

Sichere Programmierung

Projekt 2

Julian Sobott
(76511)
David Sugar
(76050)

Inhaltsverzeichnis

1	Einleitung	3
1.1	Environment	3
1.2	Befehle	3
2	Zu Aufgabe 1: C-for-Schleifen in Assembler	4
2.1	a) Analyse des C-Codes	4
2.2	b) Ausführen des Programms	4
2.3	c) Analyse des Assembler-Codes	5
2.4	d) Ausführen des Programms in gdb	6
3	Zu Aufgabe 2: Stackframes	11
3.1	a) Analyse des C-Codes	11
3.2	b) Ausführen des Programms	12
3.3	c) Stackframes	12
3.3.1	Reihenfolge der Funktionsaufrufe	12
3.3.2	Analyse der Stackframes	13
4	Zu Aufgabe 3	19
4.1	a) Analysieren Sie den in der Datei enthaltenen Source Code	19
4.1.1	Implementierung	19
4.1.2	Aufruf	20
4.2	b) Kompilieren Sie den C Code und führen Sie das Programm aus	20
4.2.1	Kompilieren	20
4.2.2	Ausführen	20
4.3	c) Führen Sie das Programm im GDB aus	21
4.3.1	Wie viele Stack Frames werden erzeugt?	21
4.3.2	Wie ist der Inhalt dieser Stack Frames?	22
4.3.3	Wie wird die Parameterübergabe in Assembler umgesetzt?	24
5	Zu Aufgabe 4	26
5.1	a) Analysieren Sie den in der Datei enthaltenen Source Code	26
5.1.1	Analyse	26
5.1.2	Floating Point Numbers	26
5.2	b) Kompilieren Sie den C Code und führen Sie das Programm aus	27
5.2.1	Kompilieren	27

5.2.2	Ausführen	27
5.3	c) Analysieren Sie das Programm mit dem GDB	28
5.3.1	Berechnet die Schleife das korrekte Ergebnis?	28
5.3.2	Welche Werte nehmen die Variablen bei der Ausführung des Programms an?	28
5.3.3	Warum wird das falsche Ergebnis berechnet?	32
5.3.4	Erstellen Sie ein modifiziertes Programm, welches ein korrektes Ergebnis liefert.	33
6	Zu Aufgabe 5: Binäre Suche	35
6.1	Kurze Erklärung von binary-search	35
6.2	a) Analyse des C-Codes	35
6.3	b) Ausführen des Programms	36
6.3.1	Segmentation fault (core dumped)	37
6.3.2	Schlussfolgerung	38
6.4	c) Ausführen des Programms in gdb	38
6.5	d) Beheben des Fehlers	40

1 Einleitung

Programmiersprachen bzw. Programme zu verstehen geht weit über eine Kenntnis der Syntax und Konventionen hinaus. Hinter jedem Programm, ob Kernel, Dämon oder Nutzer-Programm, versteckt sich eine Abfolge von Maschineninstruktionen, die von ihrem Aufbau kaum noch Ähnlichkeit mit der High-Level-Programmiersprache haben, mit der das jeweilige Programm geschrieben wurde. Dieses tiefere Verständnis zu erlangen, war Ziel dieses Praktikums.

1.1 Environment

Betriebssystem	Linux 64-Bit
Debugger	GDB mit gef Extension

1.2 Befehle

Bei der Analyse der Binärdateien mit GDB wurden u.a. folgende Befehlen verwendet.

break/b {*<addr>|<line>|<function>}

Setze eine Breakpoint an der angegebenen Stelle. Wird keine Position angegeben, wird der aktuelle Wert des Instruction-Pointer verwendet.

run/ r <arg1> <arg2> ...

Starte das Programm mit den angegebenen Argumenten.

disassemble/ disass <function>

Disassembliere die angegebene Funktion.

next/ n

Springe zur nächsten Instruktion. Betrete keine Unterprogramme.

step/ s

Springe zur nächsten Instruktion. Betrete Unterprogramm falls nächste Instruktion Unterprogrammaufruf.

x/lengthformat <address>

Untersuche length Werte an der angegebenen Adresse, interpretiert als format der Größe size.

- format
 - o - octal
 - x - hexadecimal
 - d - decimal
 - s - string
 - f - float

- ...
- size
 - b - byte - 8 Bit
 - h - half word - 16 Bit
 - w - word - 32 Bit
 - g - giant - 64 Bit
- length - $x \in [1..n]$

2 Zu Aufgabe 1: C-for-Schleifen in Assembler

2.1 a) Analyse des C-Codes

gdb-uebung-1.c

```

1  #include <stdio.h>
2
3  int main() {
4      unsigned int i;
5
6      for (i=0; i<20; i++) {
7          printf("i: %2d\n", i);
8      }
9      return 0;
10 }
```

Zu Beginn der `main()` Funktion wird eine `unsigned int` Variable, `i`, deklariert, jedoch nicht initialisiert, d.h. bis auf wenige Ausnahmen $i \in \{0..2^{32} - 1\}$.

Danach wird die Variable im Kopf der darauf folgenden For-Schleife mit 0 initialisiert. Die Schleife inkrementiert die Variable `i` am Ende jedes Schleifendurchlaufs und tritt erneut in die Schleife ein, solange `i` kleiner 20 ist. Innerhalb der Schleife wird der Wert von `i`, zum jeweiligen Zeitpunkt, formatiert mithilfe von `printf()` in der Standardausgabe ausgegeben. Dabei werden immer 2 Stellen ausgegeben, dies wird über `"%2d"` realisiert.

Potentielles Problem: Es sollte `"%2u"` verwendet werden, da `d` für die Formatierung von signed Integeren verwendet wird. In diesem Fall spielt die Formatierung aber keine Rolle.

