Vorlesungsskript zu „Vertiefung Programmieren“ Code Generierung

Dozent: Dipl.-Inf. (FH) Andreas Schmidt

Compiler Toolchain

Präprozessor, Compiler, Assembler und Linker

Code Generierung

GNU Compiler Toolchain

Assembler Code Generierung durch den Compiler

```
$ arm-none-eabi-gcc -S <Input C-File> -o <Output Asm File>
```

Der Compiler selbst führt noch keine Maschinencode Generierung im eigentlichen Sinne durch. Stattdessen erzeugt er, nach diversen Zwischen- und Optimierungsschritten, Assembler-Code, welcher spezifisch für einen bestimmten Prozessor ist.

Der generierte Assembler Code beinhaltet u.a. folgende Informationen:

- Speicherreservierungen für Variablen in der .data oder .bss Section
- Assembler Befehle
- Sog. Sprung-Labels als interne Sprungziele
- Assembler-Direktiven zur Steuerung des späteren Maschinen-Code-Generierungsprozesses

Die Code-Generierung folgt hierbei dem sog. „Application Binary Interface“ (ABI) Standard für ARM

ABI Übersicht
https://developer.arm.com/Architectures/Application%20Binary%20Interface
Procedure Call Standard
https://github.com/ARM-software/abi-aa/blob/2982a9f3b512a5bfdc9e3fea5d3b298f9165c36b/aapcs32/aapcs32.rst

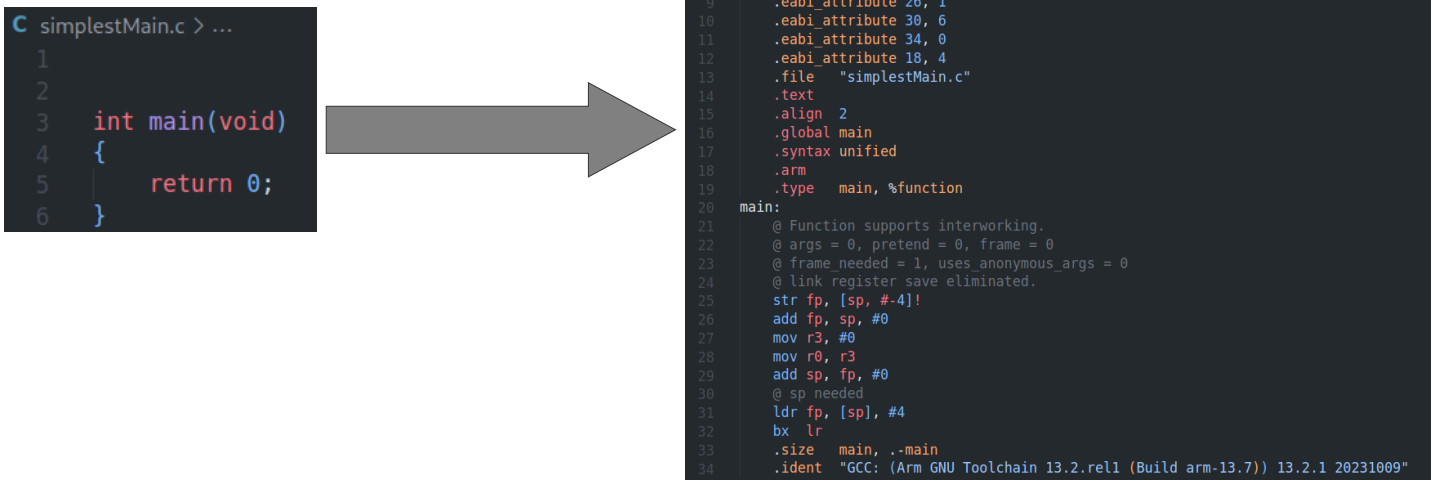
Quelle: intern

GNU Compiler Toolchain

Assembler Code Generierung durch den Compiler

```
$ arm-none-eabi-gcc -O0 -S simpleMain.c -o simpleMain.s
```

Der Compiler wird in diesem Beispiel bewusst mit deaktivierter Optimierung (Argument -O0) ausgeführt. Das stellt sicher, dass bei den einfachen Beispielen auch der gesamte Code zum Nachvollziehen generiert wird.



