Echtzeitsysteme

Übungen zur Vorlesung

System-Software-Entwicklung

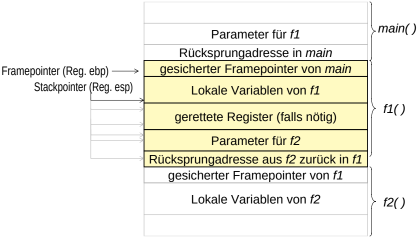
Simon Schuster **Phillip Raffeck**

Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)
Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme)
`https://www4.cs.fau.de`

Wintersemester 2019/20

Wiederholung: Stack-Aufbau

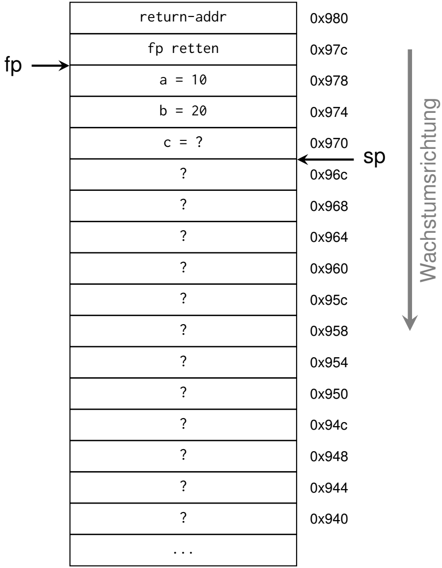
```c
int main() {
  int a, b, c;

  a = 10;
  b = 20;

  f1(a, b);

  return a;
}
```



- 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;
}
```

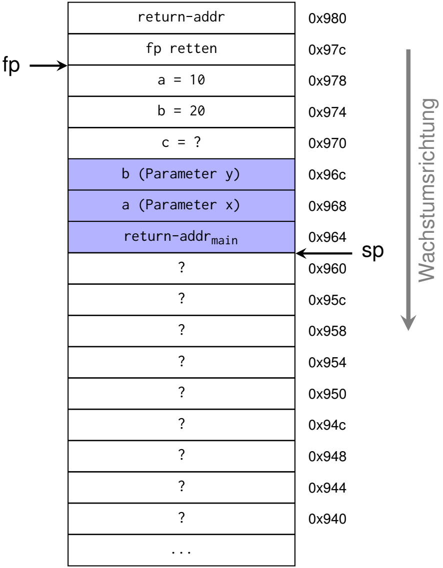


Variablenzugriff

- &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;
}
```

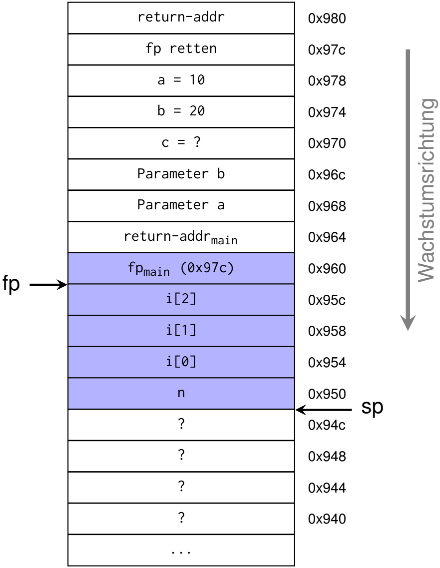


Funktionsaufruf

- 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

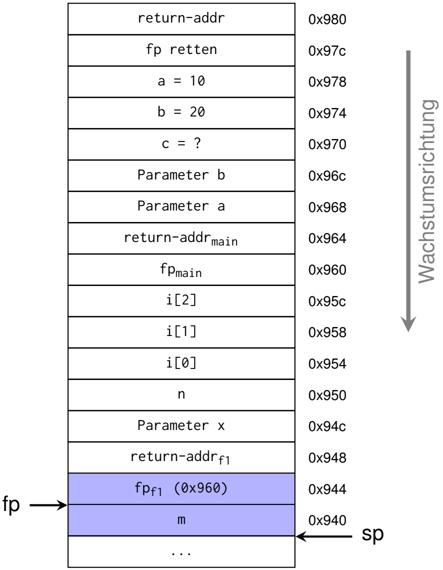
```
int main(void) {
  int a, b , c;
  a = 10;
  b = 20;
  f1(a, b);
  return a;
}

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;
}
```

	return-addr	0x980
	fp retten	0x97c
	a = 10	0x978
	b = 20	0x974
	c = ?	0x970
	Parameter b	0x96c
	Parameter a	0x968
	return-addr$_{main}$	0x964
fp →	fp$_{main}$	0x960
	i[2]	0x95c
	i[1]	0x958
	i[0]	0x954
	n	0x950
	x (Parameter z)	0x94c
	return-addr$_{f1}$	0x948 ← sp
	?	0x944
	?	0x940
	...	

Wachstumsrichtung ↓

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;
}
```

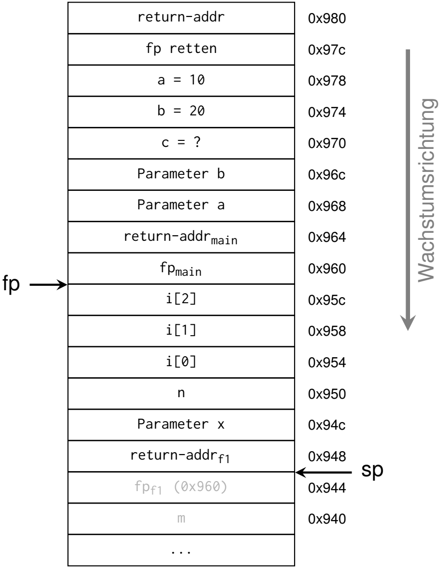


Ablauf Funktionsaufruf

