

ARM architecture (armarch)

Armarch - architecture I/II

- Architecture= Sicht des Programmierers auf den Computer
 - =instructions (Sprache) und operand locations (register und memory)
- Die **Architecture definiert nicht die Implementierung** (in der unterliegenden Hardware (siehe entsprechendes Kapitel)
- Erster Schritt zum Lernen einer architecture ist immer die Sprache (instructions)

Armach - architecture II/II

- operanden kommen vom memory, von register oder der instruction selbst
- instructions sind in 0 und 1 codiert - der sogenannten machine language
 - Menschen benutzen Buchstaben um Ihre Sprache zu kodieren, Computer nutzen Binärzahlen
- Jede instruction der ARM architecture hat 32bit

Armarch - motivation I/II

- Die ARM architecture wurde in den 1980igern von der Acorn Computer Group entwickelt
- Acorn hatte ein spin-off namens Advanced RISC Machines Ltd - heute als ARM bekannt
- Über 10 Mrd. ARM Prozessoren werden jedes Jahr verkauft
- Die architetcure wird von Flippern über Mobiltelefonen und tablets zu Robotern und server racks verwendet
- **ARM** produziert und verkauft jedoch keine Prozessoren selbst sondern **lediglich Lizenzen**
- **Andere Hersteller** (z.B. Samsung, Apple, Qualcomm) **bauen** ARM Prozessoren **mit gekaufter ARM** microarchitecture oder mit eigenen unter ARM **Lizenz** entwickelter microrachitecture

Armarch - motivation II/II

- Warum ARM?
 - sehr weit verbreitet
 - (relativ) klare architecture mit wenigen Eigenarten
 - ARM setzt die 4 Grundprinzipien der RISC architecture gut um
 - (1) regularity supports simplicity, (2) make the common case fast, (3) smaller is faster and (4) good design demands good compromises

Armarch - as(se)m(bly language) - instructions - add

- Assembly language= vom Menschen lesbare Repräsentation der machine language
- C-Code
 $a = b + c;$
- ASM-Code
`ADD a, b, c`
- Der erste Teil ist der mnemonic und bestimmt die eigentliche operation
- Die operation wird mit den source operands b und c durchgeführt und dann in den destination operand a geschrieben.

Armach - asm - instructions - sub

- C-Code

`a = b - c;`

- ASM-Code

`SUB a, b, c`

- Ähnlich zu add -> Design Principle (1): Regularity supports simplicity
 - Instructions mit der gleichen Anzahl von operands sind leichter in Hardware verarbeiten und zu kodieren

Armach - asm - comments

- Im Gegensatz zu Hochsprachen wie C gibt es in ARM Assembly nur single-line comments

- Diese beginnen mit „@“

- C-Code

```
a = b + c - d; // line com.  
/* multiple-line  
comment */
```

- ASM-Code

```
ADD t, b, c @ t = b + c  
SUB a, t, d @ a = t - d
```

- Dieser ASM-Beispielcode braucht zwei assembly instructions und eine temporäre Variable
- Die Verwendung von mehreren instructions um eine komplexere instruction zu erhalten ist üblich -> Design Principle (2): Make the common case fast
 - Der ARM instruction set optimiert den normalen Fall auf Geschwindigkeit, indem er nur einfache allgemeinverwendete instructions verwendet
- (Zur Erinnerung) ARM ist ein reduced instruction set computer (RISC)

Armach - asm - operands

- Eine instruction arbeitet mit operands (vgl. erstes Beispiel arbeitet mit den variablen a, b, c)
- Computer arbeiten aber mit 1 und 0 und nicht mit Variablennamen
- Operanden können in registers oder memory oder als constant in der instruction selbst gespeichert werden
 - Der Zugriff auf **operands in registers** oder in **constants** sind **schnell**, aber umfassen nur eine **geringe Datenmenge**
 - Auf **zusätzliche Daten** muss über den **langsameren memory** **zugegriffen** werden
- Die ARM architecture nutzt 16 registers (=register set oder register file)
 - Je weniger registers desto schneller ist der Zugriff -> Design Principle (3): Smaller is faster

constant = immediate,
also direkt verfügbar

Armach - asm - operands - register

- C-Code

`a = b + c;`

- ASM-Code

`@ R0 = a, R1 = b, R2 = c`

`ADD R0, R1, R2 @ a = b + c`

- Die ARM register werden durch die Namen Rx, also R0, R1 ... R15 angesprochen
- Die Variablen a, b, c in R0, R1, R2 platziert
- Die Instruction ADD addiert R1(b) zu R2(c) und speichert das Ergebnis in R0(a)

Armach - asm - operands - register - temporary register (B)

