

Echtzeitsysteme - Übungen

System-Software-Entwicklung

Sommersemester 2024

Eva Dengler Peter Wägemann

Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)

Lehrstuhl Informatik 4 (Systemsoftware)

<https://sys.cs.fau.de>



Lehrstuhl für Verteilte Systeme
und Betriebssysteme



Friedrich-Alexander-Universität
Technische Fakultät

Wiederholung: Stack-Aufbau

Standards in der Softwareentwicklung

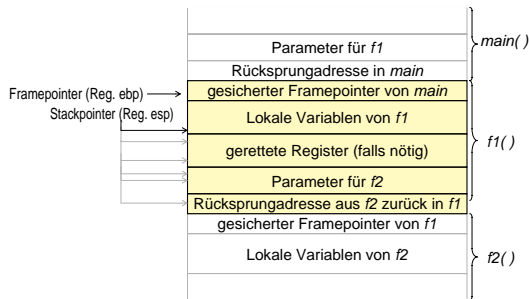
Verwendung von Fließkommazahlen

Überblick: Toolchain

Hardware

Wiederholung: Stack-Aufbau

```
1 int main() {  
2     int a, b, c;  
3  
4     a = 10;  
5     b = 20;  
6  
7     f1(a, b);  
8     return a;  
9 }  
10 }
```



■ Stack-Frame zur Verwaltung des Stacks

- Lokale Variablen
- Funktionsparameter
- Rücksprungadressen

■ Register zur Verwaltung des Stacks

- Stackpointer: Zeigt auf nächsten freien Speicherbereich
- Framepointer: Zeigt auf Beginn des aktuellen Stackframes

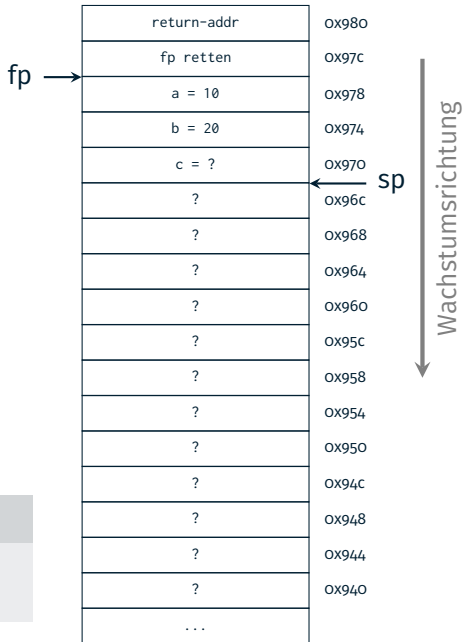
Ablauf Funktionsaufruf

```
int main(void) {  
    int a, b , c;  
    a = 10;  
    b = 20;  
    f1(a, b);  
    return a;  
}
```



```
int f1(int x, int y) {  
    int i[3];  
    int n;  
    x++;  
    n = f2(x);  
    return n;  
}
```

```
int f2(int z) {  
    int m;  
    m = 100;  
    return z+1;  
}
```



Variablenzugriff

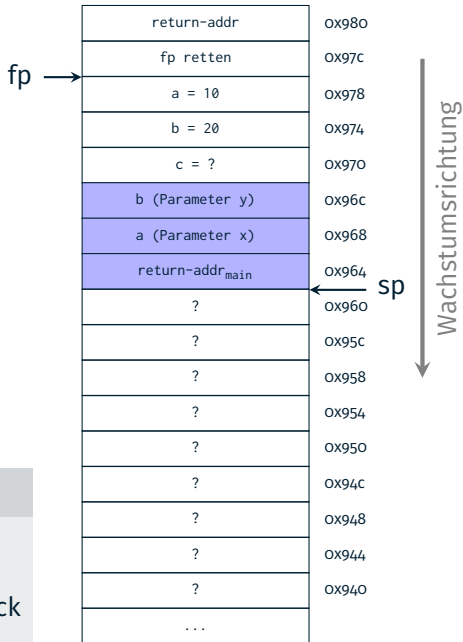
- $&a = fp - 4$

Ablauf Funktionsaufruf

```
int main(void) {  
    int a, b , c;  
    a = 10;  
    b = 20;  
    f1(a, b);  
    return a;  
}
```

```
int f1(int x, int y) {  
    int i[3];  
    int n;  
    x++;  
    n = f2(x);  
    return n;  
}
```

```
int f2(int z) {  
    int m;  
    m = 100;  
    return z+1;  
}
```



Funktionsaufruf

- Parameter auf Stack
- Rücksprungadresse auf Stack

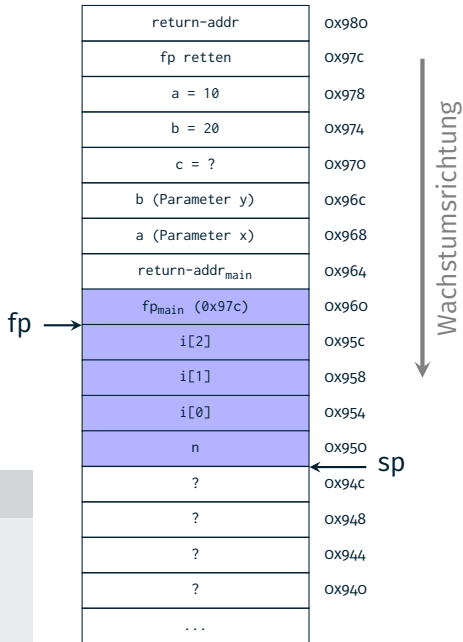
Ablauf Funktionsaufruf

```
int main(void) {  
    int a, b, c;  
    a = 10;  
    b = 20;  
    f1(a, b);  
    return a;  
}  
  
int f1(int x, int y) {  
    int i[3];  
    int n;  
    x++;  
    n = f2(x);  
    return n;  
}  
  
int f2(int z) {  
    int m;  
    m = 100;  
    return z+1;  
}
```

Parameterzugriff

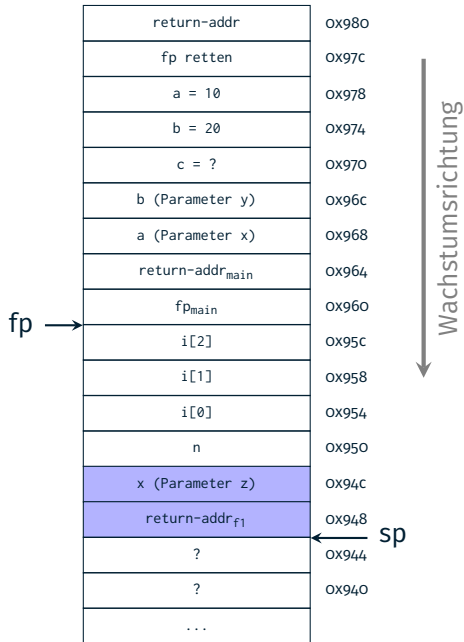
■ $&x = fp + 8$

? Schreiben an $i[4]$?