Quelle: intern

GNU Compiler Toolchain

Assembler Code Generierung durch den Compiler

simplesMain.s

```
    .cpu arm7tdmi
    .arch armv4t
    .fpu softvfp
    .eabi_attribute 20, 1
    .eabi_attribute 21, 1
    .eabi_attribute 23, 3
    .eabi_attribute 24, 1
    .eabi_attribute 25, 1
    .eabi_attribute 26, 1
    .eabi_attribute 30, 6
    .eabi_attribute 34, 0
    .eabi_attribute 18, 4
    .file   "simplestMain.c"
    .text
    .align  2
    .global main
    .syntax unified
    .arm
    .type   main, %function
main:
    @ Function supports interworking.
    @ args = 0, pretend = 0, frame = 0
    @ frame_needed = 1, uses_anonymous_args = 0
    @ link register save eliminated.
    str fp, [sp, #-4]!
    add fp, sp, #0
    mov r3, #0
    mov r0, r3
    add sp, fp, #0
    @ sp needed
    ldr fp, [sp], #4
    bx  lr
    .size   main, .-main
    .ident  "GCC: (Arm GNU Toolchain 13.2.rel1 (Build arm-13.7)) 13.2.1 20231009"
```

Alle Zeilen, welche mit einem Punkt (.) beginnen, sind sog. Assembler-Direktiven und dienen der Steuerung der späteren Machinen-Code-Generierung bzw. stellen Zusatzinformationen für spätere Verarbeitungsschritte (z.B. Linker) bereit

Hinweis zur CPU: Die Assembler Direktiven `.cpu`, `.arch` und `.fpu` legen den Zielprozessor bzw. dessen Derivat fest, für den dieser Assembler Code generiert wurde und für den auch später der Maschinencode erzeugt werden soll.

Dies kann z.B. mit dem Argument -mcpu=cortex-m4 geändert werden (Ziel CPU ist dann ein Prozessor aus der Cortex-M4 Familie) bzw. durch -march=armv7e-m

Quelle: intern

GNU Compiler Toolchain

Assembler Code Generierung durch den Compiler

```
$ arm-none-eabi-gcc -O0 -mcpu=cortex-m4 -S simplestMain.c -o simplesMain.s
```

simplesMain.s

```
1       .cpu cortex-m4
2       .arch armv7e-m
3       .fpu softvfp
4       .eabi_attribute 20, 1
5       .eabi_attribute 21, 1
6       .eabi_attribute 23, 3
7       .eabi_attribute 24, 1
8       .eabi_attribute 25, 1
9       .eabi_attribute 26, 1
10      .eabi_attribute 30, 6
11      .eabi_attribute 34, 1
12      .eabi_attribute 18, 4
13      .file   "simplestMain.c"
14      .text
15      .align  1
16      .global main
17      .syntax unified
18      .thumb
19      .thumb_func
20      .type   main, %function
21  main:
22      @ args = 0, pretend = 0, frame = 0
23      @ frame_needed = 1, uses_anonymous_args = 0
24      @ link register save eliminated.
25      push    {r7}
26      add r7, sp, #0
27      movs    r3, #0
28      mov r0, r3
29      mov sp, r7
30      @ sp needed
31      pop {r7}
32      bx  lr
33      .size   main, .-main
34      .ident  "GCC: (Arm GNU Toolchain 13.2.rel1 (Build arm-13.7)) 13.2.1 20231009"
```

Unterschied zwischen diesem und dem vorherigem Beispiel sind nicht nur die Direktiven, sondern auch der generierte Code

➔ Das liegt u.a. daran, dass der Cortex-M4 hier in diesem Fall den sog. Thumb Mode verwendet und nicht den ARM Mode

```
16      .global main
17      .syntax unified
18      .thumb
19      .thumb_func
20      .type   main, %function
```

```
16      .global main
17      .syntax unified
18      .arm
19      .type   main, %function
```

Kurz-Info zu Thumb/ARM
https://www.embedded.com/introduction-to-arm-thumb/

Quelle: intern

GNU Compiler Toolchain

Assembler Code Generierung durch den Compiler

```
$ arm-none-eabi-gcc -O0 -mcpu=cortex-m4 -S simplestMain2.c -o simplesMain2.s
```

simplestMain2.c

```c
int globalVar;

int main(void)
{
    globalVar = 10;
    return globalVar;
}
```

In dem minimalen C-Programm wird lediglich eine nicht initialisierte, globale Variable angelegt und ihr ein Wert zugewiesen.

