

Rechnerarchitektur

Befehlssatzarchitektur I

Univ.-Prof. Dr.-Ing. Rainer Böhme

Wintersemester 2021/22 · 24. November 2021

Gliederung heute

1. Von der sequenziellen Logik zum Mikroprozessor
2. **ARM-Mikroarchitektur**
3. ARM-Befehlssatz (ohne Speicherzugriff)
4. Unser erstes Assemblerprogramm

Hintergrund

Sophie Wilson und Steve Furber entwickeln die ARM-Architektur ab 1983 beim englischen Computerhersteller Acorn, heute ARM, in Cambridge.

- ARM stellt keine eigenen Chips her, sondern verkauft Lizenzen an Halbleiterhersteller, die den Prozessor an die Bedürfnisse ihrer Kunden anpassen und mit anderen Komponenten integrieren (z. B. System-on-a-Chip, SoC).
- Einige Lizenznehmer (Apple, Intel, Motorola, NXP etc.) dürfen auch den Kern weiterentwickeln.
- **Folge:** Es gibt eine Vielzahl an ARM-Varianten. → siehe z. B. Wikipedia-Artikel
- Wir behandeln ausgewählte Teile des ARMv6-Designs (32 Bit, 2002). Es ist bei Mikrocontrollern noch weit verbreitet (z. B. Raspberry Pi).
- Aktuell ist ARMv9 (64 Bit), vorgestellt im März 2021.
- ARMv9 ist abwärtskompatibel bis ARMv5.

Namenskonventionen

Diese Folie dient allein der Orientierung und ist nicht prüfungsrelevant!

ARM unterscheidet Produktfamilien nach Einsatzbereichen:

Cortex-A für Anwendungen (Smartphones, Spielkonsolen)

Cortex-M für Mikrocontroller (Haushaltsgeräte, „Internet der Dinge“)

Cortex-R für Echtzeitanwendungen (Realtime: Automotive, Industriesteuerung)

SecurCore für Sicherheitsanwendungen (Geldautomaten)

In jeder Familie gibt es Produkte, die verschiedene Designs (ARMvX) umsetzen.

ARM-Chips lassen sich mit (bis zu 16) verschiedenen **Koprozessoren** konfigurieren, z. B. für digitale Signalverarbeitung (DSP), Gleitkommaarithmetik (VFP), Java-Hardwarebeschleunigung, Virtualisierung, Speicherverwaltung, ...

Registersatz

CPUs sind Zustandsautomaten. Ihr Zustand wird in wenigen, direkt mit der Logik verbundenen **Registern** gespeichert.

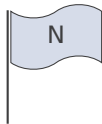
Bei ARM stehen im **User-Modus** 16 Register mit je 32 Bit zur Verfügung:

r0	zur freien Nutzung
r1	zur freien Nutzung
⋮	
r12	zur freien Nutzung
r13	reserviert für Stack-Pointer (SP)
r14	reserviert für Rücksprungadresse (Link Register, LR)
r15	reserviert für Programmzähler (PC)

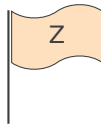
Die Verbindung mit dem (über den Systembus angebotenen) **Arbeitsspeicher** erweitert den Zustandsraum erheblich.

Flags

Die ALU setzt Flags (Bits) in einem **Statusregister**.