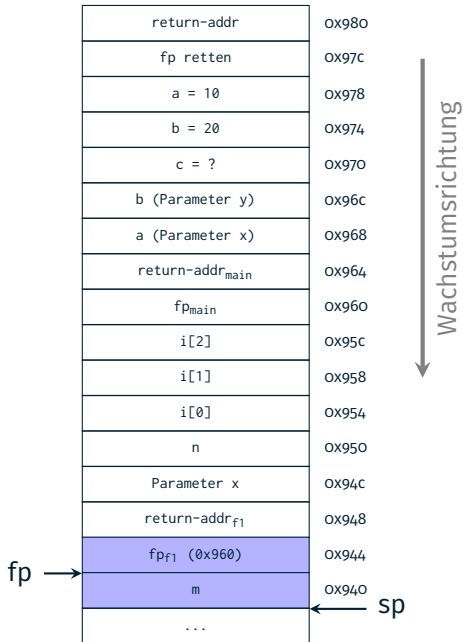
Ablauf Funktionsaufruf

```
int main(void) {  
    int a, b, c;  
    a = 10;  
    b = 20;  
    f1(a, b);  
    return a;  
}  
  
int f1(int x, int y) {  
    int i[3];  
    int n;  
    x++;  
    n = f2(x);  
    return n;  
}  
  
int f2(int z) {  
    int m;  
    m = 100;  
    return z+1;  
}
```



Ablauf Funktionsaufruf

```
int main(void) {  
    int a, b, c;  
    a = 10;  
    b = 20;  
    f1(a, b);  
    return a;  
}  
  
int f1(int x, int y) {  
    int i[3];  
    int n;  
    x++;  
    n = f2(x);  
    return n;  
}  
  
int f2(int z) {  
    int m;  
    m = 100;  
    return z+1;  
}
```



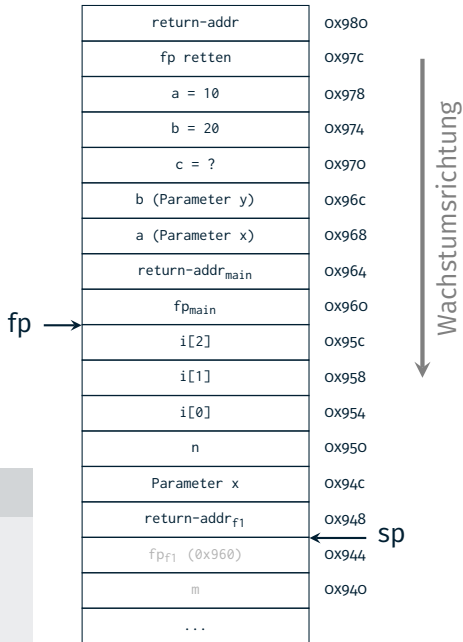
Ablauf Funktionsaufruf

```
int main(void) {  
    int a, b, c;  
    a = 10;  
    b = 20;  
    f1(a, b);  
    return a;  
}  
  
int f1(int x, int y) {  
    int i[3];  
    int n;  
    x++;  
    n = f2(x);  
    return n;  
}  
  
int f2(int z) {  
    int m;  
    m = 100;  
    return z+1;  
}
```



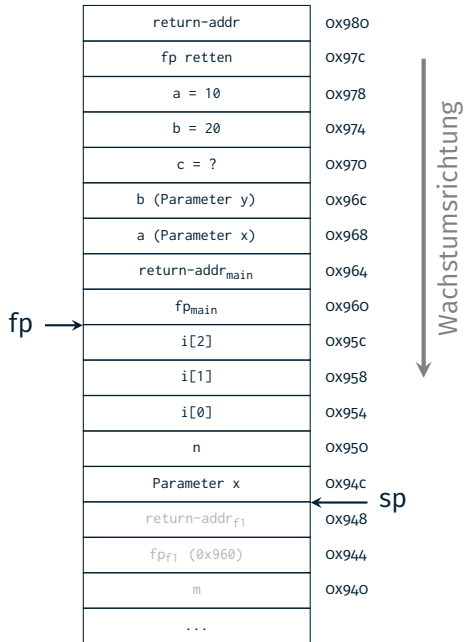
Stackframe abräumen

- $sp = fp$
- $fp = pop(sp)$



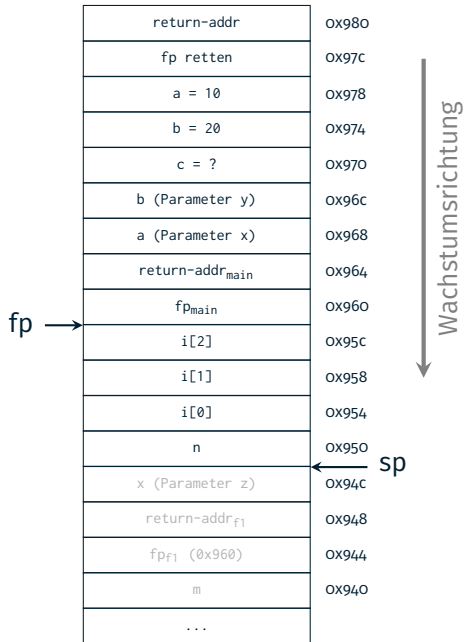
Ablauf Funktionsaufruf

```
int main(void) {  
    int a, b, c;  
    a = 10;  
    b = 20;  
    f1(a, b);  
    return a;  
}  
  
int f1(int x, int y) {  
    int i[3];  
    int n;  
    x++;  
    n = f2(x);  
    return n;  
}  
  
int f2(int z) {  
    int m;  
    m = 100;  
    return z+1;  
}
```



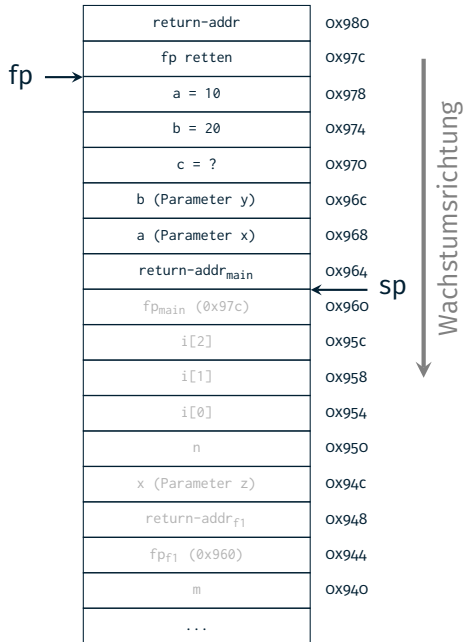
Ablauf Funktionsaufruf

```
int main(void) {  
    int a, b, c;  
    a = 10;  
    b = 20;  
    f1(a, b);  
    return a;  
}  
  
int f1(int x, int y) {  
    int i[3];  
    int n;  
    x++;  
    n = f2(x);  
    return n;  
}  
  
int f2(int z) {  
    int m;  
    m = 100;  
    return z+1;  
}
```



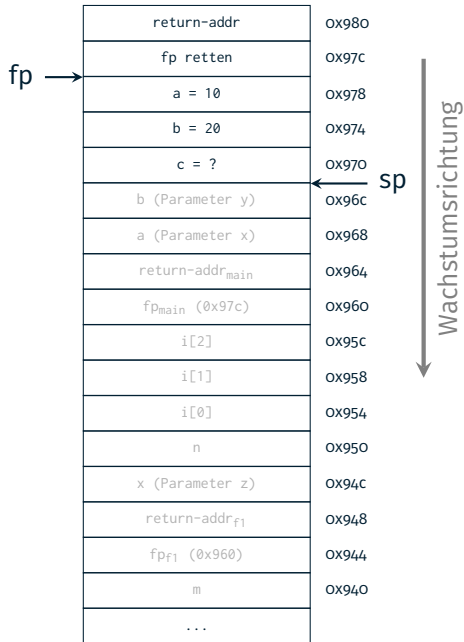
Ablauf Funktionsaufruf

```
int main(void) {  
    int a, b, c;  
    a = 10;  
    b = 20;  
    f1(a, b);  
    return a;  
}  
  
int f1(int x, int y) {  
    int i[3];  
    int n;  
    x++;  
    n = f2(x);  
    return n;  
}  
  
int f2(int z) {  
    int m;  
    m = 100;  
    return z+1;  
}
```



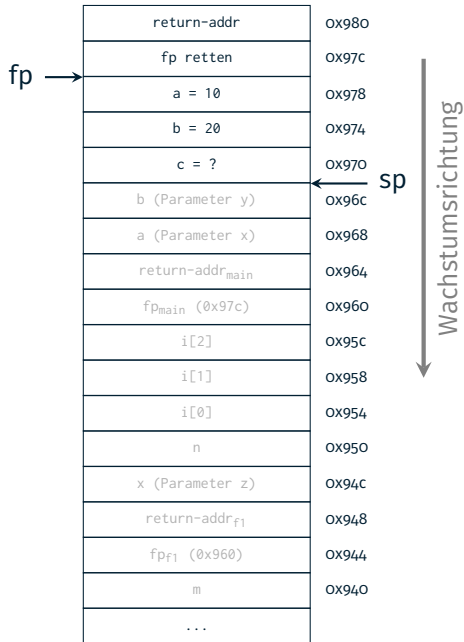
Ablauf Funktionsaufruf

```
int main(void) {  
    int a, b, c;  
    a = 10;  
    b = 20;  
    f1(a, b);  
    return a;  
}  
  
int f1(int x, int y) {  
    int i[3];  
    int n;  
    x++;  
    n = f2(x);  
    return n;  
}  
  
int f2(int z) {  
    int m;  
    m = 100;  
    return z+1;  
}
```