simplesMain2.s

```
    .cpu cortex-m4
    .arch armv7e-m
    .fpu softvfp
    .eabi_attribute 20, 1
    .eabi_attribute 21, 1
    .eabi_attribute 23, 3
    .eabi_attribute 24, 1
    .eabi_attribute 25, 1
    .eabi_attribute 26, 1
    .eabi_attribute 30, 6
    .eabi_attribute 34, 1
    .eabi_attribute 18, 4
    .file   "simplestMain2.c"
    .text
    .global globalVar
    .bss
    .align  2
    .type   globalVar, %object
    .size   globalVar, 4
globalVar:
    .space  4
    .text
    .align  1
    .global main
    .syntax unified
    .thumb
    .thumb_func
    .type   main, %function
main:
    @ args = 0, pretend = 0, frame = 0
    @ frame_needed = 1, uses_anonymous_args = 0
    @ link register save eliminated.
    push    {r7}
    add r7, sp, #0
    ldr r3, .L3
    movs    r2, #10
    str r2, [r3]
    ldr r3, .L3
    ldr r3, [r3]
    mov r0, r3
    mov sp, r7
    @ sp needed
    pop {r7}
    bx  lr
.L4:
    .align  2
.L3:
    .word   globalVar
    .size   main, .-main
    .ident  "GCC: (Arm GNU Toolchain 13.2.rel1 (Build arm-13.7)) 13.2.1 20231009"
```

In dem generierten Code ist zu erkennen, dass das Anlegen der globalen Variable vollständig über die Direktiven gesteuert wird. Dies beinhaltet u.a.

- Name (Label + .global)
- Speicher Segment (.bss)
- Größe der Variable (.size, .space)

Quelle: intern

GNU Compiler Toolchain

GNU ARM Assembler Direktiven

`.align expression [, expression]`	This is the generic .align directive. For the ARM however if the first argument is zero (ie no alignment is needed) the assembler will behave as if the argument had been 2 (ie pad to the next four byte boundary). This is for compatibility with ARM's own assembler.
`.arch name`	Select the target architecture. Valid values for name are the same as for the -march command-line option without the instruction set extension. Specifying .arch clears any previously selected architecture extensions.
`.arm`	This performs the same action as .code 32.
`.code [16|32]`	This directive selects the instruction set being generated. The value 16 selects Thumb, with the value 32 selecting ARM.
`.cpu name`	Select the target processor. Valid values for name are the same as for the -mcpu command-line option without the instruction set extension. Specifying .cpu clears any previously selected architecture extensions.
`.fpu name`	Select the floating-point unit to assemble for. Valid values for name are the same as for the -mfpu command-line option.
`.thumb`	This performs the same action as .code 16.
`.thumb_func`	This directive specifies that the following symbol is the name of a Thumb encoded function. This information is necessary in order to allow the assembler and linker to generate correct code for interworking between Arm and Thumb instructions and should be used even if interworking is not going to be performed. The presence of this directive also implies .thumb This directive is not necessary when generating EABI objects. On these targets the encoding is implicit when generating Thumb code.

Quelle: https://sourceware.org/binutils/docs/as/ARM-Directives.html

GNU Compiler Toolchain

GNU Assembler Direktiven

`.bss`	.bss tells as to assemble the following statements onto the end of the bss section. For most ELF based targets an optional subsection expression (which must evaluate to a positive integer) can be provided. In this case the statements are appended to the end of the indicated bss subsection.
`.data`	.data tells as to assemble the following statements onto the end of the data subsection numbered subsection (which is an absolute expression). If subsection is omitted, it defaults to zero.
`.global symbol`	global makes the symbol visible to ld. If you define symbol in your partial program, its value is made available to other partial programs that are linked with it. Otherwise, symbol takes its attributes from a symbol of the same name from another file linked into the same program.
`.section name`	Use the .section directive to assemble the following code into a section named name. This directive is only supported for targets that actually support arbitrarily named sections; on a.out targets, for example, it is not accepted, even with a standard a.out section name.
`.size name , expression`	This directive sets the size associated with a symbol name. The size in bytes is computed from expression which can make use of label arithmetic. This directive is typically used to set the size of function symbols.
`.space size [,fill]`	This directive emits size bytes, each of value fill. Both size and fill are absolute expressions. If the comma and fill are omitted, fill is assumed to be zero. This is the same as '.skip'.
`.text`	Tells as to assemble the following statements onto the end of the text subsection numbered subsection, which is an absolute expression. If subsection is omitted, subsection number zero is used.
`.type name , type description`	This sets the type of symbol name to be either a function symbol or an object symbol. There are five different syntaxes supported for the type description field, in order to provide compatibility with various other assemblers.

