Compilerbau

Wintersemester 2010 / 2011

Dr. Heiko Falk

Technische Universität Dortmund

Lehrstuhl Informatik 12

Entwurfsautomatisierung für Eingebettete Systeme

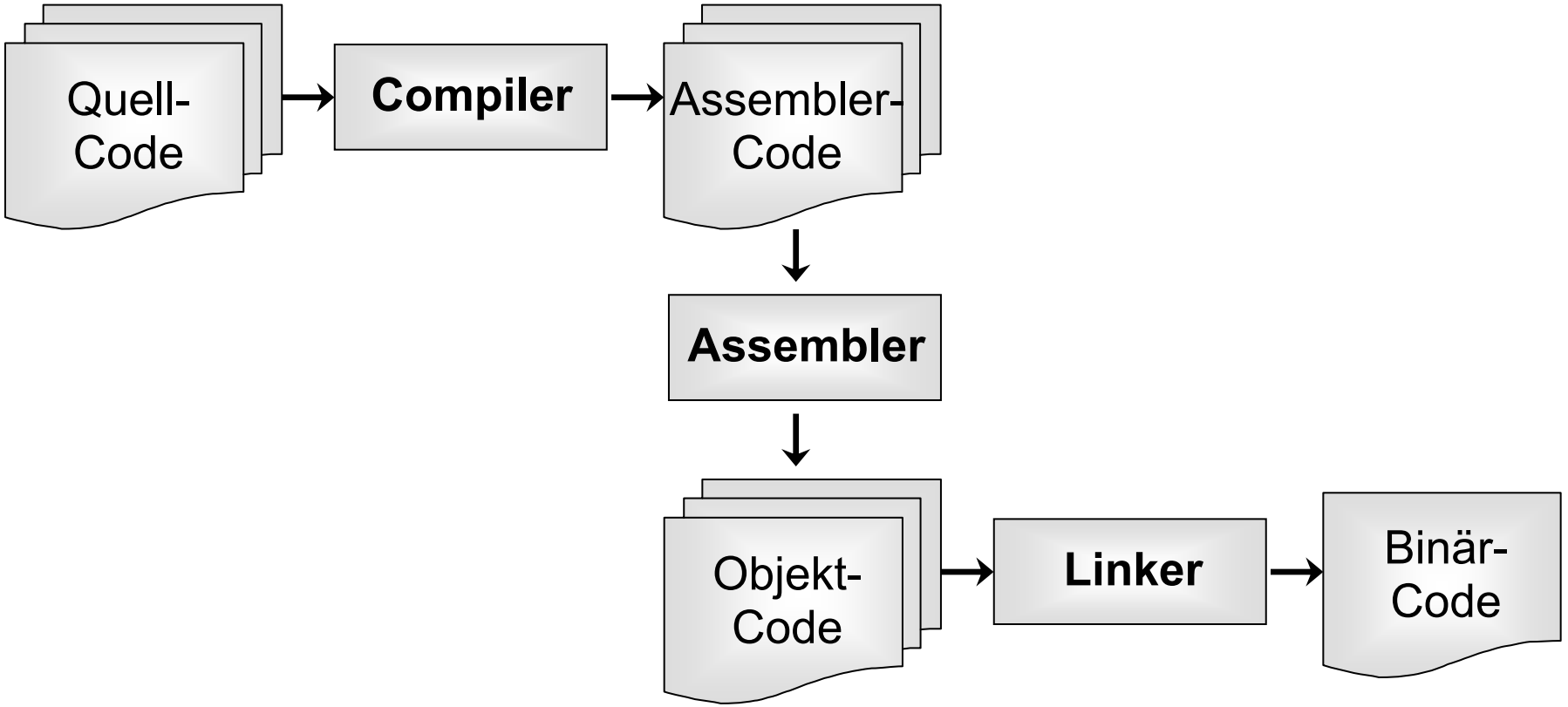
Kapitel 1

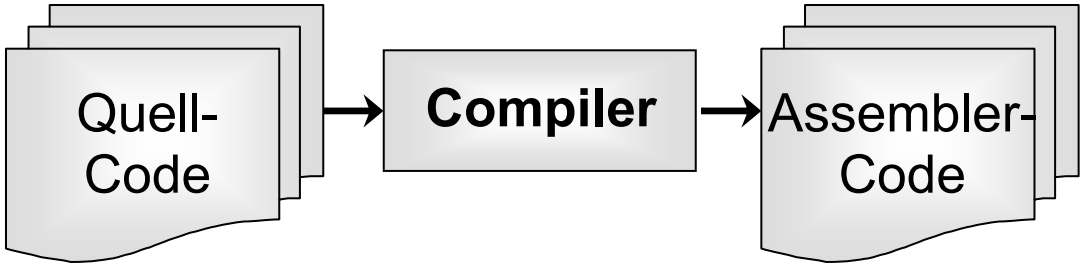
Compiler – Abhängigkeiten und Anforderungen

Gliederung der Vorlesung

- **Kapitel 1: Compiler – Abhängigkeiten und Anforderungen**
  - Werkzeuge zur Code-Generierung
  - Quellsprachen für Compiler
  - Prozessor-Architekturen
  - Anforderungen an Compiler
- **Kapitel 2: Interner Aufbau von Compilern**
- **Kapitel 3: Lexikalische Analyse (Scanner)**
- **Kapitel 4: Syntaktische Analyse (Parser)**
- **Kapitel 5: Semantische Analyse**
- **Kapitel 6: Instruktionsauswahl**
- **Kapitel 7: Register-Allokation**
- **Kapitel 8: Code-Optimierung**
- **Kapitel 9: Ausblick**