```
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;   ←
}
```

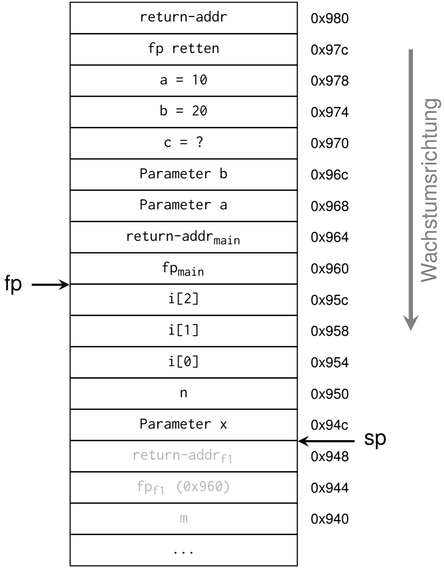
Stackframe abräumen

- sp = fp
- fp = pop(sp)



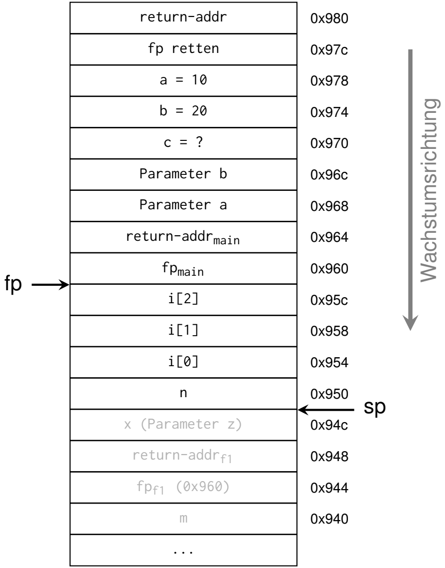
Ablauf Funktionsaufruf

```
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;
}
```



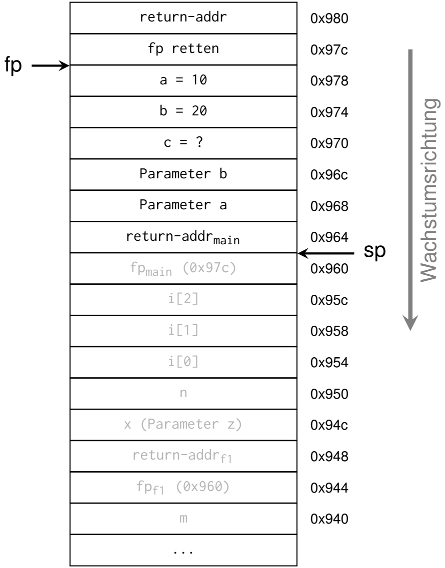
Ablauf Funktionsaufruf

```
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;
}
```



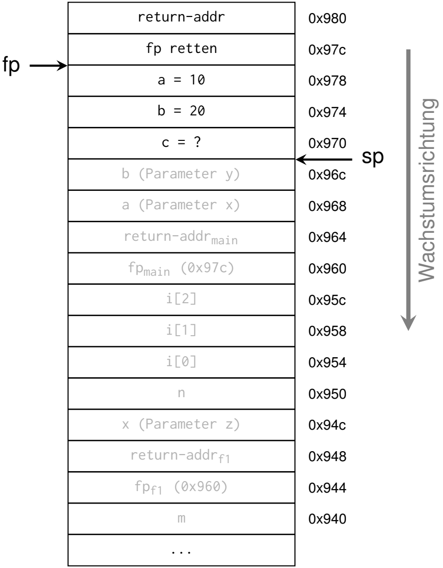
Ablauf Funktionsaufruf

```
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;
}
```



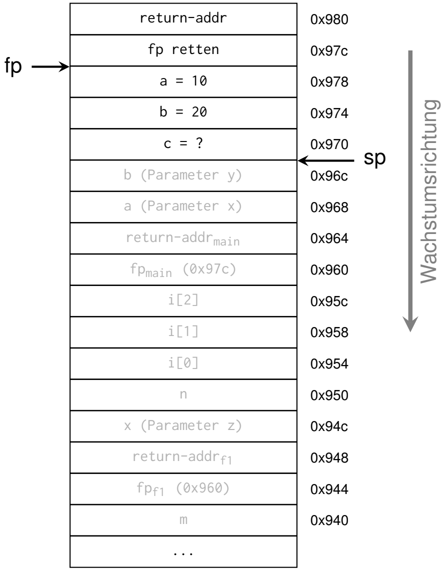
Ablauf Funktionsaufruf

```
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;
}
```



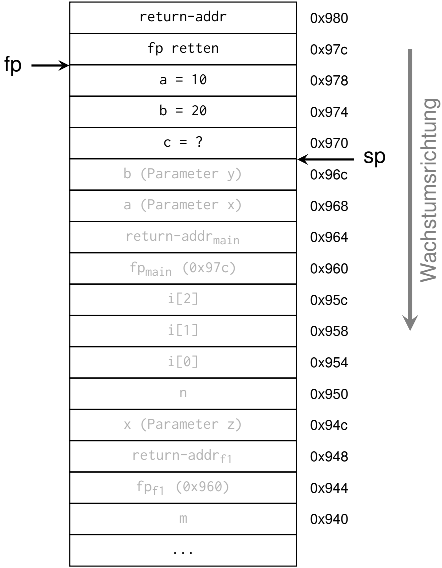
Ablauf Funktionsaufruf

