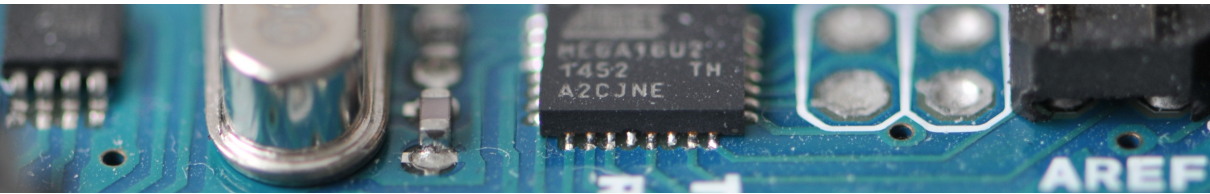
universität
innsbruck



Rechnerarchitektur

Befehlssatzarchitektur I

Univ.-Prof. Dr.-Ing. Rainer Böhme

Wintersemester 2020/21 · 25. November 2020

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 ARMv8 (64 Bit) von 2013 (erstmals im iPhone 5s).
- ARMv8 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)

SecurCore für Sicherheitsanwendungen (Geldautomaten)

In jeder Familie gibt es Produkte, die verschiede 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.

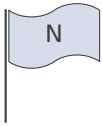
Bei ARM stehen im **User-Modus** 16 Register mit je 32 Bit zur Verfügung:

`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)

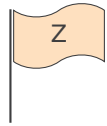
Die Verbindung mit dem (über den Systembus angebundenen) **Arbeitsspeicher** erweitert den Zustandsraum erheblich.

Flags

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

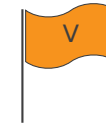


negative



zero

carry



overflow

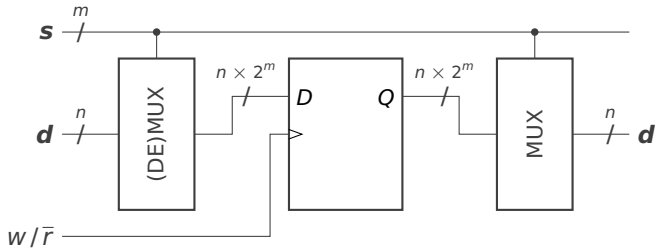
Arithmetische Operationen

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



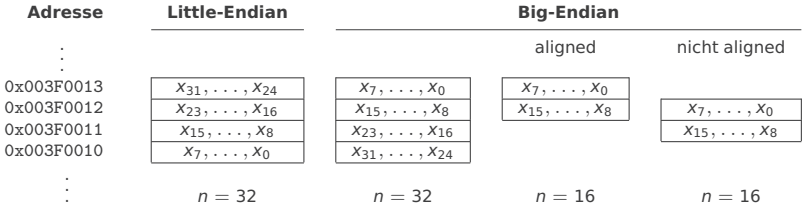
Speicheranbindung des Prozessors über

- 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”



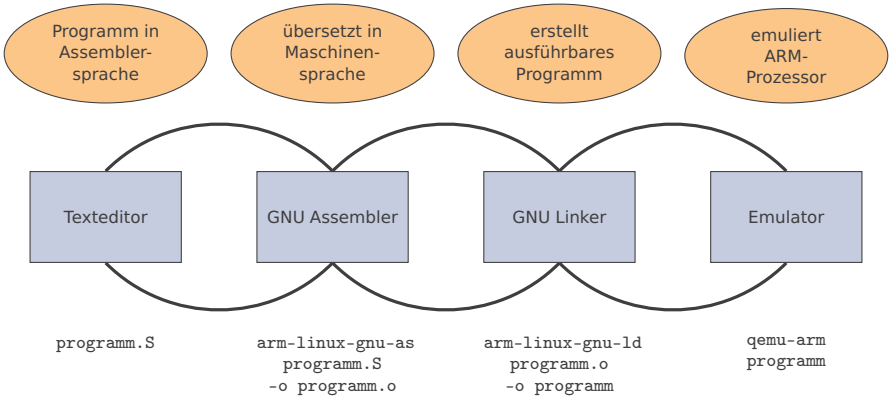
Das Kunstwort **Endianness** bezeichnet die Konvention zur **Reihenfolge** der Ablage von Bytes (8 Bit) eines **Wortes** ($n = k \times 8$ Bit) im Speicher:

- **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:

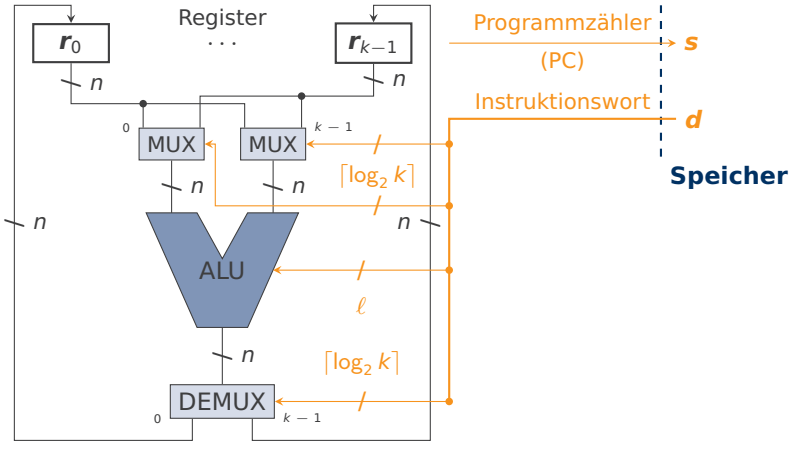
- GNU ARM Compiler Toolchain: Assembler Reference, Version 5.03, ARM Ltd. 2013
- GNU ARM Assembler Quick Reference
  http://www.ic.unicamp.br/~celio/mc404-2014/docs/gnu-arm-directives.pdf
- ARM and Thumb-2 Instruction Set: Quick Reference Card
  https://www.lri.fr/~de/ARM.pdf
- Procedure Call Standard 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 23. November 2020)

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`	$\boldsymbol{y} = \boldsymbol{a} + \boldsymbol{b}$	Addition
`ADC`	$\boldsymbol{y} = \boldsymbol{a} + \boldsymbol{b} + c$	Addition mit Übertrag
`SUB`	$\boldsymbol{y} = \boldsymbol{a} - \boldsymbol{b}$	Subtraktion
`SBC`	$\boldsymbol{y} = \boldsymbol{a} - \boldsymbol{b} + c - 1$	Subtraktion mit Übertrag
`RSB`	$\boldsymbol{y} = \boldsymbol{b} - \boldsymbol{a}$	*reverse subtract*
`RSC`	$\boldsymbol{y} = \boldsymbol{b} - \boldsymbol{a} + c - 1$	*reverse subtract* mit Übertrag
`MUL`	$\boldsymbol{y} = \boldsymbol{a} \cdot \boldsymbol{b}$	Multiplikation
`MLA`	$\boldsymbol{y} = (\boldsymbol{a} \cdot \boldsymbol{b}) + \boldsymbol{x}$	*multiply accumulate*

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`	$\boldsymbol{y} = \boldsymbol{a} \wedge \boldsymbol{b}$	bitwise AND-Verknüpfung
`ORR`	$\boldsymbol{y} = \boldsymbol{a} \vee \boldsymbol{b}$	bitwise OR-Verknüpfung
`EOR`	$\boldsymbol{y} = \boldsymbol{a} \oplus \boldsymbol{b}$	bitwise XOR-Verknüpfung
`BIC`	$\boldsymbol{y} = \boldsymbol{a} \wedge \overline{\boldsymbol{b}}$	bitwise AND-NOT (*bit clear*)

Vergleichsoperation

verwerfen Ergebnis der ALU, aktualisieren Flags

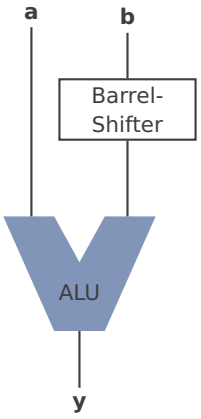
`CMP`	$\boldsymbol{a} - \boldsymbol{b}$	Vergleich
`CMN`	$\boldsymbol{a} + \boldsymbol{b}$	Vergleich mit Negation
`TST`	$\boldsymbol{a} \wedge \boldsymbol{b}$	Test
`TEQ`	$\boldsymbol{a} \oplus \boldsymbol{b}$	*test equivalence*

Registerinhalte kopieren

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

→ `MOV` und `MVN` nutzen den ersten Operanden nicht.

Ansteuerung der ALU



***b* aus Register**	***b* aus Konstante**
32 Bit	**8 Bit**
5-Bit-Zahl (vorzeichenlos)	rotiert um 4-Bit Stellen: $\{0, 2, \ldots, 30\}$
oder niedrigstes Byte eines Registers	berechnet vom Assembler

Barrel-Shifter

`LSL` – logische Linksverschiebung

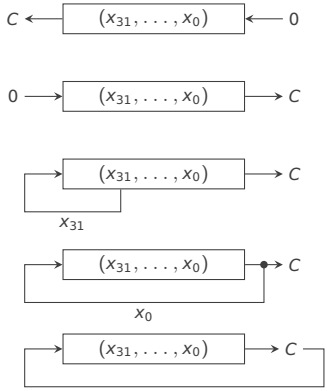
`LSR` – logische Rechtsverschiebung

`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: $\{0, 2, 4, \ldots, 30\}$