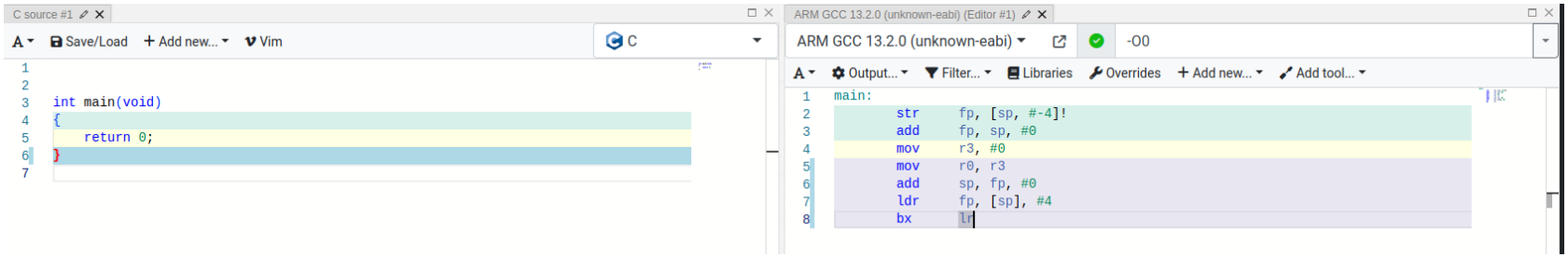
Quelle: https://sourceware.org/binutils/docs/as/

GNU Compiler Toolchain

Assembler Code Generierung durch den Compiler

Für das Kennenlernen und Untersuchen der Code-Generierung bzw. der Auswirkung bestimmte C-Code Konstrukte auf die Code-Generierung, ist die Verwendung des Kommandozeilen-Assembler meist recht mühselig.

Das Online Tool https://godbolt.org/ bietet eine einfache Web-Oberfläche mit Hilfe der C-Code Übersetzt werden kann. Dabei können unterschiedliche Compiler + Versionen eingestellt werden.



Quelle: intern

DHBW
Duale Hochschule
Baden-Württemberg

Compiler Toolchain

Präprozessor, Compiler, Assembler und Linker

Code Generierung – Realisierung von Funktionsaufrufen

Funktionsaufrufe

Branch Befehle des ARM Prozessor

- Der ARM Prozessor realisiert Funktionsaufrufe mit Hilfe sog. „Branch“ Befehlen
  - Branch Befehle sind vergleichbar mit Sprungbefehlen
  - Hierbei wird der Program Counter auf die Adresse gesetzt, an der die gewünschte Funktion im Speicher liegt
  - Im Gegensatz zu einfachen Sprungbefehlen, werden bei einem Branch Befehl für Funktionsaufrufe noch zusätzliche Aktionen durchgeführt
    - z.B. beim „Branch & Link“ (bl Instruktion) wird die Adresse nach dem aktuellen Branch Befehl im sog. Link Register gespeichert
    - Dieses Link Register enthält somit die Rücksprungadresse an die nach der Abarbeitung der aktuellen Funktion zurückgekehrt werden soll

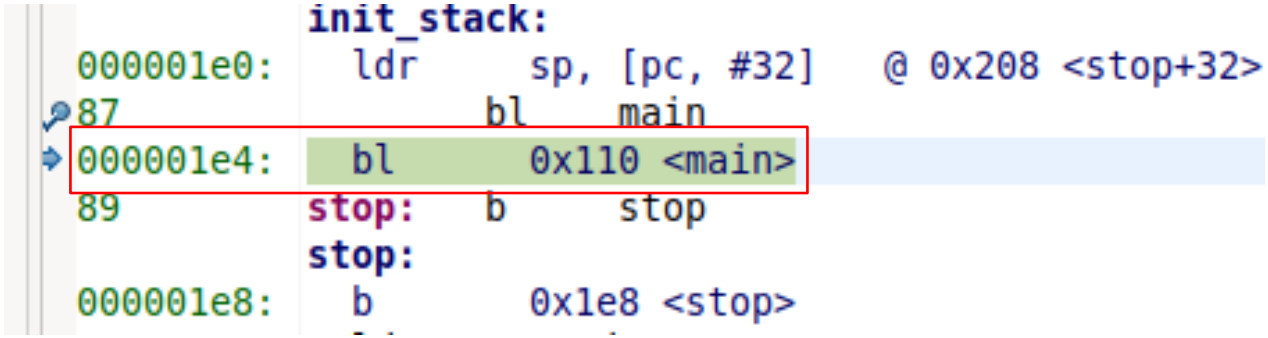
Mnemonic	Brief description
B	Branch
BL	Branch with Link
BLX	Branch indirect with Link
BX	Branch indirect
CBNZ	Compare and Branch if Non Zero
CBZ	Compare and Branch if Non Zero
IT	If-Then
TBB	Table Branch Byte
TBH	Table Branch Halfword

