Professur für Prozessleittechnik und Arbeitsgruppe Systemverfahrenstechnik
Leon Urbas

Datenstrukturen

Vorlesung Mikrorechentechnik 1

Winter Semester 2021/2022

Übersicht zeitlicher Ablauf

WiSe 2021/2022

Nr.	Datum	Flipped Classroom	Inhalt
1	11.10.2021	11.10.2021	Organisatorisches, Überblick, Einführungsbeispiel: Vom Programmcode zur TempLightPWM
2	13.10.2021	-	Entwicklungsumgebung und Werkzeugketten
3	18.10.2021	25.10.2021	ASM: Compiler, Assembler, Befehlsarchitektur ARM
4	20.10.2020	25.10.2021	Übung: Algorithmen in Assembler umsetzen (Fibonacci)
5	25.10.2021	01.11.2021	ASM: Datenstrukturen
6	03.11.2021	08.11.2021	ASM: Ansteuerung von Peripherie, GPIO Funktionssammlung
7	08.11.2021	15.11.2021	ASM: Peripherie: SPI, ADC Funktionssammlung
8	10.11.2021	15.11.2021	Übung: ASM: Memory Mapped Peripherals (Temperaturalarm)
9	15.11.2021	22.11.2021	Einführung in die Programmiersprache C
			Buß- und Bettag
10	22.11.2021	29.11.2021	Datenstrukturen und Zeiger in C
11	24.11.2021	29.11.2021	Dynamische Datenstrukturen und Objektorientierung in C
12	29.11.2021	06.12.2021	Memory Mapped Peripherals mit C
13	01.12.2021	06.12.2021	Übung: Memory Mapped Peripherals (Temperaturalarm) in C
14	06.12.2021	-	Tooling für komplexe Softwareentwicklung in Teams

Rückblende Tutorium Assembler

Vorgehen beim **Entwurf von Programmen**

- Programmfluss/Funktion als Aktivitätsdiagramm **planen**
  - Welche temporären und permanenten Variablen werden benutzt?
  - Wo werden diese Variablen günstig abgelegt (Register, Stack, Memory)?
- Testfunktionen für **Test-Driven-Design** erstellen
  - Verhalten bei Extremwerten (e.g. Minima, Maxima)
  - Verhalten bei speziellen Eingaben (ungültige Zahlenbereiche,)
  - Systematisches Abfahren aller Ausführungspfade innerhalb einer Funktion
- Programmfluss durch **methodisch**e Anwendung unserer Templates **umsetzen**
- Optional: Optimieren

Repetitorium “Einrichten von Assembler Projekten mit Eclipse”

Vertiefung von **iterativen** und **rekursiven** Umsetzungsmustern

Agenda

Ein- und mehrdimensionale Felder (array, matrix):

- Organisation und Ablage im Speicher
- Lesender und schreibender Zugriff auf einzelne Elemente
- Iteration über alle Elemente eines Feldes
- Kopieren eines Elements

Strukturen (structures):

- Organisation und Ablage im Speicher
- Lesender und schreibender Zugriff auf ein einzelnes Element einer Struktur
- Kopieren einer gesamten Struktur

Motivation

Problemstellung:

Unsere Peripherie wird mittels Registern bedient, die im Speicher des RPi eingeblendet sind. Wir wollen diese Register möglichst strukturiert „bedienen“ können.

Aus der Beschreibung im Datenblatt können wir zwei grundsätzlich verschiedene **Datenstrukturen** erkennen:

- **Strukturen**, bei denen Bits/Bytes in den Registern bestimmte Funktionen haben
  - Bsp: GPFSEL ordnet der Funktion jedes GPIOs 3 bits zu
- **Felder** aus Registern und Strukturen
  - Bsp: **GPFSEL[0..6]**, **GPSET[0..1]**, **GPCLR[0..1]**

Address	Field Name	Description	Size	Read/ Write
0x 7E20 0000	GPFSEL0	GPIO Function Select 0	32	R/W
0x 7E20 0000	GPFSEL0	GPIO Function Select 0	32	R/W
0x 7E20 0004	GPFSEL1	GPIO Function Select 1	32	R/W
0x 7E20 0008	GPFSEL2	GPIO Function Select 2	32	R/W
0x 7E20 000C	GPFSEL3	GPIO Function Select 3	32	R/W
0x 7E20 0010	GPFSEL4	GPIO Function Select 4	32	R/W
0x 7E20 0014	GPFSEL5	GPIO Function Select 5	32	R/W
0x 7E20 0018	-	Reserved	-	-
0x 7E20 001C	GPSET0	GPIO Pin Output Set 0	32	W
0x 7E20 0020	GPSET1	GPIO Pin Output Set 1	32	W
0x 7E20 0024	-	Reserved	-	-
0x 7E20 0028	GPCLR0	GPIO Pin Output Clear 0	32	W
0x 7E20 002C	GPCLR1	GPIO Pin Output Clear 1	32	W
0x 7E20 0030	-	Reserved	-	-
0x 7E20 0034	GPLEV0	GPIO Pin Level 0	32	R
0x 7E20 0038	GPLEV1	GPIO Pin Level 1	32	R
0x 7E20 003C	-	Reserved	-	-
0x 7E20 0040	GPEDS0	GPIO Pin Event Detect Status 0	32	R/W
0x 7E20 0044	GPEDS1	GPIO Pin Event Detect Status 1	32	R/W
0x 7E20 0048	-	Reserved	-	-

Übersicht Datenstrukturen

Kurzer Ausflug in die Syntax: Direktiven

Wiederholung Direktiven

Textersetzung

.equ oder .set:	Darf mehrfach gesetzt werden
.equal:	Darf einmalig gesetzt werden

Bedingtes Kompilieren von Abschnitten

.if	Selektives kompilieren von Code
.else	
.endif	

Neue Direktiven

Texteinfügung aus einer Datei

.include	Eine andere Datei hier einfügen

Ausgabe eines Fehlers beim Kompilieren

.err	Einen Fehler beim Kompilieren ausgeben

Definition Datenstruktur

Eine Datenstruktur ist eine Art der Organisation von zusammengehörigen Daten zu einer Einheit, die effektiv genutzt werden kann. Ziele einer „guten“ Organisation sind geringer **Speicherbedarf**, hohe **Geschwindigkeit** oder gute **Lesbarkeit** und **Wartbarkeit**.

Statische Datenstrukturen:

- Organisation oder Sammlung von Daten mit zur Kompilierzeit bekanntem Platzbedarf.
  - Positiv: Sehr effektive Implementierung, Adressberechnung zur Kompilierzeit
  - Negativ: Unnötig hohe Belegung von Speicher für selten eintretende Worst-Case Szenarien.
  - Beispiel: Feld (array), Struktur (structure, union), Bitfeld, Ringe, usw.

Dynamische Datenstrukturen:

- Organisation oder Sammlung von Daten derart, dass die Gesamtstruktur zur Laufzeit wachsen oder schrumpfen kann.
  - Positiv: Platzbedarf muss nicht schon zur Kompilierzeit bekannt sein
  - Negativ: Zusätzlicher Speicherplatz zur Organisation und Rechenzeit für Verwaltung
  - Beispiele: einfach oder doppelt verkettete Liste, Netze, Bäume

Datenstrukturen

In MRT1 (Assembler und C):

- **Felder/mehrdimensionale Felder** (statisch) (Stroustrup 2013)
- **Strukturen**
- Ein-/Zweifach verkettete Listen (dynamisch, C)

Vorgriff auf MRT2 (C++, per STL):

- Listen
- Vektoren (dynamisch) (Stroustrup 2013, S. 96)
- Maps

Weitere nach Aho (2000):
Grundlegende

- Listen
- Stacks/Heaps
- Queues
- Mappings

Bäume

Binäre Bäume

Mengen/Sets

Hash Maps

Graphen

- Gerichtete Graphen
- Ungerichtete Graphen

Bestandteile von Datenstrukturen

Datenstrukturen bestehen aus drei wesentlichen Komponenten:

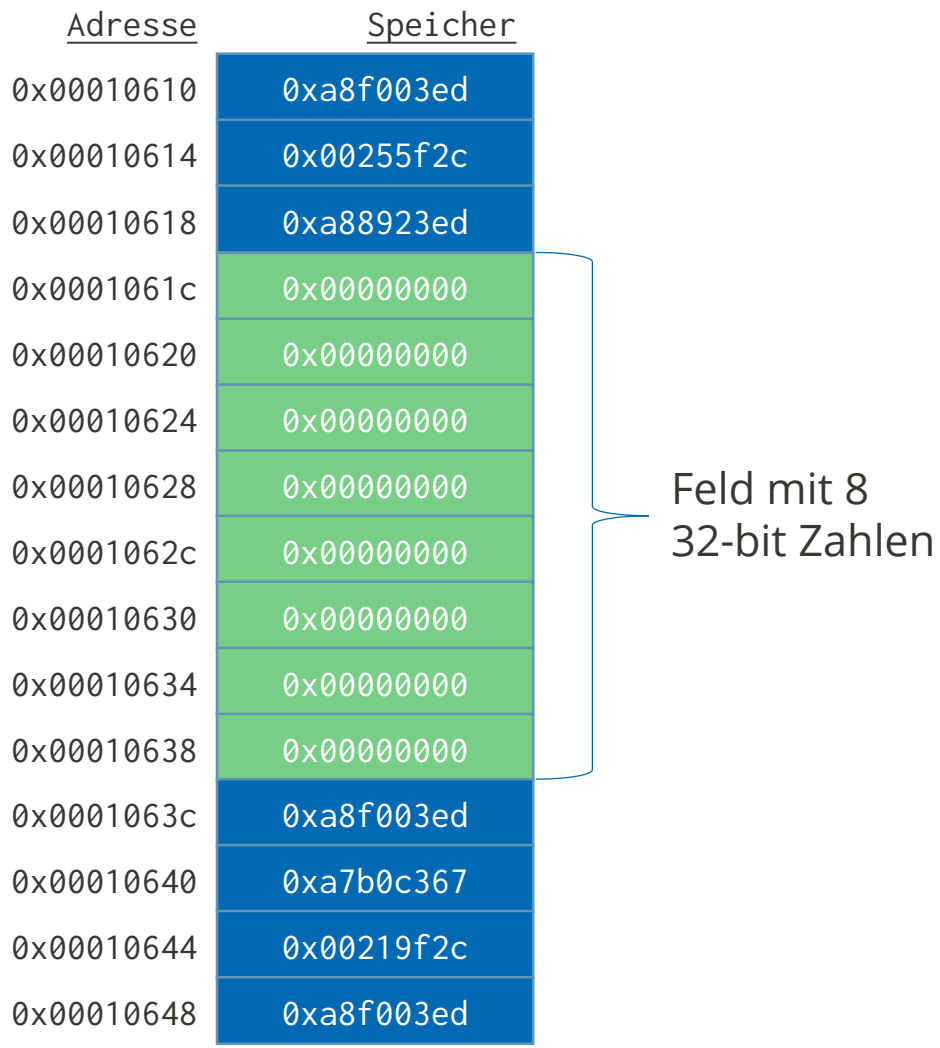
- Organisation der Daten im Speicher
  - Wie sind die Elemente im Speicher angeordnet?
  - Wie wird der Zusammenhang aufeinanderfolgender Strukturen geregelt?

- Methoden zur Organisation der Elemente der Datenstruktur
  - Einfügen & Löschen von Elementen (nur bei dynamischen Strukturen)
  - Kopieren von Strukturen

- Methoden zum Zugriff auf die Datenstruktur und ihrer Elemente
  - Lesen & Schreiben („Zugriff“) von Elementen
  - Iteration über alle Elemente

Felder (Arrays)

Felder (Arrays)



Ein Feld (Array, Field) ist **ein zusammenhängender Speicherbereich** mit **Elementen gleichen Typs und gleichem Speicherplatzbedarf** (Stroustrup 2013).