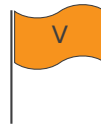
negative



zero



carry



overflow

Arithmetische Operationen

$N =$ höchstwertiges
Ergebnisbit

$Z = 1$: Ergebnis
ist Null

$C = 1$: Übertrag;
Ergebnis > 32 Bit

$V = 1$: arithmetischer
Überlauf

Logische Operationen

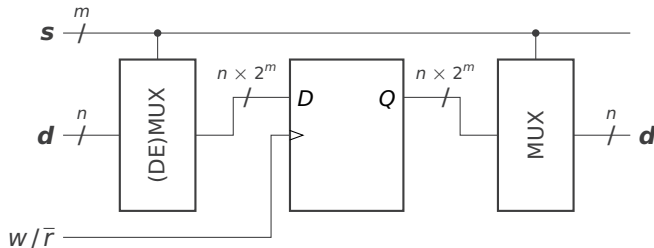
$N =$ höchstwertiges
Ergebnisbit

$Z = 1$: alle Bits im
Ergebnis sind 0

$C =$ Wert des hinaus
geschobenen Bits einer
Schiebeoperation

keine Bedeutung

Einfaches Speichermodell



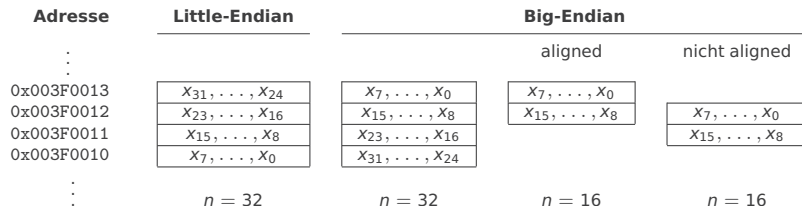
Speicheranbindung des Prozessors über

- Datenbus d der Breite n Bits, oft gleich der Registerbreite
- Adressbus s der Breite m Bits

Beispiele

- Für $n = 8$, $m = 20$: $2^{20} \times 8$ Bit = 1 MB adressierbarer Speicher
- Unser Modell-ARM sei $n = 32$, $m = 26$: $2^{26} \times 8$ Bit = 64 MB Adressraum, mit **Byte**-genauer Adressierung von 32-Bit-Wörtern

“Endianness” und “Alignment”



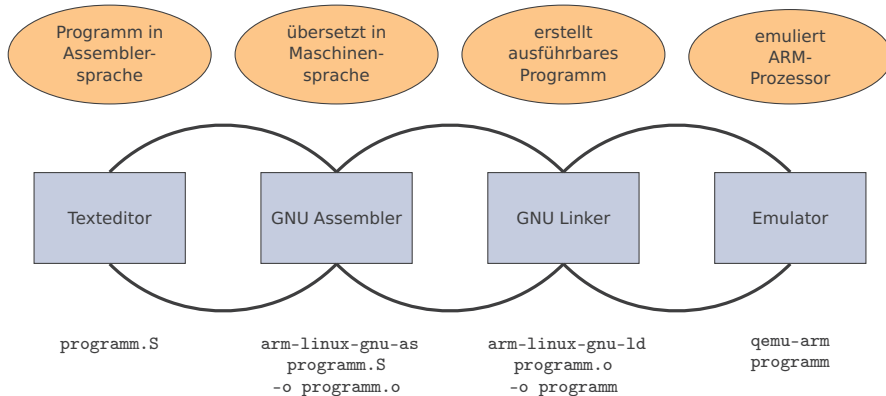
Das Kunstwort **Endianness** bezeichnet die Konvention zur **Reihenfolge** der Ablage von Bytes (8 Bit) eines **Wortes** ($n = k \times 8$ Bit) im Speicher:

- **Little-Endian:** niederwertigstes Byte zuerst, d. h. Wertigkeit nimmt mit zunehmender Adresse zu (z. B. MOS 6502, Intel x86)
- **Big-Endian:** höchstwertiges Byte zuerst, d. h. Wertigkeit nimmt mit zunehmender Adresse ab (z. B. PowerPC, Internet)

→ ARM unterstützt Big- und Little-Endian. Wir verwenden Little-Endian.

