



# Compiler für Eingebettete Systeme

## [CS7506]

Sommersemester 2014

Heiko Falk

Institut für Eingebettete Systeme/Echtzeitsysteme  
Ingenieurwissenschaften und Informatik  
Universität Ulm



## **Kapitel 2**

# **Compiler für Eingebettete Systeme**

—

# **Anforderungen & Abhängigkeiten**

# Inhalte der Vorlesung

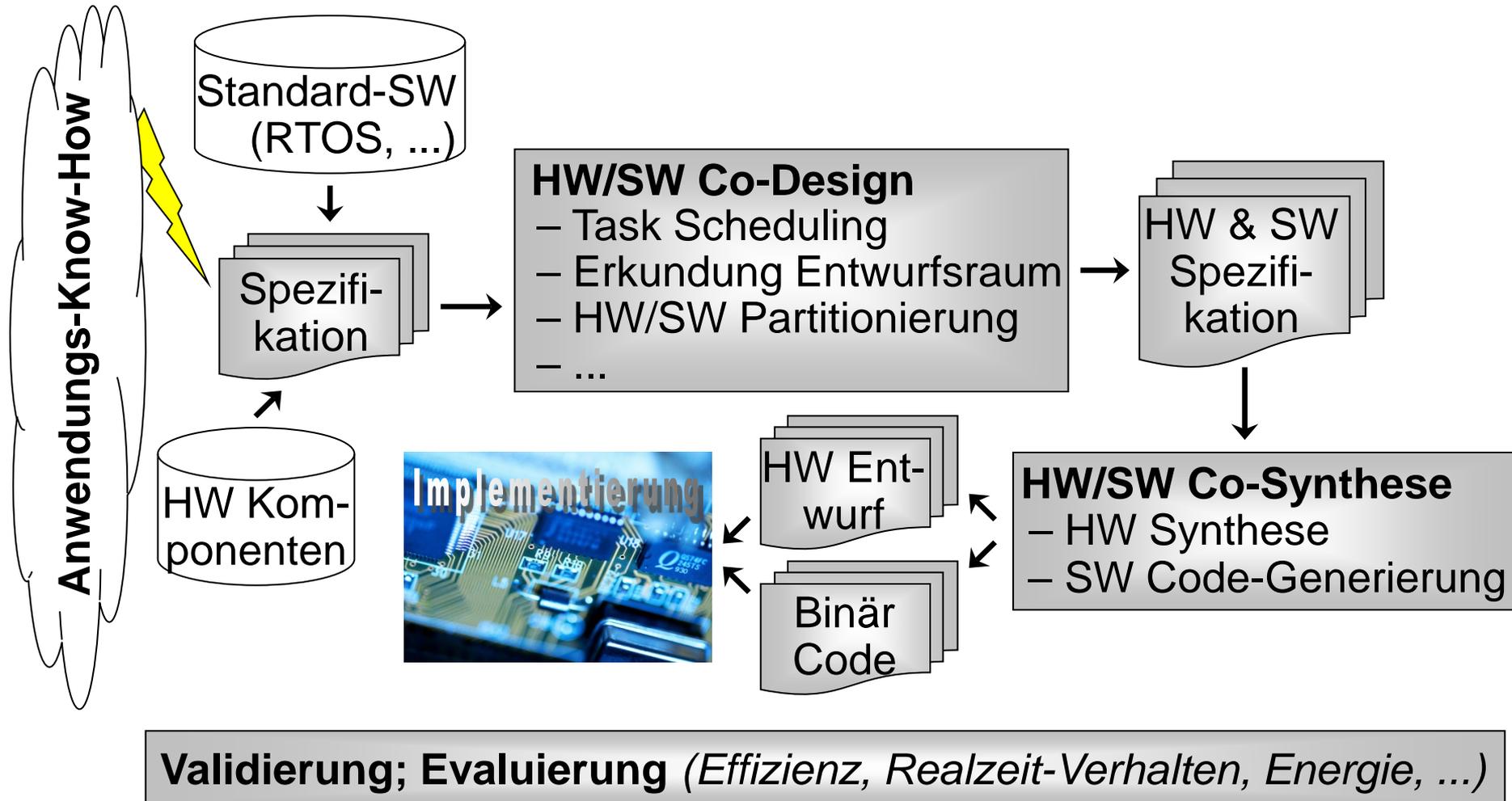
1. Einordnung & Motivation der Vorlesung
2. **Compiler für Eingebettete Systeme – Anforderungen & Abhängigkeiten**
3. Interner Aufbau von Compilern
4. Prepass-Optimierungen
5. HIR Optimierungen und Transformationen
6. Instruktionsauswahl
7. LIR Optimierungen und Transformationen
8. Register-Allokation
9. Compiler zur WCET<sub>EST</sub>-Minimierung
10. Ausblick

# Inhalte des Kapitels

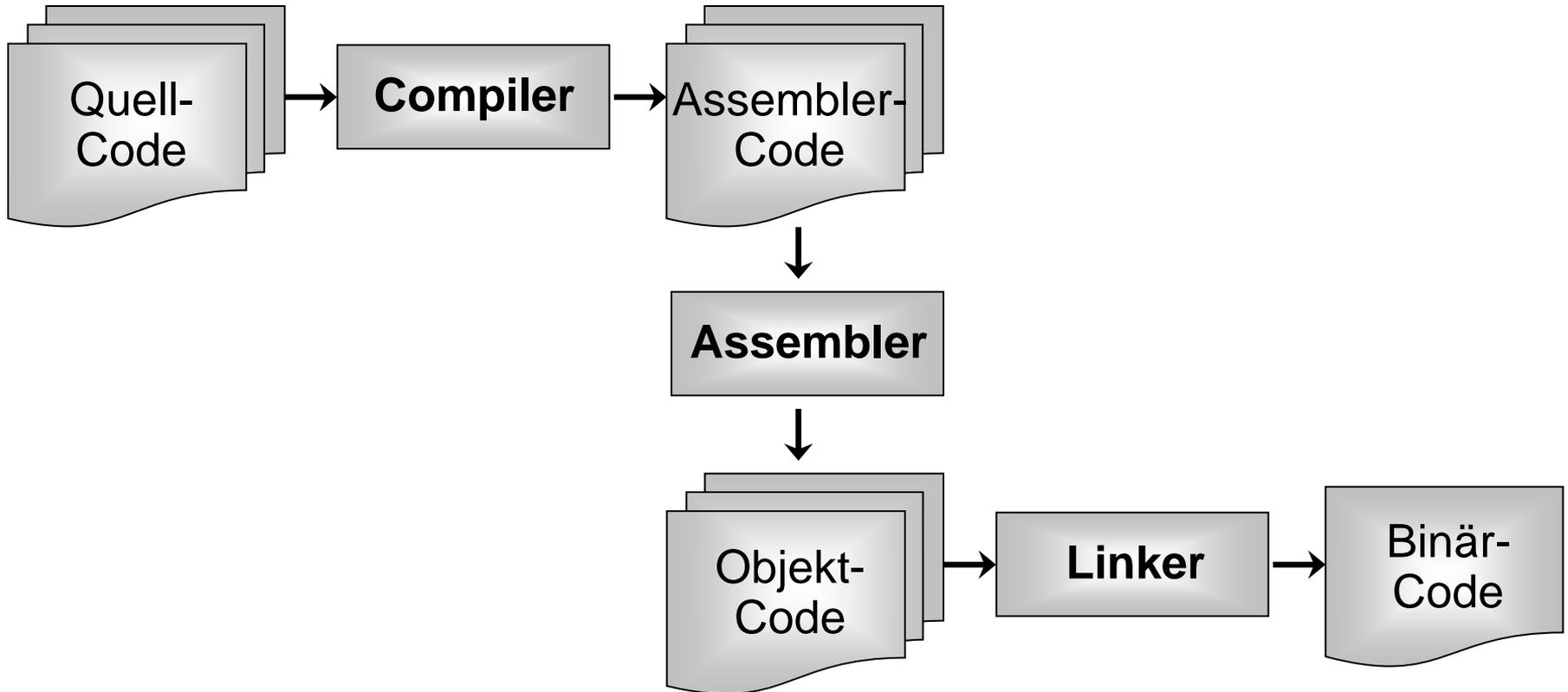
## 2. Compiler für Eingebettete Systeme – Anforderungen & Abhängigkeiten