Der Zugriff auf ein einzelnes Element der Datenstruktur erfolgt über einen **Index**

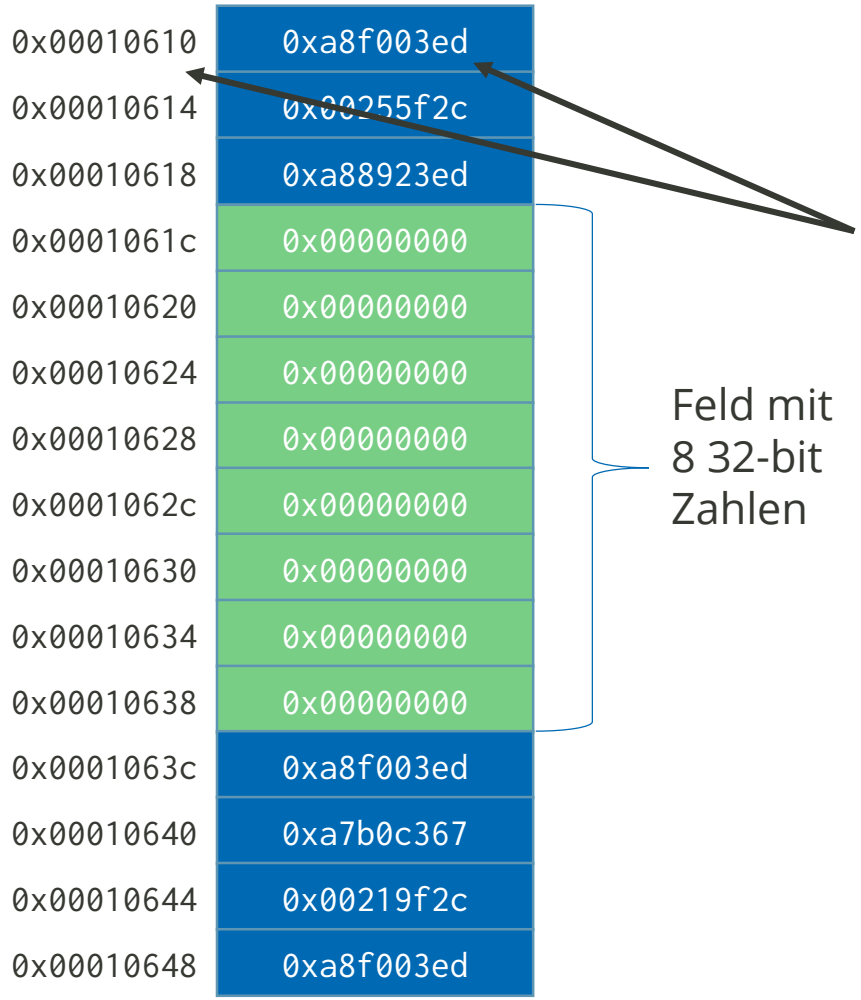
- In den von C abgeleiteten Sprachen läuft der Index **von 0 bis Elementanzahl-1**

Als **Notation** für den Zugriff auf das i'te Element im Feld FELDNAME verwenden wir die eckigen Klammern:

- `x = FELDNAME[i];`
- `Weise X den Wert des i-ten Elements des Feldes FELDNAME zu.`

Felder sind integrale Datenstrukturen der meisten Hochsprachen wie C, LUA, GO, PASCAL, FORTRAN, BASIC, Java

Definieren von eindimensionalen Feldern mit ARM_Assembler



Globale Felder werden im Quellcode im **.data**-Abschnitt definiert.

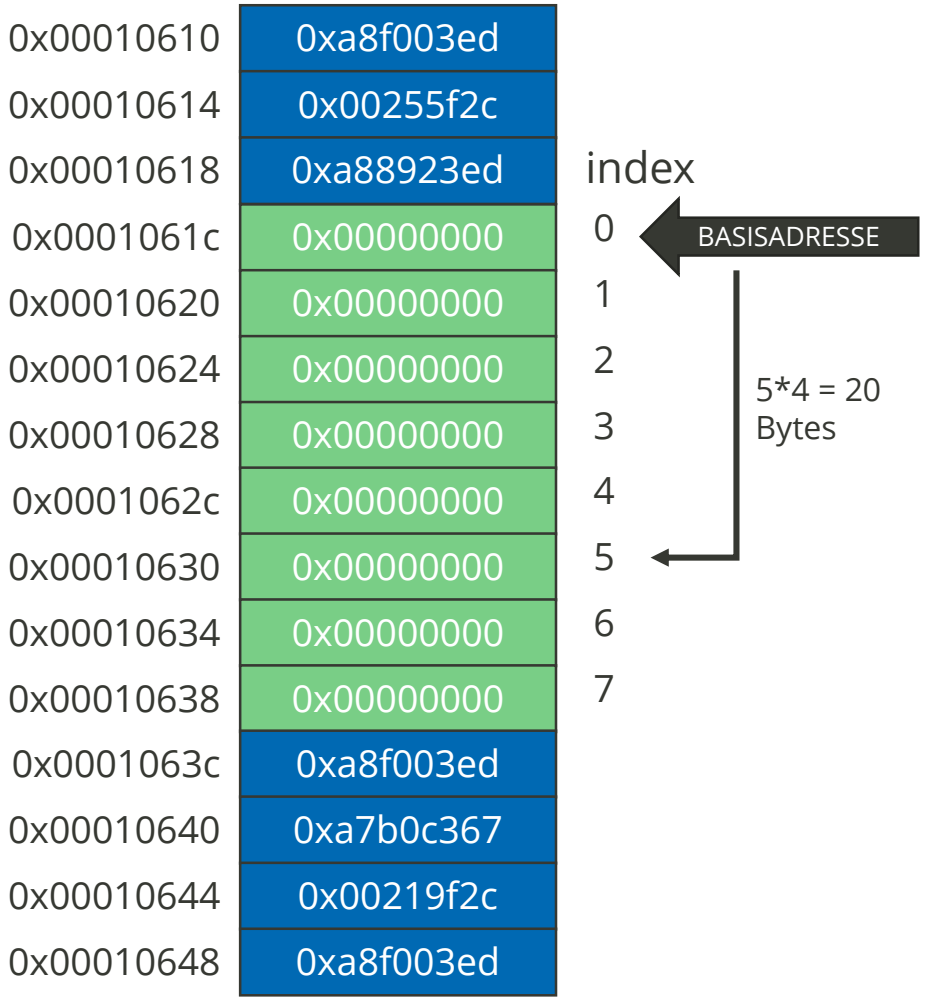
- **.skip <n>, <j>**: Lasse <n> byte frei, initialisiere alle Bytes mit <j>
- **.balign 4**: Richte Speicher an 4-byte-Grenzen aus

Zur besseren Wart- und Lesbarkeit empfiehlt es sich, das Feld mit Präprozessor-Anweisungen .equ oder .set zu parametrieren.

- **.set <ALIAS>, <WERT>**: Präprozessor-Alias
  .equ <ALIAS>, <WERT>: Präprozessor-Alias

```
.equ FIELD_TYPE_SIZE, 8
.equ FIELD_SIZE,      4
.data
.balign 4
FIELD: .skip FIELD_SIZE*FIELD_TYPE_SIZE, 0
```

Addressen von Feldelementen bestimmen



Die **Basisadresse** eines Felds ist die **Adresse des 0ten Elements**.

Die Adresse eines bestimmten Feldes wird wie folgt berechnet:

$$\texttt{ADRESSEVON(FELD[index])} = \texttt{BASISADRESSE}_{\texttt{FELD}} + \texttt{index} * \texttt{ELEMENTGRÖßE}_{\texttt{Byte}}$$

Der Zugriff auf ein Element kostet also nur O(1) und ist nicht von der Anzahl der Elemente abhängig !

Beispiel:

Basisadresse: **0x0001061c**

Größe eines Feldelements: **4 Byte**

Adresse von **FELD[5]** =

0x0001061c + 5*4 = **0x00010630**

Wenn Basisadresse und Index schon zur Übersetzungszeit bekannt sind, kann das der Compiler ausrechnen ^^