Table 34. Branch ranges

Instruction	Branch range
B label	–16 MB to +16 MB
B*cond* label (outside IT block)	–1 MB to +1 MB
B*cond* label (inside IT block)	–16 MB to +16 MB
BL{*cond*} label	–16 MB to +16 MB
BX{*cond*} Rm	Any value in register
BLX{*cond*} Rm	Any value in register

Funktionsaufrufe

Branch Befehle des ARM Prozessor



- Im Program Counter Register (PC) steht aktuell 0x1e4 – dabei handelt es sich um die Adresse, an der der als nächster auszuführender Befehl steht. In diesem Beispiel `bl 0x110 <main>`
- Wird nun der Branch & Link Befehl ausgeführt, so wird im Link Register (LR) der Wert PC + 4 gespeichert (im Beispiel also 0x1e4 + 4 = 0x1e8) und anschließend wird der PC auf 0x110 (Adresse der Main Funktion) gesetzt
- Somit wird die Programmausführung an der Adresse 0x110 fortgeführt

Funktionsaufrufe

Branch Befehle des ARM Prozessor

```
00000090: main:
00000094:   push    {r11, lr}
00000098:   add     r11, sp, #4
0000009c:   sub     sp, sp, #8
000000a0:     int mainResult = simpleFunctionCallTree(0xAB, 0xCD);
000000a4:   mov     r1, #205        @ 0xcd
000000a8:   mov     r0, #171        @ 0xab
25          bl      0xbc <simpleFunctionCallTree>
000000ac:   str     r0, [r11, #-8]
000000b0:     return 0;
000000b4:   mov     r3, #0
000000b8: }
28          mov     r0, r3
            sub     sp, r11, #4
000000bc:   pop     {r11, lr}
000000c0:   bx      lr
```

- Um aus einer Funktion „zurückzukehren“, wird z.B. die Instruktion bx verwendet. Dies ist ein sog. Indirekter Sprung, bei dem das Sprungziel in einem Register steht
- Im obigen Beispiel wir das Link Register (LR) als Rücksprungziel verwendet
  - Dort wurde vor dem Sprung in die Main Funktion die Rücksprungadresse 0x1e8 gespeichert
  - Somit springt die Ausführung mit bx lr an die Adresse 0x1e8 (der PC wird auf diese Adresse gesetzt)

GNU Compiler Toolchain

Funktionsaufruf und Parameterübergabe

```
int main()
{
    int result = sumFunction(10, 20);

    return result;
}
```

```
mov     r1, #20
mov     r0, #10
bl      sumFunction(int, int)
str     r0, [fp, #-8]
```

Die Parameter der Funktion werden, bis zu einer bestimmten Anzahl, über im ABI definierte Register übergeben

Der Funktionsaufruf erfolgt anschließend mit dem `bl` Befehl

Quelle: intern

GNU Compiler Toolchain

Funktionsaufruf und Parameterübergabe

Core registers and AAPCS usage

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8	FP	Frame Pointer or Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable-register 4.
r6	v3		Variable-register 3.
r5	v2		Variable-register 2.
r4	v1		Variable-register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

Quelle: https://github.com/ARM-software/abi-aa/blob/2982a9f3b512a5bfdc9e3fea5d3b298f9165c36b/aapcs32/aapcs32.rst

GNU Compiler Toolchain

Funktionsaufruf und Parameterübergabe

```
int main()
{
    int result = bigSumFunction(10, 20, 30, 40, 50, 60);

    return result;
}
```

```
mov     r3, #60
str     r3, [sp, #4]
mov     r3, #50
str     r3, [sp]
mov     r3, #40
mov     r2, #30
mov     r1, #20
mov     r0, #10
bl      bigSumFunction(int, int, int, int, int, int)
str     r0, [fp, #-8]
```

Die ersten 4 Parameter der Funktion werden über im ABI definierte Register übergeben, die restlichen Parameter über den Stack