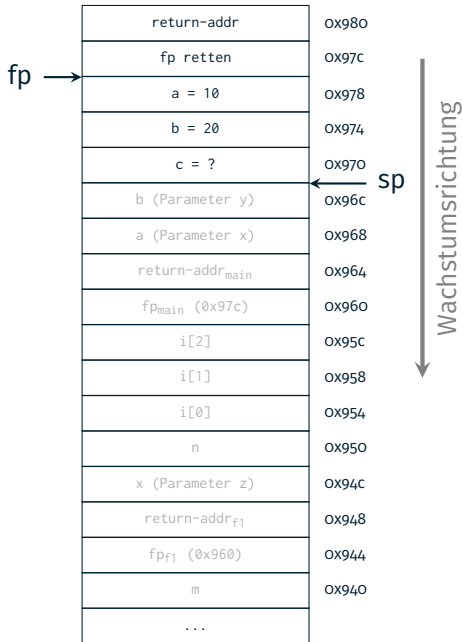
Ablauf Funktionsaufruf

```
int main(void) {  
    int a, b, c;  
    a = 10;  
    b = 20;  
    f1(a, b);  
    return a;  
} →  
  
int f1(int x, int y) {  
    int i[3];  
    int n;  
    x++;  
    n = f2(x);  
    return n;  
}  
  
int f2(int z) {  
    int m;  
    m = 100;  
    return z+1;  
}
```



Ablauf Funktionsaufruf

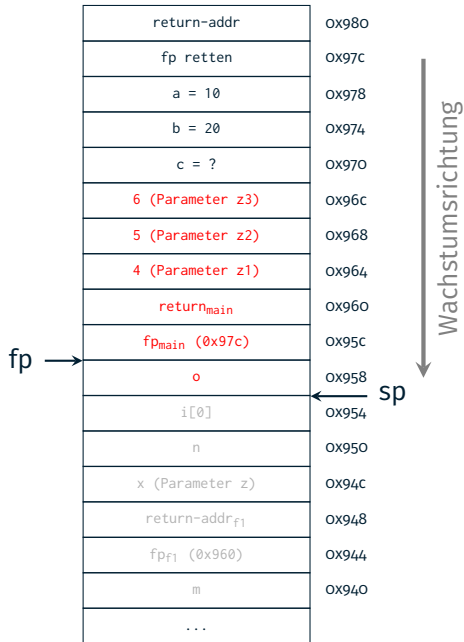
```
int main(void) {  
    int a, b, c;  
    a = 10;  
    b = 20;  
    f1(a, b);  
    f3(4, 5, 6);  
    return a;  
}  
  
int f3(int z1, int z2, int z3) {  
    int o;  
    return o;  
}
```



Was passiert bei weiterem Funktionsaufruf?

Ablauf Funktionsaufruf

```
int main(void) {  
    int a, b, c;  
    a = 10;  
    b = 20;  
    f1(a, b);  
    f3(4, 5, 6);  
    return a;  
}  
  
int f3(int z1, int z2, int z3) {  
    int o;  
    return o;  
}
```



Was passiert bei weiterem Funktionsaufruf?

Wiederholung: Stack-Aufbau

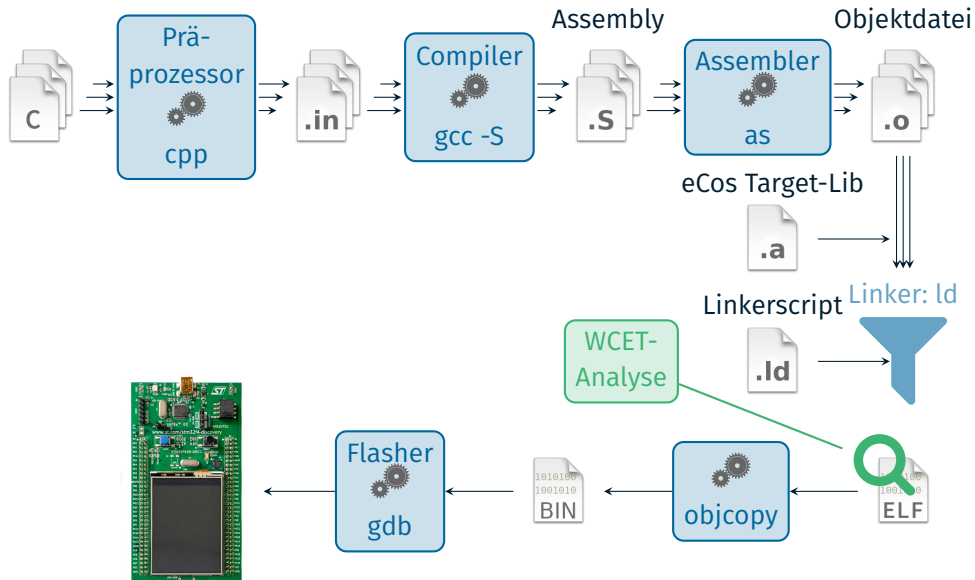
Standards in der Softwareentwicklung

Verwendung von Fließkommazahlen

Überblick: Toolchain

Hardware

EZS-Toolchain



INTERNATIONAL
STANDARD

ISO/IEC
9899

Second edition
1999-12-01

Programming languages — C

Langages de programmation — C

Processed and adopted by ASC the National Committee for Information Technology Standards (NCITS) and approved by ANSI as an American National Standard.

Date of ANSI Approval: 5/22/2000

Published by American National Standards Institute,
11 West 42nd Street, New York, New York, 10036

Copyright 2000 by Information Technology Industry Council
(ITI). All rights reserved.

These materials are subject to copyright claims of International Standardization Organization (ISO), International Electrotechnical Commission (IEC), American National Standards Institute (ANSI), and Information Technology Industry Council (ITI). Not for resale. No part of this publication may be reproduced in any form, including an electronic retrieval system, without the prior written permission of ITI. All requests pertaining to this standard should be submitted to ITI, 1250 Eye Street NW, Washington, DC 20005.

Printed in the United States of America

Reference number
ISO/IEC 9899:1999(E)



© ISO/IEC 1999

- Mehrere Iterationen:
C89, C99, C11, C18
- Früher ANSI, heute ISO/IEC Standards:
 - ANSI X3.159-1989
 - ISO/IEC 9899:1990
 - ...
- Unabhängiger Standard, von ISO entwickelt
- Beschreibt C Syntax & Semantik

6.5.5 Multiplicative operators

Syntax

multiplicative-expression:
cast-expression
multiplicative-expression * *cast-expression*
multiplicative-expression / *cast-expression*
multiplicative-expression % *cast-expression*

Constraints

Each of the operands shall have arithmetic type. The operands of the % operator shall have integer type.

Semantics

The usual arithmetic conversions are performed on the operands.

The result of the binary * operator is the product of the operands.

The result of the / operator is the quotient from the division of the first operand by the second; the result of the % operator is the remainder. In both operations, if the value of the second operand is zero, the behavior is undefined.

When integers are divided, the result of the / operator is the algebraic quotient with any fractional part discarded.⁹⁰⁾ If the quotient a/b is representable, the expression $(a/b)*b + a\%b$ shall equal a .

Source: ISO/IEC 9899:TC3, S.94

6.5.5 Multiplicative operators

Syntax

multiplicative-expression:
cast-expression
multiplicative-expression * *cast-expression*
multiplicative-expression / *cast-expression*
multiplicative-expression % *cast-expression*

Constraints

Each of the operands shall have arithmetic type. The operands of the % operator shall have integer type.

Semantics

The usual arithmetic conversions are performed on the operands.

The result of the binary * operator is the product of the operands.

The result of the / operator is the quotient from the division of the first operand by the second; the result of the % operator is the remainder. In both operations, if the value of the second operand is zero, the behavior is undefined.