2.2 b) Ausführen des Programms

Das Programm wurde kompiliert und ausgeführt. Wie erwartet werden die Zahlen von 0 bis 19 ausgegeben.

```

Praktikum2|master ⚡ ⇒ gcc gdb-uebung-1.c -o gdb-uebung-1
Praktikum2|master ⚡ ⇒ ./gdb-uebung-1
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
i: 10
i: 11
i: 12
i: 13
i: 14
i: 15
i: 16
i: 17
i: 18
i: 19

```

Abbildung 1: Ausgabe von gdb-uebung-1.c

2.3 c) Analyse des Assembler-Codes

1	<+8>:	mov	DWORD PTR [rbp-0x4],0x0
2	<+15>:	jmp	0x40113b <main+41>
3	<+17>:	mov	eax,DWORD PTR [rbp-0x4]
4	<+20>:	mov	esi, eax
5	<+22>:	mov	edi,0x402004
6	<+27>:	mov	eax,0x0
7	<+32>:	call	0x401030 <printf@plt>
8	<+37>:	add	DWORD PTR [rbp-0x4],0x1
9	<+41>:	cmp	DWORD PTR [rbp-0x4],0x13
10	<+45>:	jbe	0x401123 <main+17>

Für die Variable *i* wird Speicher auf dem Stack alloziert, die Anfangsadresse ist dabei `rbp-0x4`.

In Zeile `<+8>` wird *i* mit `0x0` initialisiert. Danach springt das Programm unbedingt in Zeile `<+41>`. Hier befindet sich nun die Überprüfung, ob die Schleife verlassen wird, d.h. $i \geq 0x14$, oder ein weiterer Schleifendurchlauf gestartet wird. Dazu wird in Zeile `<+41>`

i mit 0x13 verglichen. Ist der Wert kleiner oder gleich 0x13 wird in Zeile <+17> gesprungen und damit ein weiterer Schleifendurchlauf gestartet. Andernfalls wird die nächste Instruktion ausgeführt und damit die Schleife verlassen.

In Zeile <+17> und <+20> wird der Wert von i, vom Speicher in das `esi` Register geladen. In der darauf folgenden Zeile wird die Adresse des Formatierungsstrings ("i: %2d\n") (0x402004) in `edi` geladen.

```
1
2 gef x/s 0x402004
3 0x402004: "i: %2d\n"
```

Weiterhin wird `eax` wieder auf 0x0 zurückgesetzt. Danach wird `printf()` mit den in `edi` und `esi` geladenen Parametern aufgerufen. Schlussendlich wird i inkrementiert und daraufhin wieder verglichen (<+41>).

2.4 d) Ausführen des Programms in gdb

In dieser Aufgabe geht es nun darum das Programm `gdb-uebung-1.c` in `gdb` auszuführen. Im folgenden wird der Ablauf durch Screenshots und entsprechende Erklärungen beschrieben.

```
0x40110e <__do_global_ctors_aux+46> add    bl, bpl
0x401111 <frame_dummy+1>    mov     ss, WORD PTR [rbp+0x48]
0x401114 <main+2>          mov     ebp, esp
→ 0x401116 <main+4>        sub     rsp, 0x10
0x40111a <main+8>          mov     DWORD PTR [rbp-0x4], 0x0
0x401121 <main+15>         jmp     0x40113b <main+41>
0x401123 <main+17>         mov     eax, DWORD PTR [rbp-0x4]
0x401126 <main+20>         mov     esi, eax
0x401128 <main+22>         mov     edi, 0x402004
```

Abbildung 2: 1. Ausgabe von `gdb-uebung-1.c`

1. Hier beginnt die `main` Funktion. Als erstes wird der `rsp` Zeiger, welcher auf den Stack zeigt, um 0x10 verschoben, um entsprechen Platz auf den Stack zu allozieren.

```
0x401111 <frame_dummy+1>    mov     ss, WORD PTR [rbp+0x48]
0x401114 <main+2>          mov     ebp, esp
0x401116 <main+4>          sub     rsp, 0x10
→ 0x40111a <main+8>        mov     DWORD PTR [rbp-0x4], 0x0
0x401121 <main+15>         jmp     0x40113b <main+41>
0x401123 <main+17>         mov     eax, DWORD PTR [rbp-0x4]
0x401126 <main+20>         mov     esi, eax
0x401128 <main+22>         mov     edi, 0x402004
0x40112d <main+27>         mov     eax, 0x0
```

Abbildung 3: 2. Ausgabe von `gdb-uebung-1.c`

2. Initialisieren der Variable i mit 0x0.

```
0x401113 <main+1>      mov     rbp, rsp
0x401116 <main+4>      sub     rsp, 0x10
0x40111a <main+8>      mov     DWORD PTR [rbp-0x4], 0x0
→ 0x401121 <main+15>   jmp     0x40113b <main+41>
0x401123 <main+17>     mov     eax, DWORD PTR [rbp-0x4]
0x401126 <main+20>     mov     esi, eax
0x401128 <main+22>     mov     edi, 0x402004
0x40112d <main+27>     mov     eax, 0x0
0x401132 <main+32>     call    0x401030 <printf@plt>
```

Abbildung 4: 3. Ausgabe von gdb-uebung-1.c

3. Unbedingter Sprung in Zeile <main+41>.

```
gef> x/d $rbp-0x4
0x7fffffffddc9c: 0
```

Abbildung 5: 4. Ausgabe von gdb-uebung-1.c

4. Ausgabe von i. (Adresse: Wert). Der Wert wird in Dezimal ausgegeben.

Schritte bis zur nächsten Zeile wurden übersprungen, da sie in Aufgabe 1 b) ausführlich erklärt wurden.

```
→ 0x40113f <main+45>   jbe     0x401123 <main+17>      TAKEN [Reason: C || Z]
↳ 0x401123 <main+17>   mov     eax, DWORD PTR [rbp-0x4]
0x401126 <main+20>     mov     esi, eax
0x401128 <main+22>     mov     edi, 0x402004
0x40112d <main+27>     mov     eax, 0x0
0x401132 <main+32>     call    0x401030 <printf@plt>
0x401137 <main+37>     add     DWORD PTR [rbp-0x4], 0x1
```

Abbildung 6: 5. Ausgabe von gdb-uebung-1.c

5. Der bedingte Sprung jbe (jump below or equal) wird genommen, da $0x0 \leq 0x13$. Das heißt, das Programm springt zu <main+17>.

```
→ 0x401123 <main+17>   mov     eax, DWORD PTR [rbp-0x4]
0x401126 <main+20>     mov     esi, eax
0x401128 <main+22>     mov     edi, 0x402004
0x40112d <main+27>     mov     eax, 0x0
0x401132 <main+32>     call    0x401030 <printf@plt>
```