Adress-Berechnung Methode

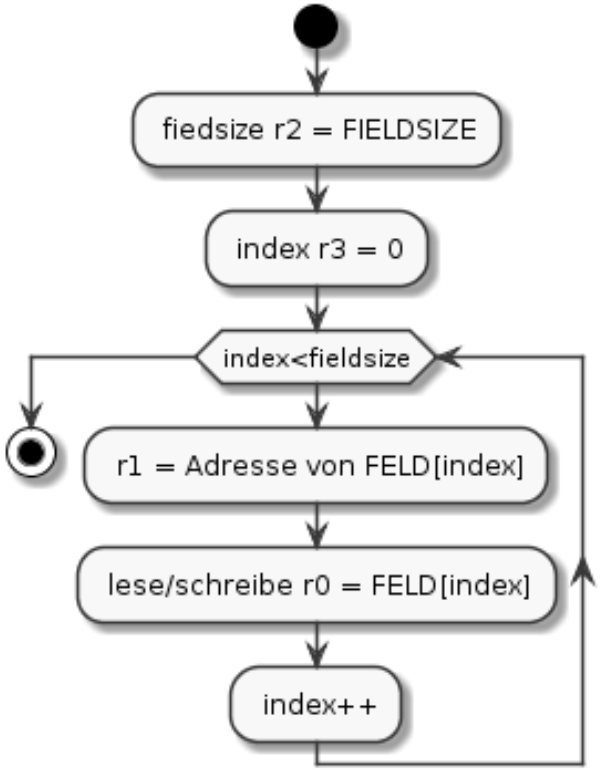
Methode bei der Assembly

Statische Adresse angeben

Mul-Instruktion verwenden

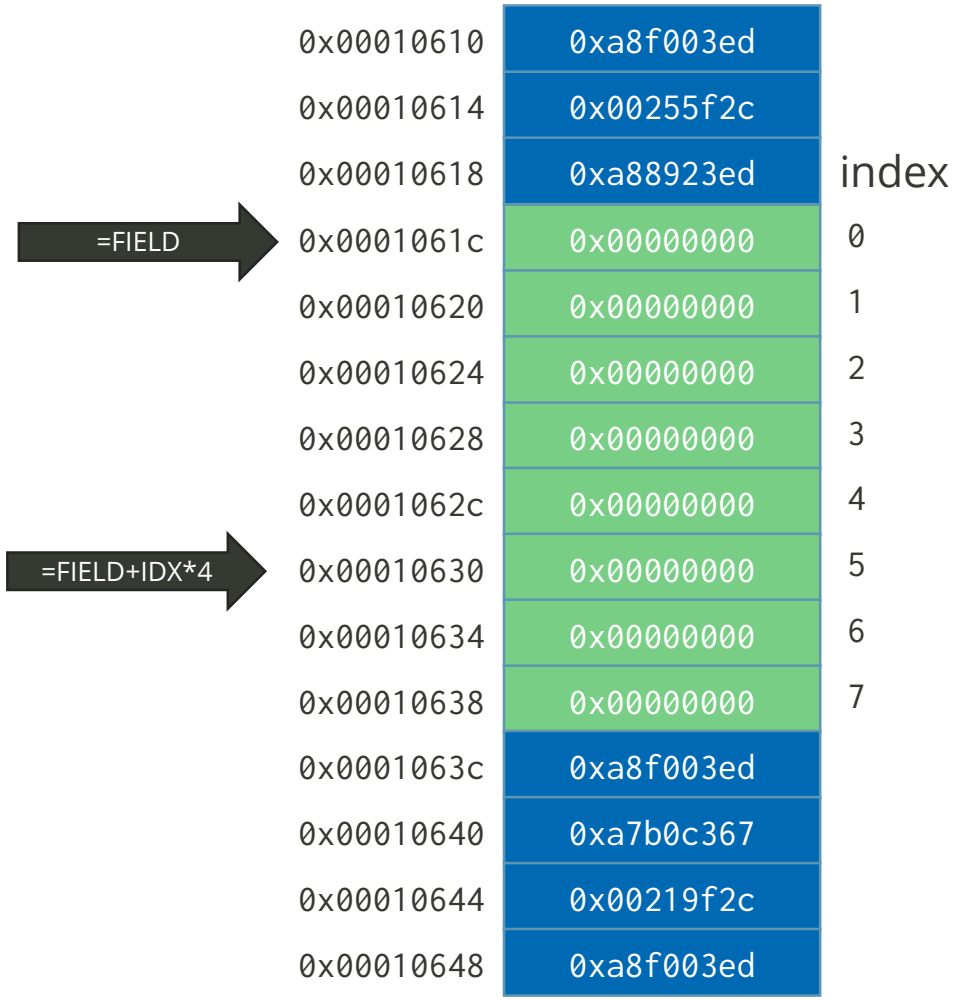
Verschiebungsanweisung verwenden

Traversieren durch ein Array



Zugriff auf konstante Elemente: Zugriff auf konstante Feldindizes

Zugriff auf Elemente in Feldern



Zugriffsverfahren für Lesen und Schreiben eines zur Kompilierzeit bekannten Elements eines Felds:

```
.equ FIELD_SIZE, 8
.equ FIELD_TYPE_SIZE, 4

.data
FIELD: .skip FIELD_SIZE*FIELD_TYPE_SIZE, 0

.text

.equ FIELD_INDEX, 5

ldr r3, =FIELD                                  // r3 <- base address

ldr r0, [r3, #(FIELD_INDEX*FIELD_TYPE_SIZE)]  // r0 <- [base + index * element_size]

add r0,  r0, #1                         // r0 <- r0 + 1

str r0, [r3, #(FIELD_INDEX*FIELD_TYPE_SIZE)]  // r0 -> [base + index * element_size]
```

Allgemeines Zugriffsmuster

Zugriff auf Elemente in Feldern

Allgemein gilt: $\text{ADRESSE}_{\text{FELD}}(\text{index}) = \text{BASIS\_ADRESSE}_{\text{FELD}} + \text{index} * \text{ELEMENTGRÖßE}_{\text{Byte}}$

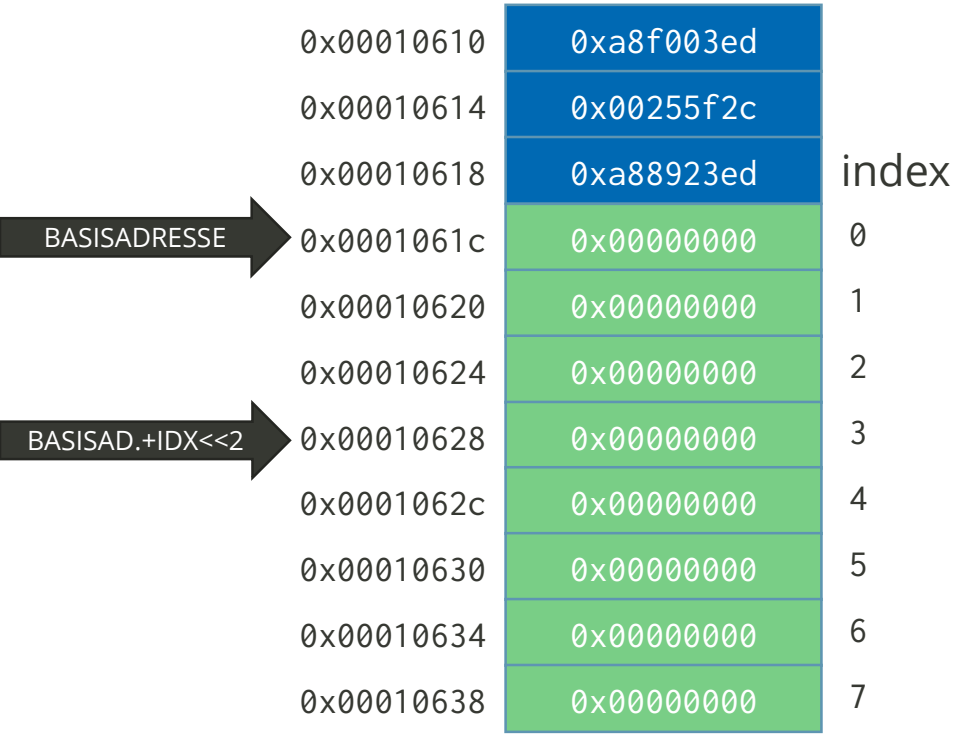
```
.equ FIELD_TYPE_SIZE, 4
.equ FIELD_INDEX, 3


mov r2, #FIELD_INDEX      // r2: FIELD_INDEX
array_example_dynamicAddress_general:
mov r0, #FIELD_TYPE_SIZE  // r0: FIELD_TYPE_SIZE
mul r0, r2, r0            // r0: FIELD_INDEX * FIELD_TYPE_SIZE
ldr r1, =FIELD            // r1: BASIS_ADRESSE
add r1, r0, r1            // r1: BASIS_ADRESSE + FIELD_INDEX * FIELD_TYPE_SIZE

ldr r0, [r1]              // r0 <- [r1]
add r0, r0, #1            // r0 <- r0 + 1
str r0, [r1]              // r0 -> [r1]
```

Optimaler Fall: Die Größe eines Elements im Feld lässt sich als 2^n ausdrücken!
Wir können mit Register-Indirekt-Addressierung und Shift das Multiplizieren umgehen

```
array_example_dynamicAddress_optimal:
.equ FIELD_TYPE_LSL, 2

ldr r3, =FIELD                          // r3 <- base address of FIELD
mov r2, #FIELD_INDEX                    // r2 <- index of element
ldr r0, [r3, r2, LSL #FIELD_TYPE_LSL]   // r0 <- [base + index << 2]
add r0, r0, #1                          // r0 <- r0 + 3
str r0, [r3, r2, LSL #FIELD_TYPE_LSL]   // r0 -> [base + index << 2]
```

Zugriff auf Elemente in Felder mit optimierter Elementgröße

Zugriff auf Elemente in Feldern



$i * 2^n = i << n$

$3 * 4 = 3 * 2^2$

$= 3 << 2$ (binär)

Hardware-Multiplier oder Multiplikation als Befehlssatzbestandteil sind in Mikrorechenarchitekturen selten. Multiplikatoren (z.B. Booth-Multiplier) brauchen mehrere Takte, sind groß und werden selten genutzt (erhöhen Chipfläche).

```
mul r0, r2, r0
```

Generelle Lösung auf Mikrorechenplattformen: Multiplikation als **Additionsschleife**...
→ SEHR **ineffizient**.

Optimierung unseres Felds: Die Größe eines Elements im Feld lässt sich als 2^n ausdrücken!
Wir können mit Register-Indirekt-Addressierung und Shift das Multiplizieren umgehen

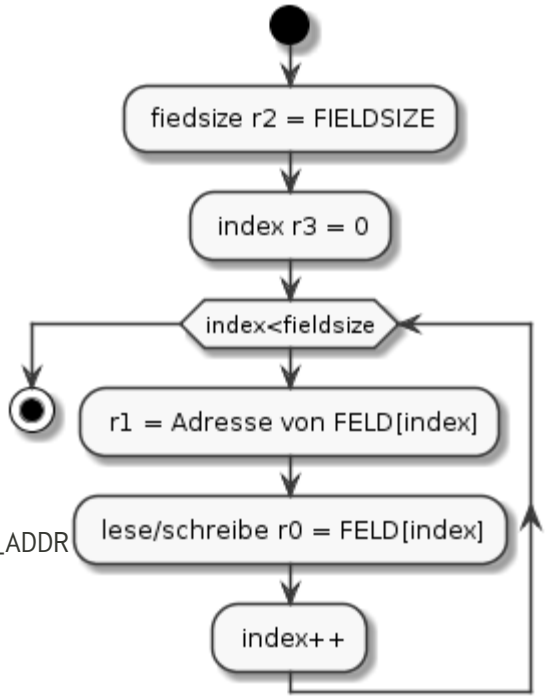
```
array_example_dynamicAddress_optimal:
.equ FIELD_INDEX, 3
.equ FIELD_TYPE_LSL, 2
ldr r3, =FIELD                              // r3 <- basis
mov r2, #FIELD_INDEX                        // r2 <- index
ldr r0, [r3, r2, LSL #FIELD_TYPE_LSL]       // r0 <- [base + index * 4 (d.h. index<<2)]
add r0, r0, #1                              // r0 <- r0 + 1
str r0, [r3, r2, LSL #FIELD_TYPE_LSL]       // r0 -> [base + index << 2]
```

Iterieren über alle Feldelemente (mit Multiplikation)

Iteration ist lediglich Synthese uns bereits bekannter

- **Template** (for-Schleife)
- Wahl eines geeigneten **Zugriffsmusters**

```
// Template: for-Schleife
// r2: base address, r3: index
array_example_dynamicAddress_iterate_init:
    ldr r2, =FIELD
    mov r3, #0
array_example_dynamicAddress_iterate_condition:
    cmp r3,#FIELD_SIZE
    bge array_example_dynamicAddress_iterate_end

    // Berechne Element-Addresse
    ldr r0, =FIELD_TYPE_SIZE
    mul r1, r0, r3
    ldr r0, =FIELD
    add r1, r1, r0  // r1 = FIELD_TYPE_SIZE*i + BASE_ADDR

    // Greife auf Element zu
    ldr r0,[r1]
    add r0, r0, r3
    str r0,[r1]

  array_example_dynamicAddress_iterate_update:
    add r3, r3, #1
    b array_example_dynamicAddress_iterate_condition
  array_example_dynamicAddress_iterate_end:
```

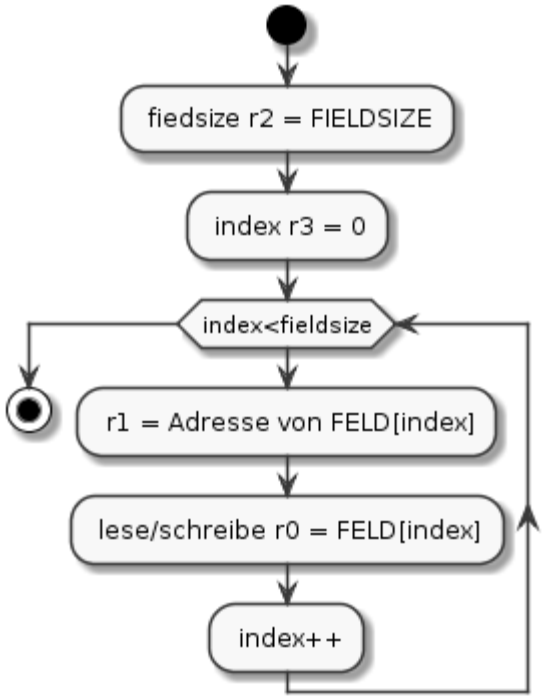


Iterieren über alle Feldelemente (mit expliziter Addition der Adresse)

Registerbelegung

- **R0: (temporär)**
- **R1: Laufende Elementaddresse**
- **R2: Erste Adresse nach allen Feldelementen**

```
// Template: for-Schleife
// r1: current address
array_add_dynamicAddress_iterate_init:
    ldr r1, =FIELD
    ldr r0, =FIELD_SIZE*FIELD_TYPE_SIZE
    add r2, r1, r0

array_add_dynamicAddress_iterate_condition:
    cmp r1, r2
    bge array_add_dynamicAddress_iterate_end

// Greife auf Element zu
    ldr r0,[r1]
    add r0, r0, #1
    str r0,[r1]

array_add_dynamicAddress_iterate_update:
    ldr r0, =FIELD_TYPE_SIZE
    add r1, r1, r0
    b array_add_dynamicAddress_iterate_condition
array_add_dynamicAddress_iterate_end:
```