When integers are divided, the result of the / operator is the algebraic quotient with any fractional part discarded.⁹⁰⁾ If the quotient a/b is representable, the expression $(a/b)*b + a\%b$ shall equal a .

Source: ISO/IEC 9899:TC3, S.94

6.5.5 Multiplicative operators

Syntax

multiplicative-expression:
cast-expression
multiplicative-expression * *cast-expression*
multiplicative-expression / *cast-expression*

3.4.3

undefined behavior

behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements

NOTE Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).

EXAMPLE An example of undefined behavior is the behavior on integer overflow.

Source: ISO/IEC 9899:TC3, S.4

Source: ISO/IEC 9899:TC3, S.94

Wiederholung: Stack-Aufbau

Standards in der Softwareentwicklung

Verwendung von Fließkommazahlen

Überblick: Toolchain

Hardware

Frage #1

Zu was wird $7/2$ ausgewertet?

1. 3.5
2. 3
3. nicht definiert in C

Frage #1

Zu was wird $7/2$ ausgewertet?

1. 3.5
2. 3
3. nicht definiert in C

Erklärung

- Standard-Typ für Ganzzahlen ist `int`
- Rest verschwindet bei Ganzzahl-Division

Frage #2

Zu was wird 2/7 ausgewertet?

- 1.
- 0
- nicht definiert in C

Frage #2

Zu was wird 2/7 ausgewertet?

- 1
- 0
- nicht definiert in C

Erklärung

- Standard-Typ für Ganzzahlen ist `int`
- Rest verschwindet bei Ganzzahl-Division

Frage #3

Zu was wird $7/2$. ausgewertet?

1. immer noch 3
2. 0
3. 3.5

Frage #3

Zu was wird $7/2$. ausgewertet?

1. immer noch 3
2. 0
3. 3.5

Erklärung

- $2.0 == 2.$ \rightsquigarrow `double` auf der rechten Seite
- 7 wird in diesem Ausdruck als `double` behandelt, auch linke Seite
- Division zweier `double` Werte

Frage #5

Zu was wird $1/2 + 1/2$ ausgewertet?

1. nicht definiert
2. \emptyset
3. 1 (dank Compileroptimierung)

Frage #5

Zu was wird $1/2 + 1/2$ ausgewertet?

1. nicht definiert
2. 0
3. 1 (dank Compileroptimierung)

Erklärung

- $int_1 / \langle \text{größerer } int_2 \rangle \rightsquigarrow 0 + 0 = 0$
- Compileroptimierung nicht C-Konform

Frage #6

Zu was wird $2 * M_PI$ ausgewertet?

1. 6
2. ungefähr 6.28
3. 6.283185307179586476925286766559005768394338798750...

Frage #6

Zu was wird `2 * M_PI` ausgewertet?

1. 6
2. ungefähr 6.28
3. 6.283185307179586476925286766559005768394338798750...

Erklärung

- `M_PI` \rightsquigarrow `double`
- `double` Standard-Typ, außer zusätzliches Literal (3.14 f)
- Begrenzter Wertebereich:
6.28318530717958600000000000000000

Frage #7

```
1 double a = 0.1;
2 double b = 0.2;
3
4 float aa = 0.1;
5 float bb = 0.2;
6
7 if (a+b == aa+bb){
8     ezs_printf("equal\n");
9 }else{
10     ezs_printf("unequal: %.30f != %.30f\n", (a+b), (aa+bb));
11 }
```

Was wird ausgegeben?

1. equal
2. unequal...

Frage #7

```
1 double a = 0.1;
2 double b = 0.2;
3
4 float aa = 0.1;
5 float bb = 0.2;
6
7 if (a+b == aa+bb){
8     ezs_printf("equal\n");
9 }else{
10     ezs_printf("unequal: %.30f != %.30f\n", (a+b), (aa+bb));
11 }
```

Was wird ausgegeben?

1. equal
2. unequal...

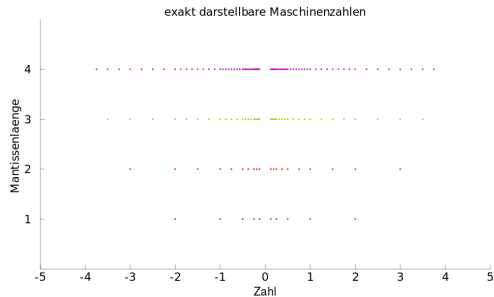
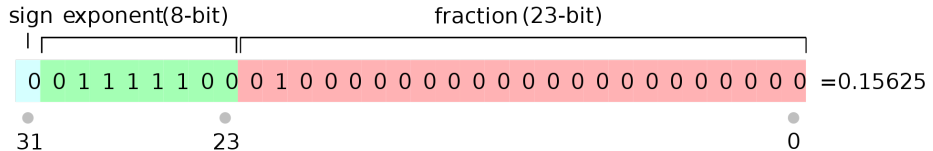
Fließkomma-Arithmetik

```
1 double a = 0.1;
2 double b = 0.2;
3
4 float aa = 0.1;
5 float bb = 0.2;
6
7 if (a+b == aa+bb){
8     ezs_printf("equal\n");
9 }else{
10    ezs_printf("unequal: %.30f != %.30f\n", (a+b), (aa+bb));
11 }
```

```
1 /* Ausgabe:
2 unequal:
3 0.3000000000000000000000000000000000000000000000000 !=
4 0.3000000119209290000000000000000000
5 */
```

- Angenommen die Einheit ist Sekunden
 - 11,9 ns Fehler durch *einzelne Berechnung*
 - Kumulation der Rundungsfehler

Begrenzte Wertebereiche – IEEE 754



■ IEEE 754

- `sizeof(float) == 4`
- `sizeof(double) == 8`

Probleme begrenzter Wertebereiche

- *What Every Computer Scientist Should Know About Floating-Point Arithmetic* [1]
- Rundungsfehler & Überläufe äußerst kritisch in *harten Echtzeitsystemen*
- Konvertierungen zwischen Größeneinheiten (sec_to_nanosec: * 1e9)
- Vermeidung des Wechsels von Größeneinheiten
- Verwendung von Festkomma-Arithmetik \rightsquigarrow VEZS
- Integer-Division ist *kein sicherer Ausweg*
 \Rightarrow *Sorgfalt bei arithmetischen Operationen in begrenzten Wertebereichen*

Wahl des Datentyps bei Berechnung des Sinus-Wertes

- Harmonische Schwingung¹: $y(t) = y_0 \cdot \sin(\omega t + \varphi_0)$ und $\omega = 2\pi f$

```
1 #define TYPE {int|double|float} ?
2 ...
3 TYPE compute_sinus(OTHER_TYPE real_time) {
4
5     TYPE f      = ...
6     TYPE omega = 2 * M_PI * f;
7     ...
8     ... sin(omega * real_time) // oder doch sinf(omega * real_time)?
9     ...
10 }
```

¹https://de.wikipedia.org/wiki/Schwingung#Harmonische_Schwingung

Wahl des Datentyps bei Berechnung des Sinus-Wertes

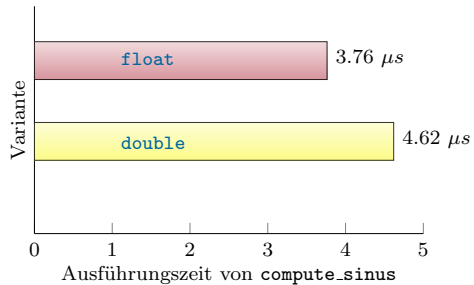
- Harmonische Schwingung¹: $y(t) = y_0 \cdot \sin(\omega t + \varphi_0)$ und $\omega = 2\pi f$

```
1 #define TYPE {int|double|float} ?
2 ...
3 TYPE compute_sinus(OTHER_TYPE real_time) {
4
5     TYPE f      = ...
6     TYPE omega = 2 * M_PI * f;
7     ...
8     ... sin(omega * real_time) // oder doch sinf(omega * real_time)?
9     ...
10 }
```