ARM-Entwicklungsumgebung

im Rechnerraum des Proseminars und auf dem ZID-GPL-Server



Dokumentation

Online verfügbar zum Nachschlagen und Selbststudium:

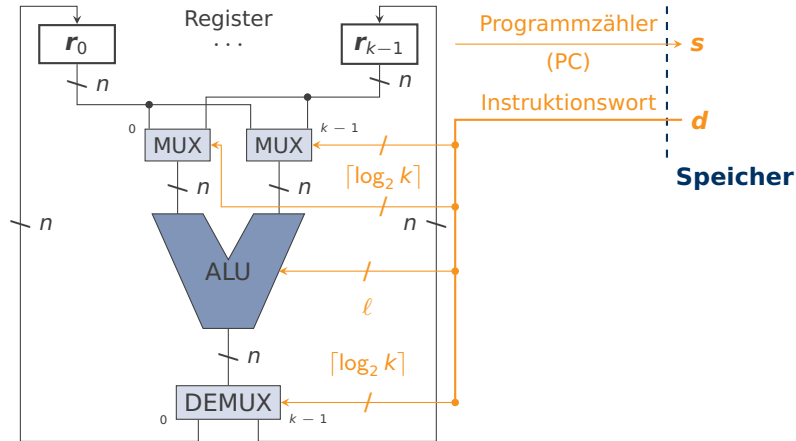
- GNU ARM Compiler Toolchain: Assembler Reference, Version 5.03, ARM Ltd. 2013
- GNU ARM Assembler Quick Reference
<http://www.ic.unicamp.br/~celio/mc404-2014/docs/gnu-arm-directives.pdf>
- ARM and Thumb-2 Instruction Set: Quick Reference Card
<https://www.lri.fr/~de/ARM.pdf>
- Procedure Call Standard for the ARM Architecture
<https://developer.arm.com/documentation/ihi0042/e/>
(Aufrufkonventionen → nächste Woche)
- Pete Cockerell: ARM Assembly Language Programming
<http://www.peter-cockerell.net/aalp/html/frames.html>

(alle Links zuletzt abgerufen am 22. November 2021)

Gliederung heute

1. Von der sequenziellen Logik zum Mikroprozessor
2. ARM-Mikroarchitektur
3. **ARM-Befehlssatz (ohne Speicherzugriff)**
4. Unser erstes Assemblerprogramm

Schaltskizze eines Mikroprozessors



Darstellung ohne Statusregister bzw. Flags, kein Speicherzugriff für Daten

Allgemeines Instruktionsformat

In menschenlesbarem **Assembler-Quelltext**

```
label:                ; Kommentar (mit // bei GNU)
    ADD [ggf. Bedingung] r0, r1, r2 [ggf. Optionen]
```

besteht eine Instruktion aus:

- **Mnemonic** (hier: ADD) für gewählte Instruktion
- **Zielregister** (hier: r0), Symbol **y**
- **Operanden** (hier: r1 und r2), Symbole **a** und **b**

Der ARM-Assembler übersetzt jede Zeile in **ein 32-Bit-Instruktionswort**.

Vom Programmierer wählbare **Labels** bezeichnen die Adresse des nachfolgenden Instruktionsworts und werden bei der Assemblierung aufgelöst (vgl. Binärkodierung beim Zustandsautomat).

Arithmetische Operationen

Mnemonic	Formel	Kommentar
ADD	$y = a + b$	Addition
ADC	$y = a + b + c$	Addition mit Übertrag
SUB	$y = a - b$	Subtraktion
SBC	$y = a - b + c - 1$	Subtraktion mit Übertrag
RSB	$y = b - a$	<i>reverse subtract</i>
RSC	$y = b - a + c - 1$	<i>reverse subtract</i> mit Übertrag
MUL	$y = a \cdot b$	Multiplikation
MLA	$y = (a \cdot b) + x$	<i>multiply accumulate</i>

Bemerkungen zur Multiplikation

- y erhält nur die niederwertigsten 32 Ergebnisbits.
- y und a können nicht das selbe Register sein. (Außerdem ist $r15$ nicht erlaubt.)
- Verwendet intern den Algorithmus von Booth mit Vorzeichen (bis 17 Taktzyklen)