- C-Code

`a = b + c - d;`

- ASM-Code

`@ R0 = a, R1 = b, R2 = c, R3 = d; R4 = t`

`ADD R4, R1, R2 @ t = b + c`

`SUB R0, R4, R3 @ a = t - d`

- R4 wird als temporäres register verwendet, dass das Zwischenergebnis b+c speichert

Armach - asm - operands - register - Übung

- Übersetzen Sie folgenden C-Code in ARM assembly.
- Annahmen: Variablen a-c befinden sich in R0-R2 und f-j in R3-R7
- C-Code:
 - a= b-c;
 - f= (g+h) - (i+j) ;
- ASM-Code (Solution)

Armarch - asm - operands - register - register set

| Name | Use |
|----------|--|
| R0 | Argument / return value / temporary variable |
| R1–R3 | Argument / temporary variables |
| R4–R11 | Saved variables |
| R12 | Temporary variable |
| R13 (SP) | Stack Pointer |
| R14 (LR) | Link Register |
| R15 (PC) | Program Counter |

Armach - asm - operands - immediates I/II

- **Immediates= constants** (Der Wert ist sofort aus der instruction verfügbar (immediately) und muss nicht aus einem register oder memory geladen werden)
- **Immediates** werden durch **vornagestelltes „#“** gekennzeichnet
- Oft werden immediates auch hexadezimal angegeben z.B. #0xAF
- **Immediates** können **8-12bit groß** sein (vgl. Abschnitt instruction encoding)

▪ C-Code

```
a = a + 4;  
b = a - 12;
```

▪ ASM-Code

```
@ R7 = a, R8 = b  
ADD R7, R7, #4 @ a = a + 4  
SUB R8, R7, #0xC @ b = a - 12
```

=12

Armach - asm - operands - immediates II/II

- Die move instruction (MOV) wird verwendet, um register in register zu kopieren oder register mit immediates zu initialisieren

- C-Code

```
i = 0;  
x = 4080;
```

- ASM-Code

```
@ R4 = i, R5 = x  
MOV R4, #0 @ i = 0  
MOV R5, #0xFF0 @ x = 4080
```

Armach - asm - operands - memory

- Daten im memory müssen zuerst in ein register geladen werden, bevor die Daten verarbeitet werden können
- Memory ist organisiert als ein array von data words
 - Die ARM architecture verwendet 32bit addresses und 32bit data words
- ARM arbeitet mit byte-addressible memory (Jedes Byte hat eine eigene einzigartige address)

Armach - asm - operands - memory - ldr

- ARM stellt die LDR instruction (load register) zur Verfügung, um ein data word aus dem memory in ein register zu laden

- C-Code
- ASM-Code

```
a = mem[2]; @ R7 = a
```

```
MOV R5, #0 @ base address = 0
```

```
LDR R7, [R5, #8] @ R7 <= data at memory  
@ address (R5 + 8)
```

- ARM bietet mehrere memory addressing modes an (vgl. Abschnitt später)

da 32bit - zweiter
Wert im Register
= 8 (binär: 1000)
NACHLESEN
UNKLAR

Armach - asm - operands - memory - str

- ARM verwendet die STR instruction (store register), um ein data word aus einem register in den memory zu speichern

- C-Code


```
mem[5] = 42;
```

- ASM-Code

```
MOV R1, #0 @ base address = 0
```

```
MOV R9, #42
```

```
STR R9, [R1, #0x14] @ value stored at memory  
@ address (R1 + 20) = 42
```



0x14 = 0001 0100 = 20

1 register = 4 bit

Armarch - asm - operands - sum instr (B)

▪ Data-processing instructions

| cmd | Name | Description | Operation |
|---|------------------|-------------|---------------------------|
| 0100 | ADD Rd, Rn, Src2 | Add | $Rd \leftarrow Rn + Src2$ |
| 0010 | SUB Rd, Rn, Src2 | Subtract | $Rd \leftarrow Rn - Src2$ |
| 1101 | Shifts: | | |
| $I = 1$ OR (instr _{11:4} = 0) | MOV Rd, Src2 | Move | $Rd \leftarrow Src2$ |

damit umgehen
lernen
-> am besten
ausdrucken.
Siehe GRIPS gibt es
Vorlage

▪ Memory instructions

| op | B | op2 | L | Name | Description | Operation |
|----|---|-----|---|---------------------|----------------|--------------------------|
| 01 | 0 | N/A | 0 | STR Rd, [Rn, ±Src2] | Store Register | $Mem[Adr] \leftarrow Rd$ |
| 01 | 0 | N/A | 1 | LDR Rd, [Rn, ±Src2] | Load Register | $Rd \leftarrow Mem[Adr]$ |

asm_instruction_sum_arm_
v05onlymain

Armach - prog(ramming) - data proc(essing) instr(uctions) - log(ical instr(uction)s I/II

