Mikroprozessortechnik

Prof. Dr. Michael Lipp

fbeit
FACHBEREICH ELEKTROTECHNIK
UND INFORMATIONSTECHNIK

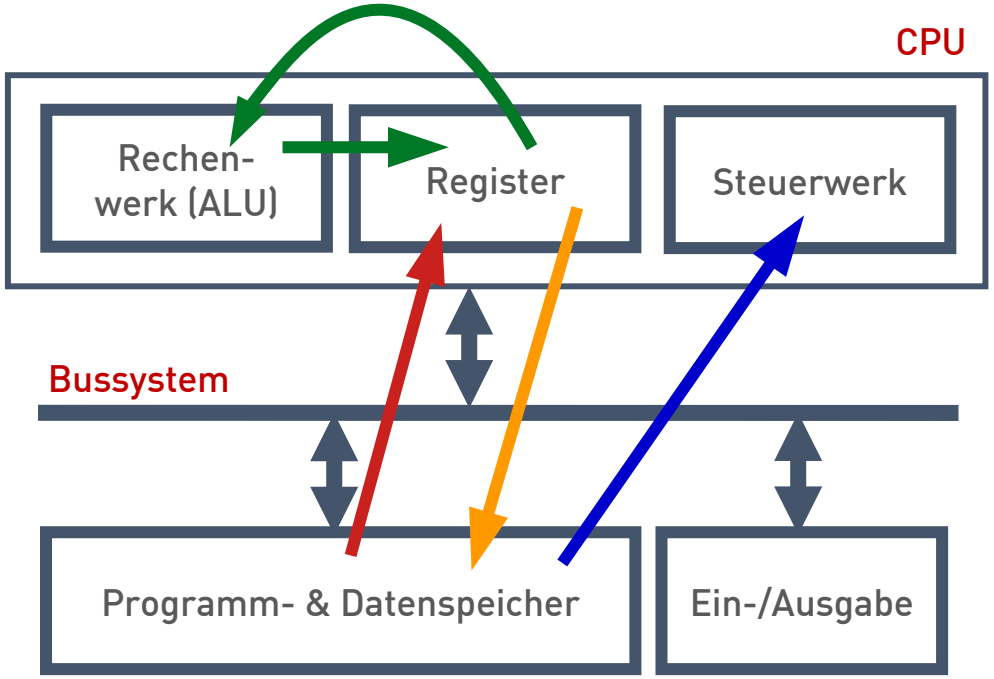
h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Code-Optimierung (Performance)

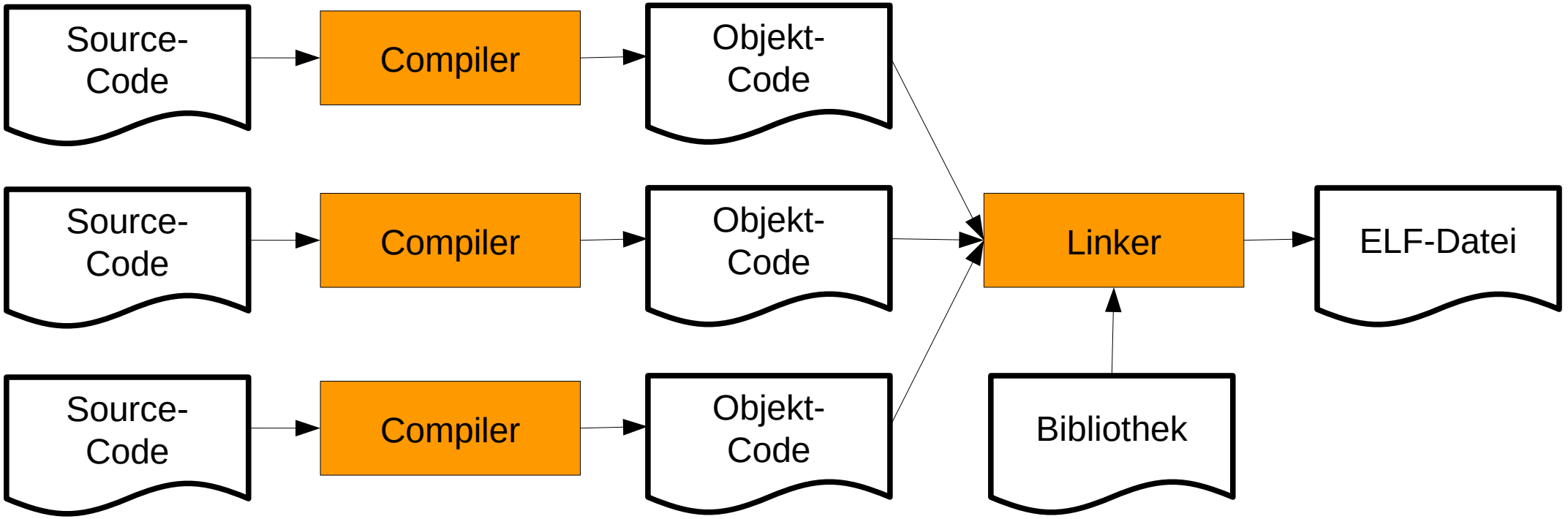
Wdh.: Arbeitsprinzip

Woher kommt das Programm?