Abbildung 7: 6. Ausgabe von gdb-uebung-1.c

6. Schreiben des Wertes von `i` in `eax` und `eax` dann in `esi`, um `i` als Parameter an die `printf` Funktion zu übergeben.

```

→ 0x401132 <main+32>      call    0x401030 <printf@plt>
↳ 0x401030 <printf@plt+0> jmp     QWORD PTR [rip+0x2fe2]      # 0x404018 <printf@got.plt>
0x401036 <printf@plt+6>  push    0x0
0x40103b <printf@plt+11> jmp     0x401020
0x401040 <_start+0>      xor     ebp, ebp
0x401042 <_start+2>      mov     r9, rdx
0x401045 <_start+5>      pop     rsi

```

Abbildung 8: 7. Ausgabe von `gdb-uebung-1.c`

```

$rsi      : 0x0
$rdi      : 0x00000000000402004 → 0x000a643225203a69 ("i: %2d"? )

```

Abbildung 9: 8. Ausgabe von `gdb-uebung-1.c`

7.+ 8. Aufrufen der `printf` Funktion mit `i=0`. Übergeben wird in `rsi` und `rdi` der Wert von `i` und ein pointer auf den format string. Dieser Aufruf führt zu folgender Ausgabe auf dem Standardoutput:

```

1 i: 0

```

```

0x401128 <main+22>      mov     edi, 0x402004
0x40112d <main+27>      mov     eax, 0x0
0x401132 <main+32>      call    0x401030 <printf@plt>
→ 0x401137 <main+37>      add     DWORD PTR [rbp-0x4], 0x1
0x40113b <main+41>      cmp     DWORD PTR [rbp-0x4], 0x13
0x40113f <main+45>      jbe     0x401123 <main+17>
0x401141 <main+47>      mov     eax, 0x0
0x401146 <main+52>      leave
0x401147 <main+53>      ret

```

Abbildung 10: 9. Ausgabe von `gdb-uebung-1.c`

9. Hier wird der Wert von `i` nun um eins erhöht.

```

gef> x/d $rbp-0x4
0x7fffffffddccc: 1

```

Abbildung 11: 10. Ausgabe von `gdb-uebung-1.c`

10. Nach der ausführung ist der Wert 1.

```
gef> x/d $rbp-0x4
0x7fffffffddccc: 0
gef> b *main+45 if *0x7fffffffddccc == 0x13
Breakpoint 2 at 0x40113f
gef> info break
```

Num	Type	Disp	Enb	Address	What
1	breakpoint	keep y		0x0000000000401116	<main+4>
	breakpoint already hit 1 time				
2	breakpoint	keep y		0x000000000040113f	<main+45>
	stop only if *0x7fffffffddccc == 0x13				

Abbildung 12: 11. Ausgabe von gdb-uebung-1.c

11. Als nächstes wollen wir das Programm bis zum letzten Durchlauf laufen lassen und dort dann einen Breakpoint setzen. Als erstes geben wir uns hierfür die Adresse für `i` aus. Diese wird benötigt, da der Conditional Breakpoint nur stoppen soll, wenn `i` einen bestimmten Wert hat. In der nächsten Zeile setzen wir den Conditional Breakpoint in die Zeile wo der bedingte Sprung ist (`<main+45>`). Als Bedingung geben wir an, dass der Wert von `i` gleich `0x13` sein soll. Wie auch in C müssen Adressen jeweils mit dem `*` dereferenziert werden. Am Ende wird noch kontrolliert ob der breakpoint richtig gesetzt wurde.

```
gef> c
Continuing.
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
i: 10
i: 11
i: 12
i: 13
i: 14
i: 15
i: 16
i: 17
i: 18
```

Abbildung 13: 12. Ausgabe von gdb-uebung-1.c

12. Lassen wir das Programm nun mit `continue (c)` laufen, sehen wir alle Schleifendurchläufe mit den entsprechenden Ausgaben. Die letzte Ausgabe ist 18 (`0x12`).

```

0x401132 <main+32>      call    0x401030 <printf@plt>
0x401137 <main+37>      add     DWORD PTR [rbp-0x4], 0x1
0x40113b <main+41>      cmp     DWORD PTR [rbp-0x4], 0x13
→ 0x40113f <main+45>      jbe     0x401123 <main+17>      TAKEN [Reason: C || Z]
↳ 0x401123 <main+17>      mov     eax, DWORD PTR [rbp-0x4]
0x401126 <main+20>      mov     esi, eax
0x401128 <main+22>      mov     edi, 0x402004
0x40112d <main+27>      mov     eax, 0x0
0x401132 <main+32>      call    0x401030 <printf@plt>
0x401137 <main+37>      add     DWORD PTR [rbp-0x4], 0x1

[#0] Id 1, Name: "gdb-uebung-1", stopped, reason: BREAKPOINT

[#0] 0x40113f → main()

Breakpoint 2, 0x000000000040113f in main ()
gef> x/d $rbp-0x4
0x7fffffffdbcc: 19

```

Abbildung 14: 13. Ausgabe von gdb-uebung-1.c

13. Der bedingte Sprung wird ein letztes Mal genommen, da der Wert von `i` gleich 19 (0x13) ist.

```

0x401132 <main+32>      call    0x401030 <printf@plt>
0x401137 <main+37>      add     DWORD PTR [rbp-0x4], 0x1
0x40113b <main+41>      cmp     DWORD PTR [rbp-0x4], 0x13
→ 0x40113f <main+45>      jbe     0x401123 <main+17>      NOT taken [Reason: !(C || Z)]
0x401141 <main+47>      mov     eax, 0x0
0x401146 <main+52>      leave
0x401147 <main+53>      ret
0x401148               nop     DWORD PTR [rax+rax*1+0x0]
0x401150 <__libc_csu_init+0> push    r15

[#0] Id 1, Name: "gdb-uebung-1", stopped, reason: SINGLE STEP

[#0] 0x40113f → main()

0x000000000040113f in main ()
gef> x/d $rbp-0x4
0x7fffffffdbcc: 20
gef> x/h $rbp-0x4
0x7fffffffdbcc: 20
gef> x/x $rbp-0x4
0x7fffffffdbcc: 0x0014

```

Abbildung 15: 14. Ausgabe von gdb-uebung-1.c

14. Die Schleife ist eine letztes Mal durchgelaufen wie erwartet und hat noch die 19 ausgegeben. Wenn wir nun aber an dem bedingten Sprung angekommen wird dies nicht mehr genommen, da die Bedingung nicht mehr zutrifft ($i = 0x14$ und somit gilt nicht mehr $i \leq 0x13$).

