsatz:

$$P \equiv U^2 \cdot f \cdot C_L$$

Taktzyklen:
Für Eintakt # Instruktionen = # Takte
Für Mehrtakt ist $\frac{\text{Takte}}{\text{Instruktion}}$ = clock cycles per instruction (CPI)
$\frac{1}{\text{CPI}} = \frac{\text{Instruktionen}}{\text{Takt}}$ = IPC Instructions per cycle

Einfaches Maß ist MIPS - Million Instructions per second.

Bei der Klassifikation nach Flynn wird unterteilt in Instruction Streams und Data Streams.

- SI - Single Instruction
- MI - Multiple Instruction (mehrere Befehle zu einem Zeitpunkt)
- SD - Single Data
- MD - Multiple Data

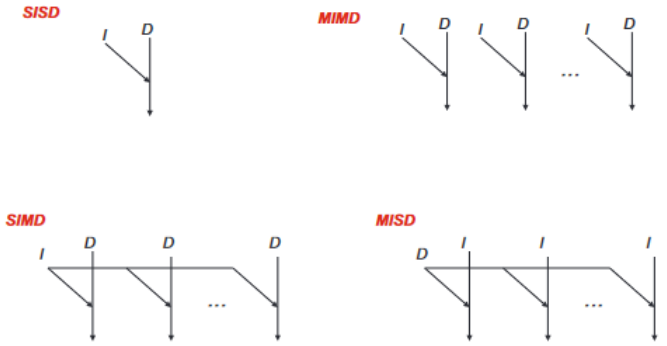

Abbildung 19: Veranschaulichung der Klassen von Flynn, MISD existiert nicht

10.2 Betriebssysteme

> Die Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften der Rechenanlage die Grundlage der möglichen Betriebsarten des digitalen Rechensystems bilden und insbesondere die Abwicklung von Programmen steuern und überwachen. - Def. Betriebssystem nach DIN 44300

Zu den Aufgaben des Betriebssystems gehören:

- Geräteüberwachung und -steuerung
- Unterbrechungssteuerung
- Ablaufsteuerung
- Datenhaltung
- Einhaltung von Qualitätsanforderungen

Der Prozessor kann sich in verschiedenen Betriebszuständen befinden:
Maschinenzustand, der Prozessor kann aus/an geschaltet sein, oder laden
Privilegierungszustand:

- Anwenderzustand: nicht privilegiert, nur eingeschränkter Befehlsvorrat
- Systemzustand: privilegiert, voller Befehlsvorrat

Ein Befehl heißt privilegiert, wenn er ausschließlich im Systemzustand ausführbar ist.

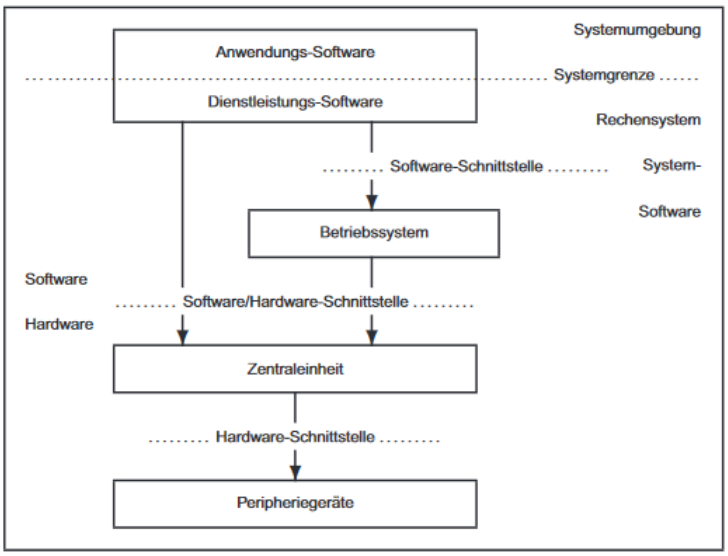

Abbildung 20: Darstellung des Schichtenmodells mit Schnittstellen zwischen Komponenten

10.3 Ausnahmebehandlung

Ohne Unterbrechungen ist die Arbeitsweise des Rechners:

1. Prozessor stößt Tätigkeit an
2. Gerät arbeitet selbständig, Prozessor muss warten
3. Prozessor arbeitet weiter

Keine Parallelität und damit schlechte Ausnutzung des Prozessors.