Werkzeuge zur Code-Generierung



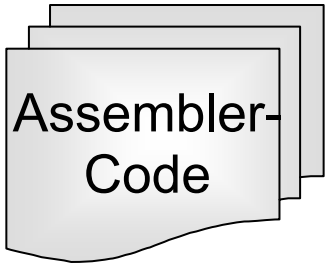


Quellcode:

- 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



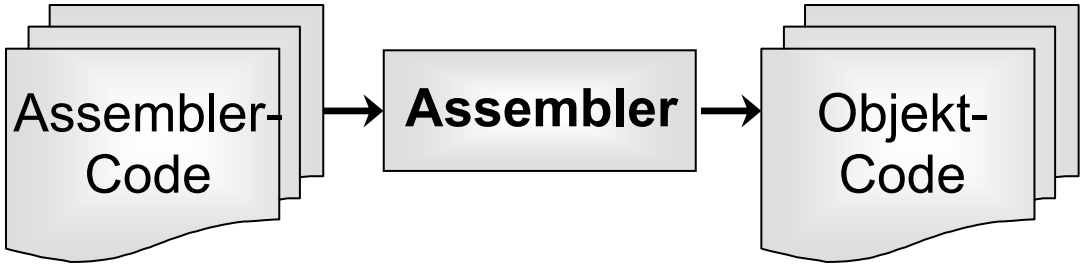

```
    .align    1
    .global   encode
    .type     encode,@function
encode:
    mov       %d15, %d5
    mov       %d12, %d4
    movh.a    %a12, HI:h
    lea       %a12, [%a12] LO:h
    movh.a    %a13, HI:tqmf
    lea       %a13, [%a13] LO:tqmf
    ld.w      %d14, [%a13] 4
    ld.w      %d10, [%a12] 4
    mul       %d14, %d10
```

- Lesbare Text-Darstellung
- Keine / wenige reale Adressen
- Statt dessen: Symbolische Adressen
  z.B. **`encode`**, **`h`**, **`tqmf`**

 # Lade Adresse von array **`h`** *nach A12*

 # Lade Adresse von array **`tqmf`** *nach A13*

 # Lade **`tqmf[1]`** *nach D14*
 # Lade **`h[1]`** *nach D10*
 # Multipliziere

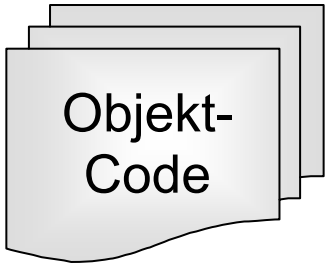


Objektcode:

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

Assembler:

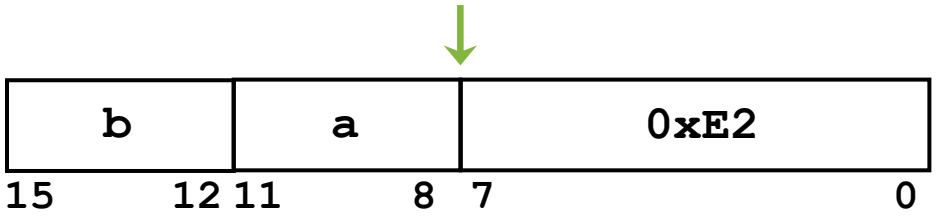
- Zeilenweise Übersetzung
  Assembler-Befehle → Maschinen-Befehle
- Innerhalb eines Assembler-Files: Adress-Auflösung



Übersetzung:

(Beispiel: Infineon TriCore 1.3)

```
mul    %d14, %d10   →   MUL    D[a], D[b] (SRR)
```



Adress-Auflösung:

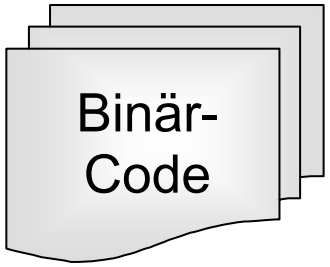
- Symbolische Adresse **`h`** in gleichem Assembler-File deklariert:
  - Symbol **`h`** ist Assembler bekannt
  - Ersetzung von **`h`** durch relative Adresse
  - Relativ innerhalb des Objekt-Files
- **`h`** ist Assembler unbekannt:
  - Adress-Auflösung erfolgt später

Binärcode:

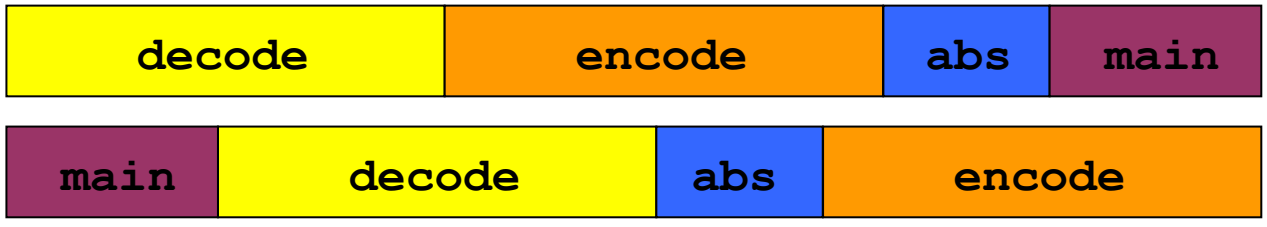
- Ausführbare Programm-Darstellung
- *Alle* symbolischen Adressen durch reale ersetzt
- Niedrigstes Abstraktionsniveau

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 hinzu
- **Beispiel Speicher-Layout des Binärcodes:**
  - Binärcode besteht aus Funktionen **`decode`**, **`encode`**, **`abs`**, **`main`**