Der Funktionsaufruf erfolgt anschließend mit dem `bl` Befehl

Quelle: intern

DHBW
Duale Hochschule
Baden-Württemberg

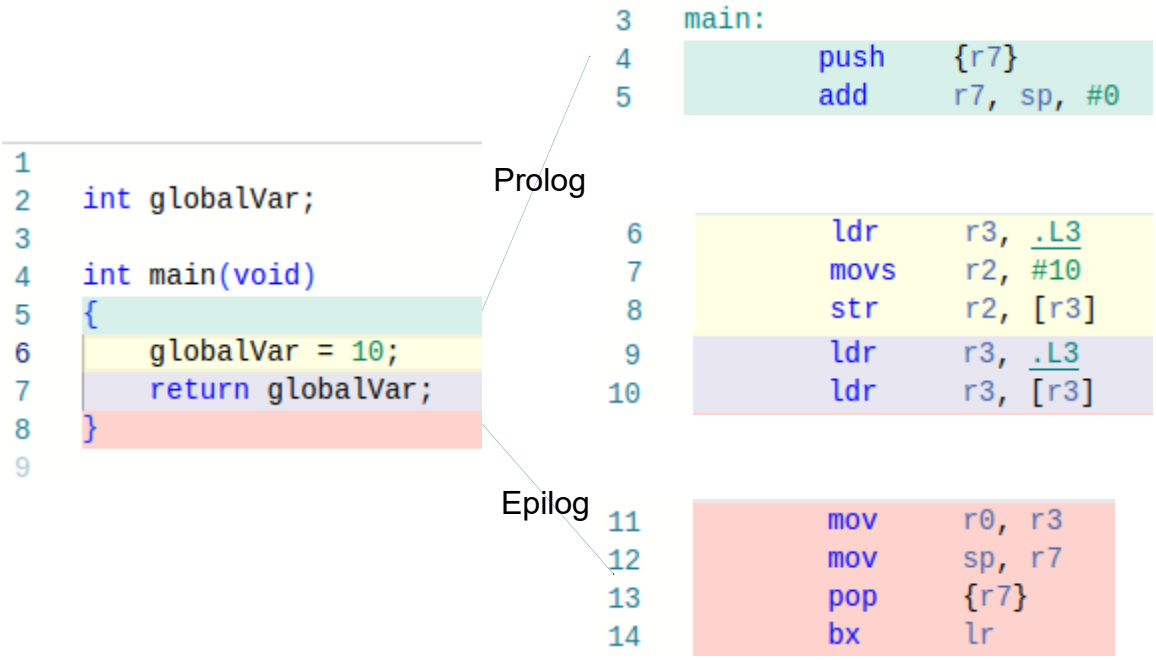
Compiler Toolchain

Präprozessor, Compiler, Assembler und Linker

Code Generierung – Prolog und Epilog

GNU Compiler Toolchain

Assembler Code Generierung – Prolog und Epilog



Der Prolog einer Funktion ist dafür verantwortlich, dass alle notwendigen Register auf dem Stack gesichert werden sowie ein entsprechender Stackframe aufgebaut wird.
Hierbei wird auch der notwendige Speicher auf dem Stack reserviert

Der Epilog einer Funktion hat die Aufgabe den Stackframe wieder „abzubauen“ und alle Register wieder herzustellen. Als Letztes springt der Epilog wieder zurück zum Aufrufer

Quelle: intern

DHBW
Duale Hochschule
Baden-Württemberg

Compiler Toolchain

Präprozessor, Compiler, Assembler und Linker

Code Generierung – Floating Point Unit

GNU Compiler Toolchain

Code Generierung mit/ohne Floating Point Unit

Zum Standard Sprachumfang von C gehört, unabhängig von den Leistungsmerkmalen der verwendeten Zielhardware, die Funktionalität von Fließkommazahlen in den Formaten `float` und `double` (IEEE 754)

Da nicht jede Zielhardware eine sog. Floating Point Unit (FPU) in Hardware realisiert hat, muss der Compiler dafür sorgen, dass dennoch mit Fließkommazahlen gearbeitet werden kann.

Dies realisiert der Compiler durch Zusatzfunktionen in Software, welche die Berechnung mit Fließkommazahlen in SW-Algorithmen realisieren

Die SW Emulation von FPU Funktionen ist aber um ein vielfaches langsamer als die Realisierung direkt in Hardware. Daher kann der Compiler bei der Code-Generierung instruiert werden, ob Code für

- SW Emulation FPU
- HW FPU

generiert werden soll

Quelle: intern

Code Generierung mit Floating Point Unit

Folgende Kommandozeilen Argumente beeinflussen die Code-Generierung für die Fließkommabehandlung

`-mfloat-abi=name`	Specifies which floating-point ABI to use. Permissible values are: ‘soft’, ‘softfp’ and ‘hard’. Specifying ‘soft’ causes GCC to generate output containing library calls for floating-point operations. ‘softfp’ allows the generation of code using hardware floating-point instructions, but still uses the soft-float calling conventions. ‘hard’ allows generation of floating-point instructions and uses FPU-specific calling conventions. The default depends on the specific target configuration. Note that the hard-float and soft-float ABIs are not link-compatible; you must compile your entire program with the same ABI, and link with a compatible set of libraries.
`-march=name[+extension…]`	‘armv7e-m’ ‘+fp’ The single-precision VFPv4 floating-point instructions. ‘+fpv5’ The single-precision FPv5 floating-point instructions. ‘+fp.dp’ The single- and double-precision FPv5 floating-point instructions. ‘+nofp’ Disable the floating-point extensions.