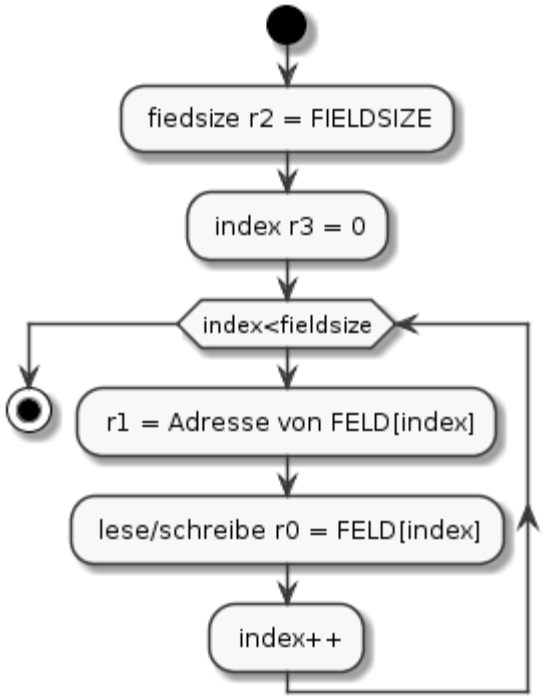
Iterieren über alle Feldelemente (mit impliziter Addition der Adresse)

Registerbelegung

- **R0: (temporär)**
- **R1: Laufende Elementaddresse**
- **R2: Erste Adresse nach allen Feldelementen**

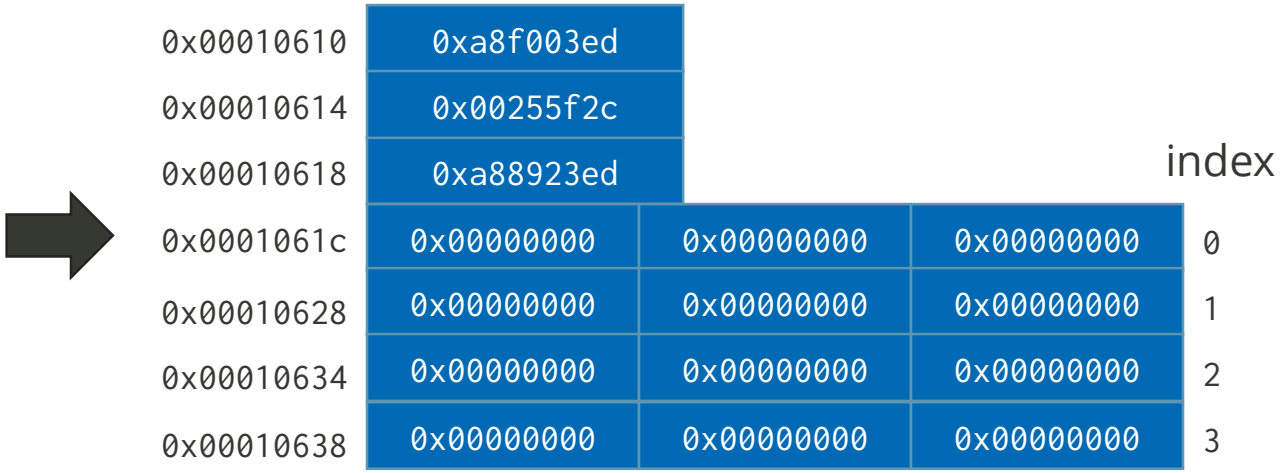
```
// Template: for-Schleife
// r1: current address
array_add_dynamicAddress_iterate_init:
    ldr r1, =FIELD
    ldr r0, =FIELD_SIZE*FIELD_TYPE_SIZE
    add r2, r1, r0

array_add_dynamicAddress_iterate_condition:
    cmp r1, r2
    bge array_add_dynamicAddress_iterate_end

// Greife auf Element zu
    ldr r0,[r1]
    add r0, r0, #1
    str r0,[r1], #FIELD_TYPE_SIZE

array_add_dynamicAddress_iterate_update:
    b array_add_dynamicAddress_iterate_condition

array_add_dynamicAddress_iterate_end:
```

indexed post-increment
1) Speichere r0 an der Adresse r1
2) Erhöhe r1 um FIELD_TYPE_SIZE



Effizienter Zugriff auf Elemente in Felder mit suboptimaler Elementgröße

Zugriff auf Elemente in Feldern



Allgemeines Zugriffsmuster:
Gilt immer, braucht MUL
Optimiertes Zugriffsmuster für Felder mit optimaler Elementgröße:
Gilt, wenn Element %2 = 0

Was passiert bei Elementgröße <> 2^n ohne MUL?

```
.equ WIERDFIELD_TYPE_SIZE, 12                                   // Größe des einzelnen Elements, hier 12 Byte
.equ WIERDFIELD_SIZE,       8                                   // Anzahl der Elemente, hier 8

.data
vector: .skip WIERDFIELD_SIZE*WIERDFIELD_TYPE_SIZE, 0   // 8 * 12 = 96 Byte
.balign 4
```

Effizienter Zugriff auf Elemente in Felder mit suboptimaler Elementgröße (2)
Zugriff auf Elemente in Feldern

Tief eingebettete Systeme haben meistens keine Multiplikationsinstruktion.

- Lösungsansatz: Multiplikation zerlegen in Sequenz von Additions- und Shift-Operationen
- Gesucht: Kürzeste Zerlegung mit minimaler Anzahl von Zykluszeiten

Beispiel: i * 12

- Größter *Teiler* von 12, der eine Zweierpotenz ist: (**4 *** 3 * i) = 12 * i
- Zerlegen des Faktors 3 in eine **Multiplikation** mit **2** und eine **Addition** : (**4 *** (**2 * i + i**)) = 12 * i

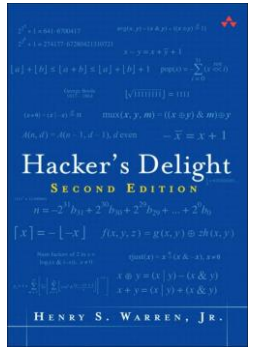
```
MOV R1,#<i>          @ R1 ← die mit 12 zu multiplizierende Zahl
LSL R2,R1,#1         @ R2 ← 2 * i
ADD R2,R2,R1         @ R2 ← 2 * i  + i
LSL R2,R2,#2         @ R2 ← 4 * ( 2 * i + i)
```

ARM Barrel-Shifter für <Operand2> erlaubt das Zusammenfassen von Addition und Shift

```
MOV R1,#<i>          @ R1 ← die mit 12 zu multiplizierende Zahl
ADD R2,R1,R1,LSL #1  @ R2 ← 2 * i  + i
LSL R2,R2,#2         @ R2 ← 4 * ( 2 * i + i)
```

Die Multiplikation einer beliebigen Zahl mit der Konstanten 12 kann mit **2 Instruktionen** realisiert werden

Division, Encodings, Textsuche...



(Warren 2013)

Base+Offset-Address-Berechnung bis 20 mit Shift-and-Add-Multiplikation

Zugriff auf Elemente in Feldern

R3=BASE Gesucht=[Basis + Index * Size]

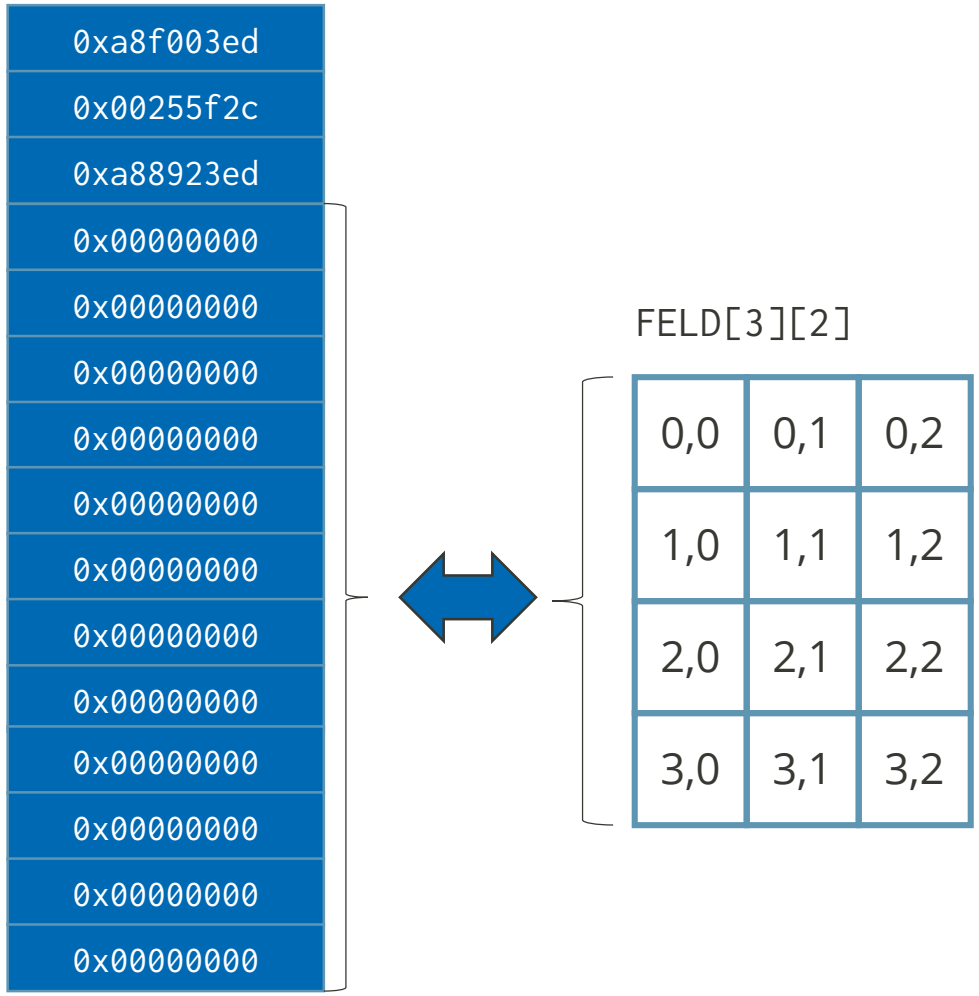
R2=Index, wird zum Offset für Loadanweisung