Mit Unterbrechungen ist die Arbeitsweise:

1. Prozessor stößt Tätigkeit an
2. Gerät arbeitet selbständig, Prozessor arbeitet parallel weiter
3. Am Ende meldet sich das Gerät mit einer Unterbrechung beim Prozessor

Ein solcher Interrupt ist ein Signal welches den Befehlszyklus des Prozessors abändert/unterbricht und diesen an einer bestimmten Stelle fortführt. Man unterscheidet grob:

- Programmbezogene Unterbrechungen (z.B. arith. Fehler, Adressfehler, falsche Befehle, ...)
- Systembezogene Unterbrechungen (z.B. E/A-Unterbrechung, Prozessoranrufe, ...)
- Maschinenfehler

Eine programmbezogene Unterbrechung trifft den Verursacher.
Systemaufrufe können auch als Interrupts zwischen nieder- und höherprivilegierten Zuständen wechseln, dadurch werden Dienste des Betriebssystems für den Benutzer verfügbar.

11 Vorlesung 11

11.1 Zahlendarstellung

Um auch reelle Zahlen darstellen zu können, muss ein Weg gefunden werden die Kommastelle zu repräsentieren.
Möglichkeiten hierzu sind:

- Festkommadarstellung
- Gleitkommadarstellung

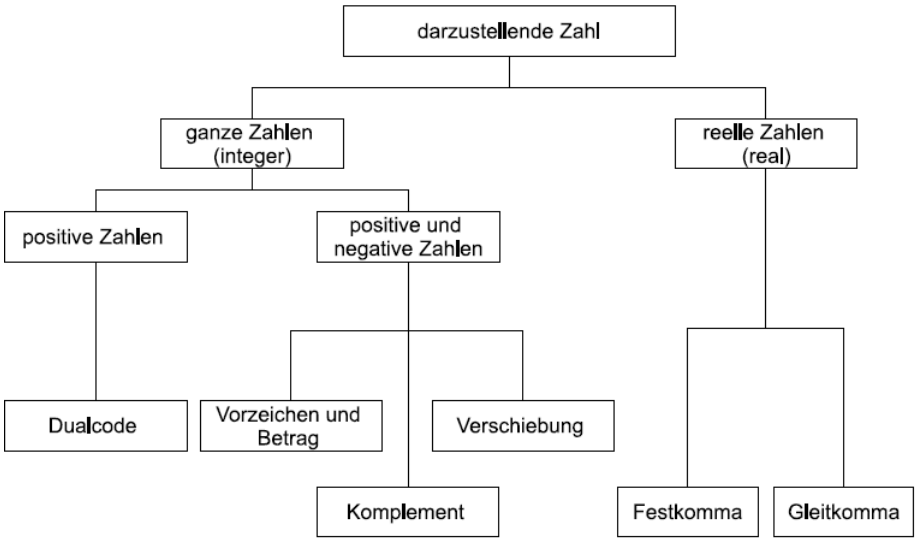

Abbildung 21: Zahlendarstellung in Rechnersystemen

Bei der Festkommadarstellung wird das Komma in der internen Zahlenabbildung weggelassen und man merkt sich wo es stehen müsste. Zum Programmbeginn wird die Kommastelle der entsprechenden Variablen definiert und bleibt dann während des Programms fest, deshalb **Fest**komma.

Bei der Gleitkommadarstellung auch halblogarithmische Darstellung ist die Kommastelle Bestandteil der Zahl und kann daher im Laufe der Zeit verändert werden. Diese Darstellung ist in ANSI/IEEE 754 spezifiziert und ist weltweit für den Datenaustausch und Rechenwerke standardisiert. Im Standard sind single, double und extended precision definiert.

Die Gleitkommadarstellung setzt sich aus dem Sign Bit (höchstwertige Position), dem biased Exponent und Fraction.