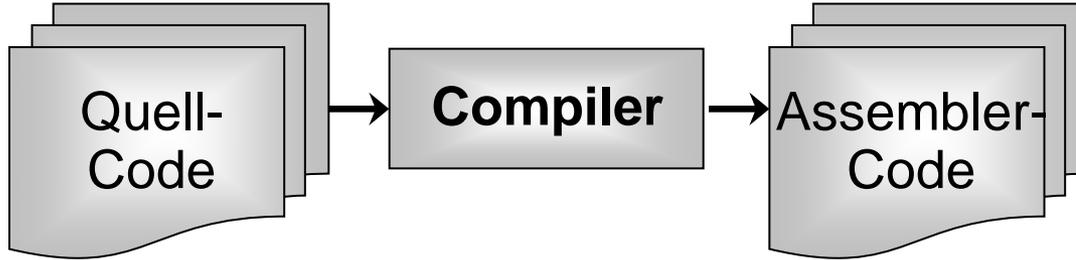
- Werkzeuge zur Code-Generierung
  - Compiler, Assembler, Linker
  - Quellcode, Assemblercode, Objektcode, Binärcode
- Quellsprachen für Compiler für Eingebettete Systeme
  - C, C++, Java
- Eingebettete Prozessoren
  - Digitale Signalprozessoren
  - Multimedia-Prozessoren
  - *Very Long Instruction Word*-Maschinen
  - Netzwerk-Prozessoren
- Anforderungen an Compiler für Eingebettete Systeme
  - Code-Qualität vs. Geschwindigkeit des Compilers

# Entwicklungsprozess Eingebetteter Systeme



# Werkzeuge zur Code-Generierung



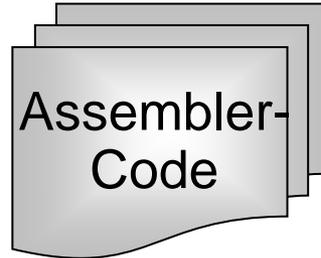


## Quellcode

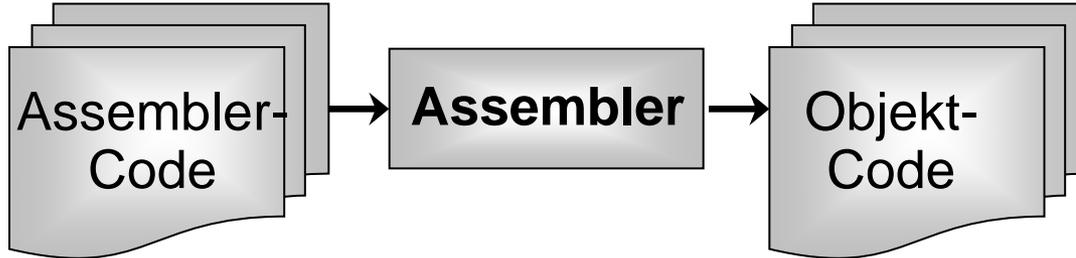
- Von Menschen les- / verstehbare Programmiersprache
- Hochsprachliche Konstrukte: Schleifen, Prozeduren, Variablen
- Hohes Abstraktionsniveau: Maschinenunabhängige Algorithmen

## Assemblercode

- Symbolischer Maschinencode
- Für Menschen eingeschränkt les- / verstehbar
- Maschinensprachen-Konstrukte: ALU-Befehle, Register, ...
- Niedriges Abstraktionsniveau: Maschinenabhängige Darstellung



<code>.align</code>	<code>1</code>	– Lesbare Textdarstellung
<code>.global</code>	<code>encode</code>	– Keine / wenige reale Adressen
<code>.type</code>	<code>encode,@function</code>	– Statt dessen: Symbolische Adressen
<code>encode:</code>		z.B. <code>encode, h, tqmf</code>
<code>mov</code>	<code>%d15, %d5</code>	
<code>mov</code>	<code>%d12, %d4</code>	
<code>movh.a</code>	<code>%a12, HI:h</code>	<i># Lade Adresse von array h nach A12</i>
<code>lea</code>	<code>%a12, [%a12] LO:h</code>	
<code>movh.a</code>	<code>%a13, HI:tqmf</code>	<i># Lade Adresse von array tqmf nach A13</i>
<code>lea</code>	<code>%a13, [%a13] LO:tqmf</code>	
<code>ld.w</code>	<code>%d14, [%a13] 4</code>	<i># Lade tqmf[1] nach D14</i>
<code>ld.w</code>	<code>%d10, [%a12] 4</code>	<i># Lade h[1] nach D10</i>
<code>mul</code>	<code>%d14, %d10</code>	<i># Multipliziere</i>

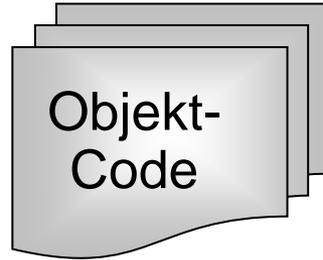


## Objektcode

- Binärdarstellung von Assemblercode, nicht mehr lesbar
- Keine Klartext-Mnemonics, statt dessen 0/1-Sequenzen
- Wenn möglich, symbolische Adressen durch reale ersetzt

## Assembler

- Zeilenweise Übersetzung  
Assembler-Befehle → Maschinen-Befehle
- Innerhalb eines Assembler-Files: Adress-Auflösung



## Übersetzung

(Beispiel: Infineon TriCore 1.3)

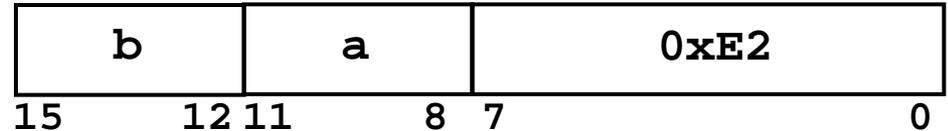
mul %d14, %d10



MUL

D[a], D[b] (SRR)

1010 1110 11100010



## Adress-Auflösung

- Symbolische Adresse **h** in gleichem Assembler-File deklariert:
  - Symbol **h** ist Assembler bekannt
  - Ersetzung von **h** durch relative Adresse, relativ in dem Objekt-File
- **h** ist Assembler unbekannt:
  - Adress-Auflösung erfolgt später

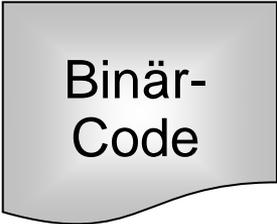


## Binärcode

- Ausführbare Programm-Darstellung
- Alle symbolischen Adressen durch reale ersetzt
- Niedrigstes Abstraktionsniveau

## Linker

- Vereinigung vieler Objektcodes und Bibliotheken zu einem ausführbaren Programm
- Symbol-Auflösung mit Hilfe von Objektcode-Bibliotheken
- Code-Anordnung im Speicher

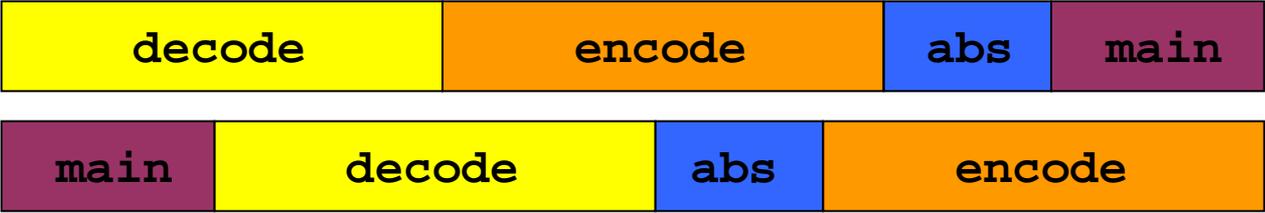


### Beispiel Symbol-Auflösung

- Objektcode enthält Sprung zu externer Funktion: `call abs`
- Suche `abs` in allen anderen Objektcodes & Bibliotheken
- Füge Code von `abs` dem Binärcode zu

### Beispiel Speicher-Layout des Binärcodes

- Binärcode besteht aus Funktionen `decode`, `encode`, `abs`, `main`



- Speicher-Anordnung definiert abschließend reale Adressen

# Inhalte des Kapitels

## 2. Compiler für Eingebettete Systeme – Anforderungen & Abhängigkeiten

- Werkzeuge zur Code-Generierung
  - Compiler, Assembler, Linker
  - Quellcode, Assemblercode, Objektcode, Binärcode
- Quellsprachen für Compiler für Eingebettete Systeme
  - C, C++, Java
- Eingebettete Prozessoren
  - Digitale Signalprozessoren
  - Multimedia-Prozessoren
  - *Very Long Instruction Word*-Maschinen
  - Netzwerk-Prozessoren
- Anforderungen an Compiler für Eingebettete Systeme
  - Code-Qualität vs. Geschwindigkeit des Compilers

# Quellsprachen für Compiler für Eingebettete Systeme

## Im Folgenden

- Kurzer Abriss über gebräuchlichste Sprachen
- Kein Anspruch auf Vollständigkeit!