R1=Register für Zwischenrechnung

Size	Zerlegung	Op1	Op2	Op3	Op4	Instruktion 1	Instruktion 2	Instruktion 3	Anzahl
2	i*2	LSL 1						LDR R0,[R3,R2,LSL#1]	1
3	i*2+i	LSL 1	ADD			ADD R2,R2,R2,LSL #1		LDR R0,[R3,R2]	2
4	i*4	LSL 2						LDR R0,[R3,R2,LSL#2]	1
5	i*4+i	LSL 2	ADD			ADD R2,R2,R2,LSL #2		LDR R0,[R3,R2]	2
6	(i*2+i)*2	LSL 1	ADD	LSL 1		ADD R2,R2,R2,LSL #1		LDR R0,[R3,R2,LSL#1]	2
7	(i*8-i)	LSL 3	SUB			RSB R2,R2,R2,#3		LDR R0,[R3,R2]	2
8	i*8	LSL 3						LDR R0,[R3,R2,LSL#3]	1
9	(i*8+i)	LSL 3	ADD			ADD R2,R2,R2,LSL #3		LDR R0,[R3,R2]	2
10	(i*4+i)*2	LSL 2	ADD	LSL 1		ADD R2,R2,R2,LSL #2		LDR R0,[R3,R2,LSL#1]	2
11	(i*4+i)*2+i	LSL 2	ADD	LSL 1	ADD	ADD R1,R2,R2,LSL #2	ADD R2,R2,R1,LSL #1	LDR R0,[R3,R2]	3
12	(i*2+i)*4	LSL 1	ADD	LSL 2		ADD R2,R2,R2,LSL #1		LDR R0,[R3,R2,LSL#2]	2
13	(i*2+i)*4+i	LSL 1	ADD	LSL 2	ADD	ADD R1,R2,R2,LSL #1	ADD R2,R2,R1,LSL #2	LDR R0,[R3,R2]	3
14	(i*8-i)*2	LSL 3	SUB	LSL 1		RSB R2,R2,R2,#3		LDR R0,[R3,R2,LSL#1]	2
15	(i*16)-i	LSL 4	SUB			RSB R2,R2,R2,#4		LDR R0,[R3,R2]	2
16	i*16	LSL 4						LDR R0,[R3,R2,LSL#4]	1
17	(i*16+i)	LSL 4	ADD			ADD R2,R2,R2,LSL #4		LDR R0,[R3,R2]	2
18	(i*8+i)*2	LSL 3	ADD	LSL 1		ADD R2,R2,R2,LSL #3		LDR R0,[R3,R2,LSL#1]	2
19	(i*8+i)*2+i	LSL 3	ADD	LSL 1	ADD	ADD R1,R2,R2,LSL #3	ADD R2,R2,R1,LSL #1	LDR R0,[R3,R2]	3
20	(i*4+i)*4	LSL 2	ADD	LSL 2		ADD R2,R2,R2,LSL #2		LDR R0,[R3,R2,LSL#2]	2

Mehrdimensionale Felder

Mehrdimensionale Felder



Ein mehrdimensionales Feld (multidimensional Array, Matrix) ist **ein zusammenhängender Speicherbereich** mit **Elementen gleichen Typs und gleicher Größe**, die mit mehr als einem Index ausgewählt werden.

Der Zugriff auf ein einzelnes Element erfolgt über **mehrere Indizes,** jeder **von 0 bis Elementanzahl-1.**

Es können n-dimensionale Felder verwendet werden, wir behandeln hier den Fall n==2

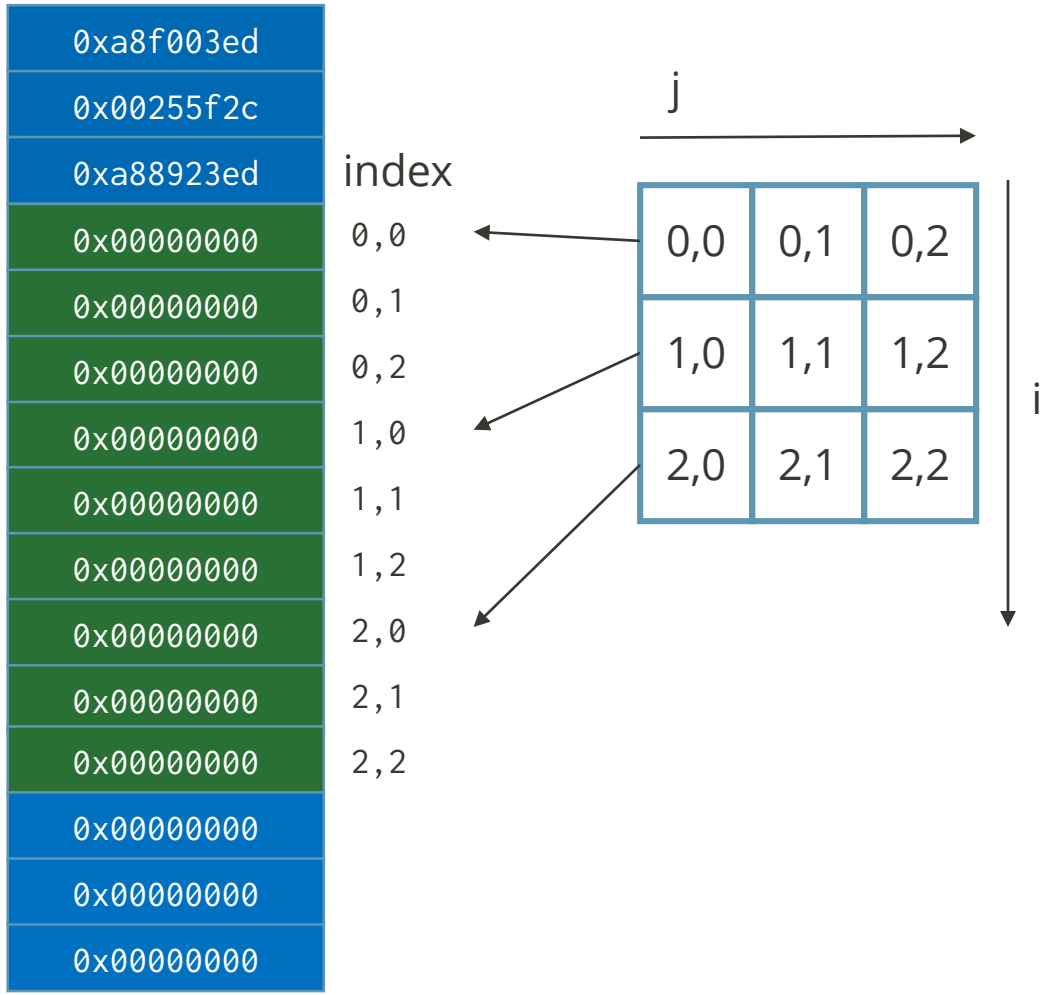
- Angelehnt an Matrizen: Spalten und Zeilen
- **Jede** Zeile und Spalte (**Dimension**) des Felds enthält die **selbe Anzahl von Elementen**

Notation für den Zugriff auf das i'te Element einer Zeile in der j'ten Spalte erfolgt der Zugriff auf Element x:

```
x = FELDNAME[i][j]
```

Abbildung n-dimensionaler Felder auf lineare Speicher

Mehrdimensionale Felder



Welche **Reihenfolge** gilt bei Anordnung der Elemente **in** unserem 1-dimensionalen **Speicher**?

An C angelehnte Sprachen:

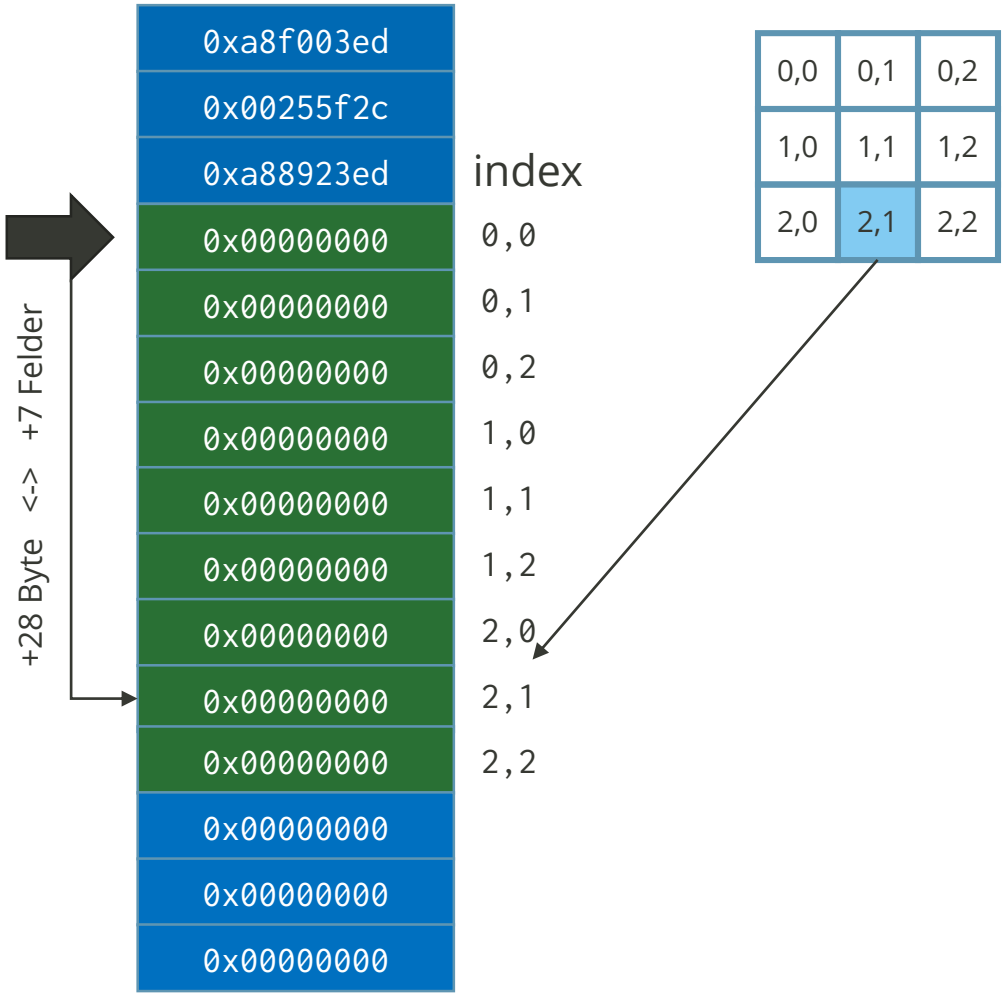
Die Elemente werden **in Zugriffsreihenfolge**, nacheinander in den Speicher geschrieben.

```
FELDNAME[i][j]
```

→ **Der letzte Index (in Leserichtung) läuft zuerst**

Zugriff auf Elemente n-dimensionaler Felder

Mehrdimensionale Felder



Die Adresse des [i,j]'ten Elements ergibt sich aus:

$$\begin{aligned}\text{ADDRESSOF(FELD[i][j])} := {} & \text{BASISADDRESSE}_{\text{FELD}} + \\ & \text{dim(i)}*i*\text{FELD\_ELEMENT\_SIZE}_{\text{BYTE}} + \\ & j*\text{FELD\_ELEMENT\_SIZE}_{\text{BYTE}}\end{aligned}$$

Wobei dim(i) die Anzahl der Elemente in der jeweilige Dimension ist.