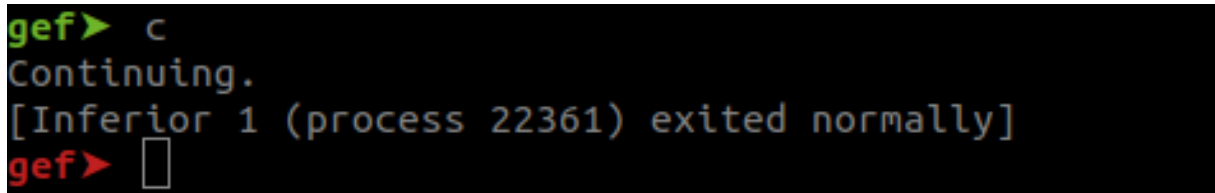
```

0x401136 <main+36>      inc     DWORD PTR [rbx-0x7cfe03bb]
0x40113c <main+42>      jge     0x40113a <main+40>
0x40113e <main+44>      adc     esi, DWORD PTR [rsi-0x1e]
→ 0x401141 <main+47>      mov     eax, 0x0
0x401146 <main+52>      leave
0x401147 <main+53>      ret
0x401148               nop     DWORD PTR [rax+rax*1+0x0]
0x401150 <__libc_csu_init+0> push    r15
0x401152 <__libc_csu_init+2> lea     r15, [rip+0x2ca7]      # 0x403e00

```

Abbildung 16: 15. Ausgabe von gdb-uebung-1.c

15. Anstatt an `<main+17>` zu springen, wurde zur nächsten Anweisung gesprungen `<main+47>`. Somit wurde die Schleife verlassen. Hier wird noch die 0 als Rückgabewert gespeichert.



```
gef> c
Continuing.
[Inferior 1 (process 22361) exited normally]
gef> 
```

Abbildung 17: 16. Ausgabe von gdb-uebung-1.c

16. Mit `continue (c)` lassen wir das Programm zuende durchlaufen und es wird normal beendet.

3 Zu Aufgabe 2: Stackframes

3.1 a) Analyse des C-Codes

`gdb-uebung-2.c`

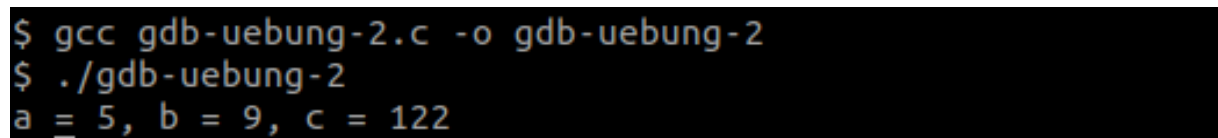
```
1  #include <stdio.h>
2
3  int f(int a, int b) {
4      return 3*a + 7*b;
5  }
6
7  int g(int a, int b) {
8      return 10*a*a - 3*b;
9  }
10
11 int h(int a, int b) {
12     return a + b + 300;
13 }
14
15 int main() {
16     int a = 5, b=9, c=0;
17
18     c = f(g(a,h(a,b)),h(b,a));
19
20     printf("a = %d, b = %d, c = %d\n", a, b, c);
21 }
```

Das Programm besteht aus drei Hilfsfunktionen `f`, `g`, `h`, die jeweils 2 `int`'s als Eingabe bekommen und mit Grundrechenarten ein Ergebnis berechnen und zurück geben. Die Berechnungen scheinen willkürlich sein.

In der `main` Funktion, werden zuerst drei `int`'s `a`, `b`, `c` initialisiert werden. Daraufhin wird in einem geschachtelten Funktionsaufruf der Wert von `c` berechnet. In den Funktionsaufrufen, werden die Rückgabewerte einer Hilfsfunktion immer direkt an die nächste Funktion, als Parameter, übergeben. Am Ende wird durch ein Aufruf der `printf` Funktion ein formatierter String mit den Werten von den Variablen ausgegeben.

3.2 b) Ausführen des Programms

Bild 18 zeigt die Ausgabe des Programms.



```
$ gcc gdb-uebung-2.c -o gdb-uebung-2
$ ./gdb-uebung-2
a = 5, b = 9, c = 122
```

Abbildung 18: Ausgabe von `gdb-uebung-2.c`

Wie erwartet sind die Werte von `a`, `b` unverändert zur ursprünglichen Initialisierung. Nur `c` hat den neuen Wert, den es zugewiesen bekommen hat.

3.3 c) Stackframes

3.3.1 Reihenfolge der Funktionsaufrufe

Als erstes soll festgestellt werden, in welcher Reihenfolge die Funktionen aufgerufen werden. Hierfür wurde in `gdb` mit `s` immer der nächste Schritt ausgeführt. Um C-Code zu sehen wurde das Programm mit Debug Informationen compiliert (Option `-g` bei `gcc`).

Hier nochmal die relevante Code Zeile:

```
1 c = f(g(a,h(a,b)),h(b,a));
```

1. `h(9, 5)` (2. `h`)
2. `h(5, 9)` (1. `h`)
3. `g(5, 314)`
4. `f(-692, 314)`

Interessant bei der Ausführung ist, dass bei dem Aufruf von `f` zuerst der 2. Parameter (`h`) ausgewertet wird und dann erst der 1. (`g`).

3.3.2 Analyse der Stackframes

Im Folgenden werden die einzelnen Stackframes nun genauer betrachtet. Jedes Stackframe wurde in einem Bild dargestellt. Die Vorlage für so ein Stackframe sieht wie folgt aus.