- Die wichtigsten logical operations sind AND, **ORR**(OR), **EOR**(XOR) und **MVN**(NOT)
- Jede dieser instructions arbeitet mit zwei source operands und einem destination operand
- Source operand 1 (Src1) ist immer ein register, Src2 ist entweder ein immediate oder ein anderes register

Armarch - prog - data proc instr - logical instr II/II (Q)

Source registers

| | | | | |
|----|-----------|-----------|-----------|-----------|
| R1 | 0100 0110 | 1010 0001 | 1111 0001 | 1011 0111 |
| R2 | 1111 1111 | 1111 1111 | 0000 0000 | 0000 0000 |

Assembly code

Result

| | | | | | |
|----------------|----|-----------|-----------|-----------|-----------|
| AND R3, R1, R2 | R3 | 0100 0110 | 1010 0001 | 0000 0000 | 0000 0000 |
| ORR R4, R1, R2 | R4 | 1111 1111 | 1111 1111 | 1111 0001 | 1011 0111 |
| EOR R5, R1, R2 | R5 | 1011 1001 | 0101 1110 | 1111 0001 | 1011 0111 |
| BIC R6, R1, R2 | R6 | 0000 0000 | 0000 0000 | 1111 0001 | 1011 0111 |
| MVN R7, R2 | R7 | 0000 0000 | 0000 0000 | 1111 1111 | 1111 1111 |

- **MVN**=MoVe and Not ->bitweises NOT auf src2
- **BIC**= Bit Clear -> **ungewünschte Teile auf Null setzen**
 - Berechnet wird bei BIC R6, R1, R2 (R1 AND (NOT R2))

Armach - prog - data proc instr - sum logical instr (B)

| cmd | Name | Description | Operation |
|------|------------------|---------------|---------------------------------|
| 0000 | AND Rd, Rn, Src2 | Bitwise AND | $Rd \leftarrow Rn \& Src2$ |
| 0001 | EOR Rd, Rn, Src2 | Bitwise XOR | $Rd \leftarrow Rn \wedge Src2$ |
| 1100 | ORR Rd, Rn, Src2 | Bitwise OR | $Rd \leftarrow Rn Src2$ |
| 1110 | BIC Rd, Rn, Src2 | Bitwise Clear | $Rd \leftarrow Rn \& \sim Src2$ |
| 1111 | MVN Rd, Rn, Src2 | Bitwise NOT | $Rd \leftarrow \sim Rn$ |

Armach - prog - data proc instr - shift instr I/II

- Shift instructions= schieben den Wert in einem register nach links oder rechts und lassen (überschüssige) bits am Ende weg
 - LSL (logical shift left, LSR (logical shift right), ASR (arithmetical shift right)
 - Die “logischen” Varianten schieben Nullen nach
 - ASR wird für 2er-Komplement verwendet und schiebt bei MSB=1 eine 1 nach ansonst 0
 - Es gibt kein ASL -> LSL wird verwendet
- Rotate instructions werden typischerweise zu den shift instructions gezählt; Was an der einen Seite rausgeschoben wird, wird an der anderen wieder reingeschoben
 - ROR (rotate right)
 - Es gibt kein ROL, da jedes „Linksrotieren“ auch durch ein entsprechendes „Rechtsrotieren“ ausgedrückt werden kann

Armach - prog - data proc instr - shift instr II/II (Q)

- Shift instr mit immediates
- Shift instr mit register

| Source register | | | | |
|-----------------|-----------|-----------|-----------|-----------|
| R5 | 1111 1111 | 0001 1100 | 0001 0000 | 1110 0111 |

| Assembly Code | | Result | | | |
|-----------------|----|-----------|-----------|-----------|-----------|
| LSL R0, R5, #7 | R0 | 1000 1110 | 0000 1000 | 0111 0011 | 1000 0000 |
| LSR R1, R5, #17 | R1 | 0000 0000 | 0000 0000 | 0111 1111 | 1000 1110 |
| ASR R2, R5, #3 | R2 | 1111 1111 | 1110 0011 | 1000 0010 | 0001 1100 |
| ROR R3, R5, #21 | R3 | 1110 0000 | 1000 0111 | 0011 1111 | 1111 1000 |

| Source registers | | | | |
|------------------|-----------|-----------|-----------|-----------|
| R8 | 0000 1000 | 0001 1100 | 0001 0110 | 1110 0111 |
| R6 | 0000 0000 | 0000 0000 | 0000 0000 | 0001 0100 |

| Assembly code | | Result | | | |
|----------------|----|-----------|-----------|-----------|-----------|
| LSL R4, R8, R6 | R4 | 0110 1110 | 0111 0000 | 0000 0000 | 0000 0000 |
| ROR R5, R8, R6 | R5 | 1100 0001 | 0110 1110 | 0111 0000 | 1000 0001 |