Beispiel **4x3 matrix**:

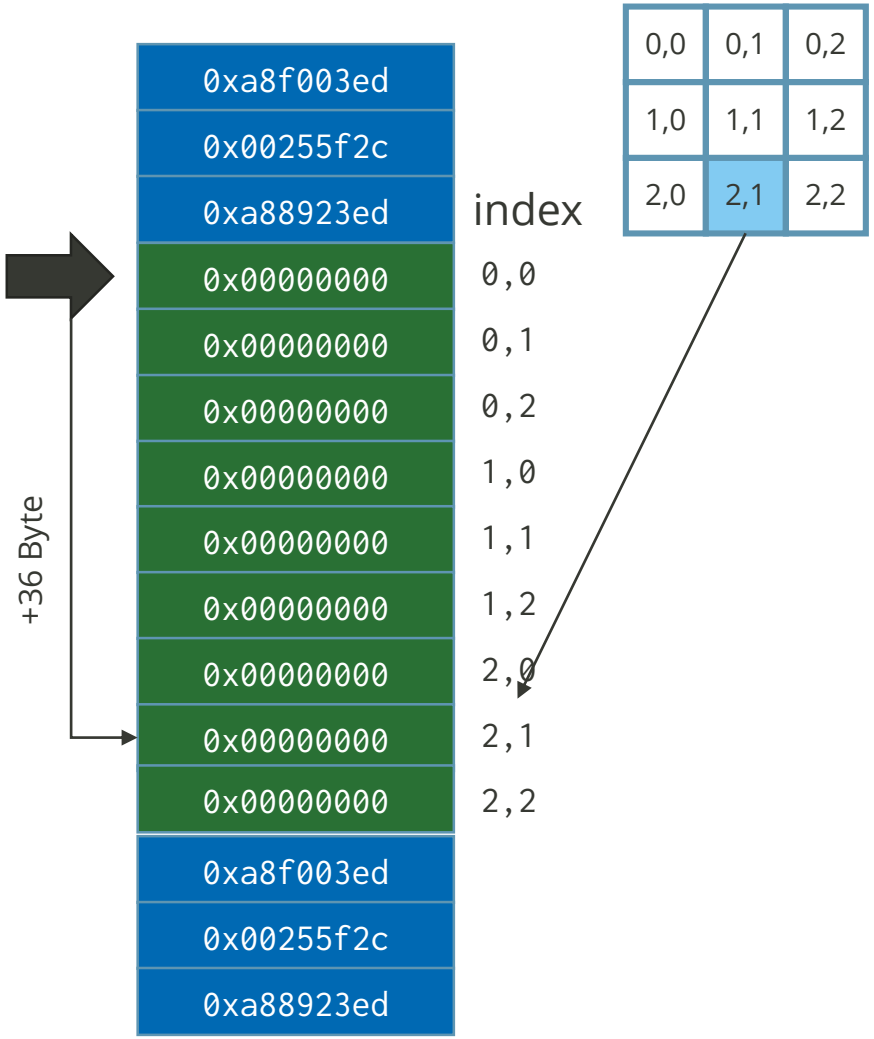
BASISADDRESSE := 500

$$\text{FELD\_ELEMENT\_SIZE}_{\text{BYTE}} := 4$$

$$\begin{aligned}\text{ADDRESSOF(FELD[2][1])} := {} & 500 + \\ & 3_{\text{Dim(i)}}*2_i*4_{\text{BYTE}} + 1_j*4_{\text{BYTE}} \\ := {} & \mathbf{500 + 28\ Byte}\end{aligned}$$

Berechnung der Adresse in Assembler

Mehrdimensionale Felder



Es gelten alle drei Techniken, die wir bereits aus eindimensionalen Feldern kennen:

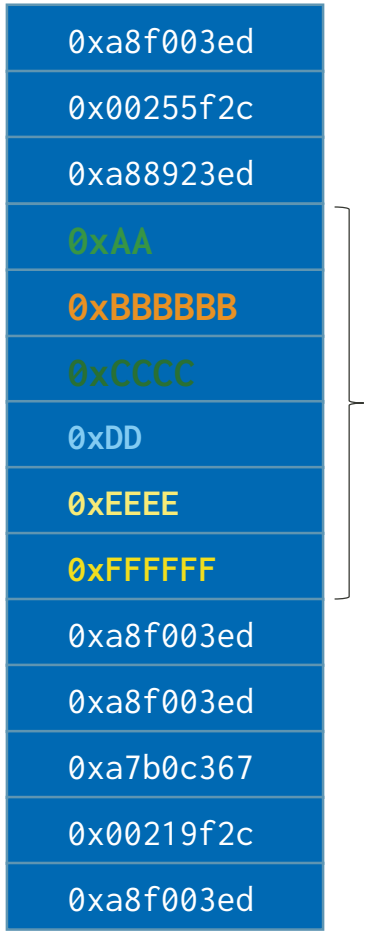
- Allgemeine Berechung (2x Multiplikation ☹)
- Optimierte Varianten für Felder, wenn jede der Dimensionen als Potenz von 2 ausdrückbar ist
- **Zerlegung der Berechnung in Addition- und Schiebeoperation**

```
// Template: Sektion
// Bereche Adresse von FELD[2,1]
matrix_example_calc2daddress:
    lsl    r1,   r10, #2        // r1 = 4i
    add r0, r9, r9, LSL #1      // r0 = r9 + r9 * 2
    add r0, r1, r0, LSL #2      // r0 = 4i + (2i+i)*4
    ldr   r1, =MATRIX           // r1 <- base address
    add r1, r0, r1              // r1 = addr of (feld[i,j]) = BASE + 4*4*j + 4*i
matrix_example_calc2daddress_end:
```

Strukturen

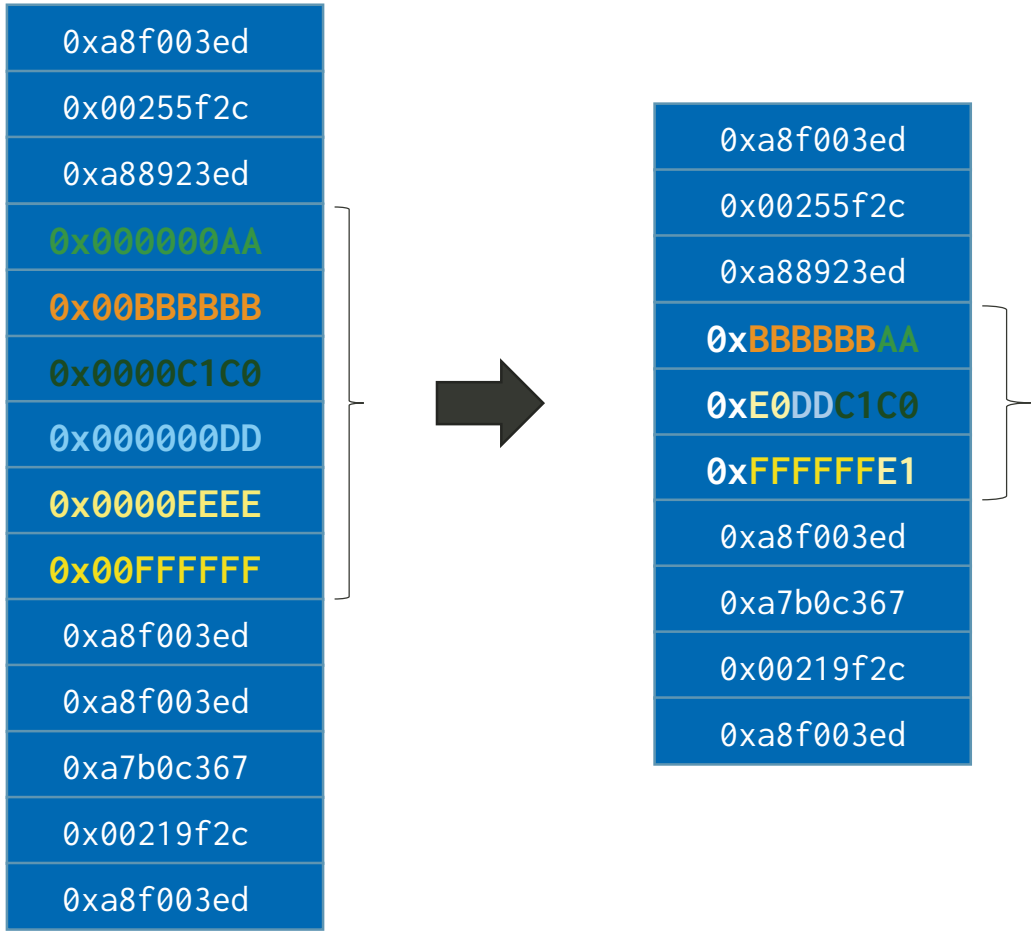
Strukturen

```
struct {
  char A;
  char B[3];
  short C;
  char D;
  short E;
  char F[3];
}
```

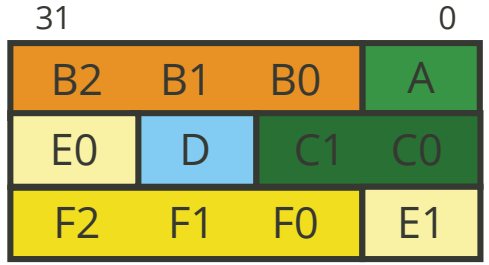


Strukturen

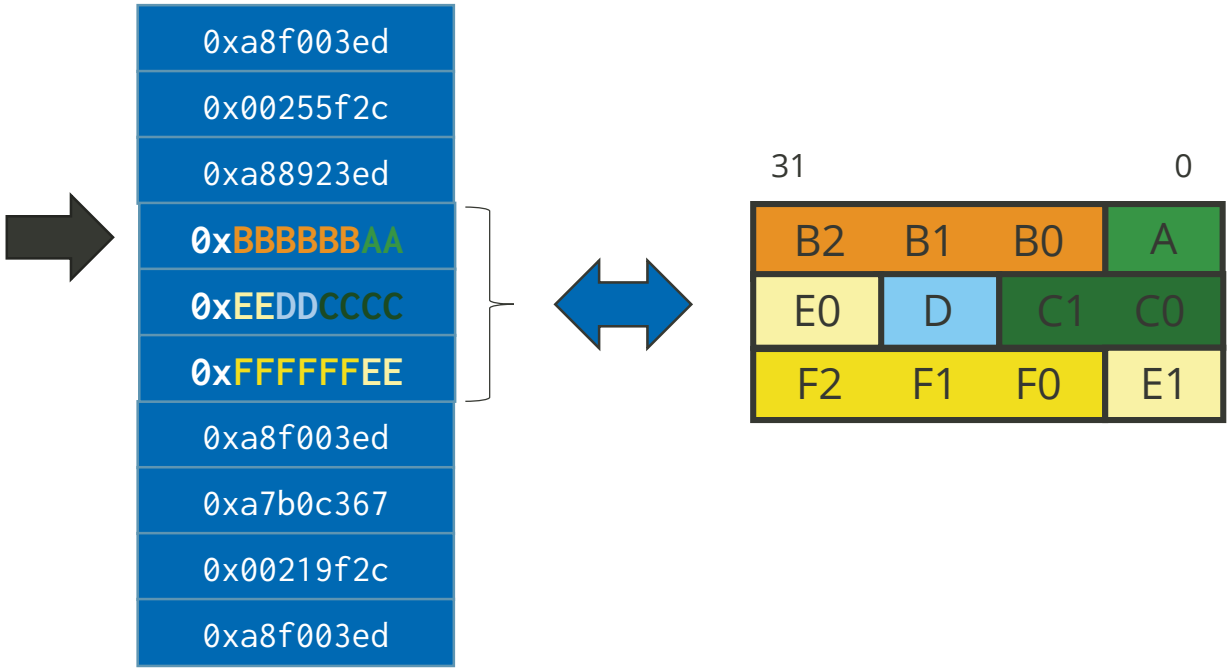
```
struct {
  char A;
  char B[3];
  short C;
  char D;
  short E;
  char F[3];
}
```




```
.balign 4
STRUKTUR:
.byte 'a'               // char A
.byte 'b','i','t'       // char B[3]
.hword 0xC1C0           // short C
.byte 'd'               // char D
.hword 0xE1E0           // short E
.byte 'f','o','r'       // char F[3]
```



Strukturen



Eine Struktur (structure, struct) ist **ein zusammenhängender Speicherbereich** mit **Elementen ungleichen Typs und variabler Größe**.

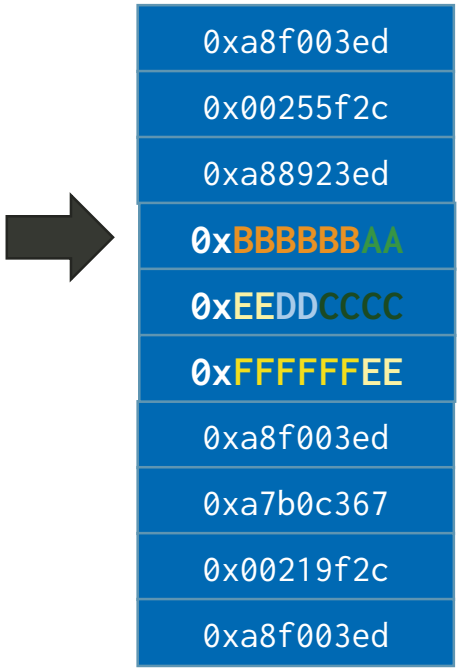
Der Zugriff auf Strukturen ist abhängig von deren Kodierung.