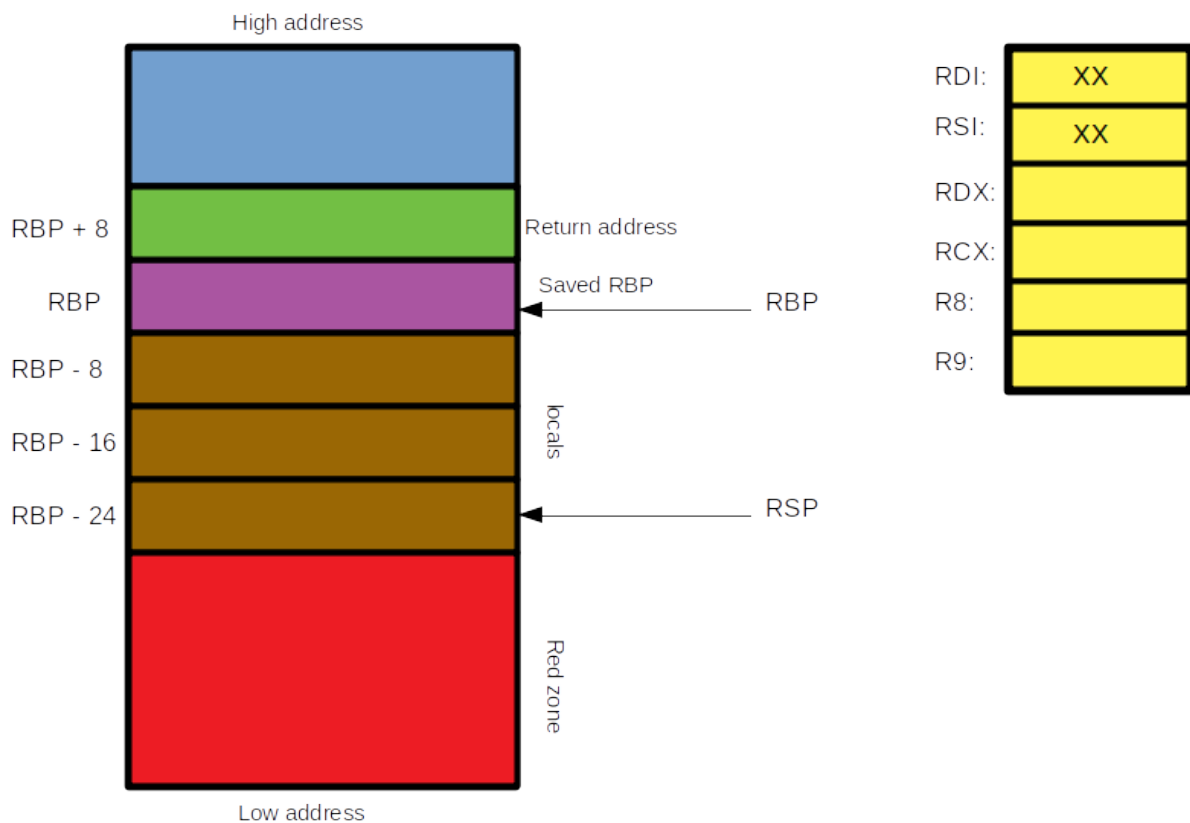


Abbildung 19: 11. Ausgabe von gdb-uebung-2.c

Hier ist links ein Ausschnitt des *Stacks* zu sehen und auf der rechten Seite ein Ausschnitt der *Register*.

Stack:

Bei dem Stack Ausschnitt ist der relevante Teil in dem sich der StackFrame befindet. Auf der linken Seite vom Stack sind die einzelnen Adressen relativ zum RBP Register angegeben und auf der rechten Seite die Bedeutung des Inhalts.

Als oberstes steht die *Rücksprungadresse* zu dieser wird zurück gesprungen, wenn die Funktion fertig ausgeführt wird. An dieser Stelle steht immer die Anweisung, die nach dem Aufruf der Funktion kommt.

Danach kommt der Inhalt des RBP Registers. Als voletztes kommen noch optional lokale Variablen. Als unterstes im Stack kommt die *RedZone*, in die nicht geschrieben werden darf.

Register:

In die Register werden die Paramter eingetragen, mit denen die Funktion aufgerufen wurde. Der erste Parameter wird in RDI, der zweite in RSI usw. gespeichert. Werden mehr als 6 Parameter übergeben, werden die restlichen Parameter im Stack gespeichert.

Nun kann jeder Inhalt des Stackframes gelesen werden. Pointer im Stack wurden aufgelöst um besser zu sehen wohin z.B. die Rücksprungadresse zeigt. Ansonsten würden an den entsprechenden Stellen Adressen stehen. Das Programm wird wieder ausgeführt und am Anfang einer Funktion werden die Daten notiert.