Armarch - prog - data proc instr - sum shift instr (B)

| cmd | Name | Description | Operation |
|--|-----------------------|------------------------|--------------------------------------|
| <i>I</i> = 0 AND (<i>sh</i> = 00; instr _{11:4} ≠ 0) | LSL Rd, Rm, Rs/shamt5 | Logical Shift Left | $Rd \leftarrow Rm \ll Src2$ |
| <i>I</i> = 0 AND (<i>sh</i> = 01) | LSR Rd, Rm, Rs/shamt5 | Logical Shift Right | $Rd \leftarrow Rm \gg Src2$ |
| <i>I</i> = 0 AND (<i>sh</i> = 10) | ASR Rd, Rm, Rs/shamt5 | Arithmetic Shift Right | $Rd \leftarrow Rm \ggg Src2$ |
| <i>I</i> = 0 AND (<i>sh</i> = 11; instr _{11:7, 4} = 0) | RRX Rd, Rm, Rs/shamt5 | Rotate Right Extend | $\{Rd, C\} \leftarrow \{C, Rd\}$ |
| <i>I</i> = 0 AND (<i>sh</i> = 11; instr _{11:7} ≠ 0) | ROR Rd, Rm, Rs/shamt5 | Rotate Right | $Rd \leftarrow Rn \text{ ror } Src2$ |

Armarch - prog - data proc instr - multiply instr

- Generell entstehen bei der Multiplikation von zwei 32bit Zahlen ein 64bit Produkt.
- Die ARM architecture hat multiply instructions die aus zwei 32bit Zahlen ein 32bit Produkt oder ein 64bit Produkt als Ergebnis liefern
- **MUL erzeugt ein 32bit Produkt** und ist **geeignet für kleine Zahlen**, die kein 64bit Ergebnis brauchen
 - Die **oberen 32bit** werden **verworfen** und die **untern 32bit** behalten
 - Beispiel: MUL R1, R2, R3
- **UMUL (unsigned multiply long) und SMULL (signed multiply long) erzeugen ein 64bit Ergebnis** und sind für **große Zahlen** geeignet
 - Die **oberen 32bit** befinden sich dann in **R2** und die **untern 32bit** in **R1**
- Jede dieser operations hat auch noch eine multiply-accumulate Variante (MLA, SMLAL, UMLAL), die das Produkt zu einer Summe dazu addiert

wird weggelassen
da zu kompliziert

Armach - prog - data proc instr - sum multiply instr (reduced) (B)

| cmd | Name | Description | Operation |
|-----|----------------------|---------------------------|---|
| 000 | MUL Rd, Rn, Rm | Multiply | $Rd \leftarrow Rn \times Rm$ (low 32 bits) |
| 100 | UMULL Rd, Rn, Rm, Ra | Unsigned Multiply Long | $\{Rd, Ra\} \leftarrow Rn \times Rm$ (all 64 bits, Rm/Rn unsigned) |
| 110 | SMULL Rd, Rn, Rm, Ra | Signed Multiply Long | $\{Rd, Ra\} \leftarrow Rn \times Rm$ (all 64 bits, Rm/Rn signed) |

Teil 2

Armarch - prog - condition flags - CPSR

- ARM instructions können optional in Abhängigkeit von ihrem (Rechen)ergebnis condition flags im Current Program Status Register (CPSR) setzen



| Flag | Name | Description | wichtig |
|------|----------|---|---------|
| N | Negative | Instruction result is negative, i.e., bit 31 of the result is 1 | wichtig |
| Z | Zero | Instruction result is zero | wichtig |
| C | Carry | Instruction causes a carry out | wichtig |
| V | oVerflow | Instruction causes an overflow | wichtig |

- Der üblichste Weg ist es, diese Flags mit der compare-Funktion (CMP) setzen zu lassen (Testfunktion)
 - Diese Funktion führt im Hintergrund eine Subtraktion durch (src2 -src1)
- Die nachfolgende instruction (typischerweise branch instruction) kann dann den status dieser flags auswerten und eine conditional execution durchführen (bedingte Ausführung)

Armach - prog - condition flags - mnemonics for conditional execution

- Wenn die conditional execution genutzt werden soll muss ein bestimmter condition mnemonic (bestimmter Suffix) an die instruction mnemonic angehängt werden
- Achtung!!! Es wird zwischen unsigned und signed compares unterschieden!!!