```
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;
}
```

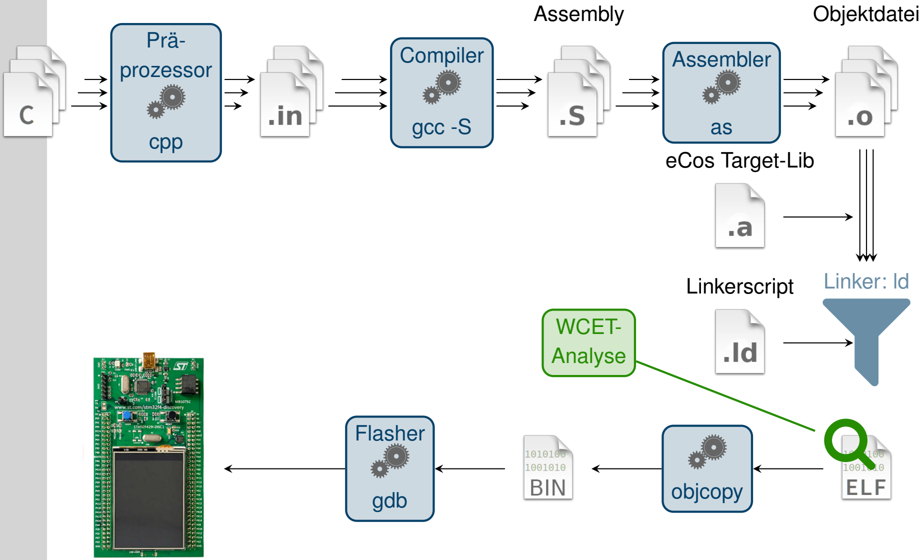
	return-addr	0x980	
	fp retten	0x97c	
	a = 10	0x978	
	b = 20	0x974	
	c = ?	0x970	
	6 (Parameter z3)	0x96c	
	5 (Parameter z2)	0x968	
	4 (Parameter z1)	0x964	
	$\texttt{return}_{\texttt{main}}$	0x960	
	$\texttt{fp}_{\texttt{main}}$ (0x97c)	0x95c	
fp →	m	0x958	
	i[0]	0x954	← sp
	n	0x950	
	x (Parameter z)	0x94c	
	$\texttt{return-addr}_{\texttt{f1}}$	0x948	
	$\texttt{fp}_{\texttt{f1}}$ (0x960)	0x944	
	m	0x940	
	...		

Wachstumsrichtung ↓

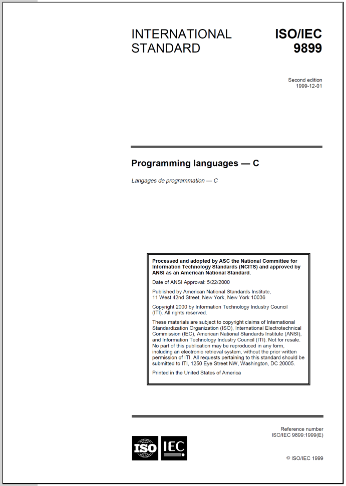
Was passiert bei weiterem Funktionsaufruf?

Übersicht

EZS-Toolchain



C Standard



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

C Standard II

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

C Standard II

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

C Standard II

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.

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

Übersicht

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. 1
2. `0`
3. nicht definiert in C

Frage #2

Zu was wird `2/7` ausgewertet?

1. 1
2. `0`
3. 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.` ⇝ `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. `0`
3. 1 (dank Compileroptimierung)

Frage #5

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

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

Erklärung

- $\texttt{int}_1$/<größerer $\texttt{int}_2$> $\rightsquigarrow$ `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 ⇝ double
- double Standard-Typ, außer zusätzliches Literal (3.14 f)
- Begrenzter Wertebereich:
  6.28318530717958600000000000000

Frage #7

```c
double a = 0.1;
double b = 0.2;

float aa = 0.1;
float bb = 0.2;

if (a+b == aa+bb){
  ezs_printf("equal\n");
}else{
  ezs_printf("unequal: %.30f != %.30f\n", (a+b), (aa+bb));
}
```

Was wird ausgegeben?

1. `equal`
2. `unequal...`

Frage #7

```c
double a = 0.1;
double b = 0.2;

float aa = 0.1;
float bb = 0.2;

if (a+b == aa+bb){
  ezs_printf("equal\n");
}else{
  ezs_printf("unequal: %.30f != %.30f\n", (a+b), (aa+bb));
}
```

Was wird ausgegeben?

1. equal
2. unequal...

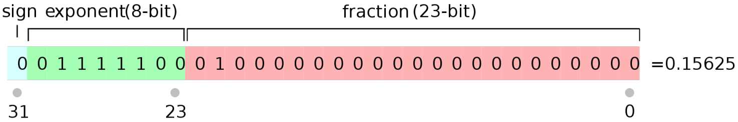
Fließkomma-Arithmetik

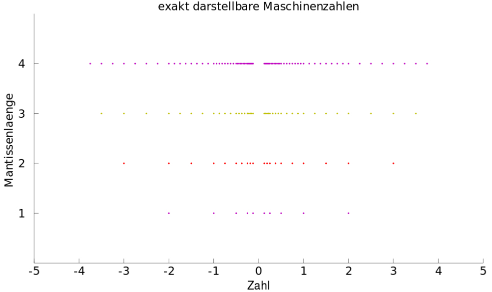
```
double a = 0.1;
double b = 0.2;

float aa = 0.1;
float bb = 0.2;

if (a+b == aa+bb){
  ezs_printf("equal\n");
}else{
  ezs_printf("unequal: %.30f != %.30f\n", (a+b), (aa+bb));
}
```

```
/* Ausgabe:
unequal:
0.300000000000000000000000000000 !=
0.300000011920929000000000000000
*/
```

- 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 ⇝ VEZS
- Integer-Division ist *kein sicherer Ausweg*

☞ *Sorgfalt bei arithmetischen Operationen in begrenzten Wertebereichen*

Wahl des Datentyps bei Berechnung des Sinus-Wertes

- Harmonische Schwingung[1]: $y(t) = y_0 \cdot sin(\omega t + \varphi_0)$ und $\omega = 2\pi f$

```
#define TYPE {int|double|float} ?
...
TYPE compute_sinus(OTHER_TYPE real_time) {

  TYPE f     = ...
  TYPE omega = 2 * M_PI * f;
  ...
  ... sin(omega * real_time) // oder doch sinf(omega * real_time)?
  ...
}
```

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

Wahl des Datentyps bei Berechnung des Sinus-Wertes

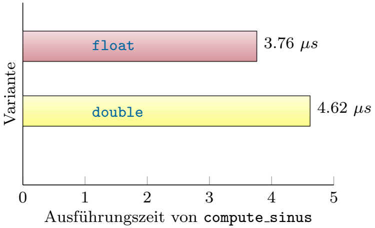
- Harmonische Schwingung[1]: $y(t) = y_0 \cdot sin(\omega t + \varphi_0)$ und $\omega = 2\pi f$

```
#define TYPE {int|double|float} ?
...
TYPE compute_sinus(OTHER_TYPE real_time) {