Wenn möglich, kümmert sich der Assembler um die Kodierung:

Beispiele

```
MOV  r0, #4096
MOV  r1, #0xfffffff0
```

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)

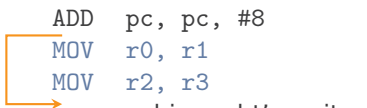
Bei Nutzung des `LDR`-Mnemonics sucht der Assembler den besten Weg zum Laden einer Konstante:

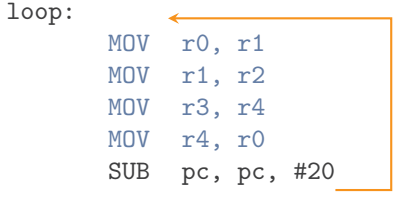
```
LDR   r0, =0x42
      ; assembliert zu  MOV r0, #0x42

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.





„Weite“ Sprünge und Rücksprünge

Steuerung des Kontrollflusses

Wer sagt, GO TO sei böse ?

Mnemonic	Kommentar
`B`	Sprung an **relative** Zieladresse (*branch*) (Assembler berechnet 26-Bit-Offset zum Label)
`BL`	wie `B`, zusätzlich **absolute** Rücksprungadresse in `r14` (`lr`) speichern (*with link*) (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 (*equal*)
0001	**NE**	$\overline{Z}$	ungleich (*not equal*)
0010	**HS CS**	C	vorzeichenlos $\geq$ (*higher or same*)
0011	**LO CC**	$\overline{C}$	vorzeichenlos $<$ (*lower*)
0100	**MI**	N	negativ (*minus*)
0101	**PL**	$\overline{N}$	positiv (*plus*)
0110	**VS**	V	Überlauf (*overflow set*)
0111	**VC**	$\overline{V}$	kein Überlauf (*overflow clear*)

Bedingungen II

(engl. *conditions*)

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

Anwendung bedingter Instruktionen

Konventionell

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

ARM-typisch

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

Konsequent: 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`, ..., `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/

Kodierung der Instruktionswörter

Jeder ARM-Assemblerbefehl wird nach diesem Schema in genau ein 32-Bit-Instruktionswort kodiert:

3 1	3 0	2 9	2 8	2 7	2 6	2 5	2 4	2 3	2 2	2 1	2 0	1 9	1 8	1 7	1 6	1 5	1 4	1 3	1 2	1 1	1 0	0 9	0 8	0 7	0 6	0 5	0 4	0 3	0 2	0 1	0 0	**Befehlstyp**
Bedingung				0	0	I	Opcode				S	Rn				Rd				2. Operand												Data Processing
Bedingung				0	0	0	0	0	0	A	S	Rd				Rn				Rs				1	0	0	1	Rm				Multiply
Bedingung				0	1	I	P	U	B	w	L	Rn				Rd				Offset												Single Data Transfer
Bedingung				1	0	0	P	U	B	w	L	Rn				Registerliste																Block Data Transfer
Bedingung				0	0	0	P	U	1	w	L	Rn				Rd				Offset 1				1	S	H	1	Offset 2				Halfword Trans Imm
Bedingung				0	0	0	P	U	0	w	L	Rn				Rd				0	0	0	0	1	S	H	1	Rm				Halfword Trans Reg
Bedingung				1	0	1	L	relative Zieladresse																								Branch
Bedingung				0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	0	0	0	1	S	H	1	Rn				Branch Exchange
Bedingung				1	1	1	1	SWI-Nummer (vom Prozessor ignoriert)																								Software Interrupt

Einige ARM-Prozessoren unterstützen zusätzlich eine kompaktere Kodierung, die 16- und 32-Bit-Worte mischt. Dieser **Thumb**- und **Thumb-2**-Kode ist meist kürzer, aber langsamer (und nicht Stoff dieser Lehrveranstaltung).

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
          len = . - msg                           0010 0A
.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 2020/21

07.10.20	1.	Einführung
14.10.20	2.	Kombinatorische Logik I
21.10.20	3.	Kombinatorische Logik II
28.10.20	4.	Sequenzielle Logik I
04.11.20	5.	Sequenzielle Logik II
11.11.20	6.	Arithmetik I
18.11.20	7.	Arithmetik II
25.11.20	8.	Befehlssatzarchitektur (ARM) I
02.12.20	9.	Befehlssatzarchitektur (ARM) II
09.12.20	10.	Prozessorarchitekturen
16.12.20	11.	Ein-/Ausgabe
13.01.21	12.	Speicher
20.01.21	13.	Leistung
27.01.21		**Klausur (1. Termin)**

danach →

inday students virtual edition