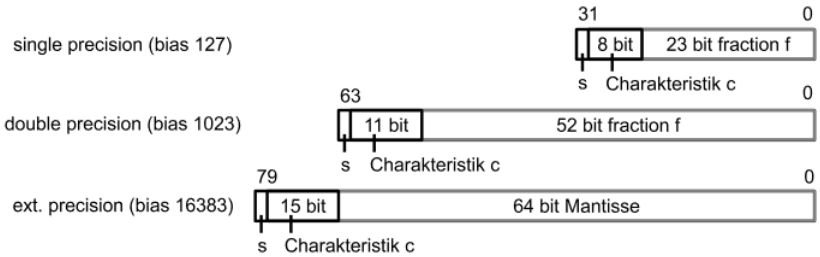

Abbildung 22: Precision im IEEE 754 Standard

Gleitkommazahlen in eingebetteten Systemen ohne Hardware-Unterstützung sollte vermieden werden.
Es existieren auch single precision Register (s0 - s31) und double precision register (d0 - d31) in Assembler, sowie passende Befehle für Addition (vadd.f64) und Loading, etc..
Der Befehl cpuinfo gibt Informationen über die CPU Architektur und Features an.

Gleitkommazahlen können entweder mit NEON oder VFP verarbeitet werden.

11.2 Leistungsbewertung 2

Leistungsmaße sind:

- Taktfrequenz
- CPI Rate (Clock Cycles per Instruction)
- MIPS Rate (Million Instructions per Second)
- MFLOPS (Million Floating-point Operations per Second)

Zur Messungen können Benchmarks verwendet werden, wobei allerdings das Problem besteht dass häufig Compiler so optimiert werden, dass gängige Benchmarks schneller laufen. Arten von Benchmarks sind:

- Reale Programme
- Kernels: kritische Auszüge aus realen Programmen
- Toy Benchmarks: z.B. Quicksort
- Synthetische Benchmarks: Spezielle Programme zur Leistungsevaluation

Benchmarks sind: SPEC, gzip und bzip.

BogoMips ist unwissenschaftlicher Benchmark, und ermittelt Wert beim Booten um Warteschleifen zu kalibrieren.

11.3 NEON

Dedizierte Funktionseinheit zur Beschleunigung der Berechnung von Ganz- und Gleitkommazahlen. Beschleunigen Anwendungen der Signalverarbeitung, Videocodierung, etc.

NEON führt Packed SIMD aus, d.h. Register werden als Vektoren eines Datentyps betrachtet.
NEON hat 32 128 Bit Register welche auch paarweise kombiniert werden können.

Die Resultate von Operationen unterliegen der Saturations-Arithmetik. Hierbei liegen alle Werte zwischen einem maximalen und minimalen Wert. Liegt ein Resultat einer Operation über dem Maximum wird es auf das Maximum gesetzt, liegt es unter dem Minimum auf das Minimum.

Neon kann auf vier Wege benutzt werden:

- NEON optimized libraries (OpenMax DL, Ne10)
- Vectorizing compiler (gcc)

- NEON intrinsics
- NEON assembly

12 Vorlesung 12

12.1 Speicher

Das Speichersystem besteht aus einer Hierarchie unterschiedlicher Speichertechnologien und -techniken. Zugriffe auf diesen Speicher sind sind sequentiell (von Neumann) bzw. kann auch parallel passieren (Harvard).

Die Speicherhierarchie lässt sich in einer sogenannten Speicherpyramide darstellen, siehe 2.

Speicher lässt sich anhand mehrerer Eigenschaften vergleichen:

Kosten und Zugriffszeit:
Kosten werden in Dollar/Bit bzw. Dollar/MByte gemessen, wobei i.d.R. gilt geringere Zugriffszeit $\rightarrow$ höhere Kosten.