  - Speicher-Anordnung definiert abschließend reale Adressen

Gliederung der Vorlesung

- **Kapitel 1: Compiler – Abhängigkeiten und Anforderungen**
  - Werkzeuge zur Code-Generierung
  - Quellsprachen für Compiler
  - Prozessor-Architekturen
  - Anforderungen an Compiler
- **Kapitel 2: Interner Aufbau von Compilern**
- **Kapitel 3: Lexikalische Analyse (Scanner)**
- **Kapitel 4: Syntaktische Analyse (Parser)**
- **Kapitel 5: Semantische Analyse**
- **Kapitel 6: Instruktionsauswahl**
- **Kapitel 7: Register-Allokation**
- **Kapitel 8: Code-Optimierung**
- **Kapitel 9: Ausblick**

Weit verbreitete Quellsprachen für Compiler

- **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: Folge von Befehlen, sequentielle Abarbeitung

```
int filtep( int rlt1, int al1, intrlt2, int al2 )
{
  long pl, pl2;
  pl = 2 * rlt1;
  pl = (long) al1 * pl;
  pl2 = 2 * rlt2;
  pl += (long) al2 * pl2;
  return( (int)(pl >> 15) );
}
```

ANSI-C: Eigenschaften

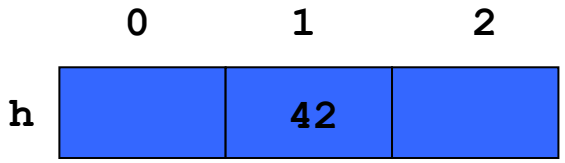
- **Standardisierte Programmiersprache:**
  - ISO/IEC 9899:1999 (E)

- **Standard-Datentypen:**
  - `signed / unsigned char`
  - `signed / unsigned short`
  - `signed / unsigned int`
  - `signed / unsigned long`
  - `signed / unsigned long long`
  - `float, double, long double`

ANSI-C: Eigenschaften

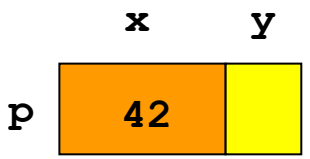
- **Zusammengesetzte Datentypen:**
  - Felder

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



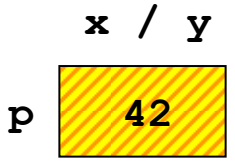
  - Strukturen

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



  - Varianten

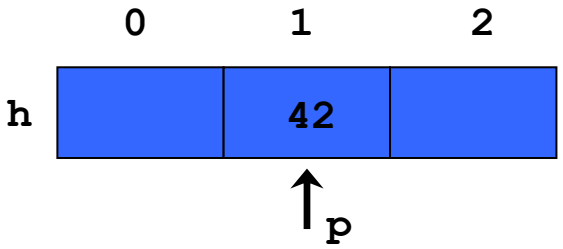
```
union point { int x; char y; } p;
p.y = 42;
```



ANSI-C: Eigenschaften

- **Zeiger & Speicherverwaltung:**
  - Zeiger

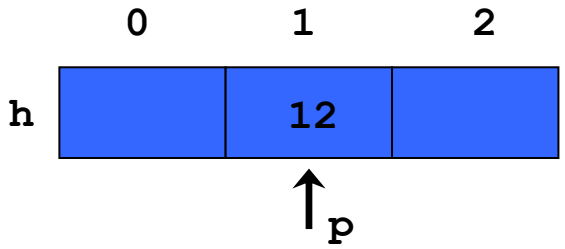
```
int h[3];
int *p = &h[1];
h[1] = 42;
*p = 12;
```



ANSI-C: Eigenschaften

- **Zeiger & 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`** ≅ 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:

    $$-8 >>_1 1 = 1000 >>_1 1 = 0100 = 4$$

    arithmetisch – MSB wird mit altem MSB gefüllt:

    $$-8 >>_a 1 = 1000 >>_a 1 = 1100 = -4$$

  - 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...

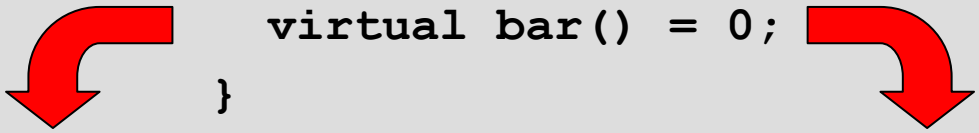
➡ ... und hier abgefangen

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

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.“

Gliederung der Vorlesung

- **Kapitel 1: Compiler – Abhängigkeiten und Anforderungen**
  - Werkzeuge zur Code-Generierung
  - Quellsprachen für Compiler
  - Prozessor-Architekturen
  - Anforderungen an Compiler
- **Kapitel 2: Interner Aufbau von Compilern**
- **Kapitel 3: Lexikalische Analyse (Scanner)**
- **Kapitel 4: Syntaktische Analyse (Parser)**
- **Kapitel 5: Semantische Analyse**
- **Kapitel 6: Instruktionsauswahl**
- **Kapitel 7: Register-Allokation**
- **Kapitel 8: Code-Optimierung**
- **Kapitel 9: Ausblick**

Prozessor-Architekturen

Grundlagen der Rechnerarchitektur:

- Sind lt. Studienplan Informationssystemtechnik bekannt. Insbes. Konzept von ALUs, Pipelines, Registern, Speichern und Grundlagen von RISC-Befehlssätzen.

☞ Da Architektur gängiger Standard-Prozessoren hinreichend bekannt, im folgenden Übersicht über Non-Standard Prozessoren aus dem Bereich Eingebetteter Systeme.