| cond | Mnemonic | Name | CondEx |
|------|--------------|-------------------------------------|-----------------------------------|
| 0000 | EQ | Equal | Z |
| 0001 | NE | Not equal | \bar{Z} |
| 0010 | CS/HS | Carry set / unsigned higher or same | C |
| 0011 | CC/LO | Carry clear / unsigned lower | \bar{C} |
| 0100 | MI | Minus / negative | N |
| 0101 | PL | Plus / positive or zero | \bar{N} |
| 0110 | VS | Overflow / overflow set | V |
| 0111 | VC | No overflow / overflow clear | \bar{V} |
| 1000 | HI | Unsigned higher | $\bar{Z}C$ |
| 1001 | LS | Unsigned lower or same | $Z \text{ OR } \bar{C}$ |
| 1010 | GE | Signed greater than or equal | $\bar{N} \oplus \bar{V}$ |
| 1011 | LT | Signed less than | $N \oplus V$ |
| 1100 | GT | Signed greater than | $\bar{Z}(\bar{N} \oplus \bar{V})$ |
| 1101 | LE | Signed less than or equal | $Z \text{ OR } (N \oplus V)$ |
| 1110 | AL (or none) | Always / unconditional | Ignored |

Armach - prog - condition flags - test functions

- Weitere Testfunktionen neben `CMP` sind `CMN` (compare negative, (src1+(-src)), `TST` (test, (AND)) und `TEQ` (Test if equal, (XOR))
- Prinzipiell kann jede data processing instruction als Testfunktion verwendet werden, indem dem instruction mnemonic ein „S“ angehängt wird
 - `N` und `Z` werden von jeder instruction gesetzt
 - `ADDS` und `SUBS` beeinflussen auch `V` und `C`
 - Shift instructions beeinflussen `C`
- Prinzipiell kann jede instruction (nicht nur branch instruction) als conditional execution verwendet werden
 - Allerdings ist dies bei neueren ARM architectures (armarch64 bzw. ARMv8) nicht mehr möglich!

Armach - prog - condition flags - mnemonics for conditional execution - example (Ü)

- **ASM-Code**

```
CMP R2, R3  
ADDEQ R4, R5, #78  
ANDHS R7, R8, R9  
ORRMI R10, R11, R12  
EORLT R12, R7, R10
```

- **CMP setzt die condition flags**

- **ADDEQ** = ADD ausgeführt wenn CMP das Ergebnis EQ (equal) hatte
- **ANDHS** = AND ausgeführt wenn CMP das Ergebnis HS (unsigned higher same) hatte
- ...

Armarch - prog - condition flags - sum test instructions (Q)

| cmd | Name | Description | Operation |
|------------------|------------------|------------------|------------------------------|
| 1000 ($S = 1$) | TST Rd, Rn, Src2 | Test | Set flags based on Rn & Src2 |
| 1001 ($S = 1$) | TEQ Rd, Rn, Src2 | Test Equivalence | Set flags based on Rn ^ Src2 |
| 1010 ($S = 1$) | CMP Rn, Src2 | Compare | Set flags based on Rn - Src2 |
| 1011 ($S = 1$) | CMN Rn, Src2 | Compare Negative | Set flags based on Rn+Src2 |

Armach - prog - branching - motivation

- Der Vorteil eines Computers gegenüber einem Taschenrechner ist, dass er Entscheidungen treffen kann
- Ein Computer führt unterschiedliche Aufgaben in Abhängigkeit von dem input aus
 - Beispiel: If/else, while loops, for loops, ... führen alle Code in Abhängigkeit von einem test aus
- ARM (und die meisten anderen architectures) führen branch instructions aus, um Code-Teile zu überspringen oder wiederholt auszuführen
- Im normalen Programmablauf wird das Programm sequentiell abgearbeitet und der Programm Counter (PC) nach jeder instruction um 4 erhöht
- Bei branch instructions, wird der PC von der branch instruction geändert

Armach - prog - branching - branch types

- ARM hat zwei Typen von branches (manchmal auch jumps genannt)
 - einen einfachen branch (B)
 - einen branch and link (BL)
- BL wird für function calls verwendet
- Die einfachen branches können unconditional (unbedingt) oder conditional (bedingt) sein

Armach - prog - branching - example unconditional branching

- **ASM-Code**

```
ADD R1, R2, #17    @ R1 = R2 + 17
B TARGET           @ branch to TARGET
ORR R1, R1, R3      @ not executed
AND R3, R1, #0xFF   @ not executed
TARGET:
SUB R1, R1, #78     @ R1 = R1 - 78
```