## Imperative Programmiersprachen

- C

## Objektorientierte Programmiersprachen

- C++
- Java

# ANSI-C: Eigenschaften

## – Rein imperativ

- Keine Objektorientierung: keine Klassen, keine Objekte
- C-Programm: Menge von Funktionen
- Funktion `main`: Ausgezeichnete Startfunktion
- Funktionen: Folgen von Befehlen, sequenzielle Abarbeitung

```
int filtep( int rlt1, int a11, int rlt2, int a12 )
{
    long p1, p12;
    p1 = 2 * rlt1;
    p1 = (long) a11 * p1;
    p12 = 2 * rlt2;
    p1 += (long) a12 * p12;
    return( (int)(p1 >> 15) );
}
```

# ANSI-C: Eigenschaften

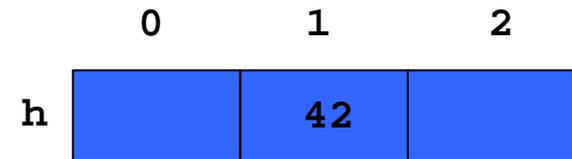
- **Standardisierte Programmiersprache**
  - ISO/IEC 9899:1999 (E)
  
- **Standard-Datentypen**
  - `signed / unsigned char`
  - `signed / unsigned short`
  - `signed / unsigned int`
  - `signed / unsigned long`
  - `signed / unsigned long long`
  - `float, double, long double, _Bool`

# ANSI-C: Eigenschaften

## – Zusammengesetzte Datentypen

### – Felder

```
int h[3];
h[1] = 42;
```



### – Strukturen

```
struct point { int x; char y; } p;
p.x = 42;
```



### – Varianten

```
union point { int x; char y; } p;
p.y = 42;
```

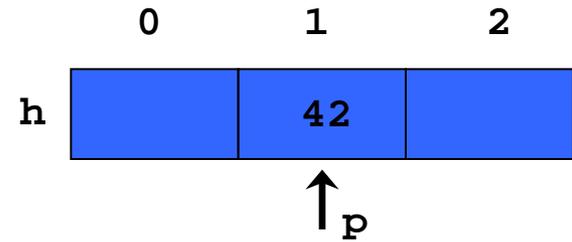


# ANSI-C: Eigenschaften

## – Zeiger und Speicherverwaltung

### – Zeiger

```
int h[3];  
int *p = &h[1];  
h[1] = 42;  
*p = 12;
```

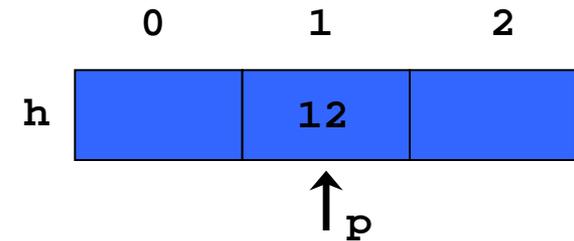


# ANSI-C: Eigenschaften

## – Zeiger und Speicherverwaltung

### – Zeiger

```
int h[3];
int *p = &h[1];
h[1] = 42;
*p = 12;
```



### – Dynamische Speicherverwaltung

```
char *p = (char *) malloc( 100 ); /* Allokation von 100 Bytes */
p[1] = 42;
free( p ); /* Speicher-Freigabe */
```

*Dynamische Speicherverwaltung explizit durch Programmierer!*

## ANSI-C: Eigenschaften

### – Architekturabhängigkeit & unspezifiziertes Verhalten

– Bit-Breite von `int`  $\cong$  Wortbreite des Prozessors

`int` auf 16-Bit Maschine: [ -32768, 32767 ]

`int` auf 32-Bit Maschine: [ -2147483648, 2147483647 ]

– Verhalten des `>>`-Operators (shift right)

*logisch* – Most Significant Bit (MSB) wird mit '0' gefüllt:

$$-8 \gg_1 1 = 1000 \gg_1 1 = 0100 = 4$$

*arithmetisch* – MSB wird mit altem MSB gefüllt:

$$-8 \gg_a 1 = 1000 \gg_a 1 = 1100 = -4$$


– Vorzeichenbehaftung von `char`:

`signed char` [ -128, 127 ] vs. `unsigned char` [ 0, 255 ]



# ANSI-C: Diskussion

## Vorteile

- Standardisierte Hochsprache, weite Verbreitung
- Viele existierende Tools zur Code-Generierung
- Viel bereits existierender Quellcode (*open source* & proprietär)
- Trotz Hochsprache: *Low-level* Programmierung möglich
- Maschinenähe
- Aufwand für Compilerentwurf noch akzeptabel



## Nachteile

- Maschinennähe, Mangelnde Portabilität von Quellcodes
- Programmierer-verantwortliche Speicherverwaltung fehleranfällig
- Keinerlei Objektorientierung



# ANSI-C++: Eigenschaften

## ANSI-C + Objektorientierung + ...

- Klassen & Objekte
- Elementfunktionen
- Konstruktoren & Destruktoren
- Vererbung
- Schutz von Klassen-Elementen: `public`, `protected`, `private`
- Virtuelle Elementfunktionen & polymorphe Klassen
- Generische Programmierung: *Templates*
- Ausnahmebehandlung: *Exceptions*



## ANSI-C++: Vorteile

- Hochsprache, erfüllt Wunsch nach OO in Eingebetteten Systemen
- Existierende ANSI-C Quellcodes können oft übernommen werden
- Weite Verbreitung
- Viele existierende Tools zur Code-Generierung
- Viel bereits existierender Quellcode (*open source* & proprietär)
- Trotz Hochsprache: *Low-level* Programmierung dennoch möglich
- Maschinennähe

# ANSI-C++: Nachteile



- Einige C++ Sprachkonstrukte führen zu großem Overhead für Eingebettete Systeme

## Beispiel Exceptions:

```
try {
    object o;
    ...
    Code...;
}
catch( E ) {
    // Fehlerbehandlung
}
```

 *Exception E* wird hier geworfen...

 ... und hier abgefangen


*o muss zerstört werden!*

- *Zeit zwischen Werfen und Fangen unklar wegen Destruktoren*
- *Erhöhter Speicherbedarf wegen interner Datenstrukturen*

# ANSI-C++: Nachteile



## Beispiel Abstrakte Basisklassen:

```

class A {
    virtual bar() = 0;
}

class B : public A {
    ...
}

class C : public A {
    ...
}

```

- *B und C enthalten je eine eigene Implementierung von `bar()`.*

```
A *foo; ...; foo->bar(); // B::bar()?? C::bar???
```

- *Hoher Laufzeitbedarf wegen Typermittlung & Methodensuche.*



## Embedded C++

### Teilmenge von C++, entworfen für Eingebettete Systeme

- Keine abstrakten Basisklassen
- Keine Ausnahmebehandlung
- Keine *Templates*
- Keine *Namespaces*

*(Features such as namespaces [...] are difficult to understand, increasing the chances of programmer errors.)*

- Keine Mehrfach-Vererbung
- Keine STL-Datenstrukturen (*Standard Template Library*)

*[Embedded C++ Slashes Code Size And Boosts Execution, [www.ghs.com/wp/ec++article2.html](http://www.ghs.com/wp/ec++article2.html)]*



# Java: Vorteile

## **Konsequent objektorientierte Programmiersprache**

- Modulare Struktur, exzellente SW-Kapselung
- Gute Daten-Typisierung
- Gute Sprachkonstrukte zur Modellierung von Verhalten & Kontrolle
- Mathematisches Modell ähnlich, aber besser als C++
- Transparenter Speicherschutz, automatische *Garbage Collection*
- Code lesbarer als C++
- Keine Zeiger
- Java Byte Code Interpreter: Hohe Portabilität

## Java: Nachteile



### Enorm hoher Ressourcenbedarf

- Nachteile von Javas OO-Konstrukten ähnlich zu C++
- Byte Code Interpretation zur Laufzeit
- *Just-In-Time* Übersetzung in Eingebetteten Systemen impraktikabel
- Realzeit-Verhalten der *Garbage Collection*?
- Derzeit: selbst schlankes Java (EmbeddedJava) für schnelle und ressourcenbeschränkte Systeme ungeeignet

### Aus Suns Lizenzbestimmungen zu Java:

*„Software is not designed or licensed for use in on-line control of aircraft, air traffic, aircraft navigation or aircraft communications; or in the design, construction, operation or maintenance of any nuclear facility.“*  
*[[java.sun.com/products/plugin/1.2/license.txt](http://java.sun.com/products/plugin/1.2/license.txt)]*