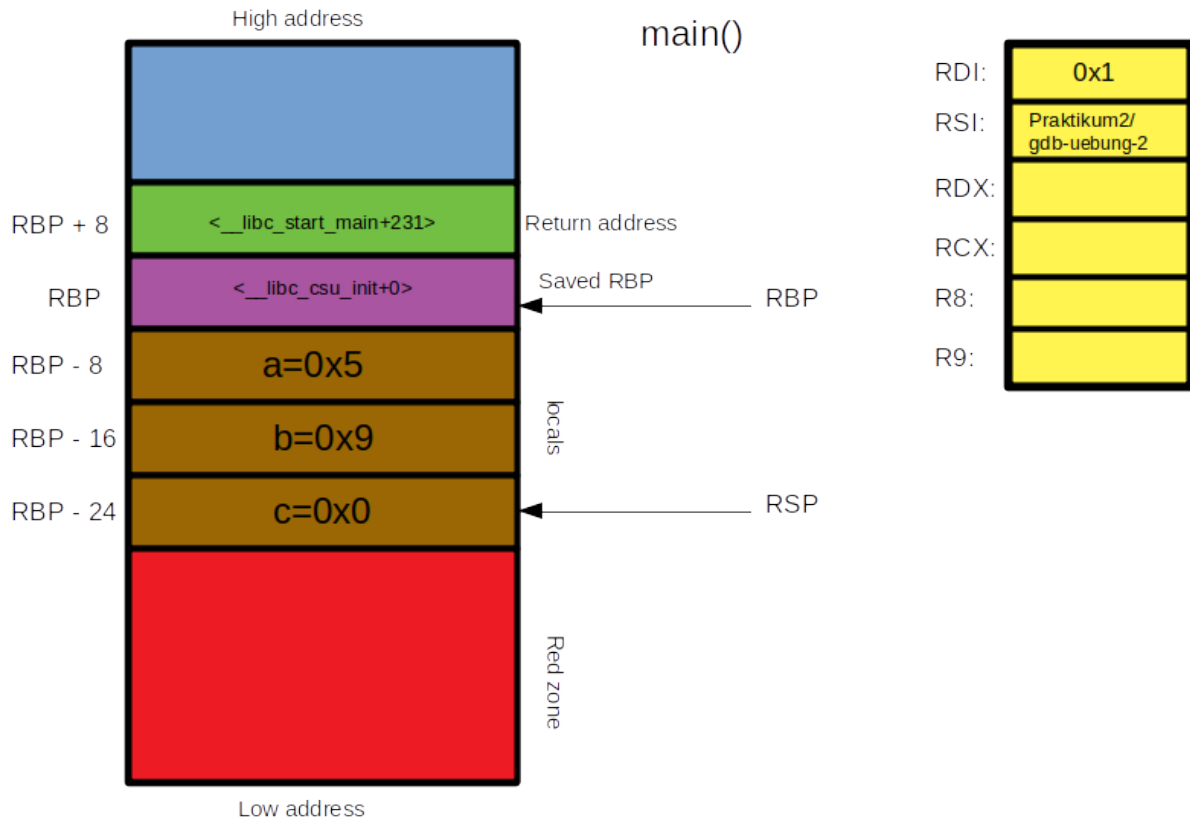


Abbildung 20: 0. Ausgabe von gdb-uebung-2.c

Zu sehen ist, dass als Parameter die die aus Parameter übergeben wurden. In unserem Fall nur die Anzahl und der Name des Programms. Führt man die Zeile in der die Variablen initialisiert werden, sieht man auch diese Werte im Stack.

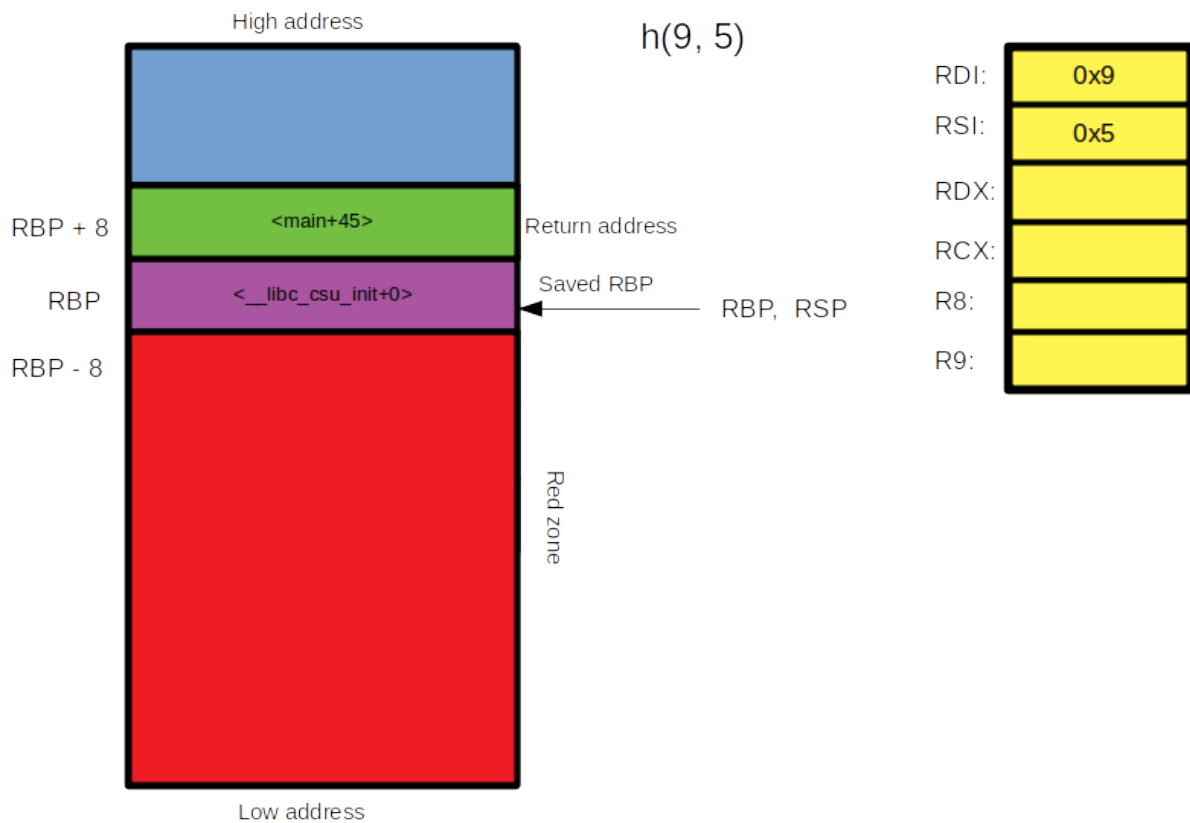


Abbildung 21: 1. Ausgabe von gdb-uebung-2.c

Die Funktion `h` wurde aufgerufen und die entsprechenden Parameter sind in den Registern zu sehen. Außerdem sieht man, dass die Rücksprungadresse nun in eine Zeile in der `main` Funktion zeigt und nicht wie zuvor in C internen Code.