- `B TARGET` sorgt dafür, dass die nächste instruction, die ausgeführt wird, die `SUB` instruction ist, die direkt nach dem label `TARGET` kommt
- Labels bezeichnen instruction locations im Programm (die beim Übersetzen in machine code in instruction addresses übersetzt werden)
- Der GCC compiler benötigt einen Doppelpunkt nach dem Label

Armach - prog - branching - example conditional branching

▪ ASM-Code

```
MOV R0, #4      @ R0 = 4
ADD R1, R0, R0  @ R1 = R0 + R0 = 8
CMP R0, R1      @ set flags based on R0 - R1 = - 4.
                 @ NZCV = 1000

BEQ THERE      @ branch not taken
ORR R1, R1, #1  @ R1 = R1 OR 1 = 9

THERE:
ADD R1, R1, #78 @ R1 = R1 + 78 = 87
```

- Die BEQ instruction hier wird nicht ausgeführt, da $R0 \neq R1$

Armach - prog - branching - conditional statements - if/else - example

- if/else statements= führt einen von zwei code blocks in Abhängigkeit von einer condition aus
 - Wenn die condition im if zutrifft wird der if-Teil ausgeführt, ansonsten der else-Teil

- **C-Code**

```
if (apples ==  
    oranges)  
    f = i + 1;  
else  
    f = f - i;
```

- **ASM-Code**

```
@ R0= apples, R1= oranges, R2= f, R3= i  
CMP R0, R1      @ apples == oranges?  
BNE L1          @ if not equal, skip if block  
ADD R2, R3, #1  @ if block: f = i + 1  
B L2            @ skip else block  
L1:  
    SUB R2, R2, R3 @ else block: f = f - i  
L2:
```

- **!!! Der Assembly-Code testet auf die umgekehrte Bedingung, da er im false-Fall wegspringt!**

Armach - prog - branching - conditional statements - if/else - general structure

- C-Code structure

```
if (test-expr)  
    then-statement  
else  
    else-statement
```

- ASM-Code structure
(described by C-like code)

```
t = test-expr;  
if (!t) goto elsebr;  
    then-statement  
goto done;  
elsebr:  
    else-statement  
done:
```

Armach - prog - branching - loops - while - example

- while-loop= führt wiederholt einen code block aus bis eine condition zutrifft

- C-Code

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

- ASM-Code

```
@ R0= pow, R1= x
MOV R0, #1      @ pow= 1
MOV R1, #0      @ x= 0
B WHILETEST
WHILE:
    LSL R0, R0, #1 @ pow= pow * 2
    ADD R1, R1, #1 @ x= x + 1
WHILETEST:
    CMP R0, #128   @ pow != 128 ?
    BNE WHILE      @ repeat loop
```

Armarch - prog - branching - loops - while - general structure

- C structure

```
while (test-expr)  
    body-statement
```

- ASM structure (described by C-like code)

```
goto test;  
loop:  
    body-statement  
looptest:  
    t = test-expr;  
    if (t)  
        goto loop;
```

Armach - prog - branching - loops - for - example

- for-loop= while-loop mit Schleifenzähler mit Initialisierung, condition check und Schleifenvariablenänderung

- C-Code

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

- ASM-Code

```
@ R0=i, R1=sum  
MOV R1, #0      @ sum=0  
MOV R0, #0      @ i=0 loop cnt init  
B FORLOOPTEST   @ goto looptest first  
FORLOOP:  
    ADD R1, R1, R0 @ sum=sum+i loop body  
    ADD R0, R0, #1 @ i=i+1 loop operation  
FORLOOPTEST:  
    CMP R0, #10    @ i<10? check condition  
    BLT FORLOOP    @ if true -> repeat loop
```

- Schleifen insbesondere for-loops sind besonders nützlich für die Verarbeitung von großen Mengen von gleichartigen Daten, die im memory gespeichert werden (array)
-

Armarch - prog - branching - loops - for - general structure

- C structure

```
for (init-expr; test-expr;  
    update-expr)  
    body-statement
```

- ASM structure (described by C-like code)

```
init-expr;  
goto test;  
loop:  
    body-statement  
    update-expr;  
test:  
    t = test-expr;  
    if (t)  
        goto loop;
```

Armach - prog - memory

- array=
Zusammenfassung/Abfolge von gleichartigen Datenelementen in einem zusammenhängenden Speicherbereich für einfachere Speicherung und einfacheren Zugriff
- Beispiel: `int scores[200]`



- Zugriff erfolgt mit indexed memory operator with scaling
 - Beispiel:
`LDR R3, [R0, R1, LSL #2]`
 - Die Resultierende Adresse für den (Pointer-Zugriff) ist:
 $\text{resultaddr} = R1 * 4 + R0$
 - LSL #2 bewirkt durch das linkschieben eine Multiplikation um 4
 - Anm: Die eckigen Klammern „[]“ stehen für die Pointer-Dereferenzierung beim ARM Assembly!!!