Quelle: intern

GNU Compiler Toolchain

Code Generierung mit Floating Point Unit

```
float floatSum(float a, float b)
{
    return a + b;
}

int main()
{
    float res = floatSum(3.14, 2.0);

    return 0;
}
```

Compiler Optionen
`-O0 -mfloat-abi=hard -march=armv7e-m+fp`

```
floatSum(float, float):
        push    {r7}
        sub     sp, sp, #12
        add     r7, sp, #0
        vstr.32 s0, [r7, #4]
        vstr.32 s1, [r7]
        vldr.32 s14, [r7, #4]
        vldr.32 s15, [r7]
        vadd.f32        s15, s14, s15
        vmov.f32        s0, s15
        adds    r7, r7, #12
        mov     sp, r7
        ldr     r7, [sp], #4
        bx      lr
```

```
main:
        push    {r7, lr}
        sub     sp, sp, #8
        add     r7, sp, #0
        vmov.f32        s1, #2.0e+0
        vldr.32 s0, .L5
        bl      floatSum(float, float)
        vstr.32 s0, [r7, #4]
        movs    r3, #0
        mov     r0, r3
        adds    r7, r7, #8
        mov     sp, r7
        pop     {r7, pc}
.L5:
        .word   1078523331
```

Quelle: intern

GNU Compiler Toolchain

Code Generierung ohne Floating Point Unit

```c
float floatSum(float a, float b)
{
    return a + b;
}

int main()
{
    float res = floatSum(3.14, 2.0);

    return 0;
}
```

Compiler Optionen
-O0

```
floatSum(float, float):
        push    {fp, lr}
        add     fp, sp, #4
        sub     sp, sp, #8
        str     r0, [fp, #-8]     @ float
        str     r1, [fp, #-12]    @ float
        ldr     r1, [fp, #-12]    @ float
        ldr     r0, [fp, #-8]     @ float
        bl      __aeabi_fadd
        mov     r3, r0
        mov     r0, r3
        sub     sp, fp, #4
        pop     {fp, lr}
        bx      lr
```

```
main:
        push    {fp, lr}
        add     fp, sp, #4
        sub     sp, sp, #8
        mov     r1, #1073741824
        ldr     r0, .L5
        bl      floatSum(float, float)
        str     r0, [fp, #-8]     @ float
        mov     r3, #0
        mov     r0, r3
        sub     sp, fp, #4
        pop     {fp, lr}
        bx      lr
.L5:
        .word   1078523331
```

Quelle: intern

GNU Compiler Toolchain

Code Generierung mit/ohne Floating Point Unit

```
        adds    r7, r7, #8
        mov     sp, r7
        pop     {r7, pc}
.L5:
        .word   1078523331
```

1078523331 = 0x4048F5C3

IEEE 753 Converter

IEEE 754 Converter, 2024-02

	Sign	Exponent	Mantissa
Value:	+1	2^{1}	1 + 0.5700000524520874
Encoded as:	0	128	4781507
Binary:	☐	☑☐☐☐☐☐☐☐	☑☐☐☑☐☐☐☑☑☑☑☐☑☐☑☑☑☐☐☐☐☐☑☑

Decimal Representation	3.14
Value actually stored in float:	3.140000104904174804687 5
Error due to conversion:	0.000000104904174804687 5
Binary Representation	01000000010010001111010111000011
Hexadecimal Representation	4048f5c3

1

-1

Quelle: intern

DHBW
Duale Hochschule
Baden-Württemberg

Compiler Toolchain

Präprozessor, Compiler, Assembler und Linker

Code Generierung – Optimierungsstufen

Code Generierung - Optimierungsstufen

Heutige moderne Compiler bieten sehr gute Optimierungsalgorithmen, um den generierten Assembler-Code hinsichtlich Laufzeit oder Speicherverbrauch zu optimieren.

Der GNU Compiler bietet hierzu eine unzählige Anzahl von einzeln aktivierbaren Algorithmen, welche jeweils auf einen anderen Aspekt hin optimieren.

Eine Übersicht ist unter https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html zu finden

Damit nicht jeder anzuwendende Optimierungsalgorithmus einzelne angegeben werden muss, bietet der Compiler eine Zusammenfassung von Algorithmen als sog. Optimierungsstufe.