# Inhalte des Kapitels

## 2. Compiler für Eingebettete Systeme – Anforderungen & Abhängigkeiten

- Werkzeuge zur Code-Generierung
  - Compiler, Assembler, Linker
  - Quellcode, Assemblercode, Objektcode, Binärcode
- Quellsprachen für Compiler für Eingebettete Systeme
  - C, C++, Java
- Eingebettete Prozessoren
  - Digitale Signalprozessoren
  - Multimedia-Prozessoren
  - *Very Long Instruction Word*-Maschinen
  - Netzwerk-Prozessoren
- Anforderungen an Compiler für Eingebettete Systeme
  - Code-Qualität vs. Geschwindigkeit des Compilers

# Eingebettete Prozessoren

- **Digitale Signalprozessoren (DSPs)**
- **Multimedia-Prozessoren**
- ***Very Long Instruction Word*-Maschinen (VLIW)**
- **Netzwerk-Prozessoren (NPUs)**

# Digitale Signalprozessoren

## Eigenschaften

- Optimiert für Digitale Signalverarbeitung (z.B. Filter, Fourier-Transformation, ...)
- Heterogene Registersätze, eingeteilt für Spezialzwecke
- Teilweise parallele Befehlsabarbeitung
- Spezielle Adressrechenwerke / Adressierungsmodi
- *Multiply-Accumulate*-Befehl ( $a = a + b * c$ )
- *Zero-Overhead Loops*
- Sättigungsarithmetik
- Effizienz und Realzeitverhalten extrem wichtig

# DSPs: Heterogene Registersätze

## Beispiel Infineon TriCore 1.3:

- Separate Adress- & Datenregister

<i>Address Registers</i>	<i>Data Registers</i>
A15	D15
A14	D14
A13	D13
A12	D12
A11	D11
A10	D10
A9	D9
A8	D8
A7	D7
A6	D6
A5	D5
A4	D4
A3	D3
A2	D2
A1	D1
A0	D0

# DSPs: Heterogene Registersätze

## Beispiel Infineon TriCore 1.3:

- Separate Adress- & Datenregister
- Register mit besonderer Bedeutung
- 64-bit Datenregister (*extended Regs*)
- Oberer & unterer Kontext (*UC & LC*): *UC* bei Funktionsaufruf automatisch gesichert, *LC* nicht

<i>Address Registers</i>		<i>Data Registers</i>		
<i>A15 (Implicit AREG)</i>		<i>D15 (Implicit DREG)</i>		E14
A14		D14		
A13		D13		E12
A12		D12		
<i>A11 (Return Addr)</i>		D11		E10
<i>A10 (Stack Ptr)</i>		D10		
<i>A9 (Global AREG)</i>		D9		E8
<i>A8 (Global AREG)</i>		D8		
A7		D7		E6
A6		D6		
A5		D5		E4
A4		D4		
A3		D3		E2
A2		D2		
<i>A1 (Global AREG)</i>		D1		E0
<i>A0 (Global AREG)</i>		D0		

## DSPs: Teilweise Parallelität

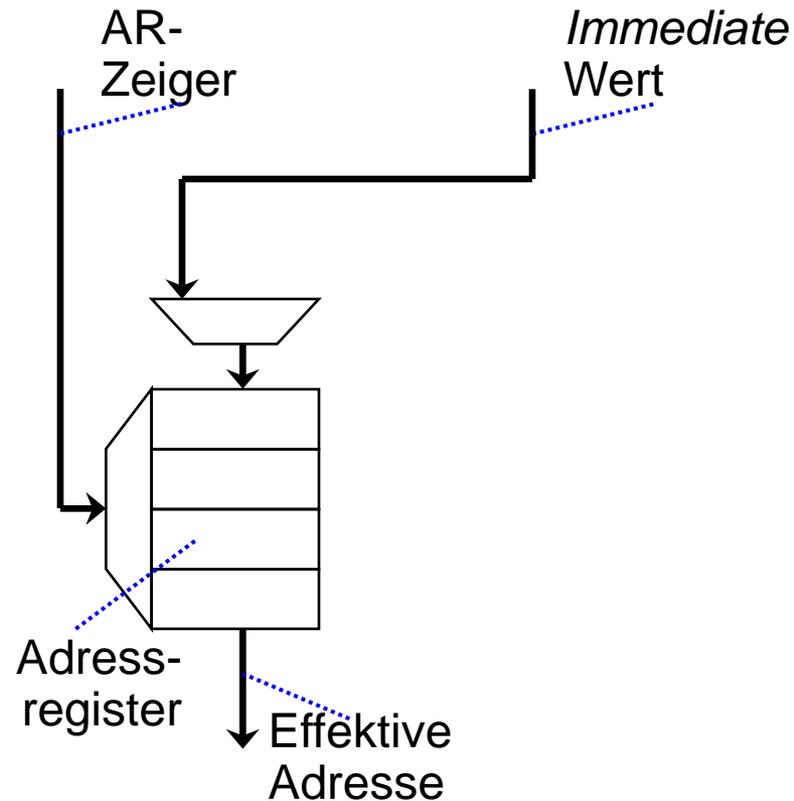
### Beispiel Infineon TriCore 1.3:

- *Integer-Pipeline:* Arithmetische Befehle, bedingte Sprünge
- *Load/Store-Pipeline:* Speicherzugriffe, Adressarithmetik, unbedingte Sprünge, Funktionsaufrufe
- *Loop-Pipeline:* Schleifen-Befehle
  
- *Teilweise Parallelität:*
  - *Pipelines* arbeiten im Idealfall unabhängig / parallel
  - Wenn nicht Idealfall:  
*Stall* in *L/S-Pipeline* → *Stall* in *I-Pipeline* und umgekehrt

## DSPs: Address Generation Units (AGUs)

### Allgemeine Architektur von Adressrechenwerken:

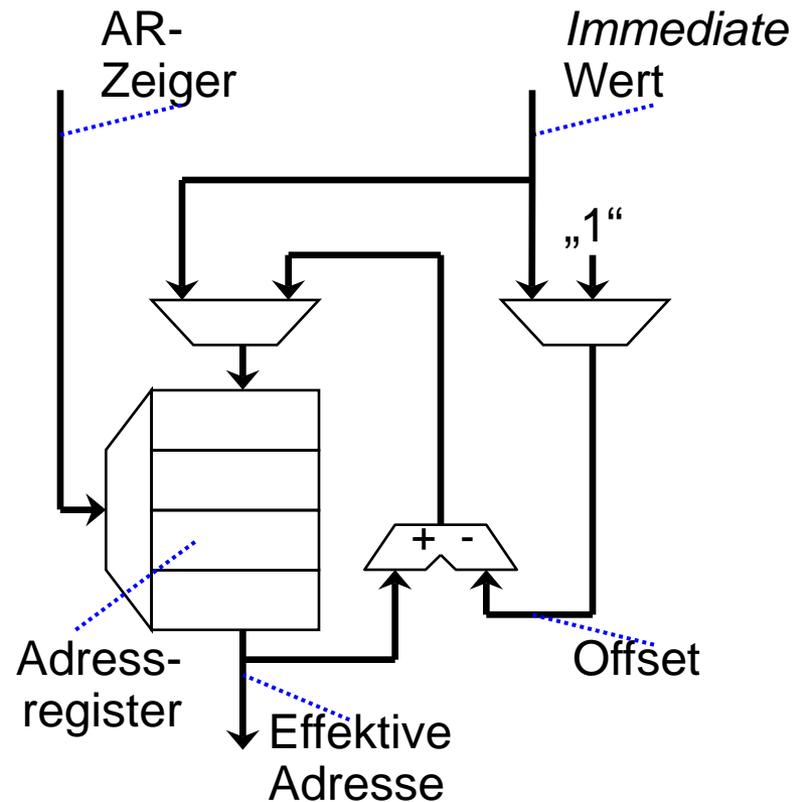
- Adressregister (AR) enthalten *effektive Adressen* zur Speicher-Adressierung
- Befehlsword codiert, welches AR zu nutzen ist (*AR-Zeiger*)
- ARs können explizit mit im Befehlsword codierten Konstanten geladen werden (*Immediates*)