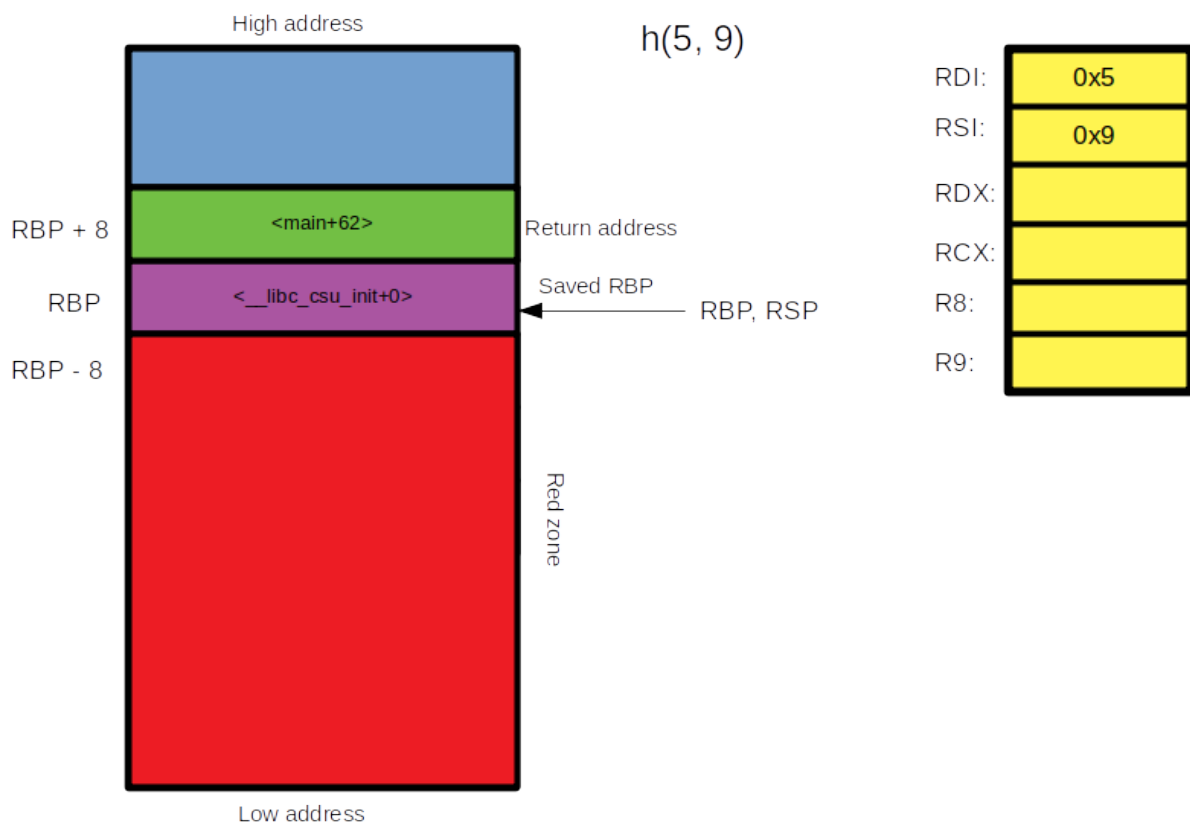


Abbildung 22: 2. Ausgabe von gdb-uebung-2.c

Zweiter Aufruf der Funktion `h` mit umgedrehten Parametern und höherer Rücksprungadresse.

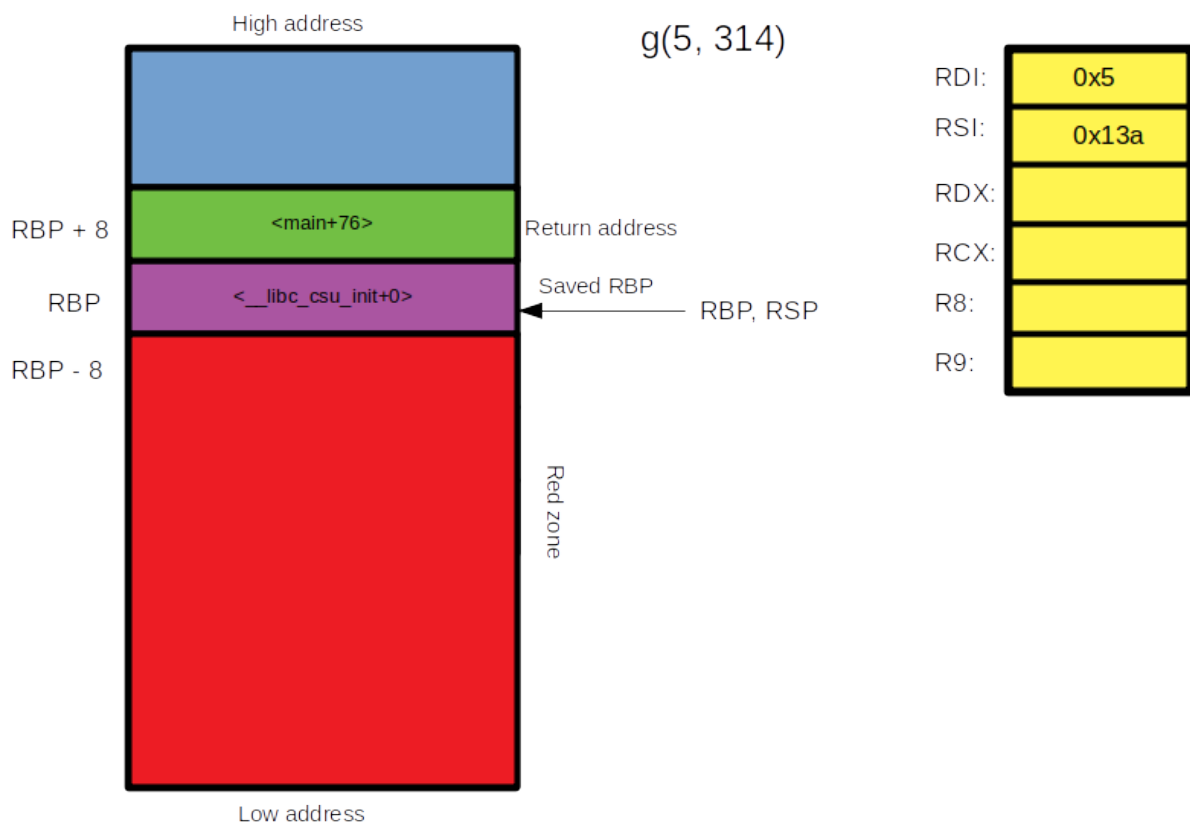


Abbildung 23: 3. Ausgabe von gdb-uebung-2.c

Aufruf der Funktion `g` mit dem Wert aus der Variable `a` und dem Rückgabewert der Funktion `h`.