- **Befehl aus Speicher holen**
- **Je nach Befehl...**
  - Daten aus Speicher oder von I/O-Register in CPU lesen
  - Daten verarbeiten
  - Daten von CPU in Speicher schreiben
- **Neuen Befehl holen usw.**

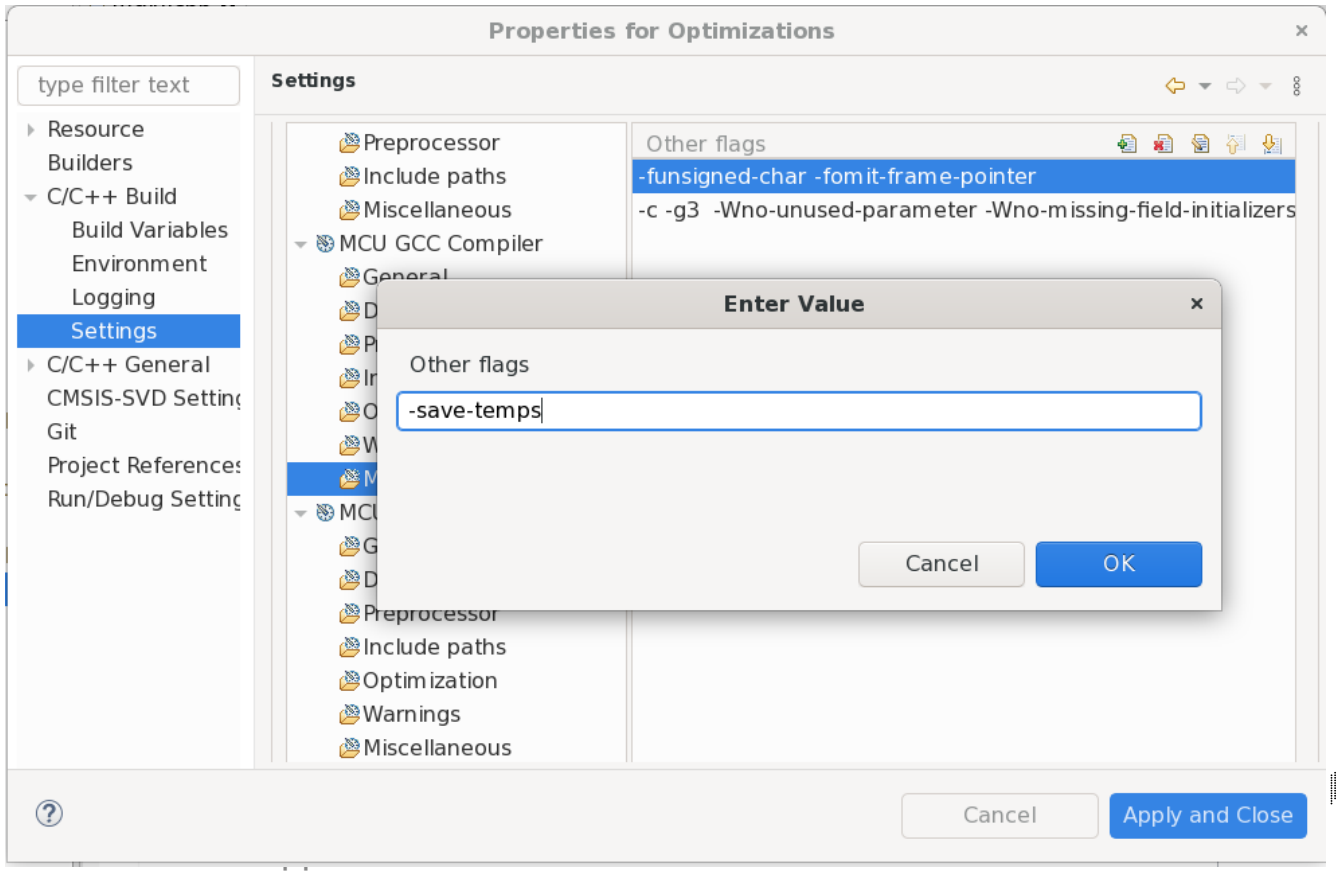


Funktionsweise Compiler/Linker



Funktionsweise Compiler/Linker

- **Compiler Internals**
  - Als Vorstufe zum Objekt-Code erzeugt der Compiler Assembler-Code
  - Kann „erhalten“ und analysiert werden