## DSPs: Address Generation Units (AGUs)

### Allgemeine Architektur von Adressrechenwerken:

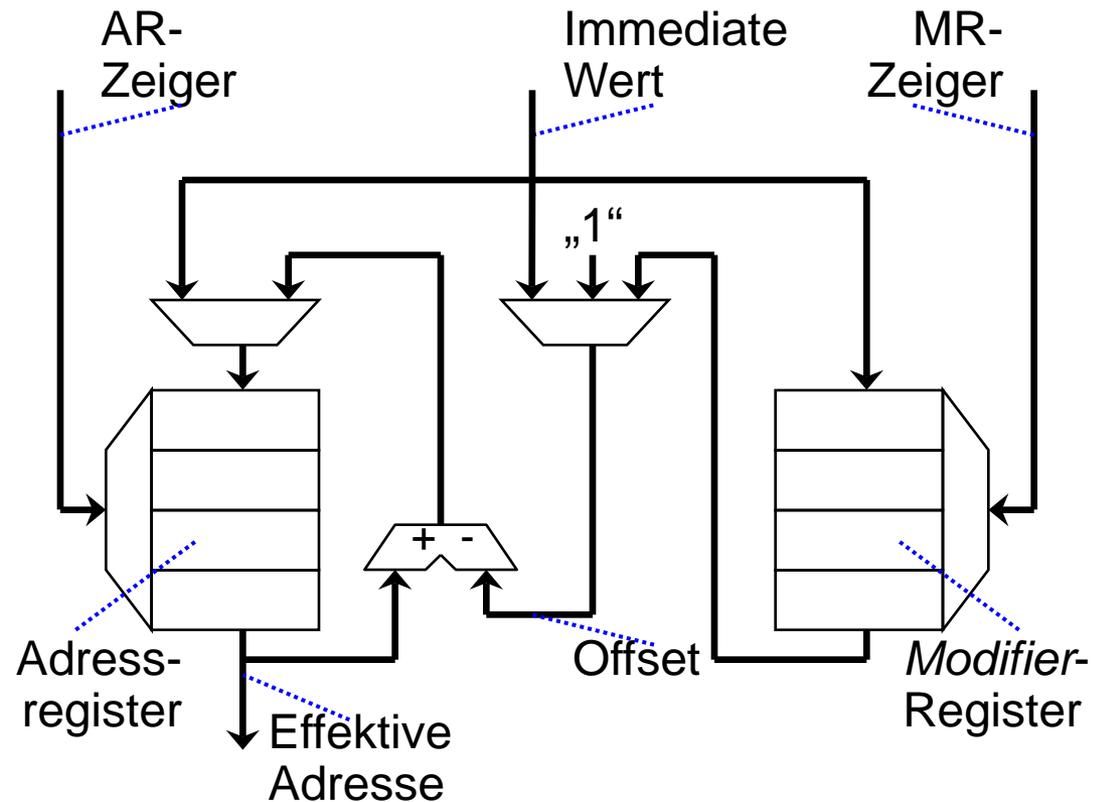
- ARs können über einfache ALU erhöht / erniedrigt werden
- Erhöhung / Erniedrigung um *Offset* als *Immediate-Wert*
- Inkrement / Dekrement um Konstante „1“ als *Offset*



## DSPs: Address Generation Units (AGUs)

### Allgemeine Architektur von Adressrechenwerken:

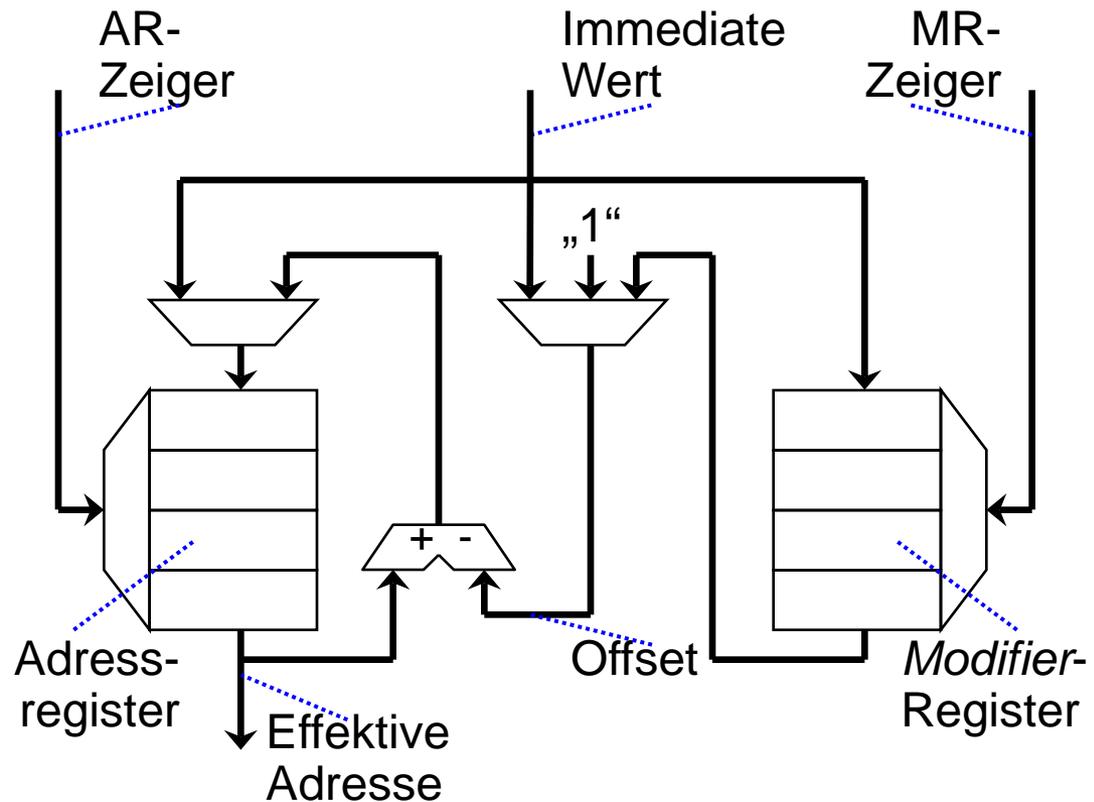
- Inkrement / Dekrement um Inhalt von *Modifier-Register (MR)*
- Befehlswort codiert, welches MR zu nutzen ist (*MR-Zeiger*)
- MRs können explizit mit *Immediate-Werten* geladen werden



## DSPs: Address Generation Units (AGUs)

### Allgemeine Architektur von Adressrechenwerken:

- AR laden:  $AR = \langle const \rangle$
- MR laden:  $MR = \langle const \rangle$
- AR ändern:  $AR \pm \langle const \rangle$
- *Auto-Increment*:  $AR \pm „1“$
- *Auto-Modify*:  $AR \pm MR$
- *“Auto“-Befehle*: Parallel zu Datenpfad, keine extra Laufzeit, hocheffizient!
- *Alle anderen*: Brauchen Extra-Instruktion für Datenpfad, weniger effizient.



## DSPs: Konventioneller Code für Schleifen

### C-Code einer Schleife:

```
int i = 10;
do {
    ...
    i--;
} while ( i );
```

### Konventioneller ASM-Code: (TriCore 1.3)

```
    mov %d8, 10;
.L0:
    ...
    add %d8, -1;
    jnz %d8, .L0;
```

### Eigenschaften

- Dekrement & bedingter Sprung: Beide in *Integer-Pipeline*  
     ☞ keine parallele Ausführung
- 2 Takte \* 10 Iterationen = mind. 20 Takte Schleifen-Overhead
- *Bei Delay-Slots für Sprünge noch mehr!*

# DSPs: Optimierter Code für Schleifen

## C-Code einer Schleife:

```
int i = 10;
do {
    ...
    i--;
} while ( i );
```

## Zero-Overhead Loops: (TriCore 1.3)

```
mov %a12, 10;
.L0:
    ...
loop %a12, .L0;
```

## Eigenschaften

- Dekrement & bedingter Sprung: Parallel in *Loop-Pipeline*
- `loop`-Befehl: Verbraucht Laufzeit nur in 1. & letzter Iteration  
☞ nur 2 Takte Schleifen-*Overhead*

## Problem der *wrap around* Arithmetik (1)

**Standard-Arithmetik führt bei Über-/Unterlauf zu *wrap around***

– Problem: Ergebnisse mit *wrap around* sind...