Armach - prog - memory - example array access with for-loop

▪ C-Code

```
int i;  
int scores[200];  
//... array init ...  
  
for(i=0; i<200; i++){  
    scores[i]=  
        scores[i]+10;  
}
```

▪ ASM-Code

```
@ R0= array base address, R1= i  
@ initialization code ...  
  
MOV R1, #0                @ i = 0  
B LOOPTEST  
LOOP:  
    @ R3 = scores[i]  
    LDR R3, [R0, R1, asl #2]  
    @ R3 = scores[i] + 10  
    ADD R3, R3, #10  
    @ scores[i] = scores[i] + 10  
    STR R3, [R0, R1, asl #2]  
    ADD R1, R1, #1          @ i = i + 1  
LOOPTEST:  
    CMP R1, #200           @ i < 200?  
    BLT LOOP               @ repeat loop
```

Armach - prog - memory - further indexing modes

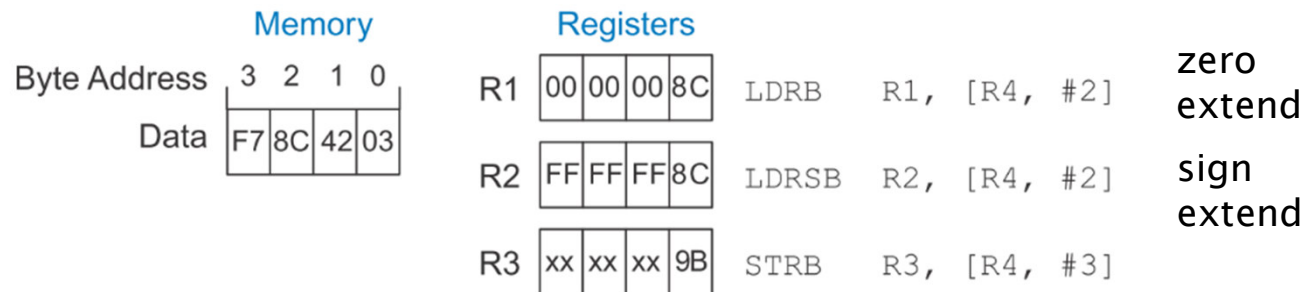
- ARM unterstützt neben der memory indexed with scaling mode noch drei weitere Adressierungsarten (offset, pre-indexed und post-indexed):

| Mode | ARM Assembly | Address | Base Register |
|------------|-------------------|-----------|----------------|
| Offset | LDR R0, [R1, R2] | $R1 + R2$ | Unchanged |
| Pre-index | LDR R0, [R1, R2]! | $R1 + R2$ | $R1 = R1 + R2$ |
| Post-index | LDR R0, [R1], R2 | R1 | $R1 = R1 + R2$ |

- Offset addressing:
 - Sie resultierende address wird aus base register +- offset berechnet
 - Der offset kann ein register sein, oder auch ein immediate (0-4095)
 - Der offset kann auch negativ verwendet werden (z.B.: -R2, #-20)
- Pre-indexed addressing:
 - wie offset addressing, aber die resultierende adress wird VOR dem memory access im base register gespeichert
- Post-indexed addressing:
 - wie offset addressing, aber die resultierende adress wird NACH dem memory access im base register gespeichert

Armarch - prog - memory - ldrb and strb

- Die instructions LDR und STR arbeiten mit 32bit
- Wenn byteweises laden/speichern gebraucht wird gibt es die Befehle LDRB (load register (zero extend) byte), LDRSB (load register sign-extend byte) und STRB (store register byte)



- Daneben gibt es noch die Befehle LDRH, LDRSH und STRH für half words (16bit)

Armach - prog - memory - ldrb and strb - Übung

- Der folgende C-Code konvertiert ein char array von lowercase zu uppercase (Subtraktion von 32); Übersetzen Sie ihn in ASM.

- C-Code

```
// carr[10] decl.  
// and init. earlier  
int i;  
  
for (i=0; i<10; i=i+1){  
    carr[i] = carr[i] - 32;  
}
```