Funktionsweise Compiler/Linker

- **Optimierung des generierten Codes**
  - Ohne Optimierung wird sehr “umständlicher“ Code generiert
  - Beispiel:

```
10 uint32_t sum(uint32_t a, uint32_t b) {
11     uint32_t sum = 0;
12     sum = a + b;
13     return sum;
14 }
```

```
24 sum:
25 .LFB0:
26     .file 1 "../myCode/c-functions.c"
27     .loc 1 10 0
28     .cfi_startproc
29     @ args = 0, pretend = 0, frame = 16
30     @ frame_needed = 0, uses_anonymous_args = 0
31     @ link register save eliminated.
32     sub sp, sp, #16
33     .cfi_def_cfa_offset 16
34     str r0, [sp, #4]
35     str r1, [sp]
36     .loc 1 11 0
37     movs    r3, #0
38     str r3, [sp, #12]
39     .loc 1 12 0
40     ldr r2, [sp, #4]
41     ldr r3, [sp]
42     add r3, r3, r2
43     str r3, [sp, #12]
44     .loc 1 13 0
45     ldr r3, [sp, #12]
46     .loc 1 14 0
47     mov r0, r3
48     add sp, sp, #16
49     .cfi_def_cfa_offset 0
50     @ sp needed
51     bx  lr
```

Live Coding

Funktionsweise Compiler/Linker

(Wdh.) AAPCS:
Parameter in r0, r1
Ergebnis in r0

„.command“: “Pseudo-Instruktionen“,
d. h. Anweisungen an den Assembler,
hier nicht weiter betrachtet

```
24 sum:
25 .LFB0
26     .
27     .
28     .
29     @
30     @ frame_needed = 0, uses_anonymous_args = 0
31     @ link register save eliminated.
32     sub sp, sp, #16
33     .cfi_def_cfa_offset 16
34     str r0, [sp, #4]
35     str r1, [sp]
36     .loc 1 11 0
37     movs    r3, #0
38     str r3, [sp, #12]
39     .loc 1 12 0
40     ldr r2, [sp, #4]
41     ldr r3, [sp]
42     add r3, r3, r2
43     str r3, [sp, #12]
44     .loc 1 13 0
45     ldr r3, [sp, #12]
46     .loc 1 14 0
47     mov r0, r3
48     add sp, sp, #16
49     .cfi_def_cfa_offset 0
50     @ sp needed
51     bx  lr
```

Platz für 4 32-bit-Werte auf Stack reservieren

„a“ und „b“ in „lokale Variablen“ (d. h. auf den Stack) kopieren

r3 mit 0 laden und auf dem Stack speichern (lokale Varoiable „sum“)

Parameter „a“ und „b“ zur Verarbeitung in Register r2 und r3 laden

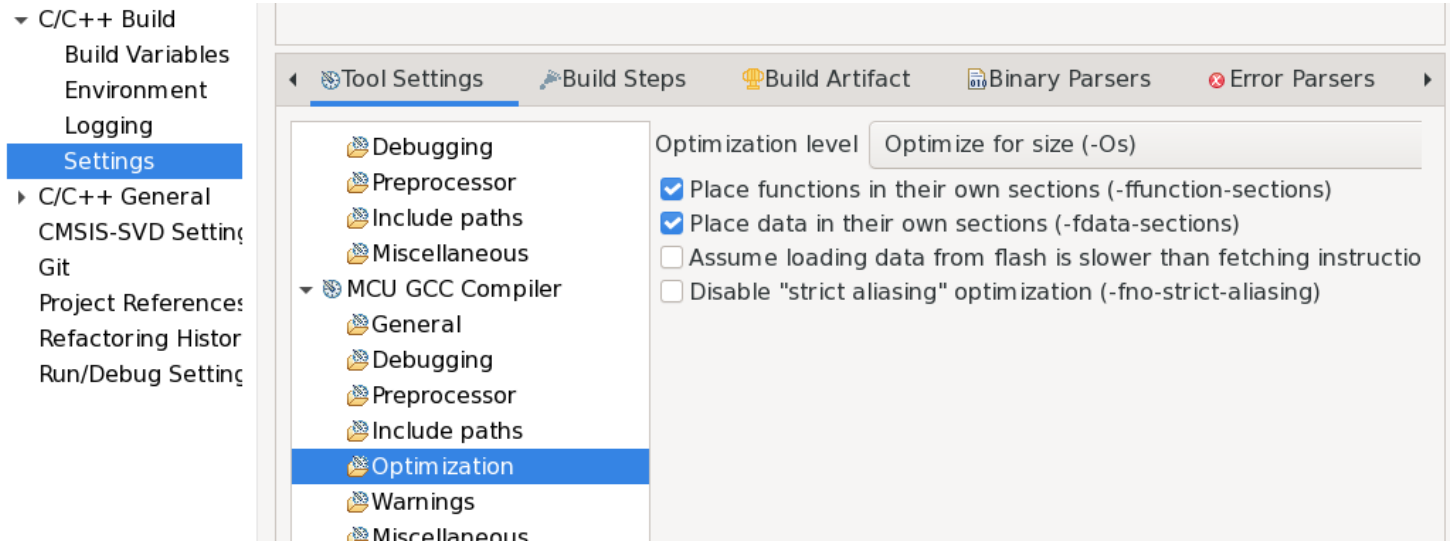
Summe bilden und in der lokalen Variablen „sum“ speichern