... nicht nur **falsch**

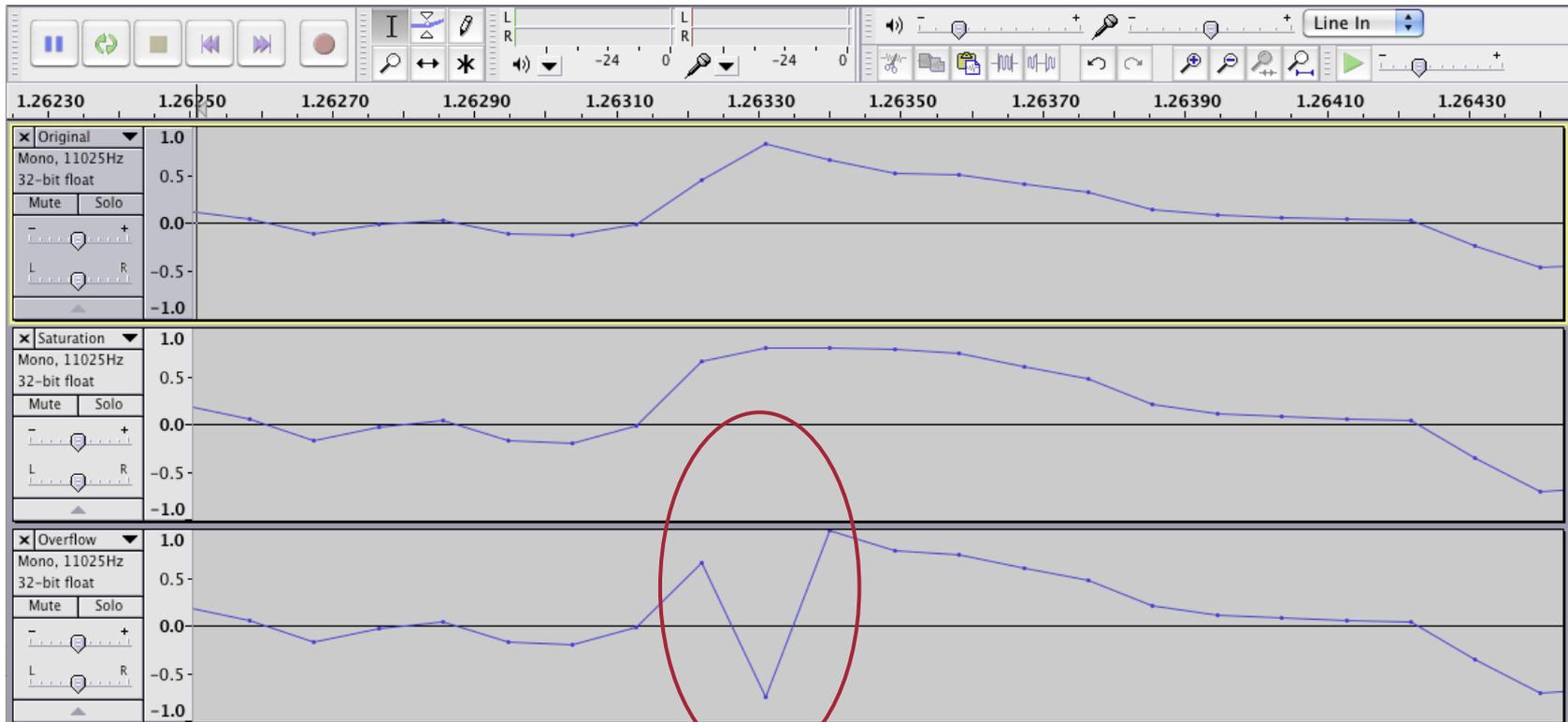
... sondern extrem **unplausibel** / nicht einmal nahe der korrekten Lösung

– Der notwendigerweise entstehende Fehler ist **maximal** (signifikanteste Bitstelle  $2^n$  geht verloren), nicht minimal! Beispiel:

$$\begin{aligned}
 (4 \text{ bit, 2er-Kompl.): } & |(7 +_{\text{wrap}} 1) - (7 +_{\text{exact}} 1)| = \\
 & |(0111_{(2)} +_{\text{wrap}} 0001_{(2)}) - 8| = \\
 & |1000_{(2)} - 8| = \\
 & |-8 - 8| = -16
 \end{aligned}$$

## Problem der *wrap around* Arithmetik (2)

- Große Fehler zwischen (mit Überlauf) berechnetem und tatsächlichem Ergebnis besonders dramatisch bei Signalverarbeitung (Verstärkung eines Audiosignals / Helligkeitsänderung eines Bildpunktes)



## Kleinerer Fehler bei Sättigungsarithmetik

Sättigungsarithmetik (*saturated arithmetic*) für Addition oder Multiplikation liefert **bei Über-/Unterlauf den jeweils maximal/minimal darstellbaren Zahlenwert.**

### Beispiele

- Betragsdarstellung (4 bit, vorzeichenlos):

$$8 +_{sat} 8 = 1000_{(2)} +_{sat} 1000_{(2)} = 7 +_{sat} 11 \rightarrow 15 \neq 18$$

$$10000_{(2)} \rightarrow 1111_{(2)} = 15 \neq 16$$

- Zweierkomplementdarstellung (4 bit, vorzeichenbehaftet):

$$7 +_{sat} 1 = 0111_{(2)} +_{sat} 0001_{(2)} = -5 -_{sat} 7 \rightarrow -8 \neq -12$$

$$1000_{(2)} \rightarrow 0111_{(2)} = 7 \neq 8$$

Insbesondere gibt es bei Sättigungsarithmetik keine Vorzeichenumkehr!

## Weiteres Beispiel

– a		0111	
b		1001	
		+	
Standard <i>wrap around</i> Arithmetik		(1)0000	
Sättigungsarithmetik		1111	
<hr/>			
<b>(a+b)/2:</b>	korrekt	1000	
	<i>wrap around</i> Arithmetik	0000	
	Sättigungsarithmetik mit >>	0111	

„fast richtig“

- Geeignet für DSP- / Multimedia-Anwendungen
  - Durch Überläufe ausgelöste *Interrupts*
    - ☞ Echtzeitbedingungen verletzt...?
  - Genaue Werte ohnehin weniger wichtig
  - *wrap around* Arithmetik liefert schlechtere Ergebnisse

# Sättigungsarithmetik: Bewertung

## Vorteil

- Plausible Ergebnisse bei Bereichsüberschreitungen

## Nachteile

- Aufwändiger in der Berechnung
- Assoziativität etc. sind verletzt

Sättigungsarithmetik und „Standardarithmetik“ können auf DSPs in der Regel wahlweise benutzt werden (es existieren entsprechende Befehlsvarianten)

„Sättigung“ im IEEE 754 *floating point* Standard:

- Bei Über-/Unterlauf entsteht  $\pm$  „unendlich“ als Ergebnis
- Weitere Operationen ändern diesen Wert nicht mehr!

# DSPs: Realzeiteigenschaften

## Das Zeitverhalten des Prozessors sollte vorhersagbar sein!

Eigenschaften, die Probleme verursachen:

- Zugriff auf gemeinsame Ressourcen
  - *Caches* mit Ersetzungsstrategien mit problematischem Zeitverhalten
  - *Unified caches* für Code und Daten gleichzeitig (Konflikte zwischen Daten und Befehlen)
  - Fließbänder (*pipelines*) mit *stall cycles* („*bubbles*“)
  - *Multi-cores* mit unvorhersagbaren Kommunikationszeiten
- Sprungvorhersage, spekulative Ausführung
- *Interrupts*, die zu jedem Zeitpunkt möglich sind
- Speicherauffrischen (*refresh*) zu jeder Zeit
- Befehle mit datenabhängigen Ausführungszeiten

☞ **So viele dieser Eigenschaften wie möglich vermeiden**

# Multimedia-Prozessoren

## Eigenschaften

- Optimiert z.B. für Bild- & Tonverarbeitung
- Bekannte kommerzielle Produkte:  
Intel MMX, SSE oder SSE2; AMD 3DNow!; Sun VIS;  
PowerPC AltiVec; HP MAX
- Motivation: Multimedia-Software nutzt oft nicht die gesamte Wortlänge eines Prozessors (d.h. `int`), sondern nur Teile (z.B. `short` oder `char`).
- SIMD-Prinzip: *Single Instruction, Multiple Data*
- Parallele Bearbeitung mehrerer „kleiner“ Daten durch 1 Befehl