Logische Operationen und Vergleiche

Mnemonic	Formel	Kommentar
AND	$y = a \wedge b$	bitweise AND-Verknüpfung
ORR	$y = a \vee b$	bitweise OR-Verknüpfung
EOR	$y = a \oplus b$	bitweise XOR-Verknüpfung
BIC	$y = a \wedge \bar{b}$	bitweise AND-NOT (<i>bit clear</i>)

Vergleichsoperation

verwerfen Ergebnis der ALU, aktualisieren Flags

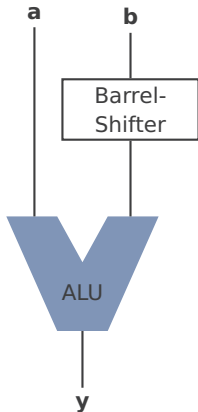
CMP	$a - b$	Vergleich
CMN	$a + b$	Vergleich mit Negation
TST	$a \wedge b$	Test
TEQ	$a \oplus b$	<i>test equivalence</i>

Registerinhalte kopieren

Mnemonic	Formel	Kommentar
MOV	$y = b$	Registerinhalt kopieren
MVN	$y = \overline{b}$	bitweise invertierte Kopie

→ MOV und MVN nutzen den ersten Operanden nicht.

Ansteuerung der ALU



***b* aus Register**

32 Bit

5-Bit-Zahl
(vorzeichenlos)
oder niedrigstes Byte
eines Registers

***b* aus Konstante**

8 Bit

rotiert um 4-Bit Stellen:
{0, 2, ..., 30}
berechnet vom
Assembler

Barrel-Shifter

LSL – logische Linksverschiebung

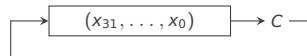
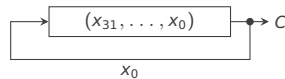
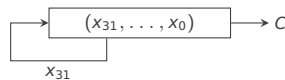
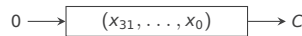
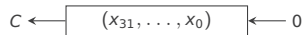
LSR – logische Rechtsverschiebung

ASR – arithmetische Rechtsverschiebung

ROR – Rechtsrotation

RRX – erweiterte Rechtsrotation
(um genau 1 Bit)

ASL – arithmetische Linksverschiebung: Synonym für LSL



Effiziente Multiplikation mit Konstanten

Mit dem Barrel-Shifter können Multiplikationen mit $2^k \pm 1$ **in einem Taktzyklus** (statt 17 bei MUL) berechnet werden.

Beispiele

```
MOV  r2, r0, LSL #2      ; r2 = r0 * 4
ADD  r9, r5, r5, LSL #3  ; r9 = r5 * 9
RSB  r9, r5, r5, LSL #3  ; r9 = r5 * 7
SUB  r10, r9, r8, LSR #4 ; r10 = r9 - r8 : 16
MOV  r12, r4, ROR r3     ; r12 = r4 um r3 Bits
                           nach rechts rotiert
```

Immediate-Werte

(engl. für „unmittelbar“; auch: direkte Werte, Programmkonstanten)

Assembler-Notation mit vorangestellter Raute #: MOV r0, #13

Besonderheit bei ARM

Jedes Instruktionswort ist 32 Bit lang. Damit stehen nur 12 Bit für den zweiten Operanden **b** zur Verfügung.

- 8 Bit davon werden für Konstanten verwendet
- 4 Bit für ROR-Verschiebung in Vielfachen von 2: {0, 2, 4, ..., 30}

Wenn möglich, kümmert sich der Assembler um die Kodierung:

Beispiele

```
MOV  r0, #4096
MOV  r1, #0xffffffff0
```

entsprechen

```
MOV  r0, #0x40, ROR #26
MVN  r1, #15
```

Hörsaalfragen



24 82 94 16

Welche dieser Konstanten können über MOV oder MVN geladen werden?

- a. #508
- b. #510
- c. #1023
- d. #1024

Zugang: <https://arsnova.uibk.ac.at> mit Zugangsschlüssel **24 82 94 16**. Oder scannen Sie den QR-Kode.