  TYPE f     = ...
  TYPE omega = 2 * M_PI * f;
  ...
  ... sin(omega * real_time) // oder doch sinf(omega * real_time)?
  ...
}
```

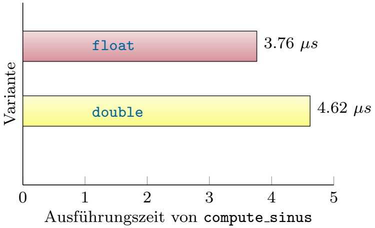
- *`float` oder `double` für Realzeit sinnvoll? Was ist `OTHER_TYPE`?*
- Konfiguration von `float` und `double` sinnvoll
- *Laufzeit von `compute_sinus()`?*

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

Vergleich der Laufzeiten

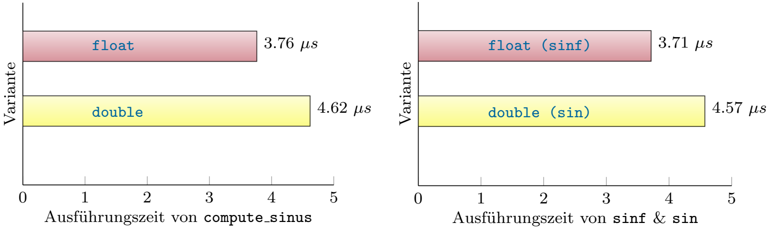


Vergleich der Laufzeiten



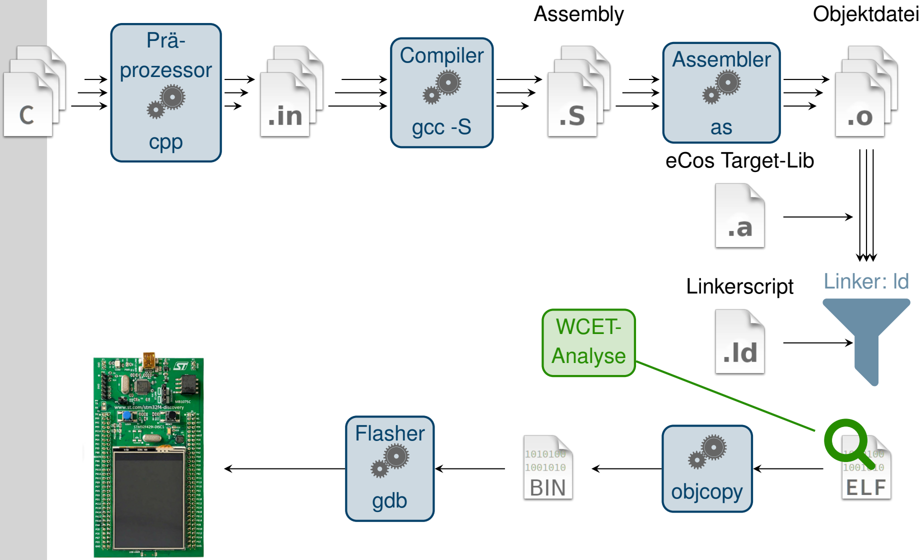
- Laufzeitzuwachs um *23 %* bei Wechsel `float` → `double`
- Soft Float? Hard Float? hier: Soft Float

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 Abhängigkeit der Wortbreite (32-Bit Cortex-M4, 8-Bit AVR)
- Spezialbibliothek für Signalverarbeitung mit Integer-Arithmetik
- Spezielle Hardware-Einheiten zur Signalverarbeitung

EZS-Toolchain



Quellcode

```c
#define FOO 42

#include "example.h"

#if defined(FOO)
  int i = FOO;
#else
  int i = 0;
#endif
```

Expandiert

```c
// Inhalt example.h
void example();

int i = 42;
```

Präprozessor

- Vorverarbeitungsschritt vor der Übersetzung
  - Konfigurationsabhäniger Code `#if(def)`
  - Definierbare Konstanten `#define`
  - Auflösen von `#include`-Direktiven
- Reine Zeichernersetzung/Textmanipulation

Übersetzer

Quellcode

```c
volatile extern int i;
int j = 42;

int main(int argc, ...)
{
  i = 0;
  if(argc % 2) {
    i = 1;
  }
  return i + j;
}
```

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

Übersetzer II

Beispiel: Schleifenaufrollen

Unoptimiert

```
for(i = 0; i < 40; i++) {
  x++;
}
x++;
x++;
```

Größenoptimiert

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

Laufzeitverhalten

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

⇝ invalidiert z.T. Annotationen und Annahmen über Laufzeitverhalten

Assembler

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 Machinencode (binär)
- 1:1 Übersetzung
- z.T. Macroassembler: Komplexbefehle zu Instruktionsfolge

Linker

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

$\rightsquigarrow$ Zusammenführung der Funktionen und Variablen aus Objektdateien

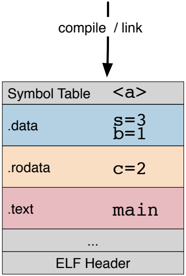
$\rightsquigarrow$ Vergabe globaler Adressen gemäß Konfiguration

$\rightsquigarrow$ Auflösen der Adressen im Code

Flasher: Speicherorganisation auf einem Mikrocontroller

```c
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)
}
```

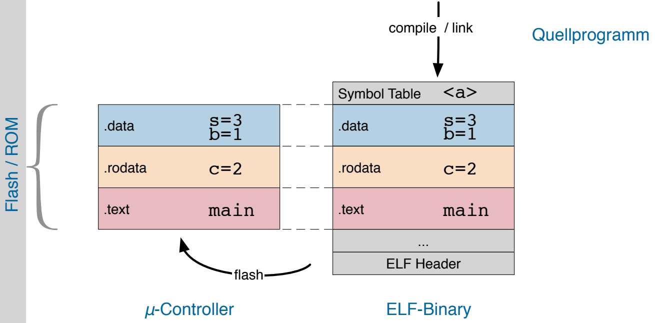
Quellprogramm



ELF-Binary

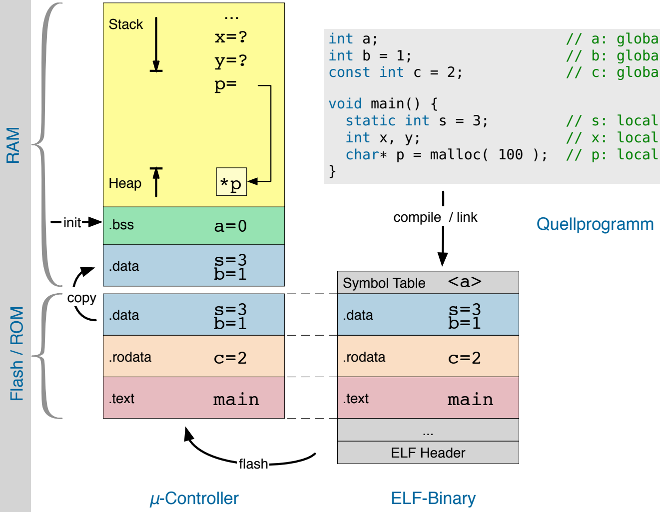
Flasher: Speicherorganisation auf einem Mikrocontroller