## SISD vs. SIMD-Ausführung

### Aufgabe: Addiere zweimal je 2 `short`-Variablen

– *SISD-Prinzip (Single Instruction, Single Data):*

Lade erste 2 Summanden in Register,

`int`-Addition,

Lade zweite 2 Summanden in Register,

`int`-Addition

☞ Kosten: 2 volle Additionen

– *SIMD-Prinzip (Single Instruction, Multiple Data):*

Lade erste 2 Summanden in obere Halb-Register,

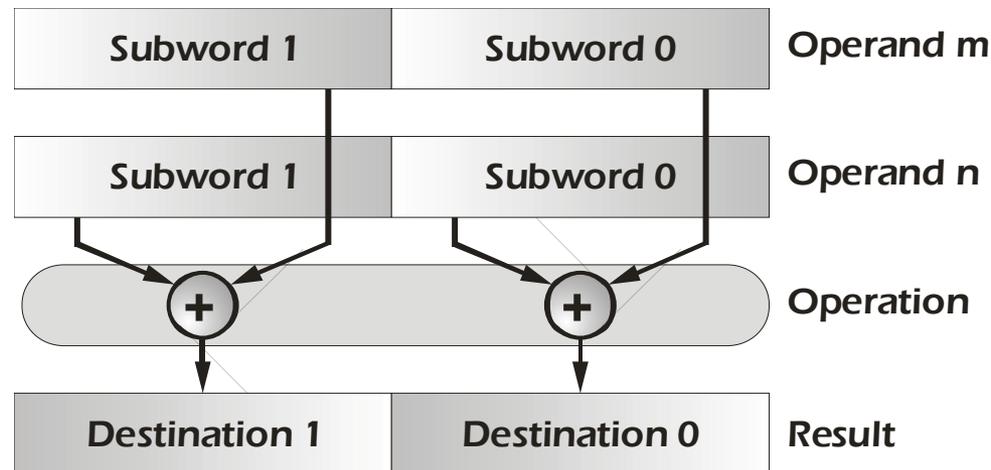
Lade zweite 2 Summanden in untere Halb-Register,

SIMD-Addition

☞ Kosten: 1 Addition

# Veranschaulichung SIMD-Addition

## SIMD Halbwort-Addition:



- SIMD-Instruktionen auch für Viertel-Worte gebräuchlich:  
☞ 4 parallele `char`-Additionen bei 32-bit Prozessor

# *Very Long Instruction Word (VLIW)*

## **Motivation**

*Performance*-Steigerung durch erhöhte Parallelität

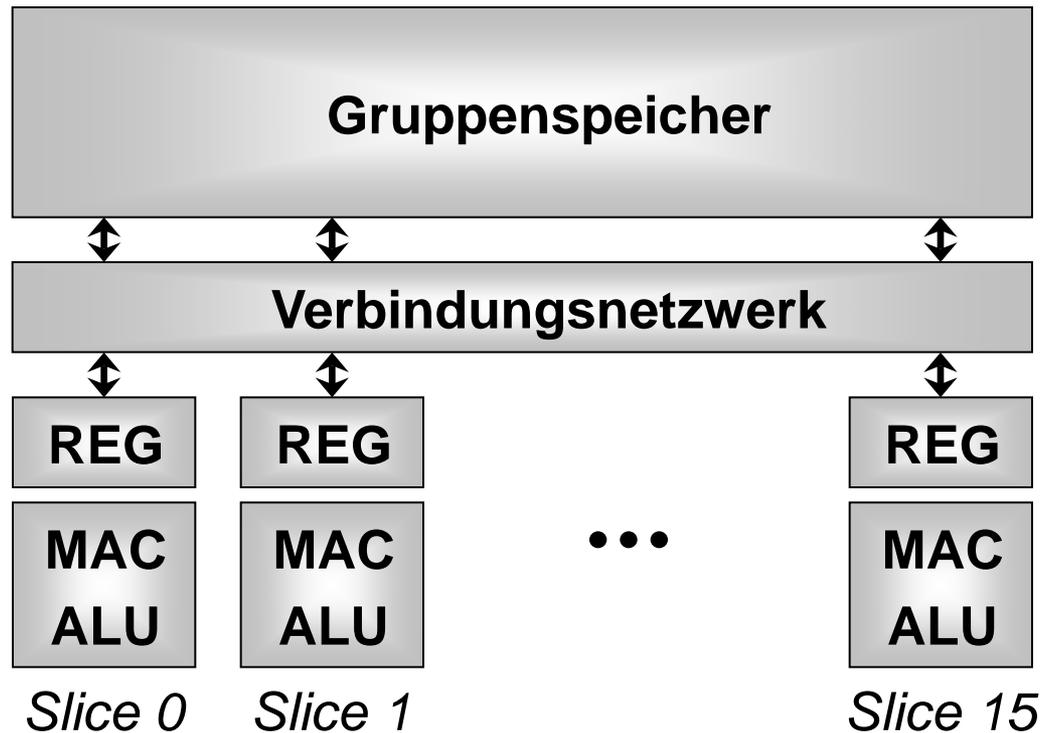
## **Konventionelle Prozessoren:**

- 1 integer-ALU
- 1 Multiplizier-Einheit
- 1 (heterogenes) Register-File

## **VLIW-Prozessoren:**

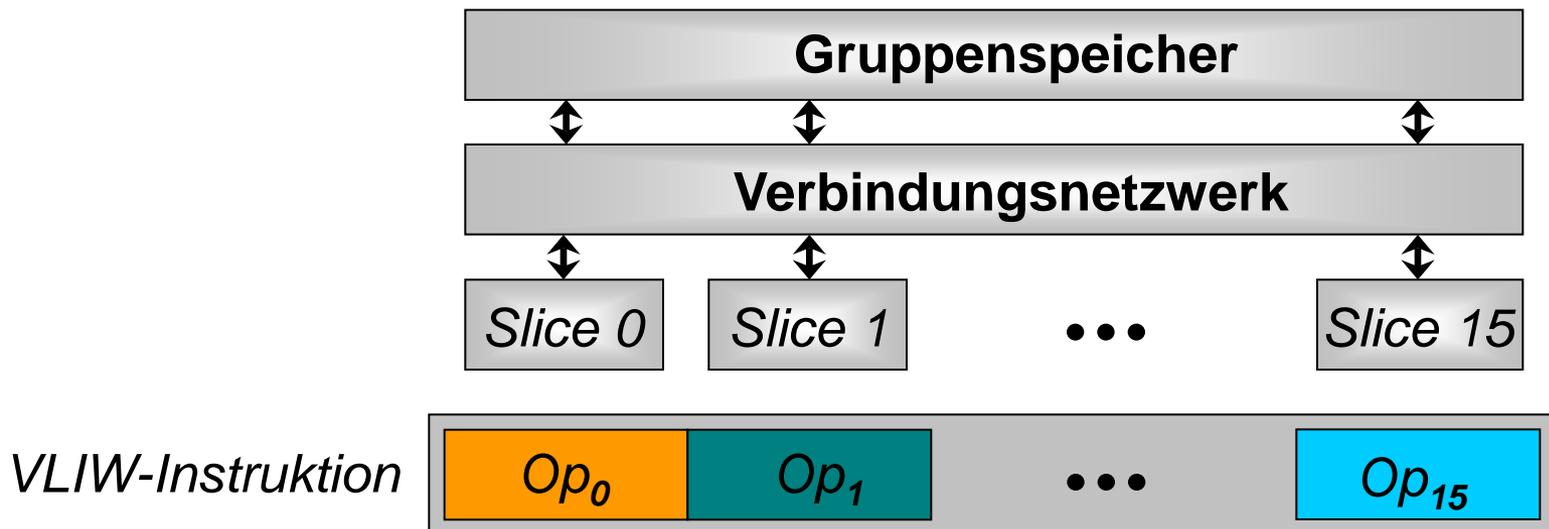
- $n$  integer-ALUs
- $n$  Multiplizier-Einheiten
- $n$  (heterogene) Register-Files
- Verbindungsnetzwerk

# Beispiel: M3 VLIW-Prozessor



## VLIW-Befehlswort

- 1 Befehlswort enthält 1 VLIW-Instruktion
- 1 VLIW-Instruktion enthält  $n$  VLIW-Operationen
- Jede Operation steuert genau eine *Functional Unit (FU)*
- Starre Zuordnung von Operationen im Befehlswort zu FUs:  
Operation 0  $\leftrightarrow$  FU 0, Operation 1  $\leftrightarrow$  FU 1, ...



# Netzwerk-Protokolle

## Kommunikation zwischen entfernten Prozessoren

- Kommunikationsmedium fehleranfällig
- Nutzdaten werden in Pakete zerteilt
- Pakete werden mit Zusatz-Informationen versehen (*Header*)

## Beispiel IPv4-Header:

0	7	15	23	26	31
Version		Länge		Service-Art	
Kennzeichnungsnummer			Flags		Fragment Offset
Gültigkeitsdauer		Protokoll		CRC	
Senderadresse					
Zieladresse					

# Bit-Pakete

## Bit-Pakete in Protokoll-*Headern*

- *Header* zerfallen in Bereiche unterschiedlicher Bedeutung
- Solche Bit-Bereiche sind nicht nach Prozessor-Wortbreiten angeordnet
- Bit-Paket:
  - Menge aufeinanderfolgender Bits
  - beliebiger Länge
  - an beliebiger Position startend
  - u.U. Wortgrenzen überschreitend

☞ Effiziente Manipulation von Daten auf Bit-Ebene notwendig!