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

DSPs: Heterogene Registersätze

Beispiel Infineon TriCore 1.3:

- Separate Adress- & Datenregister

Address Registers	*Data Registers*
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

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

DSPs: Teilweise Parallelität

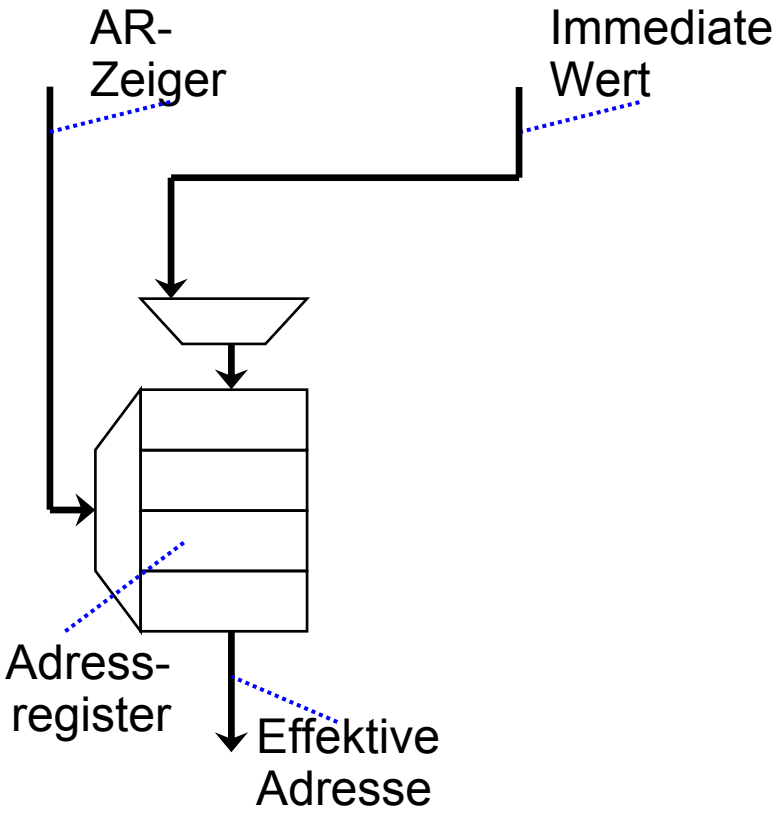
Beispiel Infineon TriCore 1.3:

- *Integer-Pipeline:* Arithmetische Befehle
  Bedingte Sprünge
- *Load/Store-Pipeline:* Speicherzugriffe, Adress-Arithmetik
  Unbedingte Sprünge, Funktionsaufrufe
- *Loop-Pipeline:* Schleifen-Befehle
- *Teilweise Parallelität:*
  - Pipelines arbeiten im Idealfall unabhängig / parallel
  - Wenn nicht Idealfall:
    Stall in L/S-Pipeline $\rightarrow$ Stall in I-Pipeline und umgekehrt

DSPs: Address Generation Units *(AGUs)*

- **Allgemeine Architektur von Adressrechenwerken:**

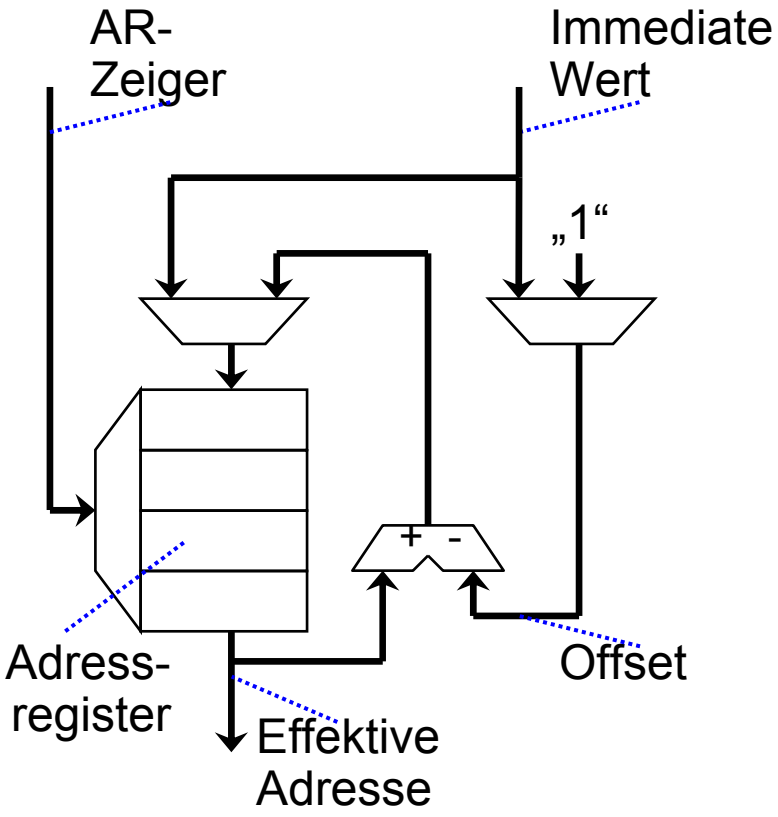
- Adressregister enthalten *effektive Adressen* zur Speicher-Adressierung
- Befehlswort codiert, welches AR zu nutzen ist *(AR-Zeiger)*
- ARs können explizit mit im Befehlswort codierten Konstanten geladen werden *(Immediates)*