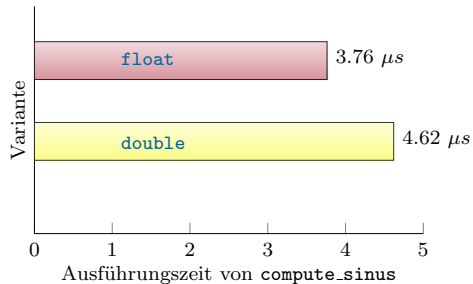
- *float oder double für Realzeit sinnvoll? Was ist OTHER_TYPE?*
- *Laufzeit von compute_sinus()?*

¹https://de.wikipedia.org/wiki/Schwingung#Harmonische_Schwingung

Vergleich der Laufzeiten

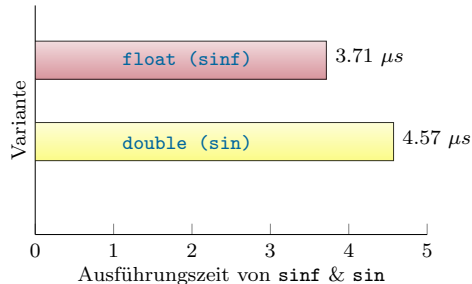
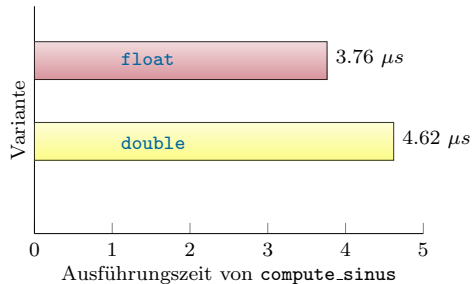


Vergleich der Laufzeiten



- Laufzeitzuwachs um 23 % bei Wechsel `float` → `double`
- Soft Float? Hard Float? hier: Soft Float

Vergleich der Laufzeiten



- Laufzeitzuwachs um 23 % bei Wechsel `float` → `double`
- Soft Float? Hard Float? hier: Soft Float
- Noch mehr Optimierungspotential? Wo wird die Laufzeit verbraucht?
 - 99 % der Gesamtlaufzeit für `sinf` und `sin`
- Wahl des Datentyps in Abh. der Wortbreite (32-Bit Cortex-M4, 8-Bit AVR)
- Spezialbibliothek für Signalverarbeitung mit Integer-Arithmetik
- Spezielle Hardware-Einheiten zur Signalverarbeitung

Wiederholung: Stack-Aufbau

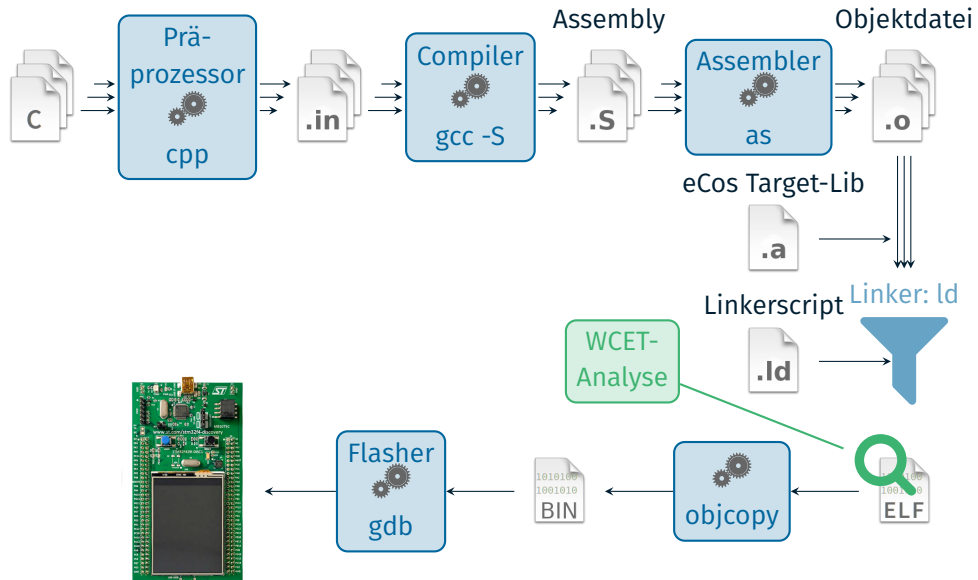
Standards in der Softwareentwicklung

Verwendung von Fließkommazahlen

Überblick: Toolchain

Hardware

EZS-Toolchain



Quellcode

```
1 #define FOO 42
2
3 #include "example.h"
4
5 #if defined(FOO)
6     int i = FOO;
7 #else
8     int i = 0;
9 #endif
```

Expandiert

```
1 // Inhalt example.h
2 void example();
3
4 int i = 42;
```

Präprozessor

- Vorverarbeitungsschritt vor der Übersetzung
 - Konfigurationsabhängiger Code `#if(def)`
 - Definierbare Konstanten und Makros `#define`
 - Auflösen von `#include`-Direktiven
- Reine Zeichenersetzung/Textmanipulation

Quellcode

```
1 volatile extern int i;  
2 int j = 42;  
3  
4 int main(int argc, ...)  
5 {  
6     i = 0;  
7     if(argc % 2) {  
8         i = 1;  
9     }  
10    return i + j;  
11 }
```

Assembly

```
...  
ldr r3, [fp, #-8]  
and r3, r3, #1  
cmp r3, #0  
beq .L2  
ldr r3, .L4  
mov r2, #1  
str r2, [r3]  
.L2:  
...
```

Übersetzer

- Interpretation des Quelltexts gemäß Semantik laut Standard
- Umwandlung in Befehlssatz der Zielplattform
- Aufrufe gemäß Application Binary Interface (ABI)
- **Optimierung des Kompilats**

Beispiel: Schleifenaufrollen

Unoptimiert

```
1 for(i = 0; i < 40; i++) {  
2   x++;  
3 }  
4 x++;  
5 x++;
```

Größenoptimiert

```
1 for(i = 0; i < 42; i++) {  
2   x++;  
3 }
```

Laufzeitverhalten

- Optimierungen verändern Kontrollflussstrukturen
 - Schleifenaufrollen (siehe oben)
 - Schleifentauschen (loop interchange)
 - Schleifenneigen (loop skewing)
 - if-conversion
 - ...

↪ invalidiert z.T. Annotationen und Annahmen über Laufzeitverhalten

Assembly

```
...  
ldr r3, [fp, #-8]  
and r3, r3, #1  
cmp r3, #0  
beq .L2  
ldr r3, .L4  
mov r2, #1  
str r2, [r3]  
.L2:  
...
```

Objektdatei

```
...  
e51b3008  
e2033001  
e3530000  
0a000002  
e59f3028  
e3a02001  
e5832000  
...
```

Assembler

- Umwandlung der textuellen Repräsentation in Maschinencode (binär)
- 1:1 Übersetzung
- z.T. Macroassembler: Komplexbefehle zu Instruktionsfolge

Objektdatei

```
$ nm test.o
          U i      # extern int i
00000000 D j
00000000 T main

$ nm i.o
00000004 C i      # Definition int i = 0
```

Binary

```
$ nm test.elf
00018a84 B i
00018634 D j
000081ec T main
...
```

Linker

- Variablen/Funktionen über Objektdateien verteilt
- ↪ Zusammenführung der Funktionen und Variablen aus Objektdateien
- ↪ Vergabe globaler Adressen gemäß Konfiguration
- ↪ Auflösen der Adressen im Code