Empfohlene Vorgehensweise

Verwendung der LDR-Ladelogik (ARM-spezifisch)

Bei Nutzung des LDR-Mnemonics sucht der Assembler den besten Weg zum Laden einer Konstante:

```
LDR  r0, =0x42
      ; assembliert zu  MOV  r0, #0x42

LDR  r0, =0xffffffff
      ; assembliert zu  MVN  r0, #0x00


LDR  r0, =0x55555555
      ; assembliert zu  LDR  r0, [pc, Offset zu Konstantenpool]
      :
      ; DCD  0x55555555 (Assembler-spezifische Pseudo-Instruktion)
```

LDR vertiefen wir nächste Woche beim Thema Speicherzugriff.

Einfache Sprünge

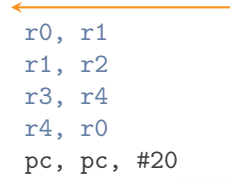
Bei ARM ist der Programmzähler $r15 / pc$ ein Register wie jedes andere.

```
ADD pc, pc, #8
MOV r0, r1
MOV r2, r3
; hier geht's weiter
```

An orange arrow starts at the beginning of the third instruction 'MOV r2, r3' and points back to the beginning of the first instruction 'ADD pc, pc, #8', indicating a jump.

loop:

```
MOV r0, r1
MOV r1, r2
MOV r3, r4
MOV r4, r0
SUB pc, pc, #20
```

An orange arrow starts at the end of the 'SUB pc, pc, #20' instruction and points back to the beginning of the 'MOV r0, r1' instruction, forming a loop.

„Weite“ Sprünge und Rücksprünge

Steuerung des Kontrollflusses

Wer sagt, GO TO sei böse?

Mnemonic	Kommentar
B	Sprung an relative Zieladresse (<i>branch</i>) (Assembler berechnet 26-Bit-Offset zum Label)
BL	wie B, zusätzlich absolute Rücksprungadresse in r14 (lr) speichern (<i>with link</i>) (dient zum Aufruf von Unterprogrammen)

Sprünge an absolute Adressen können durch MOV in r15 realisiert werden,
z. B. Rücksprung aus Unterprogramm: MOV r15, r14 oder MOV pc, lr.

→ Alle Instruktionsworte müssen im Speicher “aligned” sein.

Bedingte Ausführung von Instruktionen

Besonderheit des ARM-Instruktionssatzes

Alle Instruktionen haben ein 4-Bit-Feld, das Bedingungen angibt, unter denen die Instruktion ausgeführt wird.

- Viele Architekturen erlauben dies nur für Sprünge (engl. *branches*).
- Bei ARM kommt diese Logik für jede Instruktion zum Einsatz.
- Nicht ausgeführte Instruktionen benötigen einen Taktzyklus.
- Deutliche Ersparnis gegenüber Verzweigungen, welche die Pipeline blockieren (3 Taktzyklen zum Füllen)
- **Assembler-Konvention:** Bedingung wird als Suffix an das Mnemonic angehängt

Einschränkung: Gilt nicht im Thumb-Modus (nicht Stoff dieser Lehrveranstaltung)

Bedingungen I

(engl. *conditions*)

Kodierung	Suffixe	Flags	Bedeutung
0000	EQ	Z	gleich (<i>equal</i>)
0001	NE	\bar{Z}	ungleich (<i>not equal</i>)
0010	HS CS	C	vorzeichenlos \geq (<i>higher or same</i>)
0011	LO CC	\bar{C}	vorzeichenlos $<$ (<i>lower</i>)
0100	MI	N	negativ (<i>minus</i>)
0101	PL	\bar{N}	positiv (<i>plus</i>)
0110	VS	V	Überlauf (<i>overflow set</i>)
0111	VC	\bar{V}	kein Überlauf (<i>overflow clear</i>)

Bedingungen II

(engl. *conditions*)