Der GNU Compiler unterstützt aktuell 5 Stufen:

- O0 – Keine Optimierung
- O1 – Code Größen Reduzierung ohne Einfluss auf die Compilier-Zeit
- O2 – Alle Optimierungen die keinen sog. Space-Speed-Tradeoff erzeugen
- O3 – Maximale Optimierung
- Os – Optimierung auf Code-Größe und nicht auf Laufzeit

Quelle: intern

GNU Compiler Toolchain

Code Generierung - Optimierungsstufen

Auch wenn die Optimierungsstufen in den allermeisten Fällen einiges an Speicherverbrauch reduzieren und Laufzeit gewinnen können, so sind sie dennoch mit Vorsicht zu genießen.

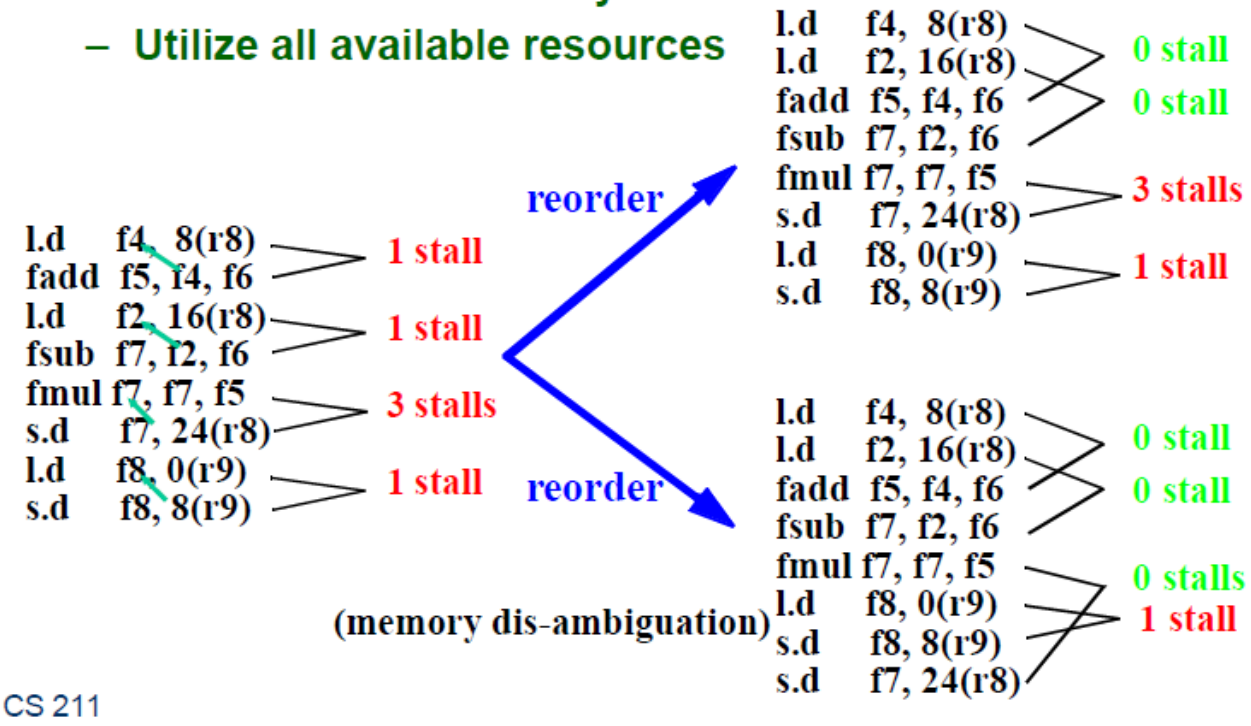
Bei hohen Optimierungsstufen kann der Compiler u.a. Code-Reihenfolgen mit Sprüngen „optimieren“ was dazu führen kann, dass der Debugger während der Schrittweisen Ausführung zwischen Zeilen springt, da sich der High-Level C-Code nicht mehr auf den Assembler-Code abbilden lässt

Häufig ist die Verwendung von Optimierungsstufen in Safety-kritischen Anwendungen eingeschränkt. Viele Compiler-Hersteller beschränken für solche Anwendungen die nutzbaren Optimierungsstufen auf die erste oder maximale zweite Stufe.

Quelle: intern

Code Generierung - Optimierungsstufen

- **Rearrange code sequence to minimize execution time**
  - Hide instruction latency
  - Utilize all available resources



Quelle: https://www2.seas.gwu.edu/~bhagiweb/cs211/lectures/compiler.pdf