Flasher: Speicherorganisation auf einem Mikrocontroller

```
int a; // a: global, uninitialized
int b = 1; // b: global, initialized
const int c = 2; // c: global, const

void main() {
    static int s = 3; // s: local, static, initialized
    int x, y; // x: local, auto; y: local, auto
    char* p = malloc( 100 ); // p: local, auto; *p: heap (100 byte)
}
```

compile / link

Quellprogramm

Symbol Table <a>	
.data	s=3 b=1
.rodata	c=2
.text	main
...	
ELF Header	

ELF-Binary

Flasher: Speicherorganisation auf einem Mikrocontroller

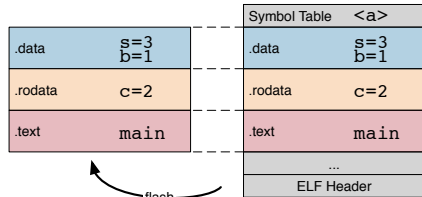
```
int a;                // a: global, uninitialized
int b = 1;           // b: global, initialized
const int c = 2;     // c: global, const

void main() {
    static int s = 3; // s: local, static, initialized
    int x, y;         // x: local, auto; y: local, auto
    char* p = malloc( 100 ); // p: local, auto; *p: heap (100 byte)
}
```

compile / link

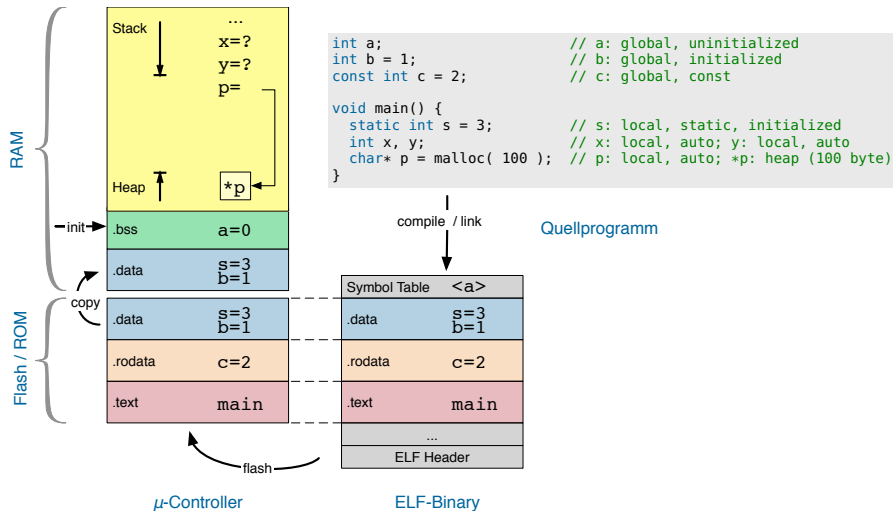
Quellprogramm

Flash / ROM

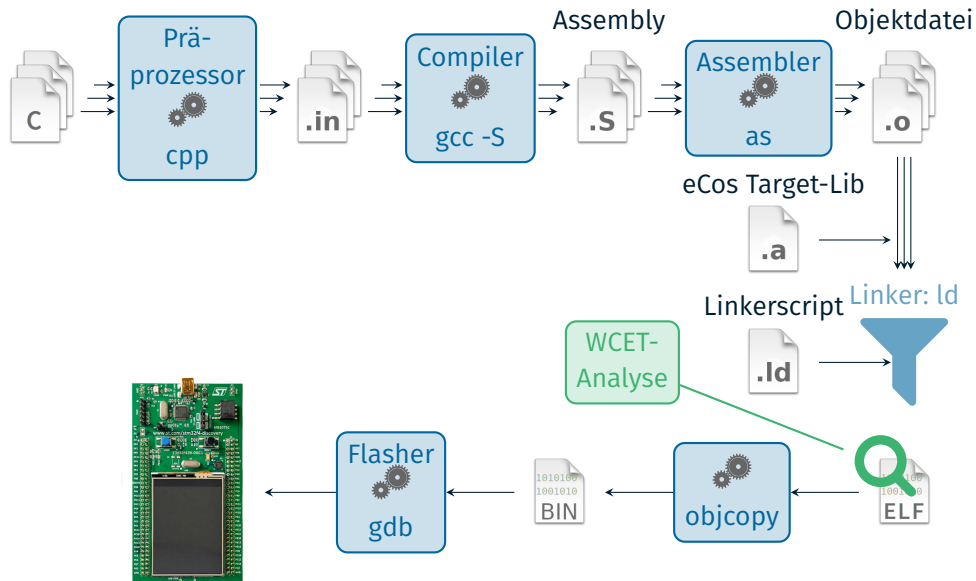


ELF-Binary

Flasher: Speicherorganisation auf einem Mikrocontroller



EZS-Toolchain



Instruktionsatz, Operationslaufzeiten

Logical	AND	AND Rd, Rn, <op2>	1
	Exclusive OR	EOR Rd, Rn, <op2>	1
	OR	ORR Rd, Rn, <op2>	1

Source: ARM, Cortex M4 Reference Manual ropo, S.30

Instruktionssatz, Operationslaufzeiten

Logical	AND	AND Rd, Rn, <op2>	1
	Exclusive OR	EOR Rd, Rn, <op2>	1
Divide	Signed	SDIV Rd, Rn, Rm	2 to 12 ^a
	Unsigned	UDIV Rd, Rn, Rm	2 to 12 ^a

- a. Division operations use early termination to minimize the number of cycles required based on the number of leading ones and zeroes in the input operands.

Source: ARM, Cortex M4 Reference Manual ropo, S.30 & S.33

Logical	AND	AND Rd, Rn, <op2>	1
	Exclusive OR	EOR Rd, Rn, <op2>	1
Divide	Signed	SDIV Rd, Rn, Rm	2 to 12 ^a
	Unsigned	UDIV Rd, Rn, Rm	2 to 12 ^a

3.3.1 Cortex-M4 instructions

The processor implements the ARMv7-M Thumb instruction set. Table 3-1 shows the Cortex-M4 instructions and their cycle counts. The cycle counts are based on a system with zero wait states.

Source: ARM, Cortex M4 Reference Manual ropo, S.29

Logical	AND	AND Rd, Rn, <op2>	1
	Exclusive OR	EOR Rd, Rn, <op2>	1
Divide	Signed	SDIV Rd, Rn, Rm	2 to 12 ^a
	Unsigned	UDIV Rd, Rn, Rm	2 to 12 ^a

3.3.1 Cortex-M4 instructions

Instruktionslaufzeiten

- Zyklendauern aus Datenblättern
- Jedoch: Meist nicht vollständig
- Annahme hier: Zero-Wait-States \leadsto Kein Warten auf Speicher
- \leadsto Konkrete Hardwaremodellierung für jedes Board erforderlich