ARM Assembler bietet uns hier keine Sprachmittel.

Der Zugriff auf ein Element einer Struktur erfolgt über den Offset des Feldes zur Basisadresse:

ADDRESSOF(FIELDNAME) :=
BASISADRESSE + ELEMENTOFFSET

Zugriff auf Elemente in Strukturen




```
struct {
  char A;
  char B[3];
  short C;
  char D;
  short E;
  char F[3];
}
```

```
// Bekannt durch Platzbedarf der Elemente
.equ STRUKTUR_BYTEOFFSET_A, 0
.equ STRUKTUR_BYTEOFFSET_B, 1
.equ STRUKTUR_BYTEOFFSET_C, 4
.equ STRUKTUR_BYTEOFFSET_D, 6
.equ STRUKTUR_BYTEOFFSET_E, 7
.equ STRUKTUR_BYTEOFFSET_F, 9

// Berechnen der Adresse von z.B. E
ldr  r0, =STRUKTUR
ldrh r0, [r0, #STRUKTUR_BYTEOFFSET_E]
```

Eine Struktur (structure, struct) ist **ein zusammenhängender Speicherbereich** mit **Elementen ungleichen Typs und variabler Größe**.

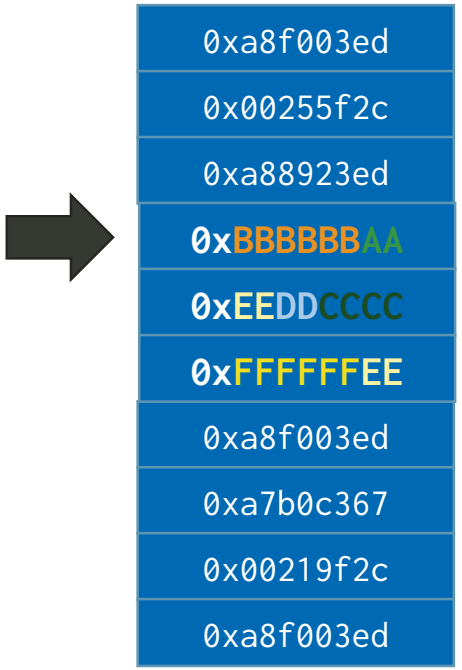
ARM Assembler bietet uns keine Sprachmittel für die.

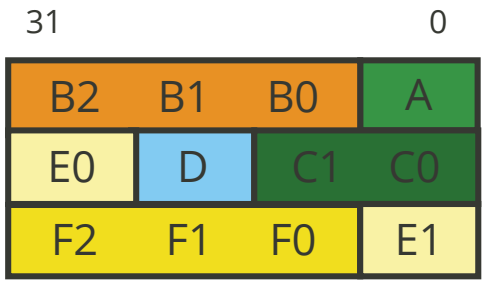
Lösung: Wir definieren Offsets für alle Felder!

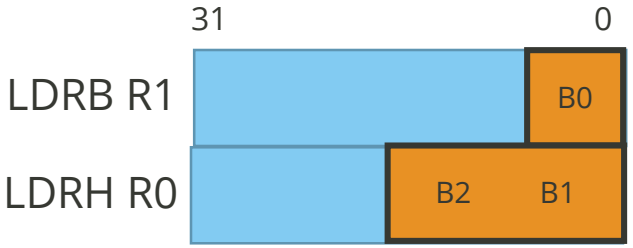
Der Zugriff auf ein Element einer Struktur erfolgt über den Offset des Feldes zur Basisadresse:

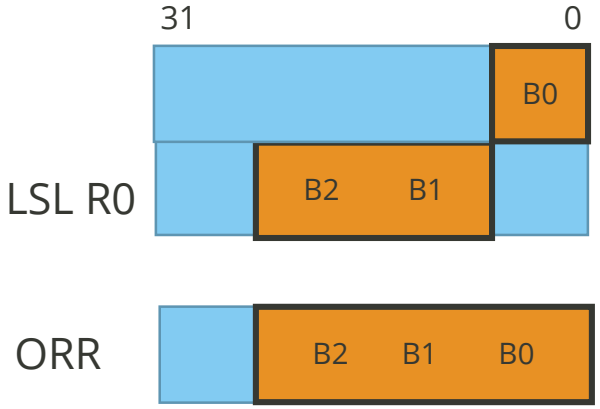
ADDRESSOF(FIELDNAME) :=
BASISADRESSE + **ELEMENTOFFSET**

Zugriff auf Elemente in Strukturen










```
// Bekannt durch Kodierung des Struktur
.equ STRUKTUR_BYTEOFFSET_A, #0
.equ STRUKTUR_BYTEOFFSET_B, #1
.equ STRUKTUR_BYTEOFFSET_C, #4
.equ STRUKTUR_BYTEOFFSET_D, #6
.equ STRUKTUR_BYTEOFFSET_E, #7
.equ STRUKTUR_BYTEOFFSET_F, #9

// Lade B
ldr  r0, =STRUKTUR
ldrb r1, [r0, #STRUKTUR_BYTEOFFSET_B]
ldrh r0, [r0, #STRUKTUR_BYTEOFFSET_B + 1]
lsl  r0, #8
orr  r0, r0, r1
```

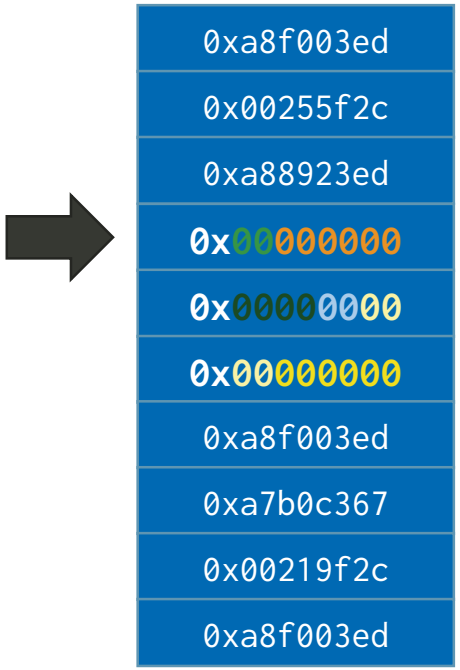
Load-/Store-Instruktionen sind immer abhängig von der Größe des Elements:

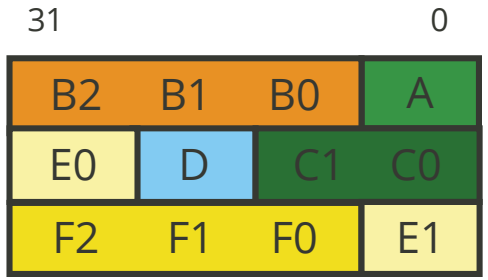
- 1 Byte: ldr**b**/str**b** (byte)
- 2 Byte: ldr**h**/str**h** (halfword)
- 4 Byte: ldr/str (word, native Wortbreite auf ARM)

Was machen wir bei 3 Byte?

- Zerlegen in eine Sequenz von 1/2/4-Operation
  - Byte-Zugriff für den ersten Teil
  - Halbwort-Zugriff für zweiten Teil
  - Schieben des zweiten Teils an die richtige Stelle
  - ORR-Verknüpfen des ersten und des zweiten Teils

Alternative: Zugriff auf A und B, danach Ausblenden von A
(ist jetzt allerdings abhängig von A, nicht empfehlenswert)

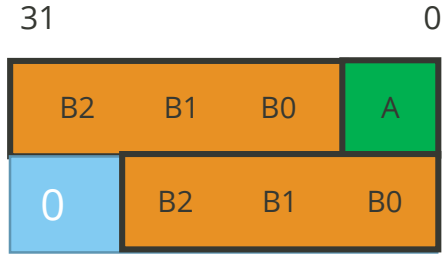




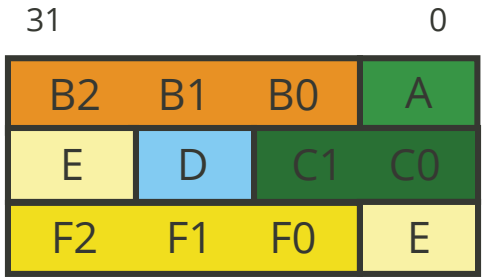

```
// Bekannt durch Kodierung des Struktur
.equ STRUKTUR_BYTEOFFSET_A, 0
.equ STRUKTUR_BYTEOFFSET_B, 1
.equ STRUKTUR_BYTEOFFSET_C, 4
.equ STRUKTUR_BYTEOFFSET_D, 6
.equ STRUKTUR_BYTEOFFSET_E, 7
.equ STRUKTUR_BYTEOFFSET_F, 9

// Lade B
ldr  r0, =STRUKTUR
ldr  r1, [r0, #STRUKTUR_BYTEOFFSET_A]
lsr  r1, #8
```

LDR R1,...

LSR R1, #8



Übungsaufgabe: Speicheroperation für Strukturen



Setzen Sie für den Zugriff auf die abgebildete Struktur für die Felder A, C und F um.

Praxisbezug

- Der Zugriff muss aus mehreren Operationen (komplementär) zum Lade-Beispiel der Vorlesung umgesetzt werden.
- Die Instruktionen und Adressmodi sollten Ihnen bereits bekannt sein. Benutzen Sie die Übung als Prüfpunkt, ob Sie alle notwendigen Instruktionen und Adressmodi kennen.

Hilfsmittel/Templates

- Diese Vorlesungsfolien und Beispiele
- ARM Cheat Sheet: https://github.com/oowekyala/arm-cheatsheet/blob/master/arm-cheatsheet.pdf

Zeitaufwand

- **etwa 20 Minuten**

Bitfelder

Datenstrukturen mit Elementen unterschiedlicher Größe

Die **Peripherie-Register** unseres Raspberry Pi (und jedes Microcontrollers) werden in Speicherbereiche abgebildet. Einzelne **Register** sind dabei in oft **in Bitfelder zerlegbar**. Diese Felder aus Bitfelder bilden ihrerseits Datenstrukturen.

Beispiel: GPFSEL-register

31											0	
0	0	GPFSEL09	GPFSEL08	GPFSEL07	GPFSEL06	GPFSEL05	GPFSEL04	GPFSEL03	GPFSEL02	GPFSEL01	GPFSEL00	0x7E200000
0	0	GPFSEL19	GPFSEL18	GPFSEL17	GPFSEL16	GPFSEL15	GPFSEL14	GPFSEL13	GPFSEL12	GPFSEL11	GPFSEL10	0x7E200004
						...						
0	0	GPFSEL49	GPFSEL48	GPFSEL47	GPFSEL46	GPFSEL45	GPFSEL44	GPFSEL43	GPFSEL42	GPFSEL41	GPFSEL40	0x7E200010

Hochsprache wie C bieten uns passende Beschreibungsmuster für diese Datenstrukturen. **Assembler** bietet **keine Funktionen für den Zugriff auf solche Strukturen**.

Trotzdem können **Zugriffsmuster für den Zugriff**

- **auf Register,** organisiert als im Feld
- **auf Bitfelder** im Register, organisiert in nicht an Bytes ausgerichteten Bereichen

hergeleitet werden.

Vorgriff auf C: Darstellung von Felder und Bitfelder in Hochsprachen

C bietet uns praktische Beschreibungs- und Zugriffsmittel für Strukturen und Felder.