Wert von „sum“ nach r3 kopieren

R3 nach r0 kopieren (Rückgabewert)
Speicher auf Stack wieder „freigeben“

Funktionsweise Compiler/Linker

- **Optimierung des generierten Codes**
  - Mit „Optimize for Size“ sehr effizienter Code (Einstellen über Project/Properties)
  - Variablen evtl. „weg-optimiert“




```
24 sum:
25 .LFB0:
26     .file 1 "../myCode/c-functions.c"
27     .loc 1 10 0
28     .cfi_startproc
29     @ args = 0, pretend = 0, frame = 0
30     @ frame_needed = 0, uses_anonymous_args = 0
31     @ link register save eliminated.
32 .LVL0:
33     .loc 1 14 0
34     add r0, r0, r1
35 .LVL1:
36     bx  lr
```

Funktionsweise Compiler/Linker

- **Optimierung des generierten Codes**
  - Optimierung geht u. U. sehr weit
  - Beispiel „sinnlose Schleife“:

```
16 void emptyLoop() {
17     for (int i = 0; i < 1000; i++) {
18     }
19 }
```

```
48 emptyLoop:
49 .LFB1:
50     .loc 1 16 0
51     .cfi_startproc
52     @ args = 0, pretend = 0, frame = 0
53     @ frame_needed = 0, uses_anonymous_args = 0
54     @ link register save eliminated.
55 .LVL2:
56     .loc 1 19 0
57     bx  lr
```

Details zum ARM-Befehlssatz

ARM Befehlssatz

- **RISC Architektur**
  - ARM implementiert eine RISC-Architektur (Reduced Instruction Set Computer)
    - Relativ geringe Anzahl von Befehlen, die effizient (in einem CPU-Takt-Zyklus) ausgeführt werden können
    - Komplexe Operationen müssen aus einfachen Befehlen zusammengesetzt werden
  - Alternative: CISC (Complex Instruction Set Computer)
    - Befehlssatz enthält auch komplexe Instruktionen
    - Instruktionen können z. B. mehrere ALU-Operationen erfordern
    - Realisierung z. B. mit „Mikro-Code“

ARM Befehlssatz

- **Binäre Befehlskodierungen in ARM-Prozessoren**
  - ARM: 32-Bit Befehlssatz
    - Benötigt viel Speicherplatz
    - Sehr effizient, da ein Befehl sehr viele Optionen beinhaltet
  - Thumb: 16-Bit Befehlsatz
    - 30% bis 40% weniger Speicher
    - Weniger effizient, manchmal mehrere Befehle erforderlich um ARM-Befehl zu nachzubilden (ca. 30% Geschwindigkeitsverlust)
    - Wechsel ARM/Thumb bei manchen ARM-Prozessoren möglich
      - Modus an Bit 0 des PC erkennbar (wird beim Speicherzugriff ignoriert), 1 → Thumb
  - Thumb2: 32/16-Bit Befehlssatz (vergl. [Av7M-RM])
    - „Optimale Mischung“

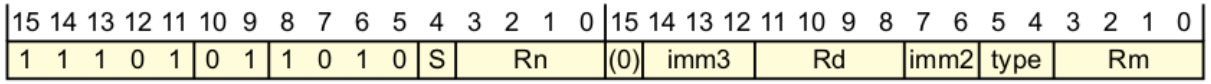
Thumb2 Befehlssatz (32-Bit Beispiel)

(aus [Av7M-RM])

A7.7.2 ADC (register)