- Zugriffszeit: Durchschnittliche Zeit um ein Wort aus dem Speicher zu lesen
- Zykluszeit: Minimale Zeit zwischen zwei Speicherzugriffen
- Bandbreite: maximale Datenmenge die pro Sekunde übertragen werden kann (Byte/sec)

Zugriffsverfahren:

- Random Access:
  - Dynamischer RAM (DRAM) z.B. Hauptspeicher
  - Statischer RAM (SRAM): schneller und teurer als DRAM, z.B. Cache
- serieller Zugriff: Festplatten, optische Platten, Magnetband

Veränderbarkeit des Speichers:

- Read-only: Kein Überschreiben, nur lesen, Inhalt beim Fabrikationsprozess festgelegt
- Read-write: Inhalt veränderbar, wird als Haupt- und Cache-Speicher verwendet
- Read-mostly: PROM-Halbleiterspeicher, wird für BIOS benutzt

Permanenz:

- Flüchtiger Speicher: geht bei Stromausfall verloren (z.B. Register, Hauptspeicher)
  - dynamische Speicher: Periodisch Spannung anlegen um Daten zu erneuern (Refreshing)
  - statische Speicher: kein Refreshing nötig
- Nicht flüchtige Speicher: Nicht von angelegtem Strom abhängig (z.B. ROM, Festplatte)

Speichertechnologien:

SRAM speichert Informationen mit zwei gekoppelten Invertern (Bistabile Schaltung). Die Inverter werden jeweils mit sechs Transistoren welche in einer 6T-Zelle angeordnet sind realisiert.

DRAM speichert Informationen in einem Kondensator, welcher auch als 1T-Zelle bezeichnet wird. Der Kondensator entlädt sich und muss damit Refreshed werden um diese wieder aufzufrischen. Der Auf-/Entladeprozess ist hierbei eine Kurve

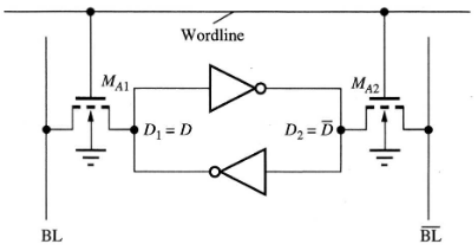

Abbildung 23: SRAM Speicher

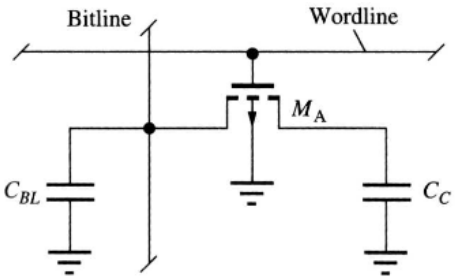

Abbildung 24: DRAM Speicher

12.2 Speicherorganisation

Eigenschaften eines RAM-Chips sind durch die Anzahl an adressierbarer Plätze und Breite jedes adressierbaren Platzes in Bit gegeben. Damit bezeichnet 256K $\times$ 1 SRAM einen SRAM mit $256K = 2^{18}$ Einträge mit 1 Bit Breite, d.h. es gibt 18 Adresseingänge mit einem 1 Bit Datenein-/ausgang.

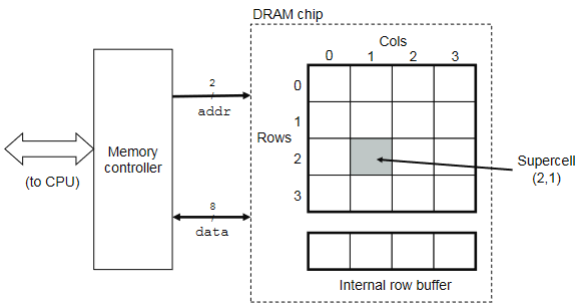

Abbildung 25: Abstrakte Repräsentation eines DRAM Chips

Der Zugriff geschieht in zwei Phasen einem Row Address Strobe (RAS), wo die gesuchte Zeile in den internen Buffer kopiert wird. Anschließend wird beim Column Address Strobe die entsprechende Spalte selektiert und über den Datenkanal zurückgegeben.
Mehrere solcher Module können kombiniert werden um größere Wörter abzuspeichern.
Der Hauptspeicher ist mittels eines Busses mit der I/O Bridge und der CPU verbunden.

12.3 Lokalität

Das Lokalitätsprinzip beschreibt, dass bei der Programmausführung nur auf einen relativ geringen Teil des Speichers zugegriffen wird.