Für unregelmäßige Datenstrukturen unterscheiden wir zwischen

- Der Bitfeldkodierung eines Registers
- Einem Feld aus Datenstrukturen

```c
// Dieser Code dient nur der Darstellung von C,
//    er wird nicht auf unserem Pi in dieser Form funktionieren !!!

#define GPIO_PERIPHERAL_BASE_ADDRESS     0x7E200000
#define GPIO_PERIPHERAL_FSEL_OFFSET         0x00000000
#define GPIO_FSEL_MODE_OUTPUT         0b000;

typedef struct gpio_fsel_reg_t {
          unsigned FSELx0 : 3;
          unsigned FSELx1 : 3;
          unsigned FSELx2 : 3;
          unsigned FSELx3 : 3;
          unsigned FSELx4 : 3;
          unsigned FSELx5 : 3;
          unsigned FSELx6 : 3;
          unsigned FSELx7 : 3;
          unsigned FSELx8 : 3;
          unsigned FSELx9 : 3;
          unused           : 2;
} gpio_fsel_reg;

volatile gpio_fsel_reg *gpio_fsel_regs[5] = (unsigned int *)
GPIO_BASE_ADDRESS;

...
gpio_fsel_regs[1].FSELx3 = GPIO_FSEL_MODE_OUTPUT;
...
```

Zugriff auf Feldelemente

GPFSEL-register

31									8	6	5	3	2	0	
0	0	GPFSEL09	GPFSEL08	GPFSEL07	GPFSEL06	GPFSEL05	GPFSEL04	GPFSEL03	GPFSEL02		GPFSEL01		GPFSEL00		0x7E200000
0	0	GPFSEL19	GPFSEL18	GPFSEL17	GPFSEL16	GPFSEL15	GPFSEL14	GPFSEL13	GPFSEL12		GPFSEL11		GPFSEL10		0x7E200004
						...									
0	0	GPFSEL49	GPFSEL48	GPFSEL47	GPFSEL46	GPFSEL45	GPFSEL44	GPFSEL43	GPFSEL42		GPFSEL41		GPFSEL40		0x7E200010

Beispiel: Index GPFSEL[i] mit i := Nummer des GPIO

- Feldgröße 4
- 3 bits/FSEL → 10 Felder
- Berechnung der **Addresse des Feldelements**: $BASISADDRESSE_{GPIOS} + \left\lfloor \frac{\text{NUMMER}_{GPIO}}{10} \right\rfloor \cdot 4_{Byte/Feldelement}$
- Bestimmen der **Position des Bitfeldes im Feldelement**: $(\text{NUMMER}_{GPIO} \ \% \ 10) \ * 3_{Bit/Strukturelement}$

Zugriff auf Elemente in Bitfelder

Bitmasken

Wir wollen bei lesendem und schreibendem Zugriff NUR das entsprechende Bitfeld verändern.

Schreiben:

- Unseren neuen Wert für das Bitfeld an die korrekte Position schieben
- Maske zum Ausblenden der existenten Bits an der passenden Position erstellen
- Feldelement lesen

- Position des Bitfeldes im Register
  → Schiebeoperationen
  Beispiel FSEL: index * 3 Bit
- Größe des Bitfeldes
  → Definition einer Zugriffsmaske
  Beispiel FSEL: 0b111

bekannt sein.

Zugriff auf Elemente in Bitfelder: Berechnungen DIV und MOD

Bitmasken

ARM besitzt keine DIV oder MOD Instruktionen.

- Beide sind außerhalb von Digitalen Signalprozessoren (DSPs) unüblich

Beide Operationen können zusammen in einer Schleife umgesetzt werden.

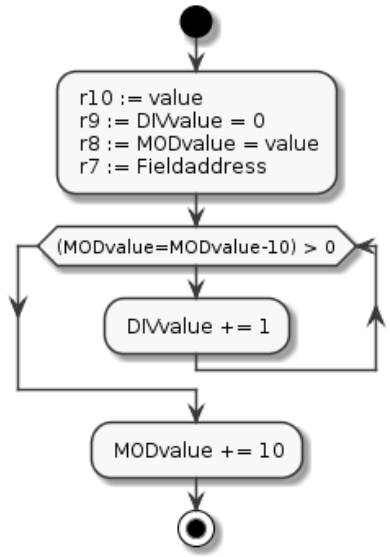
z.B.div=27/10 , mod=27%10

Init: div = 0, mod = 0

Test: mod < 0;

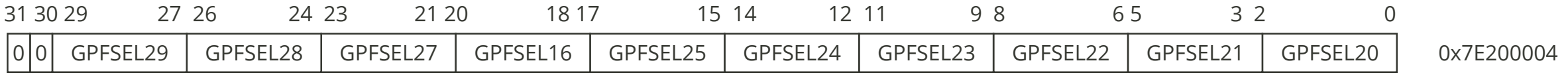
Loop 1: mod = 27-10 = 17
div = div+1=1
Loop 2: mod = 17-10 = 7
div = div+1 = 2
Loop 3: mod = 7 – 10 = -3
div = div +1 = 3

```
structs_example_calcFieldAddr: // template: Sektion
    structs_example_calcFieldAddr_divloop_init:
        // while (r9:div=0, r0:mod=27)
    mov r0, r10;
    mov r9, #0
    structs_example_calcFieldAddr_divloop_test:
      // Subtrahiere, solange r0 positiv ist
      subs r0, r0, #10
      bmi  structs_example_calcFieldAddr_divloop_done
      add  r9, r9, #1
      b    structs_example_calcFieldAddr_divloop_test
    structs_example_calcFieldAddr_divloop_done:
    add r0, r0, #10 // r0 ist ins negative gerutsch,
korrigiere letzte subraktion
    mov r8, r0          //r8 = mod 27%10

    ldr r0, =FAKE_GPIOFSEL_BASE
    add r7, r0, r9, LSL #2   // r9= 4*(GPIONR/10) +
FAKE_GPIOFSEL_BASE
  structs_example_calcFieldAddr_done:
```



Lesen von Elemente in Bitfeldern

Zugriff auf Elemente in Bitfelder



- Feldelement lesen
- Gesuchtes Bitfeld nach Rechts ausrichten (für nachfolgende Vergleiche oder arithmetische Operationen)
  - Abhängig von Bitfeldern/Bits rechtsseitig unseres Felds
- Nicht interessante Bitfelder per AND ausblenden/auf 0 Setzen mit maske z.B 0b111

Beispiel: GPIO27

```
Lesen:      0b00 110 101 101 010 101 011 011 100 110 110

Ausrichten: 0b00 110 101 101 010 101 011 011 100 110 110 >> 3* 27%10
            0b00 000 000 000 000 000 000 000 110 101 101

Maske:      0b00 000 000 000 000 000 000 000 000 000 111
AND:        0b00 000 000 000 000 000 000 000 110 101 101

            0b00 000 000 000 000 000 000 000 000 000 101
```

Schreiben von Bitfeldern: Mit Copy-Modify-Update

Zugriff auf Elemente in Bitfelder

Wir dürfen nur unser GPIO verändern, der Rest wird eventuell gerade benutzt…
Vorgehen:

- Maske bilden
- Unseren Wert an die passende Stelle Schieben
- Lese aktuellen Wert aus dem Register
- Ändern von Bits mit den Instruktionen
  1. Bitwise clear:
     BIC Rd, Ra, <operand2>
     // Rd ← Ra AND (NOT <operand2>)
  2. Setze unsere Bits:
     ORR Rd, Rs, <operand2>
     // Rd ← Ra OR <operand2>
- Schreibe den Wert in das Register

Beispiel: GPIO27

```
Maske: r1 = 0b00 000 000 000 000 000 000 000 000 000 111 << 3*GPIO%10
            0b00 000 000 111 000 000 000 000 000 000 000

Wert:  r2 = 0b00 000 000 000 000 000 000 000 000 000 110 << 3*GPIO%10
            0b00 000 000 110 000 000 000 000 000 000 000

Lesen: r0 = 0b00 110 101 101 010 101 011 011 100 110 110

BIC r0,r0, #0b00 000 000 111 000 000 000 000 000 000 000
  (NOT     #0b11 111 111 000 111 111 111 111 111 111 111)
  (r0 AND   0b00 110 101 000 010 101 011 011 100 110 110)

ORR r0,r0, #0b00 000 000 110 000 000 000 000 000 000 000
            0b00 110 101 110 010 101 011 011 100 110 110

Schreiben in den Speicher
```

Kritische Abschnitt

Zusammenfassung

Zusammenfassung

Felder

- sind **zusammenhängende Speicherbereiche** aus **Feldelemente gleichen Typ und gleicher Größe**
- Die Elemente werden durch Indizes angesprochen
- In Assembler müssen die Indizes in Offsets zur Basisadressen umgerechnet werden
  - ADRESSE(INDEX) = BASIS + INDEX * ELEMENT_GRÖSSE
  - ADRESSE(INDEX1,INDEX2) = BASIS + (INDEX1 * DIMENSION2 + INDEX2) * ELEMENT_GRÖSSE

Strukturen

- sind **zusammenhängende** Speicherbereiche aus **Elementeen unterschiedliche Größe** und/oder **Bitfeldern** bestehen
- Elementstruktur wird durch (feste) Offsets zur Basisadresse definiert.
  - Bei **Registern** brauchen wir lediglich die **Basisaddresse** + das **konstante Feldoffset**
  - Bei **Bitfeldern** benötigen wir zusätzlich die **Bitposition für Schiebeoperationen**
- Lesen: Read, Shift, Mask
- Schreiben: Mask, Prepare Value, Copy, Update, Modify

Übungsaufgabe: Zugriffsbibliothek für GPIOs Teil 1

Bauen Sie das Vorlesungsbeispiel des GPIO-Zugriffs zu einer generischen Funktion

- Zum Ändern von GPFSEL mit `gpfun(GPIObasisadresse, GPIONr, FUNC)`
- Zum Ändern von GPSET mit `gpset(GPIObasisadresse, GPIONr)`
- Zum Ändern von GPCLR mit `gpclr (GPIObasisadresse, GPIONr)`

Aus. Benutzen Sie einen von Ihnen definierten Speicherbereich in .data zum Testen der Interaktion.

Praxisbezug

– Wir werden nächstes mal diese Funktionen gemeinsam herleiten und besprechen, nachdem Memory Mapping umgesetzt wurde – wir erhalten dabei eine neue, erst zur Laufzeit bekannte Basisadresse für die Peripherie

Hilfsmittel/Templates

– Diese Vorlesungsfolien und Beispiele

Zeitaufwand

– **etwa 60 Minuten**

Ausblick

Das nächste Mal: **GPIO und Memory Mapping**

Mit was für Speicheradressen hantieren wir bislang?

→ Wir können aus unserem Addressraum offensichtlich nicht auf die Peripherie zugreifen

Welche Rolle spielen

- **Physische Speicheradressen** aus dem Datenblatt
- **Abgebildete Speicheradresse** (Kernel)
- **Seiten-Abgebildeter Speicher** (Paged Memory)

Wie funktioniert **Memory Mapping**?
Wie können wir Bereiche des abgebildeten Speichers in unseren zur Laufzeit benutzten Adressraum holen?

Umsetzung einer rudimentären Zugriffsbibliothek für die Bedienung der GPIOs (bcm2836_gpiolib)

Prof. Leon Urbas , leon.urbas@tu-dresden.de

Vielen Dank für Ihre Aufmerksamkeit

Beispielcode der Slides auf github.com

Lehrveranstaltung MRT1/2 auf OPAL

Quellen

(Aho 2000) Aho, A. V. (2000). *Data Structures and Algorithms*. Pearson Education Singapore.

(Stroustrup 2013) Stroustrup, B. (2013). *The C++ Programming Language: The C++ Programm Lang_p4*. Addison-Wesley.

(Warren 2013) Warren, H. S. (2013). *Hacker's delight* (Second Edition). Upper Saddle River, Addison-Wesley.