DSPs: Address Generation Units *(AGUs)*

- **Allgemeine Architektur von Adressrechenwerken:**

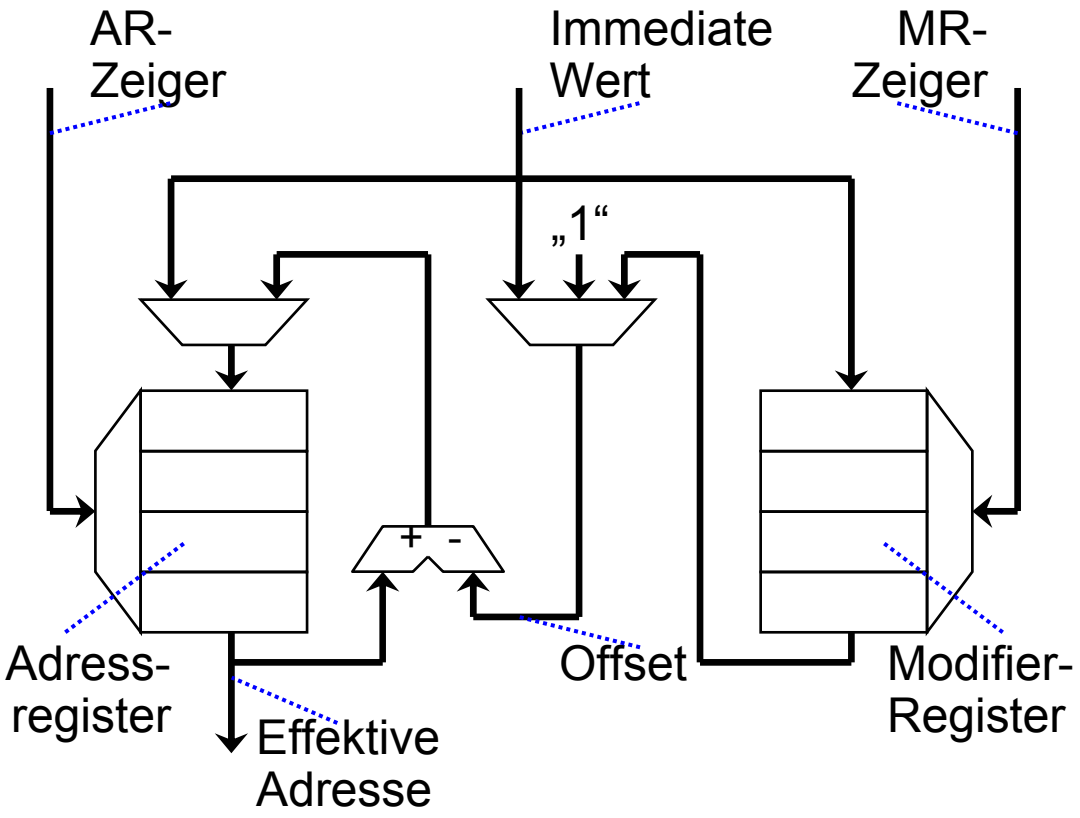
- ARs können über einfache ALU erhöht / erniedrigt werden
- Erhöhung / Erniedrigung um *Offset* als Immediate-Wert
- Inkrement / Dekrement um Konstante “1” als Offset



DSPs: Address Generation Units *(AGUs)*

- **Allgemeine Architektur von Adressrechenwerken:**

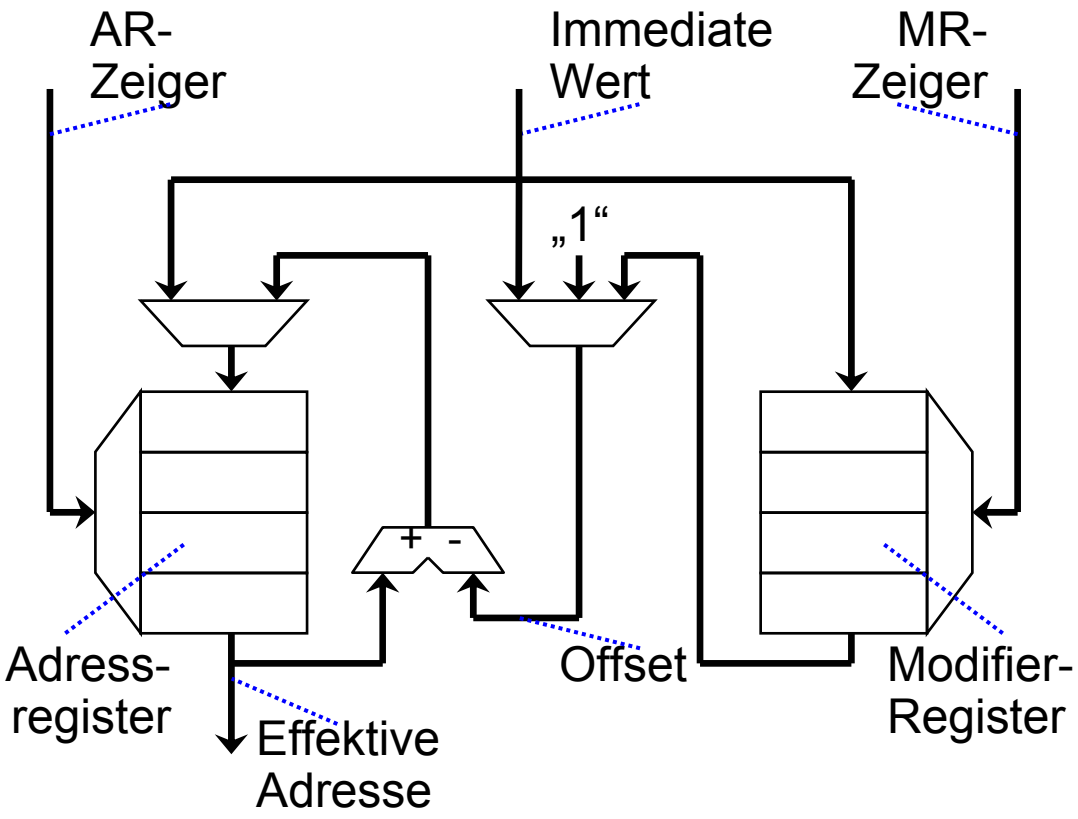
- Inkrement / Dekrement um Inhalt von Modifier-Register *(MR)*
- Befehlswort codiert, welches MR zu nutzen ist *(MR-Zeiger)*
- MRs können explizit mit Immediate-Werten geladen werden



DSPs: Address Generation Units *(AGUs)*

- AR laden: AR = <*const*>
- MR laden: MR = <*const*>
- AR ändern: AR ± <*const*>
- *Auto-Increment:* AR ± "1"
- *Auto-Modify:* AR ± MR

- *"Auto"-Befehle:* Parallel zu Datenpfad, keine extra Laufzeit, hocheffizient!
- *Alle anderen:* Brauchen Extra-Instruktion für Datenpfad, minder effizient.