Wiederholung: Stack-Aufbau

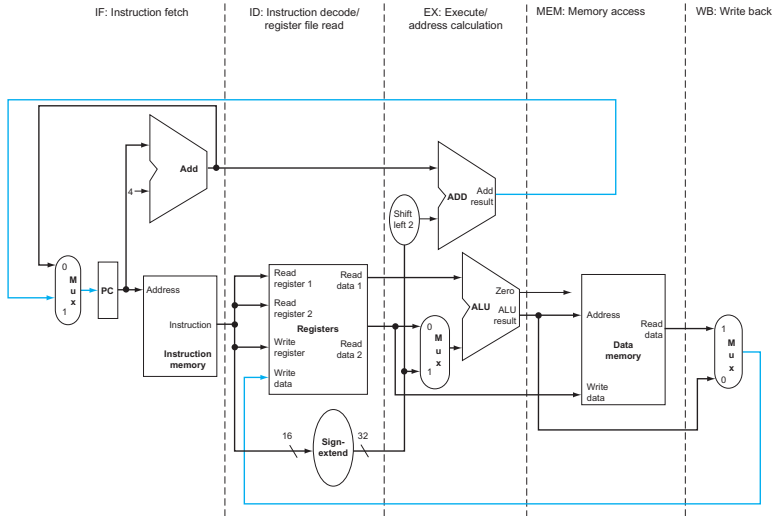
Standards in der Softwareentwicklung

Verwendung von Fließkommazahlen

Überblick: Toolchain

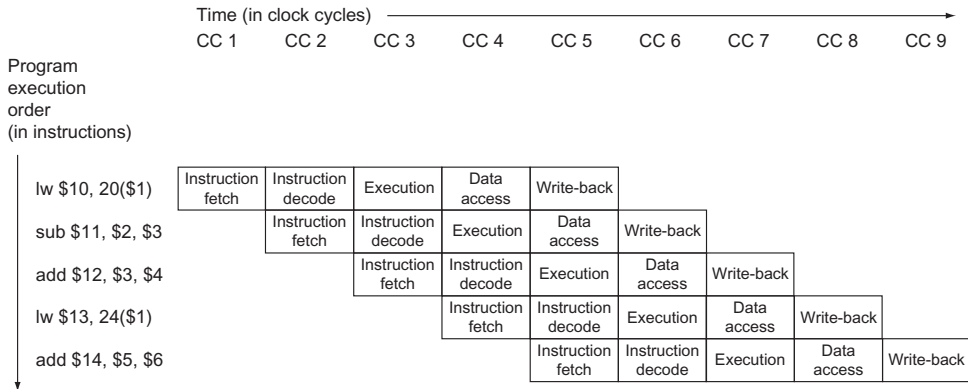
Hardware

MIPS: Single-Cycle



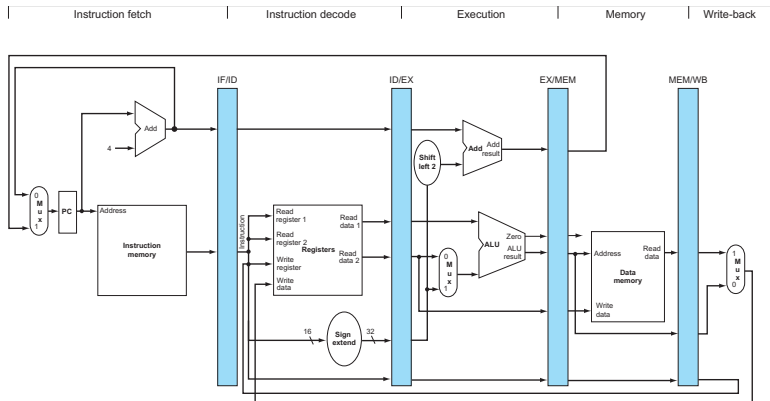
Source: D. A. Patterson und J. L. Hennessy, Computer organization and design: the hardware/software interface, 4th ed., 2012

MIPS: Pipelining



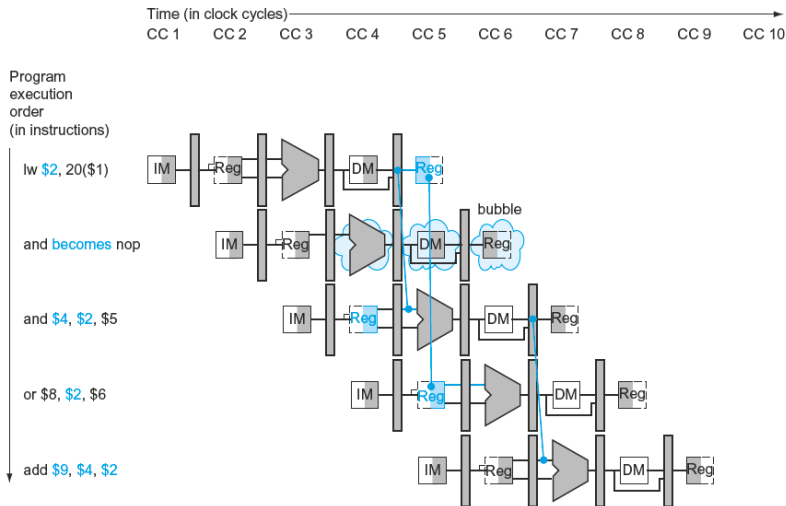
Source: D. A. Patterson und J. L. Hennessy, Computer organization and design: the hardware/software interface, 4th ed., 2012

MIPS: Pipelining



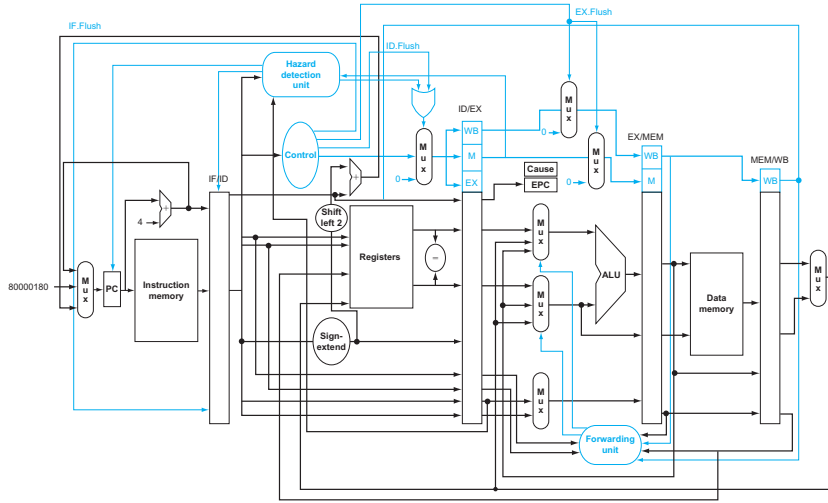
Source: D. A. Patterson und J. L. Hennessy, Computer organization and design: the hardware/software interface, 4th ed., 2012

MIPS: Pipelining



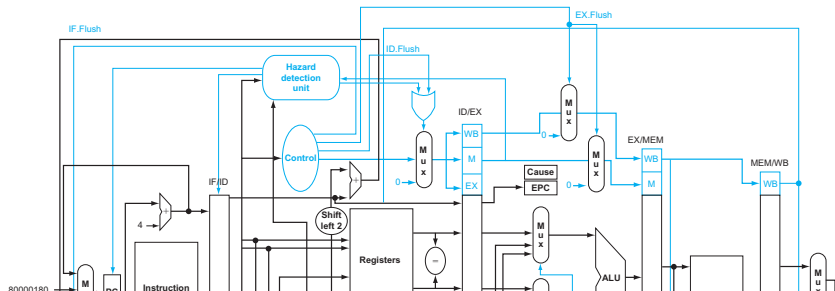
Source: D. A. Patterson und J. L. Hennessy, Computer organization and design: the hardware/software interface, 4th ed., 2012

MIPS: Pipelining



Source: D. A. Patterson und J. L. Hennessy, Computer organization and design: the hardware/software interface, 4th ed., 2012

MIPS: Pipelining



👉 All dieses Wissen muss dem Analysetool bekannt sein

Source: D. A. Patterson und J. L. Hennessy, Computer organization and design: the hardware/software interface, 4th ed., 2012

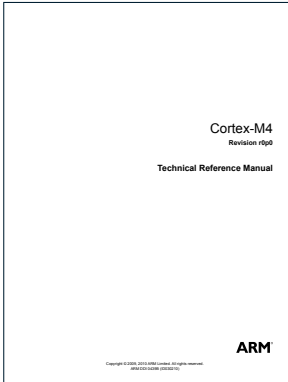
Eigenschaften von CPU-Architekturen

- Pipelining
- Caching
- Sprungvorhersage
- Mikroprogrammierbar vs. Fixed-Function
- Out-of-Order-Prozessoren
- Transaktionaler Speicher
- Superskalarität
- Mehrkernarchitekturen
- Hyperthreading
- ...

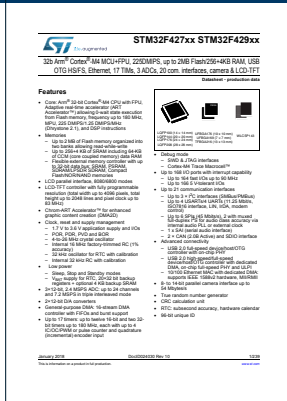
Eigenschaften von CPU-Architekturen

- Pipelining
 - Caching
 - Sprungvorhersage
 - Mikroprogrammierbar vs. Fixed-Function
 - Out-of-Order-Prozessoren
 - Transaktionaler Speicher
 - Superskalarität
 - Mehrkernarchitekturen
 - Hyperthreading
 - ...
- ☞ All diese Funktionalitäten müssen dem Entwickler bekannt sein
- ☞ Berücksichtigung in der WCET-Analyse