Encoding T2 ARMv7-M

```
ADC{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}
```




```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
ADC{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

where:

S	If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.
<c><q>	See *Standard assembler syntax fields* on page A7-177.
<Rd>	Specifies the destination register. If <Rd> is omitted, this register is the same as <Rn>.
<Rn>	Specifies the register that contains the first operand.
<Rm>	Specifies the register that is optionally shifted and used as the second operand.
<shift>	Specifies the shift to apply to the value read from <Rm>. If <shift> is omitted, no shift is applied and both encodings are permitted. If <shift> is specified, only encoding T2 is permitted. The possible shifts and how they are encoded are described in *Shifts applied to a register* on page A7-182.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], shifted, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

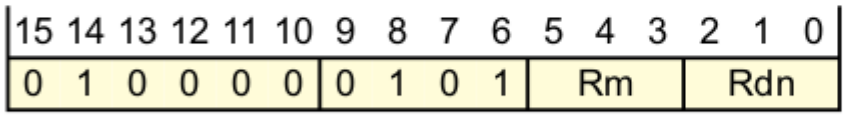
Thumb2 Befehlssatz (16-Bit Beispiel)

A7.7.2 ADC (register) (aus [Av7M-RM])

Encoding T1 All versions of the Thumb instruction set.

`ADCS <Rdn>,<Rm>` Outside IT block.

`ADC<c> <Rdn>,<Rm>` Inside IT block.



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	0	1	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

Kurzform wird vom Assembler i. allg. automatisch verwendet, wenn das gewünschte Verhalten damit erreicht werden kann

Thumb2 Befehlssatz

- **Befehlsvarianten (verfügbar für die meisten Befehle)**
  - Aktualisierung der Zustands-Bits (Z, N, C usw.) durch Anhängen von „S“ an den Befehls-Mnemonics
  - Bedingte Ausführung durch Anhängen eines Kürzels für eine Bedingung (Mnemonic Extension, s. folgende Folie)
    - Bedingung beim ARM-Befehlssatz im Befehl kodiert,
    - beim Thumb Befehlssatz mit Hilfsbefehl kodiert (IfThen(IT)-Block)
    - Performanter als Sprungbefehl

Thumb2 Befehlssatz

cond	Mnemonic extension	Meaning, integer arithmetic	Meaning, floating-point arithmetic[a]	Condition flags
0000	EQ	Equal	Equal	Z == 1
0001	NE	Not equal	Not equal, or unordered	Z == 0
0010	CS [b]	Carry set	Greater than, equal, or unordered	C == 1
0011	CC [c]	Carry clear	Less than	C == 0
0100	MI	Minus, negative	Less than	N == 1
0101	PL	Plus, positive or zero	Greater than, equal, or unordered	N == 0
0110	VS	Overflow	Unordered	V == 1
0111	VC	No overflow	Not unordered	V == 0
1000	HI	Unsigned higher	Greater than, or unordered	C == 1 and Z == 0
1001	LS	Unsigned lower or same	Less than or equal	C == 0 or Z == 1
1010	GE	Signed greater than or equal	Greater than or equal	N == V
1011	LT	Signed less than	Less than, or unordered	N != V
1100	GT	Signed greater than	Greater than	Z == 0 and N == V
1101	LE	Signed less than or equal	Less than, equal, or unordered	Z == 1 or N != V
1110	None (AL) [d]	Always (unconditional)	Always (unconditional)	Any

a. Unordered means at least one NaN operand.
b. HS (unsigned higher or same) is a synonym for CS.
c. LO (unsigned lower) is a synonym for CC.
d. AL is an optional mnemonic extension for always, except in IT instructions. See *IT* on page A7-236 for details.

Thumb2 Befehlssatz

A7.7.38 IT (aus [Av7M-RM])

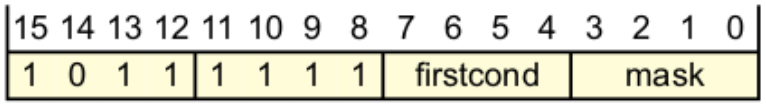
If Then makes up to four following instructions (the *IT block*) conditional. The conditions for the instructions in the IT block can be the same, or some of them can be the inverse of others.

IT does not affect the condition flags. Branches to any instruction in the IT block are not permitted, apart from those performed by exception returns.

16-bit instructions in the IT block, other than CMP, CMN, and TST, do not set the condition flags. The AL condition can be specified to get this changed behavior without conditional execution.

Encoding T1 ARMv7-M

IT{x{y{z}}} <firstcond> Not permitted in IT block




```
if mask == '0000' then SEE "Related encodings";
if firstcond == '1111' || (firstcond == '1110' && BitCount(mask) != 1) then UNPREDICTABLE;
if InITBlock() then UNPREDICTABLE;
```

Related encodings See *If-Then, and hints* on page A5-135.

Thumb2 Befehlssatz

Assembler syntax

`IT{x{y{z}}}<q>  <firstcond>`

where:

`<x>`	Specifies the condition for the second instruction in the IT block.
`<y>`	Specifies the condition for the third instruction in the IT block.
`<z>`	Specifies the condition for the fourth instruction in the IT block.
`<q>`	See *Standard assembler syntax fields* on page A7-177.
`<firstcond>`	Specifies the condition for the first instruction in the IT block.

Each of `<x>`, `<y>`, and `<z>` can be either:

`T`	Then. The condition attached to the instruction is `<firstcond>`.
`E`	Else. The condition attached to the instruction is the inverse of `<firstcond>`. The condition code is the same as `<firstcond>`, except that the least significant bit is inverted. `E` must not be specified if `<firstcond>` is `AL`.

Thumb2 Befehlssatz

- **IT Beispiel**