- ASM-Code

Armarch - prog - memory - sum memory instructions (Q)

| op | B | op2 | L | Name | Description | Operation |
|----|-----|-----|---|-----------------------|------------------|-------------------------------|
| 01 | 0 | N/A | 0 | STR Rd, [Rn, ±Src2] | Store Register | Mem[Adr] ← Rd |
| 01 | 0 | N/A | 1 | LDR Rd, [Rn, ±Src2] | Load Register | Rd ← Mem[Adr] |
| 01 | 1 | N/A | 0 | STRB Rd, [Rn, ±Src2] | Store Byte | Mem[Adr] ← Rd _{7:0} |
| 01 | 1 | N/A | 1 | LDRB Rd, [Rn, ±Src2] | Load Byte | Rd ← Mem[Adr] _{7:0} |
| 00 | N/A | 01 | 0 | STRH Rd, [Rn, ±Src2] | Store Halfword | Mem[Adr] ← Rd _{15:0} |
| 00 | N/A | 01 | 1 | LDRH Rd, [Rn, ±Src2] | Load Halfword | Rd ← Mem[Adr] _{15:0} |
| 00 | N/A | 10 | 1 | LDRSB Rd, [Rn, ±Src2] | Load Signed Byte | Rd ← Mem[Adr] _{7:0} |
| 00 | N/A | 11 | 1 | LDRSH Rd, [Rn, ±Src2] | Load Signed Half | Rd ← Mem[Adr] _{15:0} |

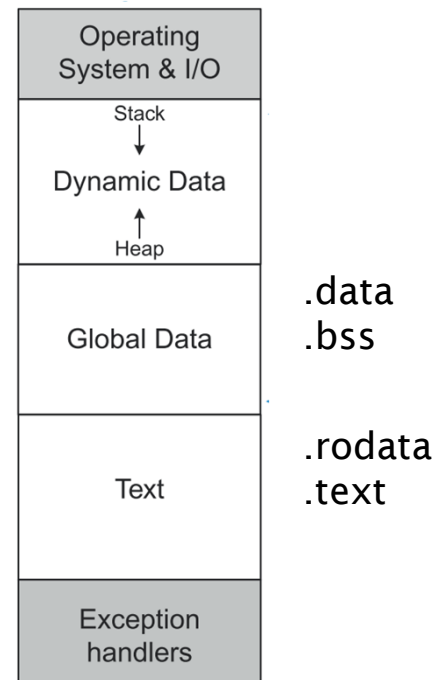
Armarch - prog - memory - defining data elements

- Variablen und Konstanten werden durch spezielle directives definiert

| Directive | Data Type |
|-----------|---|
| .ascii | Text string |
| .asciz | Null-terminated text string |
| .byte | Byte value |
| .double | Double-precision floating-point number |
| .float | Single-precision floating-point number |
| .int | 32-bit integer number |
| .long | 32-bit integer number (same as .int) |
| .octa | 16-byte integer number |
| .quad | 8-byte integer number |
| .short | 16-bit integer number |
| .single | Single-precision floating-point number (same as .float) |

Armach - prog - memory - variables

- Man unterscheidet globale und locale Variablen (vgl. C)
- Globale Variablen werden im Segment bzw. section `.data` definiert (wenn read only -> `.rodata`)
- Lokale Variablen werden auf dem stack angelegt (vgl. Abschnitt function call)



Armach - prog - memory - global initialized variables - example

```
.section .data
height:
    .int 5
length:
    .int 6
dataarr:
    .int 7,8,9 #int array
res:
    .int 0

.section .rodata
fstring:
    .asciz "the result is %d\n"
```

```
.section .text
.globl myfn
myfn:
    ...
    ldr r1, =height
    ldr r2, =length
    add r0, r1, r2
    ldr r3, =res
    str r0, []
    ...
```

- !!! Das „=“ vor dem label wirkt hier als eine Art Adressoperator (vgl. „&“ in C).
- Eigentlich passiert hier aber noch viel mehr im Hintergrund (Details siehe Abschnitt machine code)

Armarch - prog - memory - global uninitialized variables

- Uninitialisierte Daten oder Daten initialisiert mit 0 werden im Segment oder section `.bss` mit der directive `.comm` bzw `.lcomm` definiert
- Syntax: `.comm <symbol>, <length>`

| Directive | Description |
|---------------------|--|
| <code>.comm</code> | Declares a common memory area for data that is not initialized |
| <code>.lcomm</code> | Declares a local common memory area for data that is not initialized |

- Vorteil: im Gegensatz zu `.data` sind `.bss` nicht Teil vom executable file -> code-Größe ist kleiner!

Armach - prog - memory - global uninitialized variables - example

```
.section .bss
    .comm res, 4
```

```
.section .data
height:
    .int 5
length:
    .int 6
```

```
.section .text
.globl myfn
myfn:
    ...
    ldr r1, =height
    ldr r2, =length
    add r0, r1, r2
    ldr r3, =res
    str r0, []
    ...
```

- !!! Das „=“ vor dem label wirkt hier als eine Art Adressopertor (vgl. „&“ in C).
- Eigentlich passiert hier aber noch viel mehr im Hintergrund (Details siehe Abschnitt machine code)