Kodierung	Suffix	Flags	Bedeutung
1000	HI	$C \cdot \bar{Z}$	vorzeichenlos $>$ (<i>higher</i>)
1001	LS	$\bar{C} + Z$	vorzeichenlos \leq (<i>lower or same</i>)
1010	GE	$NV + \bar{N}\bar{V}$	\geq mit Vorzeichen (<i>greater or equal</i>)
1011	LT	$N\bar{V} + \bar{N}V$	$<$ mit Vorzeichen (<i>less than</i>)
1100	GT	$\bar{Z}NV + \bar{Z}\bar{N}\bar{V}$	$>$ mit Vorzeichen (<i>greater than</i>)
1101	LE	$N\bar{V} + Z + \bar{N}V$	\leq mit Vorzeichen (<i>less or equal</i>)
1110	AL	1	ohne Bedingung (<i>always</i>)
1111	NV	0	reserviert (<i>never</i>)

Anwendung bedingter Instruktionen

Konventionell

```
CMP  r3, #7
BEQ  skip
ADD  r0, r1, r2
skip: ...
```

ARM-typisch

```
CMP    r3, #7
ADDNE  r0, r1, r2
...
```

Konsequenz: Für jede Instruktion wird festgelegt, ob sie Flags setzt (Suffix: **S**). Bedingungen bleiben bei Bedarf über mehrere Instruktionen erhalten.

Schleife

```
loop: ...
      SUBS  r1, r1, #1
      BNE  loop
```

Ausnahme: CMP braucht kein S.

Systemaufrufe

„Vorteil von Assembler: Man kann alles machen.“

„Nachteil von Assembler: Man muss alles machen.“

In vielen Fällen stellt das **Betriebssystem** grundlegende Funktionen bereit.

Die Schnittstelle ist abhängig von Architektur und Betriebssystem.

- **ARM** nutzt die Instruktion SWI (*software interrupt*) zum Aufruf von Funktionen im privilegierten Modus (SVC).
- **Linux** definiert, welche Funktion abhängig von den Werten in den Registern $r0, \dots, r7$ ausgeführt wird.

Beispiel: $r0=0, r7=1$ zum geordneten Beenden des Programms.

- Diese Schnittstelle steht auch im ARM-Emulator zur Verfügung.

<http://thinkingeek.com/2014/05/24/arm-assembler-raspberry-pi-chapter-19/>

Gliederung heute

1. Von der sequenziellen Logik zum Mikroprozessor
2. ARM-Mikroarchitektur
3. ARM-Befehlssatz (ohne Speicherzugriff)
4. **Unser erstes Assemblerprogramm**

Unser erstes Assemblerprogramm

Hello Innsbruck!

```
.data
msg:
.ascii    "Hello Innsbruck!\n"                0000 48 65 6C 6C 6F 20 49 6E
                                                0008 6E 73 62 72 75 63 6B 21
                                                0010 0A

        len = . - msg

.text
.align
.global  _start
_start:
/* write syscall */
    MOV     r0, #1                            0014 E3A00001
    LDR     r1, =msg                          0018 E59F1016
    LDR     r2, =len                          001c E3A02012
    MOV     r7, #4                            0020 E3A07004
    SWI     #0                                0024 EF000000

/* exit syscall */
    MOV     r0, #0                            0028 E3A00000
    MOV     r7, #1                            002c E3A07001
    SWI     #0                                0030 EF000000
```


Syllabus – Wintersemester 2021/22

06.10.21	1. Einführung
13.10.21	2. Kombinatorische Logik I
20.10.21	3. Kombinatorische Logik II
27.10.21	4. Sequenzielle Logik I
03.11.21	5. Sequenzielle Logik II
10.11.21	6. Arithmetik I
17.11.21	7. Arithmetik II
24.11.21	8. Befehlssatzarchitektur (ARM) I
01.12.21	9. Befehlssatzarchitektur (ARM) II
15.12.21	10. Ein-/Ausgabe
12.01.22	11. Prozessorarchitekturen
19.01.22	12. Speicher
26.01.22	13. Leistung
02.02.22	Klausur (1. Termin)