```c
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)
}
```

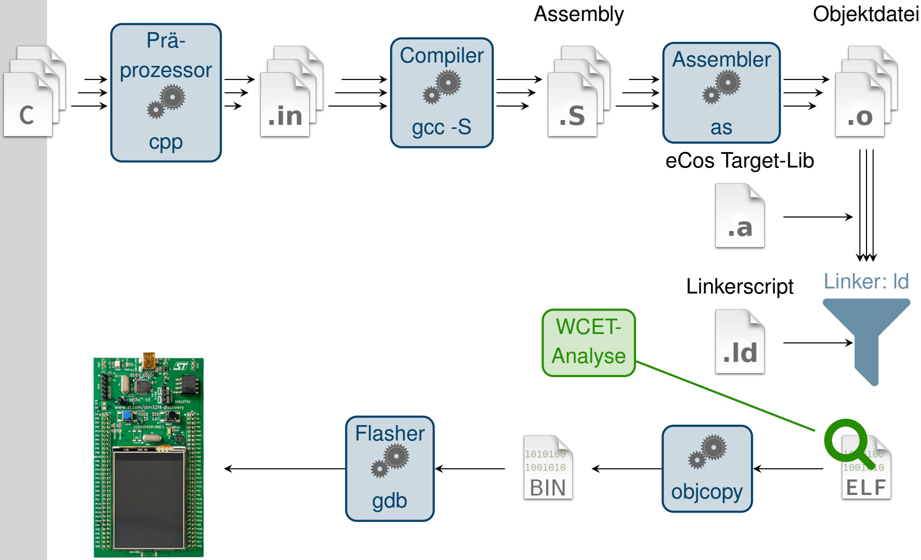


Flasher: Speicherorganisation auf einem Mikrocontroller