```
30  ldr r0, =5
31  subs r0, 7
32
33  // Absolutwert bilden
34  it mi              // Nächste Instruktion nur ausgeführt, wenn N==1
35  rsbmi r0, r0, 0    // Reverse subtract: r0 = 0 - r0
```

Thumb2 Befehlssatz

- **Befehle zum Laden und Speichern**
  - LDR/STR
    - Variante: LDRH/STRH (Laden/Speichern Halbwort [16 Bit])
    - Variante: LDRB/STRB (Laden/Speichern Byte)
  - Adressierung
    - Immediate: Adresse = Basis-Register-Wert + Offset
      - C-Analogon: Basisregister enthält Startadresse eines Arrays, Index konstant
    - Register: Adresse = Basis-Register-Wert + Offset-Register-Wert
      - C-Analogon: Basisregister enthält Startadresse eines Arrays, Variable als Index
    - Scaled Register: Wie „Register“ aber addierter Register-Wert wird vorher bitweise geschoben
      - C-Analogon: Pointer-Arithmetik

Thumb2 Befehlssatz

- **Befehle zum Laden und Speichern (Fortsetzung)**
  - Optionale Basis-Register Änderung
    - Pre-Indexed: Offset wird vor Zugriff zum Basisregister addiert (oder vom Basisregister subtrahiert)
    - Post-Indexed: Offset wird nach Zugriff zum Basisregister addiert (oder vom Basisregister subtrahiert)
  - C-Analogon: Pre-/Post-Inkrement

Thumb2 Befehlssatz

- **Beispiel**

```
30 void clearValues(int32_t* values, uint16_t nbOfValues) {
31     while (nbOfValues-- > 0) {
32         *values++ = 0;
33     }
34 }
```

```
120 clearValues:
121 .LFB3:
122     .loc 1 30 0
123     .cfi_startproc
124     @ args = 0, pretend = 0, frame = 0
125     @ frame_needed = 0, uses_anonymous_args = 0
126     @ link register save eliminated.
127 .LVL9:
128     add r1, r0, r1, lsl #2
129 .LVL10:
130     .loc 1 32 0
131     movs    r3, #0
132 .LVL11:
133 .L8:
134     .loc 1 31 0
135     cmp r0, r1
136     bne .L9
137     .loc 1 34 0
138     bx  lr
139 .L9:
140     .loc 1 32 0
141     str r3, [r0], #4
142 .LVL12:
143     b   .L8
```

- Beachten Sie die Optimierung (keine Laufvariable!)

Thumb2 Befehlssatz

- **Arithmetisch/Logische Befehle**
  - ADC – Add with Carry (zwei Register oder Register und Wert)
  - ADD – Add (zwei Register oder Register und Wert)
  - ADR – Addiert Wert zu PC und speichert Ergebnis in Register
  - ADC – Bitweises UND (zwei Register oder Register und Wert)
  - ASR – Arithmetic Shift Right (zwei Register oder Register und Wert)
  - ...

Thumb2 Befehlssatz

- **„Besondere“ Verarbeitungsbefehle**
  - BFC – Bit Field Clear (löscht eine Folge zusammenhängender Bits)
  - BFI – Bit Field Insert (kopiert eine Folge zusammenhängender Bits)
  - BIC – Bit Clear (bitweise Und-Verknüpfung von Register und Wert)
  - CLZ – Count Leading Zeros (führende Nullen zählen)
- **Potential für Lösungen „besser als der Compiler“ (?)**
  - CLZ als „intrinsic“ verfügbar (kein Assembler-Code notwendig)
    - Wird genutzt wie Funktionsaufruf, Compiler generiert aber Assembler-Instruktion

```
unsigned char __CLZ(unsigned int val)
```

Thumb2 Befehlssatz

- **Potential für Lösungen „besser als der Compiler“ (?) (Fs.)**
  - BIC für Löschen von Bits
    - Benutzt der Compiler!

```
47 uint32_t testBfc(uint32_t val) {
48     return val & ~0x1f0;
49 }
```

```
240 testBfc:
241 .LFB139:
242     .loc 1 47 0
243     .cfi_startproc
244     @ args = 0, pretend = 0, frame = 0
245     @ frame_needed = 0, uses_anonymous_args = 0
246     @ link register save eliminated.
247 .LVL21:
248     .loc 1 49 0
249     bic r0, r0, #496
250 .LVL22:
251     bx  lr
```

Thumb2 Befehlssatz

- **Sprung-Befehle**
  - B – Branch (Angehängte Bedingungen nutzen!)
  - BL – Branch with Link
  - BLX – Branch with Link and Exchange (BL mit Wechsel Befehlssatz)
  - BX – Branch and Exchange (Sprung mit Wechsel Befehlssatz)

Thumb2 Befehlssatz

- **Konstante in Register laden**
  - Kein Befehl um einen beliebigen 32 Bit Wert in ein Register zu laden
    - Lässt sich nicht mit einem 32 Bit Befehlswort realisieren
  - Befehl MOV kann 16 Bit Werte laden
  - Tatsächlich kommt das Laden großer Ganzzahlwerte in Programmen sehr selten vor
  - Der „Pseudo-Befehl“ LDR mit „=Wert“ als Argument wird vom Assembler ja nach Wert übersetzt in
    - einen MOV-Befehl oder
    - in das Erzeugen einer Konstanten mit PC-relativem Laden

Thumb2 Befehlssatz

```
LDR r0,=0x23456789
```

Disassembler:

```
08000248:   ldr     r0, [pc, #4]    ; (0x8000250)
0800024a:   bx      lr
63          }
0800024c:   movs    r0, #0
0800024e:   bx      lr
08000250:   str     r1, [r1, #120]  ; 0x78
```

Bei der PC-relativen Adressierung ist der Basiswert PC+4 (ein Sonderfall, sonst ist es immer der Wert im Registers)

Disassembler zeigt bei 0x8000250 einen Befehl an, weil er nicht „weiß“, dass dort eine Konstante steht. Im Memory-View in Eclipse sieht man:

```
0x08000250  23456789
```

Inline Assembler

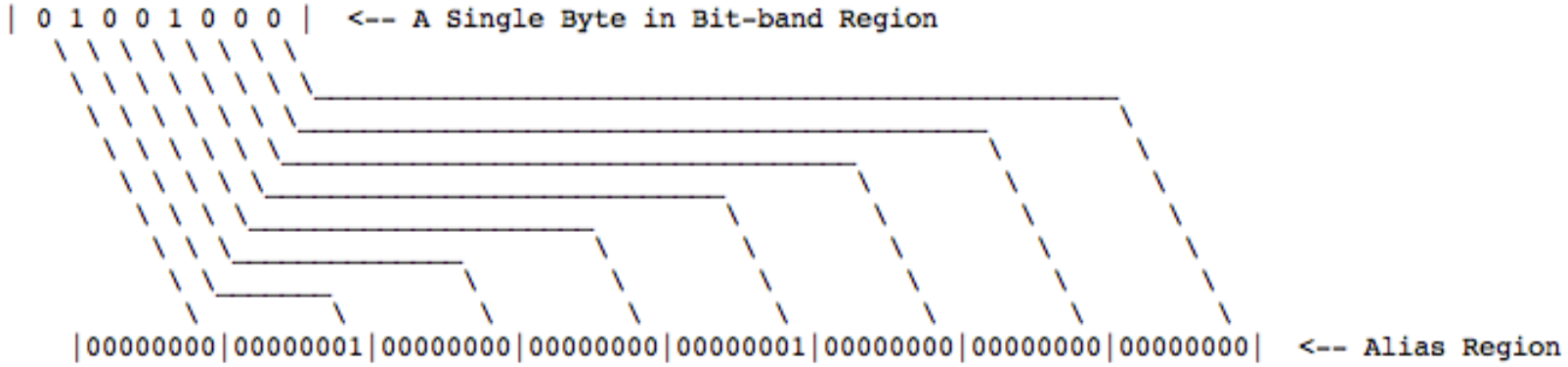
- **Generell ist eigene Assembler-Programmierung bei modernen, optimierenden Compilern (und „gutem“ C/C++-Code, s. nachfolgender Abschnitt) selten sinnvoll**
- **Falls doch erforderlich, Nutzung möglich mit**
  - Extra Quelltext (bereits behandelt)
  - Inline Assembler (für wenige Anweisungen)

```
51 int32_t addASM(int32_t a, int32_t b) {
52     __asm__ ("add r0, r1\n\t"
53              "bx lr");
54     // Dummy, verhindert Compiler Warnung.
55     return 0;
56 }
```

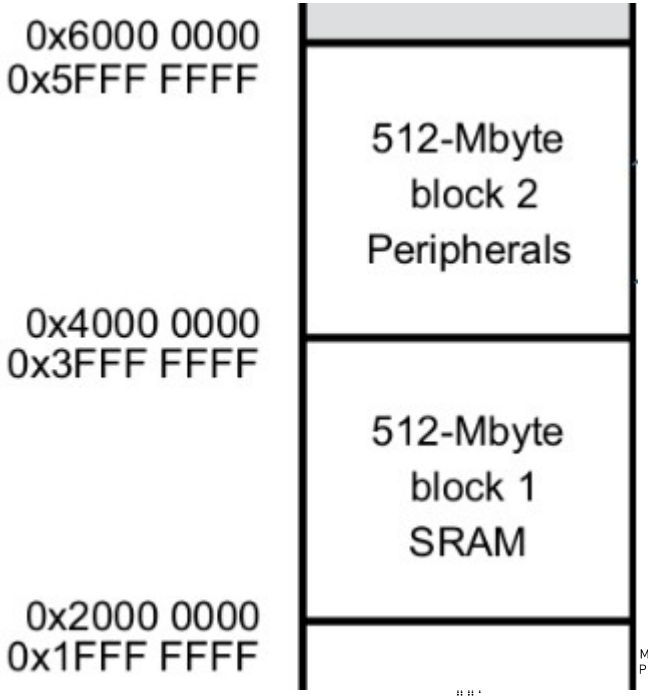
Optimierung mit Bitbanding

Bitbanding

- **Setzen/Löschen eines Bits im Speicher erfordert einen Read-Modify-Write-Zyklus**
- **Beim Bitbanding wird jedem Bit im Speicher ein Wort im Adressraum (alias region) zugeordnet**
  - Verfügbar für den SRAM- und den Peripherals-Block (vergl. Vorlesung 4)



Quelle: https://spin.atomicobject.com/2013/02/08/bit-banding/



Bitbanding

- **Berechnung des zugeordneten Worts**

  bit_word_addr = *bit_band_base* + (*byte_offset* x 32) + (*bit_number* × 4)

  where:

  – *bit_word_addr* is the address of the word in the alias memory region that maps to the targeted bit

  – *bit_band_base* is the starting address of the alias region

  – *byte_offset* is the number of the byte in the bit-band region that contains the targeted bit

  – *bit_number* is the bit position (0-7) of the targeted bit

- **Bit-Manipulation**
  - Schreiben einer „1“ in das Wort der „alias region“ setzt das Bit,
  - schreiben einer „0“ löscht das Bit
  - Lesen ist auch möglich

Bitbanding

- **Keine vordefinierten Makros verfügbar**