Man unterscheidet

- **zeitliche (temporale) Lokalität**: Nach einem Zugriff auf einen bestimmten Datensatz wird mit großer Wahrscheinlichkeit bald erneut darauf zugegriffen (z.B. durch Schleifen)

- **räumliche Lokalität**: Nach einem Zugriff wird mit großer Wahrscheinlichkeit auf einen in unmittelbarer Nähe liegenden Speicher zugegriffen (z.B. Matrizen, sequentielle Instruktionen)

Gut geschriebene Programme haben eine gute Lokalität, den das Lokalitätsprinzip hat eine enorme Auswirkung auf die Performanz eines Rechnersystems. Auch hier kann man zwischen der Lokalität der Daten und der der Befehle unterscheiden.

Beispiele für Lokalität:

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

Dieses Beispiel hat gute räumliche Lokalität, da Elemente des Vektors sequentiell gelesen werden. Jedoch schlechte zeitliche Lokalität, da wir jedes Vektorelement nur einmal lesen.

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

Hier haben wir sowohl schlechte räumliche, als auch schlechte zeitliche Lokalität, denn wir summieren die Werte nicht so wie im Speicher liegen (Zeilenweise), sondern Spaltenweise. Außerdem lesen wir auch hier alle Elemente nur einmal.

Da auch Befehle Daten sind, kann man sagen dass Schleifen gute räumliche und zeitliche Lokalität haben, da wir sie mehrmals durchlaufen.

Bei einer guten Lokalität kann sich die Laufzeit eines Programmes signifikant verbessern.

12.4 Cache

Der Cache ist ein kleiner und schneller Speicher (SRAM), welcher auf der Ebene k eine Teilmenge der Daten eines größeren und langsameren Speicher auf Level k+1 speichert. Wenn die Lokalität nicht beachtet wird, muss durch mehrere Cache Ebenen hindurch auf Daten zugegriffen werden.

Werden gesuchte Daten auf einer Ebene k gefunden, spricht man von einem **Cache Hit** und der Wert wird direkt aus Ebene k gelesen. Dadurch ergibt sich ein Geschwindigkeitsvorteil.

Werden die Daten nicht gefunden spricht man von einem **Cache Miss** und die Daten müssen aus dem Level k+1 geholt werden. Ist der k-te Cache voll, müssen Daten dieses Caches überschrieben werden. Dies nennt man Ersetzung und dafür existieren verschiedene Strategien.

- Zufallsersetzung
- Least-recently used (LRU) Ersetzung

Desto weiter wir im Cache absteigen müssen, desto größer wird die Latenz. Auch zwischen den L1, L2, L3 Caches existieren Unterschiede.

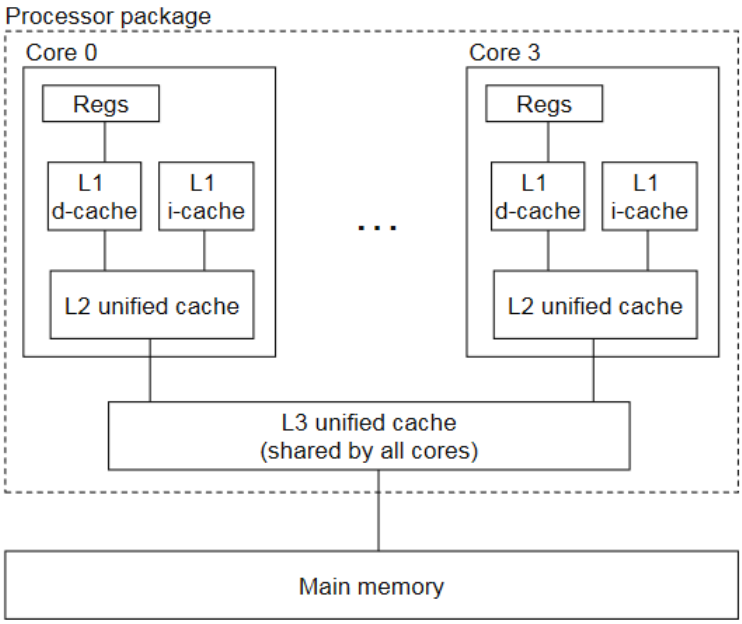

Abbildung 26: Beim ARM A53 hat man getrennte Caches für Daten (d-cache) und Befehle (i-cache) auf Level 1 mit Größen L1: 2x 32KB, L2: 256KB, L3: 8MB