```c
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)
}
```



EZS-Toolchain



Instruktionssatz, 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 r0p0, S.30

Instruktionssatz, Operationslaufzeiten

Logical	AND	`AND 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 r0p0, S.30 & S.33

Instruktionssatz, Operationslaufzeiten

Logical	AND	`AND 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 r0p0, S.29

Instruktionssatz, Operationslaufzeiten

Logical	AND	`AND 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 r0p0, S.29

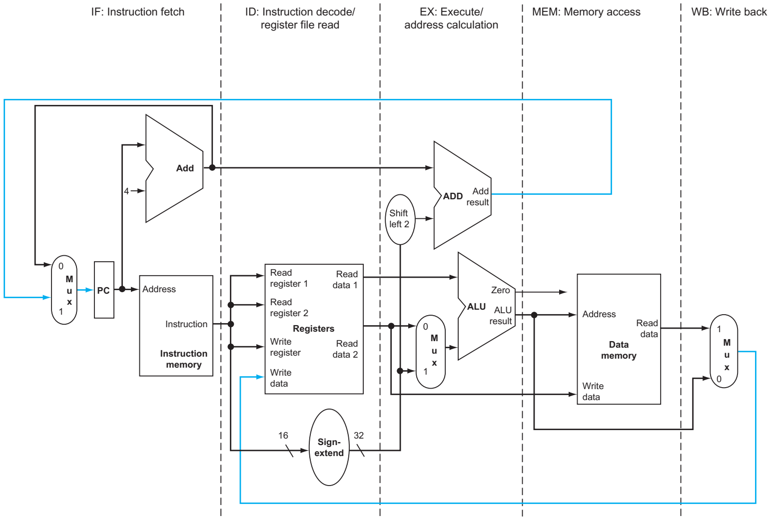
Instruktionslaufzeiten

- Zyklendauern aus Datenblättern
- Jedoch: Meist nicht vollständig
- Annahme hier: Zero-Wait-States ⇝ Kein Warten auf Speicher

⇝ Konkrete Hardwaremodellierung für jedes Bord erforderlich

Übersicht

MIPS: Single-Cycle



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

MIPS: Pipelining

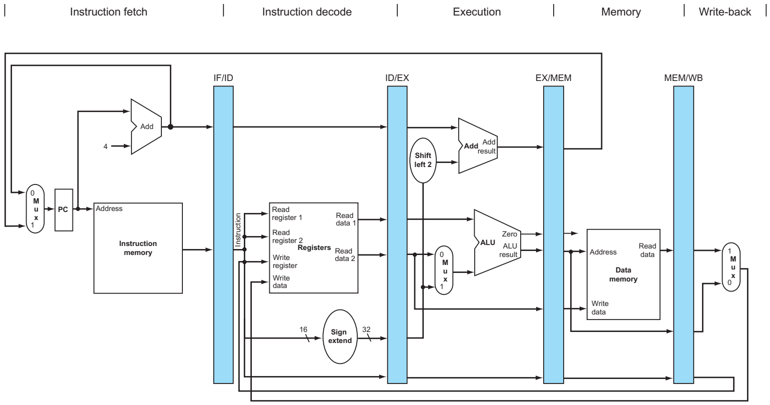
Time (in clock cycles) →

Program execution order (in instructions) ↓

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
lw $10, 20($1)	Instruction fetch	Instruction decode	Execution	Data access	Write-back				
sub $11, $2, $3		Instruction fetch	Instruction decode	Execution	Data access	Write-back			
add $12, $3, $4			Instruction fetch	Instruction decode	Execution	Data access	Write-back		
lw $13, 24($1)				Instruction fetch	Instruction decode	Execution	Data access	Write-back	
add $14, $5, $6					Instruction fetch	Instruction decode	Execution	Data access	Write-back

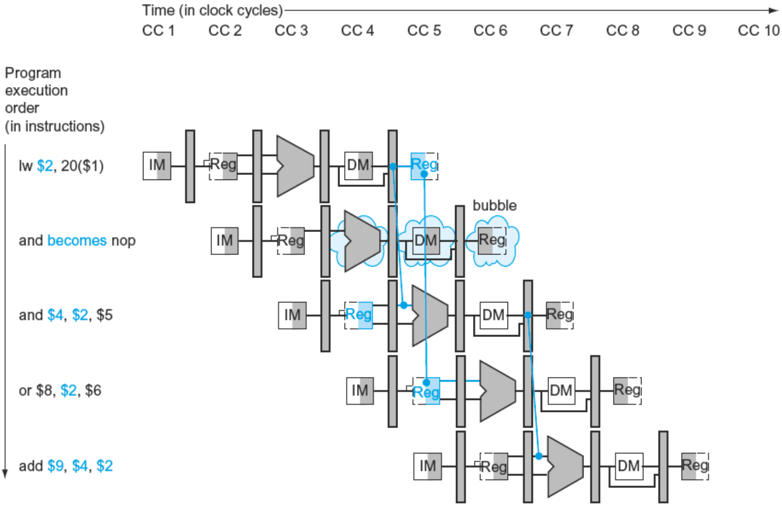
Source: D. A. Patterson und J. L. Hennessy, Computer organization and design: the hardware/software interface, 4th ed., 2012

MIPS: Pipelining



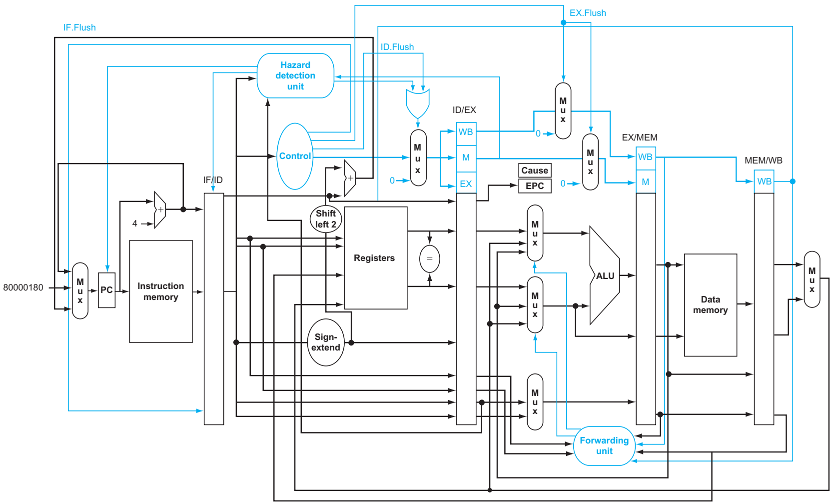
Source: D. A. Patterson und J. L. Hennessy, Computer organization and design: the hardware/software interface, 4th ed., 2012

MIPS: Pipelining



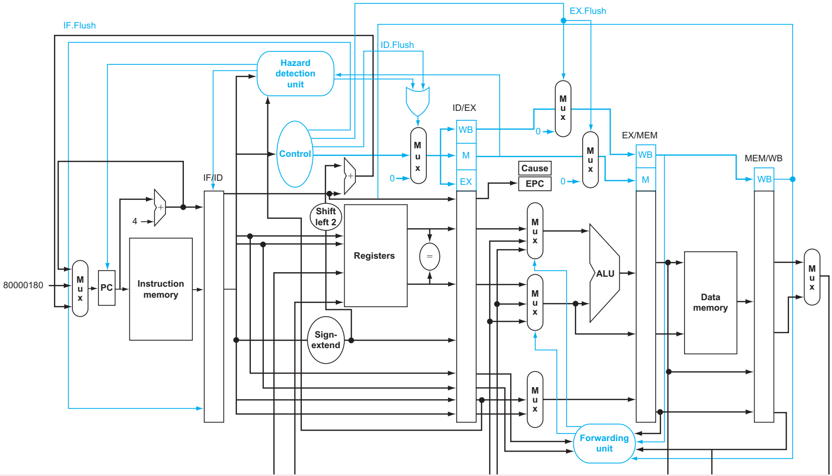
Source: D. A. Patterson und J. L. Hennessy, Computer organization and design: the hardware/software interface, 4th ed., 2012

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
- ...

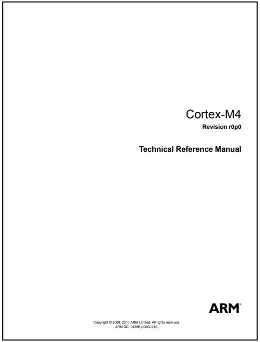
Eigenschaften von CPU-Architekturen

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

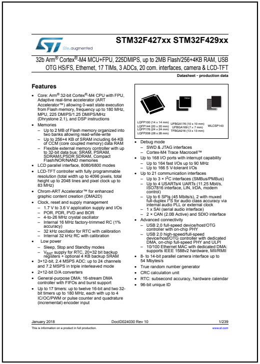
☞ All diese Funktionalitäten müssen dem Entwickler bekannt sein

☞ Berücksichtigung in der WCET-Analyse

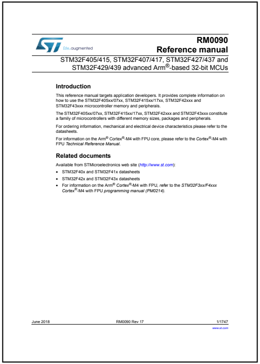
Referenzen



Cortex-M4
Revision r0p0
Technical Reference Manual
ARM
Copyright © 2009, 2010 ARM Limited. All rights reserved.
ARM DDI 0439B (ID030210)



STM32F427xx STM32F429xx
32b Arm® Cortex®-M4 MCU+FPU, 225DMIPS, up to 2MB Flash/256+4KB RAM, USB OTG HS/FS, Ethernet, 17 TIMs, 3 ADCs, 20 com. interfaces, camera & LCD-TFT
Datasheet - production data
Features
January 2018 DocID024030 Rev 10 1/239



RM0090
Reference manual
STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced Arm®-based 32-bit MCUs
Introduction
Related documents
June 2018 RM0090 Rev 17 1/1747

ARM: Cortex M4 – Technical Reference Manual

111 Seiten

Prozessorinterna

ST: STM32F427xx STM32F429xx Datasheet

240 Seiten

Boardspezifika

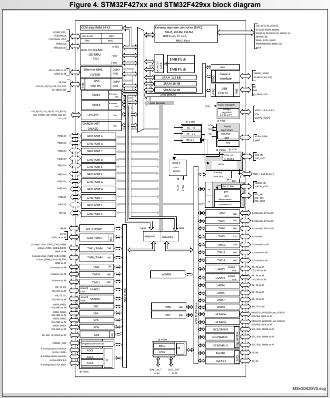
ST: RM0090 Reference manual

1747 Seiten

"Complete Information on STM32F4xxx"

Speichertopologie STM32F429i-DISC1