```
#define SRAM_BB_ALIAS(address, bit)       ((int32_t*)(SRAM_BB_BASE) \
    + (((char*)(address) - (char*)(SRAM_BASE)) << 3) + (bit))

#define PERIPH_BB_ALIAS(address, bit) ((int32_t*)(PERIPH_BB_BASE) \
    + (((char*)(address) - (char*)(PERIPH_BASE)) << 3) + (bit))
```

- **Beispiel:**

```
47      volatile uint8_t value = 0;
48      *SRAM_BB_ALIAS(&value, 2) = 1;
49      leds = value;
```

C/C++-Wissen

- **Schlüsselwort `volatile`**
  - Generell geht der Compiler davon aus, dass sich Werte im Speicher nur ändern, wenn sie vom erzeugten Code geschrieben werden
  - Wird ein Wert aus dem Speicher in ein Register geladen, muss er bei mehrmaliger Verwendung nicht nochmal geladen werden
  - Annahme stimmt hier nicht: Der Wert von `value` ändert sich (durch den Zugriff auf `value` über bitbanding) obwohl der Variablen `value` im Code kein neuer Wert zugewiesen wird
  - Wird eine Variable als `volatile` deklariert, muss der Compiler bei jedem Zugriff den Wert neu aus dem Speicher laden

Bitbanding

- **Peripherie-spezifische Alternativen**
  - Bei den GPIO-Registern können einzelne Bits im ODR über das „Bit Set Reset Register“ gesetzt oder zurückgesetzt werden (vergl. Vorlesung 2)
  - Noch effizienter als Bitbanding

Optimierungen im C/C++-Quell-Code

Performanter C/C++-Quellcode

- **Generell ist die Optimierung im Compiler gut**
  - Die eingesetzten Algorithmen sind einzeln oder gruppenweise einschaltbar
  - Unterschiedliche Compiler können unterschiedlich gute Ergebnisse liefern
- **Beste Ergebnisse bekommt man, wenn man die Optimierung unterstützt**

Performanter C/C++-Quellcode

- **Beispiel aus „ARM Befehlssatz“**

```
30 void clearValues(int32_t* values, uint16_t nbOfValues) {
31     while (nbOfValues-- > 0) {
32         *values++ = 0;
33     }
34 }
```

```
120 clearValues:
121 .LFB3:
122     .loc 1 30 0
123     .cfi_startproc
124     @ args = 0, pretend = 0, frame = 0
125     @ frame_needed = 0, uses_anonymous_args = 0
126     @ link register save eliminated.
127 .LVL9:
128     add r1, r0, r1, lsl #2
129 .LVL10:
130     .loc 1 32 0
131     movs    r3, #0
132 .LVL11:
133 .L8:
134     .loc 1 31 0
135     cmp r0, r1
136     bne .L9
137     .loc 1 34 0
138     bx  lr
139 .L9:
140     .loc 1 32 0
141     str r3, [r0], #4
142 .LVL12:
143     b   .L8
```

- **Schlechteres Ergebnis für:**

```
36 void clearValuesIdx(int32_t* values, uint16_t nbOfValues) {
37     for (uint16_t i = 0; i < nbOfValues; i++) {
38         values[i] = 0;
39     }
40 }
```

  - Berechnung der Position im Array (`values[i]`) wird vom Optimierer nicht in Nutzung von Pointer „umgewandelt“

Performanter C/C++-Quellcode

- **„Kurze“ Funktionen (Methoden) können im Header-File „`inline`“ definiert werden**
  - Compiler generiert bei Nutzung der Funktion keinen Funktionsaufruf („bl“) sondern den kompletten Assembler-Code für die Funktionsimplementierung
  - Sinnvoll, wenn Implementierung kürzer als Funktionsaufruf (inklusive Übergabe Parameter und Rückgabewert)
  - Typisches Beispiel: „getter“ und „setter“ (ohne Prüfung Zusicherung)
    - Für „inline“ Funktion (Methode) wird typischer Weise ein einziger LDR/STR-Befehl erzeugt

```
11 class InlineDemo {
12 private:
13     int normalAttr;
14     int inlineAttr;
15
16 public:
17     int getNormalAttr();
18
19     int getInlineAttr() {
20         return inlineAttr;
21     }
22 };
```