# Bit-Pakete

## *Network Processing Units (NPUs)*

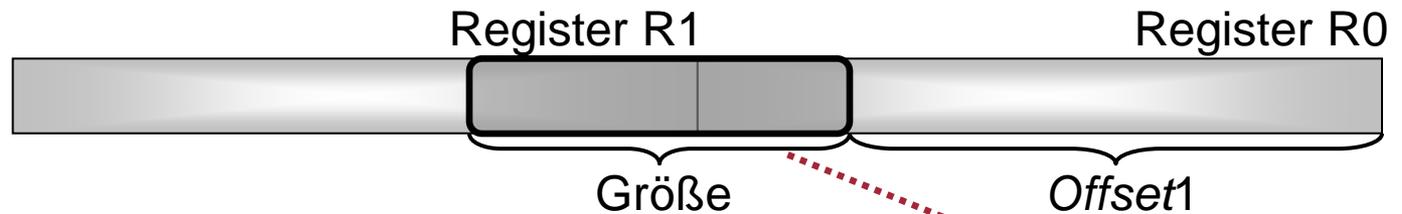
- Software zur Protokollverarbeitung:  
Hoher Code-Anteil für Verarbeitung von Bit-Paketen
- Typischer C-Code (GSM-Kernel, TU Berlin):

```
xmc[0] = (*c >> 4) & 0x7;  
xmc[1] = (*c >> 1) & 0x7;  
xmc[2] = (*c++ & 0x1) << 2;  
xmc[2] |= (*c >> 6) & 0x3;  
xmc[3] = (*c >> 3) & 0x7;
```

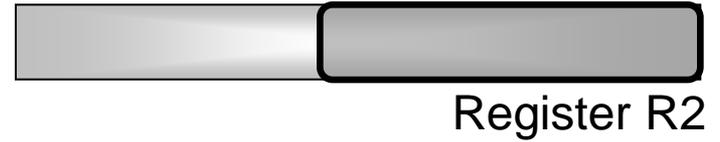
- Befehlssatz von NPUs:  
Spezial-Instruktionen zum Extrahieren, Einfügen & Bearbeiten von Bit-Paketen

# Operationen auf Bit-Paketen

## Extrahieren von Bit-Paketen

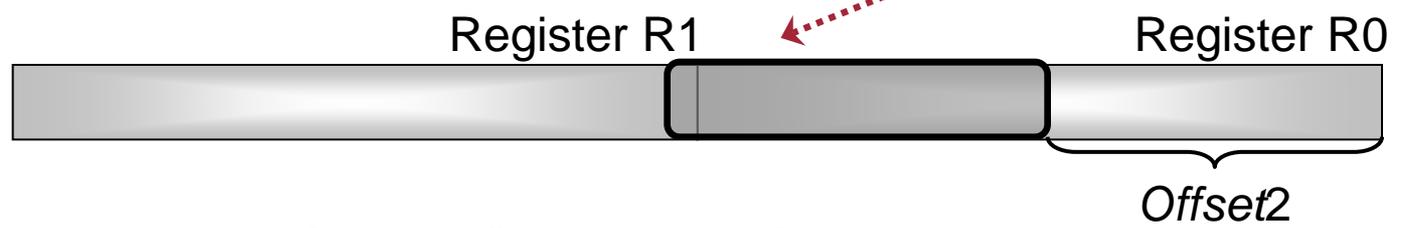
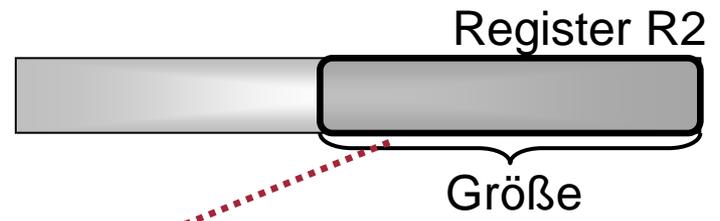


```
extr R2, R0, <Offset1>, <Größe>;
```



## Einfügen von Bit-Paketen

```
insert R0, R2, <Offset2>, <Größe>;
```



# Inhalte des Kapitels

## 2. Compiler für Eingebettete Systeme – Anforderungen & Abhängigkeiten

- Werkzeuge zur Code-Generierung
  - Compiler, Assembler, Linker
  - Quellcode, Assemblercode, Objektcode, Binärcode
- Quellsprachen für Compiler für Eingebettete Systeme
  - C, C++, Java
- Eingebettete Prozessoren
  - Digitale Signalprozessoren
  - Multimedia-Prozessoren
  - *Very Long Instruction Word*-Maschinen
  - Netzwerk-Prozessoren
- Anforderungen an Compiler für Eingebettete Systeme
  - Code-Qualität vs. Geschwindigkeit des Compilers

# Wichtigste Anforderungen an Compiler für ES

## Maximale Code-Qualität

- Laufzeit-Effizienz
- Geringer Energieverbrauch
- Geringe Codegröße
- Maximale Parallelisierung
- Echtzeitfähigkeit
- ...

## Sinnvolle Maßnahmen

- Bestmögliche Abbildung der Quell- auf die Zielsprache
- Präsenz starker Compiler-Optimierungen
- Wiederverwendung von Code-Fragmenten
- Maximale Nutzung schneller und kleiner Speicher
- Einbeziehung der WCET (*Worst-Case Execution Time*)
- ...

# Nebensächliche Anforderung

## Geschwindigkeit des Compilers

- Situation bei *Desktop*-Rechnern:
  - ✓ Großer Umfang verfügbarer Ressourcen
  - ✓ Code-Qualität von geringerem Interesse
  - ✓ Compiler sollen schnell korrekten Code generieren
  
- Situation bei Eingebetteten Systemen:
  - ❑ Code-Qualität von maximalem Interesse
  - ❑ Compiler sollen hoch-optimierten Code generieren
  - ❑ Compiler werden im ES-Entwicklungsprozess seltener aufgerufen als bei *Desktop*-Rechnern

👉 **Hohe Laufzeiten Optimierender Compiler akzeptabel!**

# Literatur

## Werkzeuge zur Code-Generierung

- John R. Levine, *Linkers & Loaders*, Morgan Kaufmann, 2000.  
ISBN 1-55860-496-0

## Programmiersprachen

- Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, Prentice Hall, 1988.  
ISBN 0-13-110362-8
- *Embedded C++ Home Page*,  
<http://www.caravan.net/ec2plus>, 2002.
- *The Real-Time Specification for Java*,  
<http://www.rtsj.org>, 2007.

# Literatur

## Prozessoren & Befehlssätze

- Peter Marwedel, *Eingebettete Systeme*, Springer, 2007.  
ISBN 978-3-540-34048-5
- Rainer Leupers, *Code Optimization Techniques for Embedded Processors*, Kluwer Academic Publishers, 2000.  
ISBN 0-7923-7989-6
- Jens Wagner, *Retargierbare Ausnutzung von Spezialoperationen für Eingebettete Systeme mit Hilfe bitgenauer Wertflussanalyse*, Dissertation Universität Dortmund, Informatik 12, 2006.

# Zusammenfassung (1)

## Werkzeuge zur Code-Generierung

- Assembler und Linker als weitere Werkzeuge neben Compiler
- Adress-Auflösung, Speicher-Layout und *Libraries*

## Quellsprachen für Compiler für Eingebettete Systeme

- ANSI-C: weitverbreitete imperative Programmiersprache; maschinennahe Programmierung möglich; fehleranfällige Speicherverwaltung; nicht-spezifizierte Sprachdetails
- C++: ähnlich wie C; hoher *Overhead* wegen mancher objektorientierten Konstrukte
- Java: sehr hoher *Overhead* wegen Byte Code Interpretation; keine Echtzeitfähigkeit wegen *Garbage Collection*
- ANSI-C gebräuchlichste Programmiersprache für Eingebettete Systeme

## Zusammenfassung (2)

### Eingebettete Prozessoren

- Z.T. sehr spezielle Befehlssätze (*multiply-accumulate*, *insert/extract*, SIMD)
- Spezielle Register und Adressrechenwerke
- Hoher Grad an Parallelität (mehrere *Pipelines*, Vielzahl funktionaler Einheiten)

### Anforderungen an Compiler für Eingebettete Systeme

- Code-Qualität primäre Anforderung
- Compiler-Laufzeit nur sekundär, im Gegensatz zu Compilern für *Desktop-Rechner*