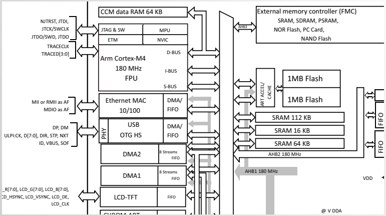
Figure 4. STM32F427xx and STM32F429xx block diagram



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

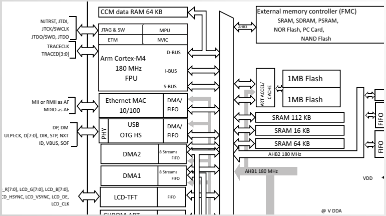
Source: ST: STM32F427xx STM32F429xx Datasheet, S.20

Speichertopologie STM32F429i-DISC1

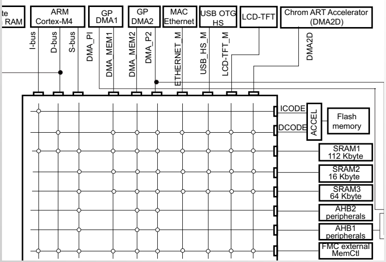


Source: ST: STM32F427xx STM32F429xx Datasheet, S.20

Speichertopologie STM32F429i-DISC1

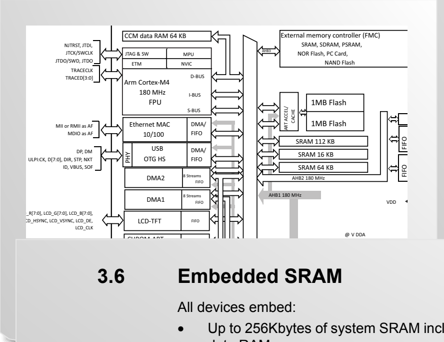


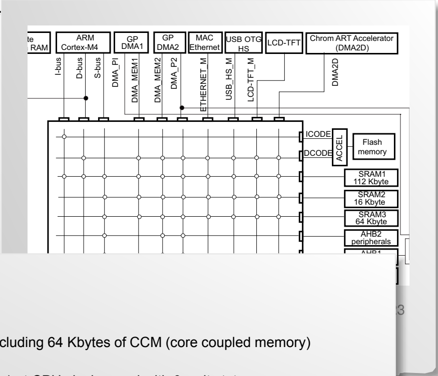
Source: ST: STM32F427xx STM32F429xx Datasheet, S.20



Source: ST: STM32F427xx STM32F429xx Datasheet, S.23

Speichertopologie STM32F429i-DISC1





3.6 Embedded SRAM

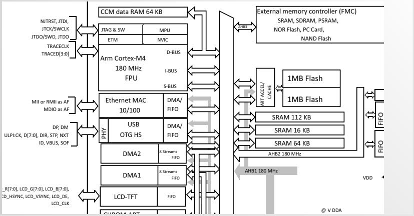
All devices embed:

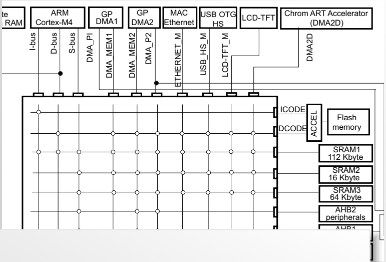
- Up to 256Kbytes of system SRAM including 64 Kbytes of CCM (core coupled memory) data RAM

  RAM memory is accessed (read/write) at CPU clock speed with 0 wait states.

Source: ST: STM32F427xx STM32F429xx Datasheet, S.22

Speichertopologie STM32F429i-DISC1





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.

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

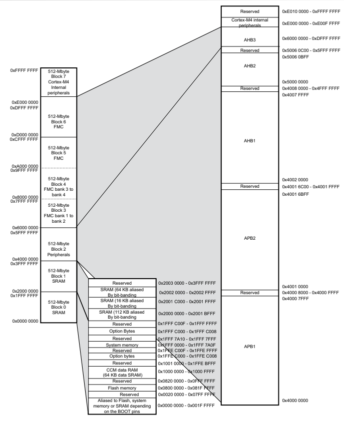
The ART Accelerator™ is a memory accelerator which is optimized for STM32 industry-standard Arm® Cortex®-M4 with FPU processors. It balances the inherent performance advantage of the Arm® Cortex®-M4 with FPU over Flash memory technologies, which normally requires the processor to wait for the Flash memory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

Speicherlayout STM32F429i-DISC1

Figure 19. Memory map



Source: ST: STM32F427xx STM32F429xx Datasheet, S.53

APB2	0x4001 3800 - 0x4001 3BFF	SYSCFG
	0x4001 3400 - 0x4001 37FF	SPI4
	0x4001 3000 - 0x4001 33FF	SPI1
	0x4001 2C00 - 0x4001 2FFF	SDIO
	0x4001 2400 - 0x4001 2BFF	Reserved
	0x4001 2000 - 0x4001 23FF	ADC1 - ADC2 - ADC3
	0x4001 1800 - 0x4001 1FFF	Reserved
	0x4001 1400 - 0x4001 17FF	USART6
	0x4001 1000 - 0x4001 13FF	USART1

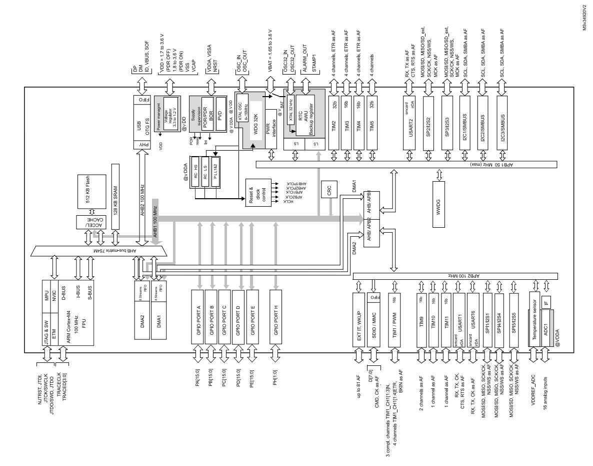
Source: ST, STM32F427xx STM32F429xx Datasheet, S.55

Peripherie

- Im Adressraum eingeblendet
- Am Peripheriebus (APBx)
- ⤳ Anderes Zugriffsverhalten als Speicher

Beispiel: USART

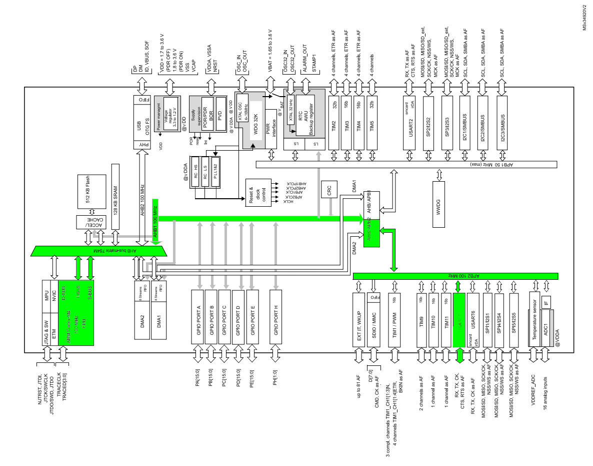
Blockdiagramm



Source: ST, STM32F427xx STM32F429xx Datasheet, S.20

Beispiel: USART

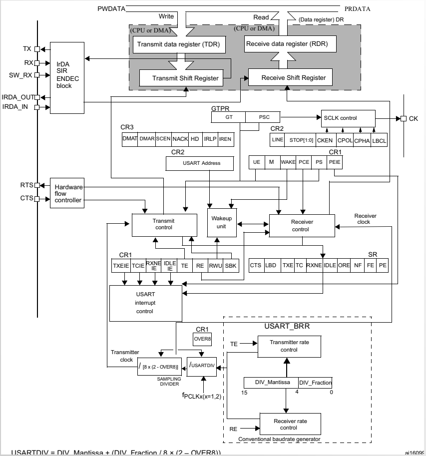
Blockdiagramm



Source: ST, STM32F427xx STM32F429xx Datasheet, S.20

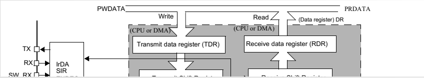
Beispiel: USART

Innerer Aufbau



Source: ST: RM0090 Reference manual, S.989

Beispiel: USART



30.6.1 Status register (USART_SR)

Address offset: 0x00

Reset value: 0x0000 00C0

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						CTS	LBD	TXE	TC	RXNE	IDLE	ORE	NF	FE	PE
						rc_w0	rc_w0	r	rc_w0	rc_w0	r	r	r	r	r

Bit 7 **TXE**: Transmit data register empty

This bit is set by hardware when the content of the TDR register has been transferred into the shift register. An interrupt is generated if the TXEIE bit =1 in the USART_CR1 register. It is cleared by a write to the USART_DR register.

0: Data is not transferred to the shift register

1: Data is transferred to the shift register)

Note: This bit is used during single buffer transmission.

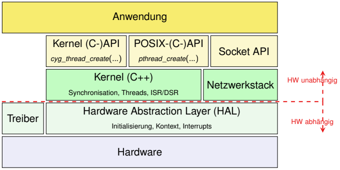
Source: ST: RM0090 Reference manual, S.1007 & 1008

Board Support Package