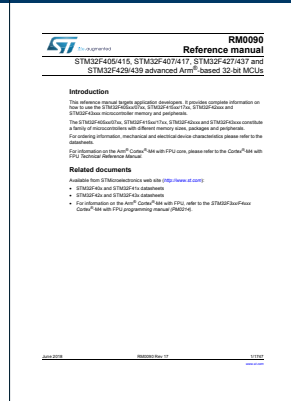
Referenzen



ARM: Cortex M4 –
Technical Reference
Manual
111 Seiten
Prozessorinterna



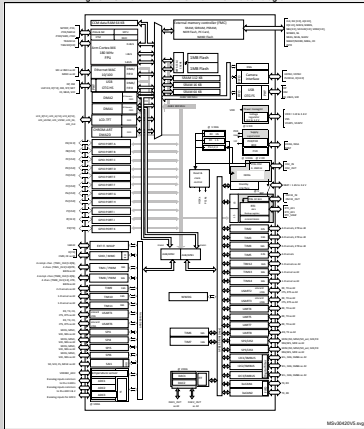
ST: STM32F427XX
STM32F429XX
Datasheet
240 Seiten
Boardspezifika



ST: RM0090 Reference
manual
1747 Seiten
"Complete
Information on
STM32F4xxx"

Speichertopologie STM32F429i-DISC1

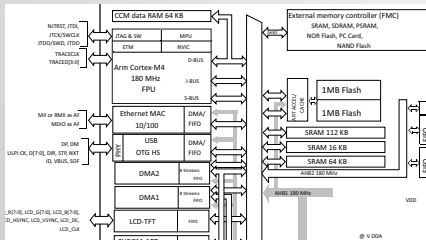
Figure 4. STM32F427xx and STM32F429xx block diagram



1. The timers connected to APB2 are clocked from TIMCLK up to 180 MHz, while the timers connected to APB1 are clocked from TIMCLK either up to 90 MHz or 180 MHz depending on TIMPRE bit configuration in the RCC_DCKCFGR register.

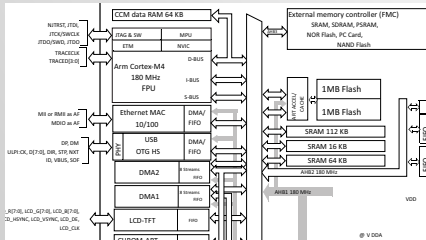
Source: ST: STM32F427xx STM32F429xx Datasheet, S.20

Speichertopologie STM32F429i-DISC1

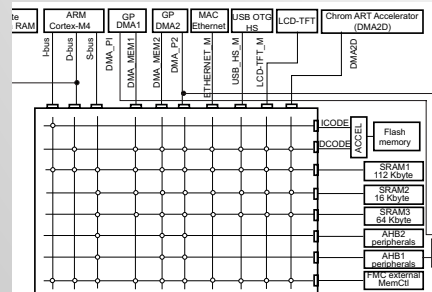


Source: ST: STM32F427xx STM32F429xx Datasheet, S.20

Speichertopologie STM32F429i-DISC1

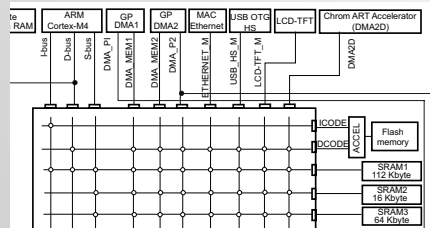
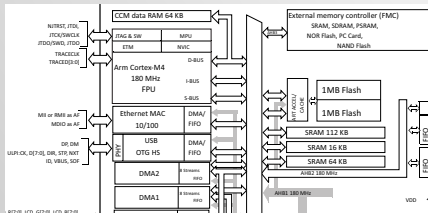


Source: ST: STM32F427xx STM32F429xx Datasheet



Source: ST: STM32F427xx STM32F429xx Datasheet

Speichertopologie STM32F429i-DISC1



Source

3.6 Embedded SRAM

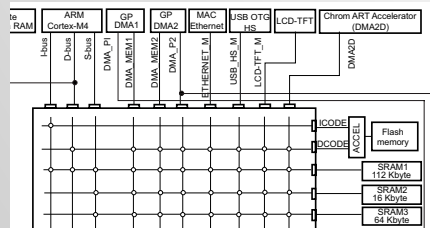
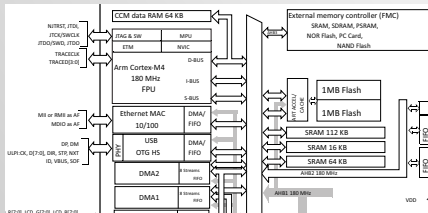
All devices embed:

- Up to 256Kbytes of system SRAM including 64 Kbytes of CCM (core coupled memory) data RAM
- RAM memory is accessed (read/write) at CPU clock speed with 0 wait states.

Source: ST: STM32F427xx STM32F429xx Datasheet, S.22

9xx Datasheet

Speichertopologie STM32F429i-DISC1



Source:

3.6 Embedded SRAM

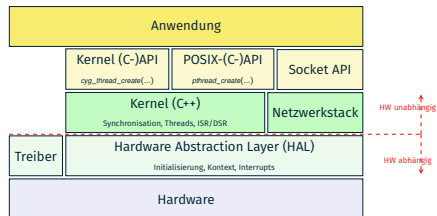
All devices embed:

3.2 Adaptive real-time memory accelerator (ART Accelerator™)

The ART Accelerator™ is a memory accelerator which is optimized for STM32 industry-standard Arm® Cortex®-M4 with FPU processors. It balances the inherent performance advantage of the Arm® Cortex®-M4 with FPU over Flash memory technologies, which normally requires the processor to wait for the Flash memory at higher frequencies.

To release the processor full 225 DMIPS performance at this frequency, the accelerator implements an instruction prefetch queue and branch cache, which increases program execution speed from the 128-bit Flash memory. Based on CoreMark benchmark, the

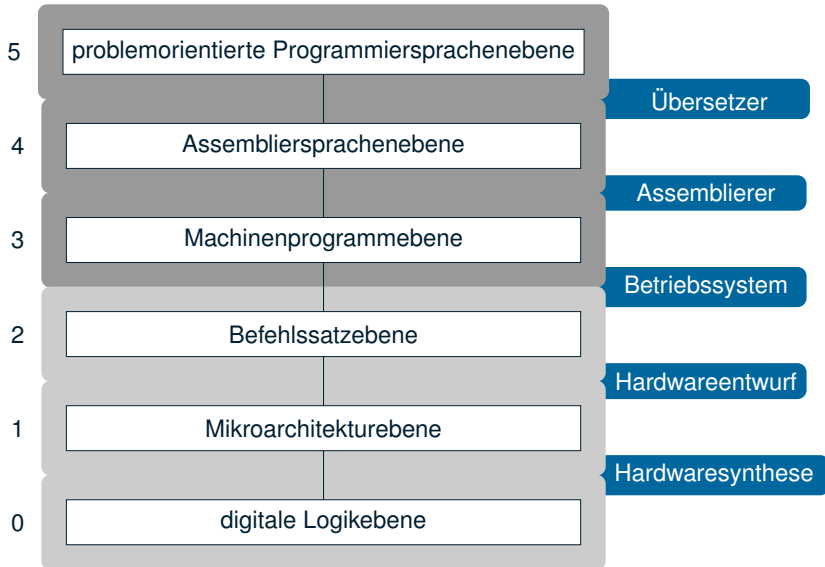
Source: ST: STM32F427xx STM32F429xx Datasheet, S.21



Betriebssystem

- in jedem Fall Ablaufplaner
 - oft Treiber/BSP mitgeliefert
 - ggf. interne Kontrollflüsse/Fäden/Unterbrechungen
 - meist konfigurierbar
- ~> Großer Einfluss auf Zeitverhalten des Gesamtsystems

Ebenen



- Systemsoftwareentwicklung benötigt holistisches Wissen über
 - Werkzeugkette
 - Betriebssystem
 - Zielarchitektur
 - Echtzeittheorie

↪ Umfasst Interna, nicht immer verfügbar
 - Entwickler muss all diese Einflussfaktoren kennen:
 - Zur Entwicklung
 - Zur Analyse
- ↪ Annahmen durch statische Analyse kontinuierlich verifizieren
- ↪ Nur so erhalten wir ein **sicheres** Echtzeitsystem

Danke fürs Zuhören!

Fragen?

[1] David Goldberg.

What every computer scientist should know about floating-point arithmetic.

ACM Computing Surveys (CSUR), 23(1):5–48, 1991.