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!*

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

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

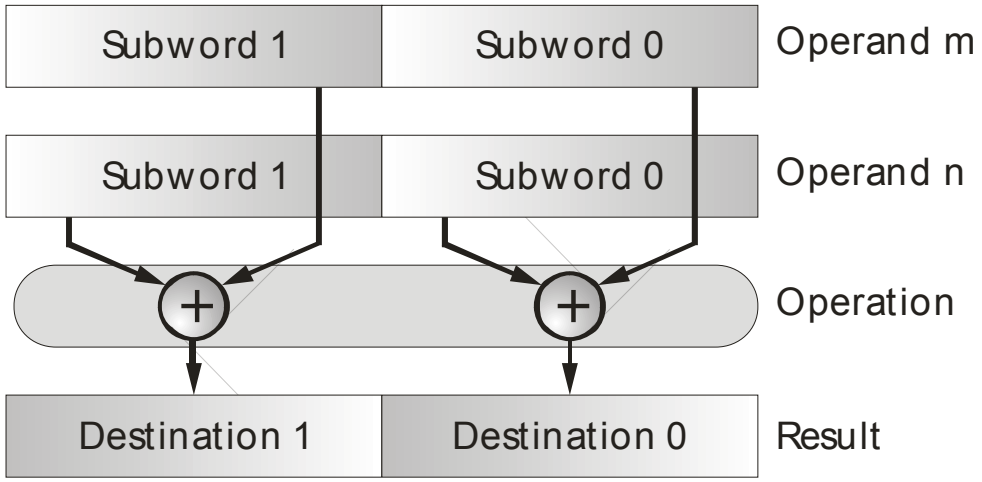
SISD vs. SIMD-Ausführung

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:

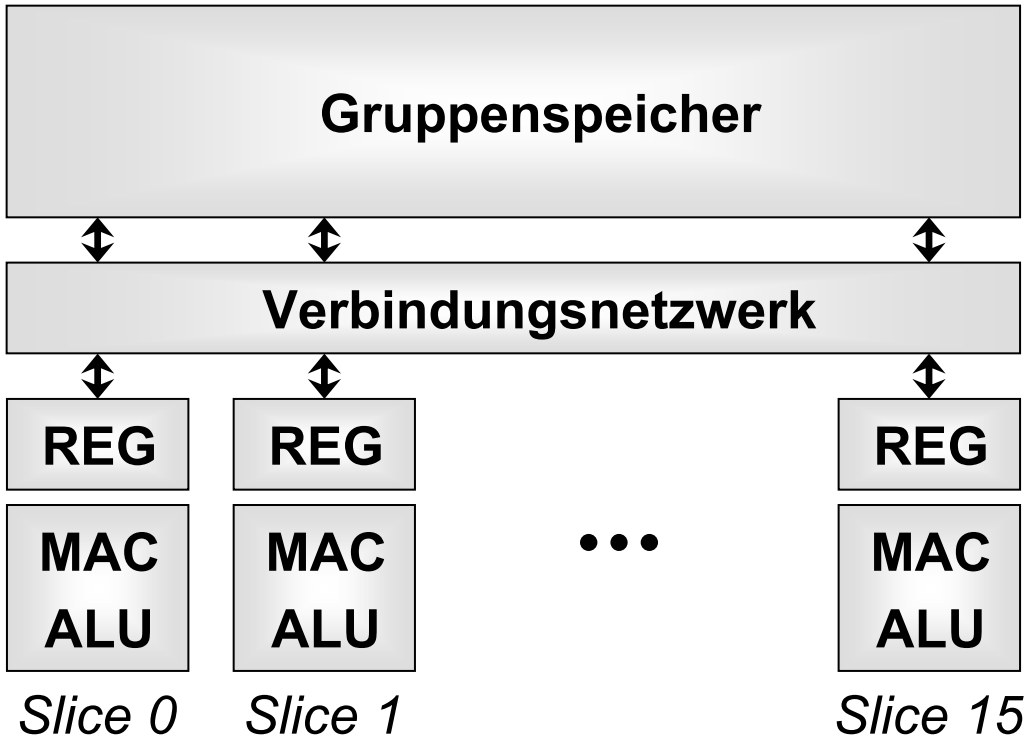


- SIMD-Instruktionen auch für Viertel-Worte gebräuchlich:
  ☞4 parallele **`char`**-Additionen bei 32-bit Prozessor