```
stm32f411e-discovery
|-- Release_Notes.html
|-- stm32f411e_discovery_accelerometer.c
|-- stm32f411e_discovery_accelerometer.h
|-- stm32f411e_discovery_audio.c
|-- stm32f411e_discovery_audio.h
|-- STM32F411E-Discovery_BSP_User_Manual.chm
|-- stm32f411e_discovery.c
|-- stm32f411e_discovery_gyroscope.c
|-- stm32f411e_discovery_gyroscope.h
`-- stm32f411e_discovery.h
```

Board Support Package

- Vom Hersteller vorgegeben
- Ansteuerung für Boardperipherie
- Meist permissive Lizenzen

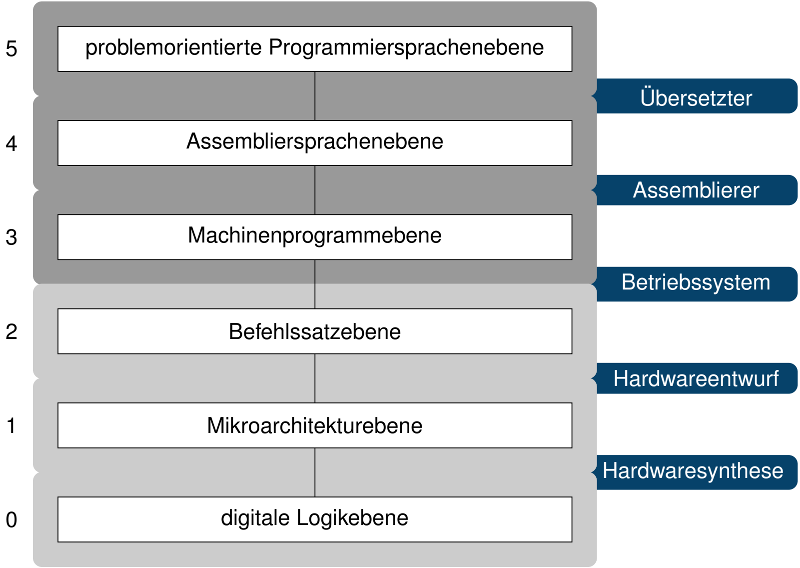


Betriebssystem

- in jedem Fall Ablaufplaner
- oft Treiber/BSP mitgeliefert
- ggf. interne Kontrollflüsse/Fäden/Unterbrechungen
- meist konfigurierbar

⇝ Großer Einfluss auf Zeitverhalten des Gesamtsystems

Ebenen


5
problemorientierte Programmiersprachenebene
4
Assembliersprachenebene
3
Machinenprogrammebene
2
Befehlssatzebene
1
Mikroarchitekturebene
0
digitale Logikebene
Übersetzter
Assemblierer
Betriebssystem
Hardwareentwurf
Hardwaresynthese

Fazit

- 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

Fragen...

Literatur

[1] David Goldberg.
What every computer scientist should know about floating-point arithmetic.
ACM Computing Surveys (CSUR), 23(1):5–48, 1991.