Very Long Instruction Word *(VLIW)*

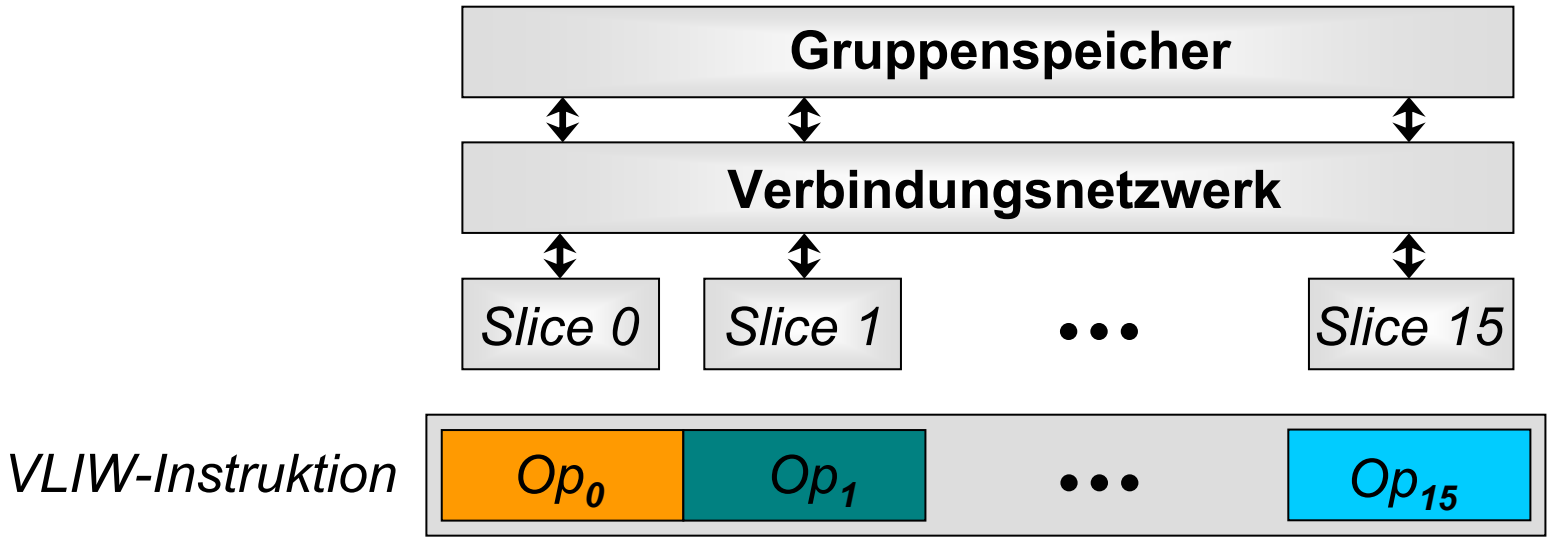
- **Motivation:**
  Performance-Steigerung durch erhöhte Parallelität
- **Konventionelle Prozessoren:**
  - 1 integer-ALU
  - 1 Multiplizier-Einheit
  - 1 (heterogenes) Register-File
- **VLIW-Prozessoren:**
  - *n* integer-ALUs
  - *n* Multiplizier-Einheiten
  - *n* (heterogene) Register-Files
  - Verbindungsnetzwerk

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 ↔ FU 0, Operation 1 ↔ FU 1, ...

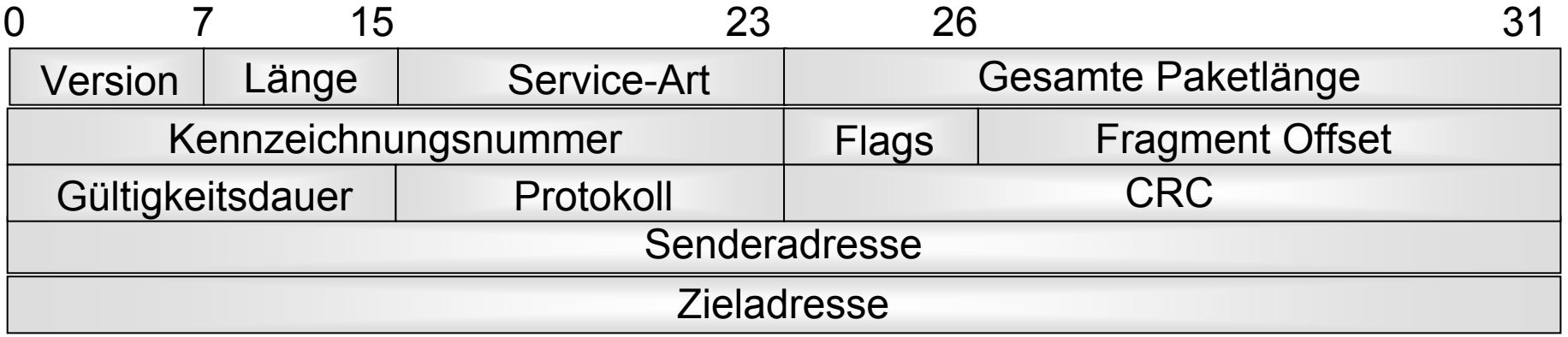


Netzwerk-Protokolle

Kommunikation zwischen entfernten Prozessoren:

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

- **Beispiel IPv4-Header:**



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

Bit-Pakete

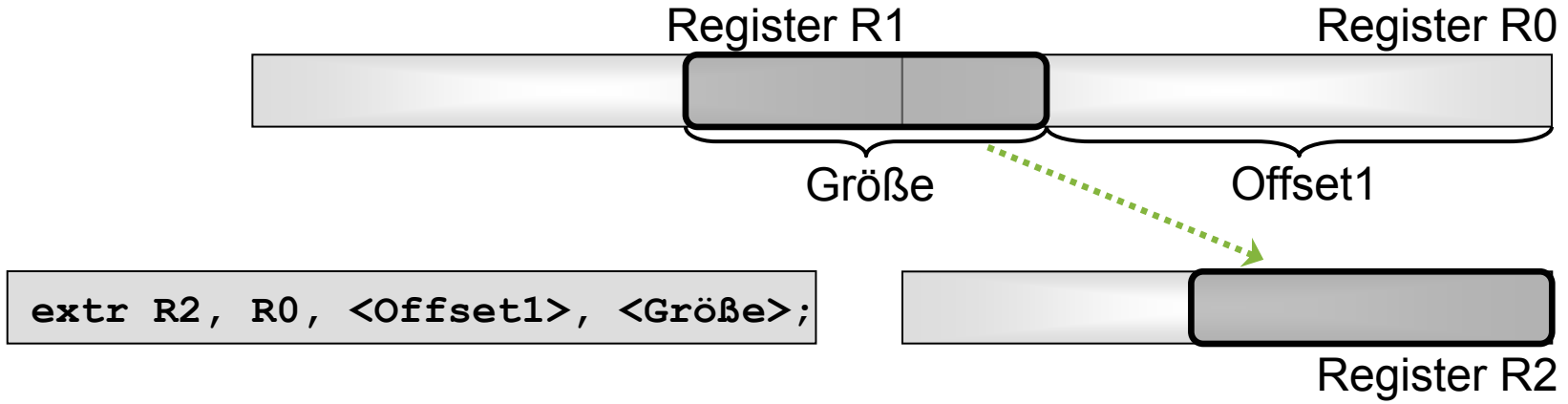
- **Network Processing Units *(NPUs)***
  - Software zur Protokollverarbeitung:
    Hoher Code-Anteil für Verarbeitung von Bit-Paketen
  - Typischer C-Code (GSM-Kernel, TU Berlin):

```
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;
```

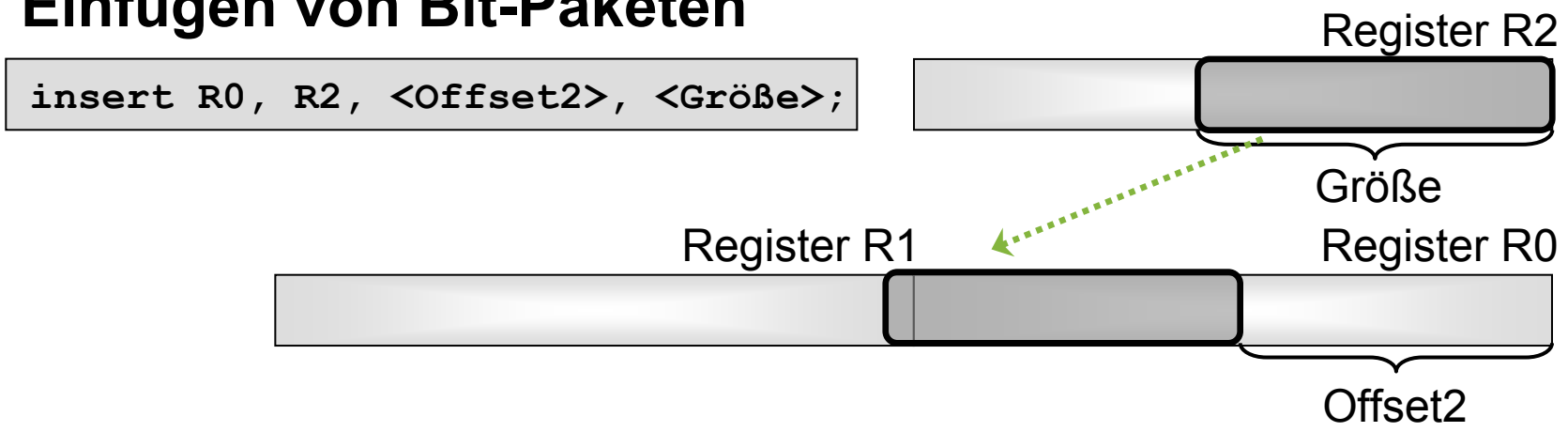
  - Befehlssatz von NPUs:
    Spezial-Instruktionen zum Extrahieren, Einfügen & Bearbeiten von Bit-Paketen

Operationen auf Bit-Paketen

- **Extrahieren von Bit-Paketen**



- **Einfügen von Bit-Paketen**



Gliederung der Vorlesung

- **Kapitel 1: Compiler – Abhängigkeiten und Anforderungen**
  - Werkzeuge zur Code-Generierung
  - Quellsprachen für Compiler
  - Prozessor-Architekturen
  - Anforderungen an Compiler
- **Kapitel 2: Interner Aufbau von Compilern**
- **Kapitel 3: Lexikalische Analyse (Scanner)**
- **Kapitel 4: Syntaktische Analyse (Parser)**
- **Kapitel 5: Semantische Analyse**
- **Kapitel 6: Instruktionsauswahl**
- **Kapitel 7: Register-Allokation**
- **Kapitel 8: Code-Optimierung**
- **Kapitel 9: Ausblick**

Wichtigste Anforderung an Desktop-Compiler

Geschwindigkeit des Compilers

- Desktop- bzw. Server-Systeme verfügen über große Ressourcen.
- Code-Qualität ist zwar nicht unwichtig, aber wegen der hohen Ressourcen nicht unbedingt zwingend.
- In typischem Entwicklungsprozess für Desktop-Systeme wird Übersetzung sehr häufig durchgeführt. *(„Mal eben schauen, was passiert, wenn ich hier etwas ändere…“)*

☞ Daher: Laufzeit des Compilers für Desktop-Systeme wichtig(st)es Kriterium. (Übersetzung starten und Kaffee kochen wird oft als inakzeptabel empfunden)

☞ **Schnelle Laufzeiten und hohe Effizienz bei Implementierung des Compilers wichtig!**

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

- B. W. Kernighan, D. 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

- Werkzeuge zur Code-Generierung neben Compilern
- ANSI-C als gebräuchlichste Programmiersprache für ES
- Charakteristika Eingebetteter Prozessoren
- Code-Qualität als primäre Anforderung an Compiler für Eingebettete Systeme