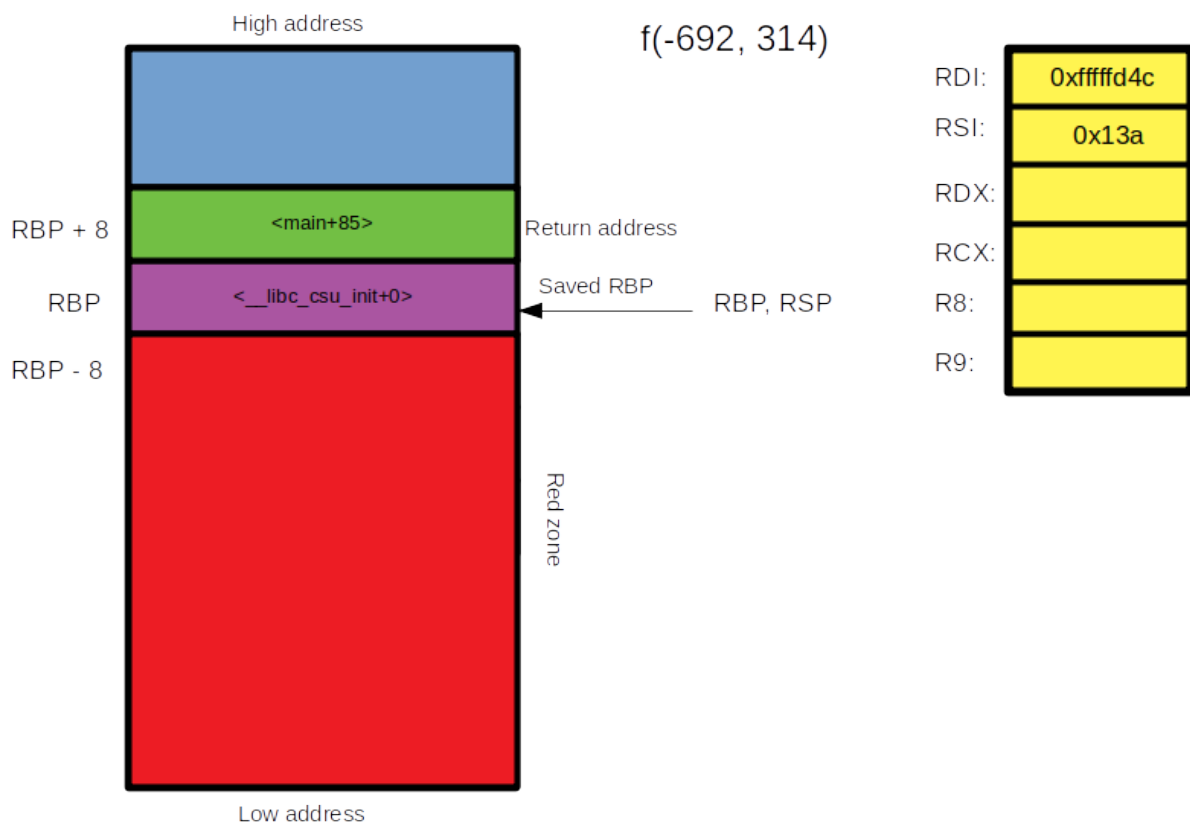


Abbildung 24: 4. Ausgabe von gdb-uebung-2.c

Aufruf der Funktion `f` mit den Rückgabewerten der Funktionen `g`, `h`.

4 Zu Aufgabe 3

4.1 a) Analysieren Sie den in der Datei enthaltenen Source Code

Das bereitgestellte Quelldatei `gdb-uebung-3.c` enthält die rekursive Implementierung eines **Factorial-Algorithmus**, damit ist $f(n) = n!$.

Def: $n! = n * (n - 1) * (n - 2) * \dots * 1 = \prod_{i=1}^n i$, $n \in \mathbb{N}$

4.1.1 Implementierung

```
1 unsigned int f(unsigned int i) {  
2     if (i>1) {  
3         return i * f(i-1);  
4     } else {  
5         return 1;  
6     }  
7 }
```

Abbildung 25: factorial function

Die Funktion, oben abgebildet, nimmt einen vorzeichenlosen Integer als Argument und gibt als Ergebnis ebenfalls einen vorzeichenlosen Integer zurück. Dabei ist **int** jedoch betriebssystemabhängig definiert. Für die meisten Systeme kann jedoch angenommen werden, dass es sich dabei um ein **4 Byte** großes Wort handelt, d.h für den Rückgabewert kommen Werte innerhalb des Wertebereichs $W = [0, 2^{32} - 1]$ in Frage. Aufgrund des extrem schnellen Wachstums von $n!$ ist dies ein sehr beschränkender Faktor, der bei der Nutzung unbedingt mit zu berücksichtigen ist, da es schnell zu einem Überlauf und damit zu einer Verfälschung des Ergebnisses kommen kann. In Zeile 2 wird danach geprüft, ob der übergebene Wert größer 1 ist. Sollte dies der Fall sein, wird das Produkt von i und dem Ergebnis des Rekursiven Aufrufs von $f(i - 1)$ zurückgegeben. Andernfalls wird die Konstante 1 als Rückgabewert der Funktion genutzt, siehe Zeile 5.

4.1.2 Aufruf

```
1 int main() {
2     unsigned int i=5, r=0;
3
4     r = f(i);
5
6     printf("i = %d, f(i) = %d\n", i, r);
7 }
```

Abbildung 26: invocation of f()

In der **main()** Funktion wird die in ?? beschriebene Funktion **f(unsigned int)** aufgerufen und dabei **i = 5** als Argument übergeben. Das Ergebnis des Aufrufs wird der Variable **int r** zugewiesen, siehe Zeile 4. Danach wird **i** und das Ergebnis **r** mittels **printf()** auf der Kommandozeile ausgegeben.

4.2 b) Kompilieren Sie den C Code und führen Sie das Programm aus

4.2.1 Kompilieren

Das Kompilieren des Quellcodes innerhalb von **gdb-uebung-3.c** kann mittels des folgenden Befehls auf der Kommandozeile ausgeführt werden.

```
1 $ gcc gdb-uebung-3.c -o gdb-uebung-3
```

Der in diesem Fall genutzte Compiler heißt **gcc** (GNU compiler collection). Dabei ist **gdb-uebung-3.c** der Name der Quelldatei. Mittels **-o gdb-uebung-3** wird der gewünschte Name, der zu erstellenden Programmdatei, angegeben.

4.2.2 Ausführen

Das Programm kann nun auf der Kommandozeile ausgeführt werden.

```
1 $ ./gdb-uebung-3
2 i = 5, f(i) = 120
```

Zum Verifizieren des Ergebnisses kann dieses auch noch einmal Händisch berechnet werden, $\prod_{i=1}^5 i = 1 * 2 * 3 * 4 * 5 = 2 * 3 * 20 = 2 * 60 = 120$. Das Ergebnis stimmt, die Funktionen scheinen auf den ersten Blick also richtig implementiert. Um nachzuvollziehen, wie das Ergebnis zustande kommt, kann der folgende Graph betrachtet werden.

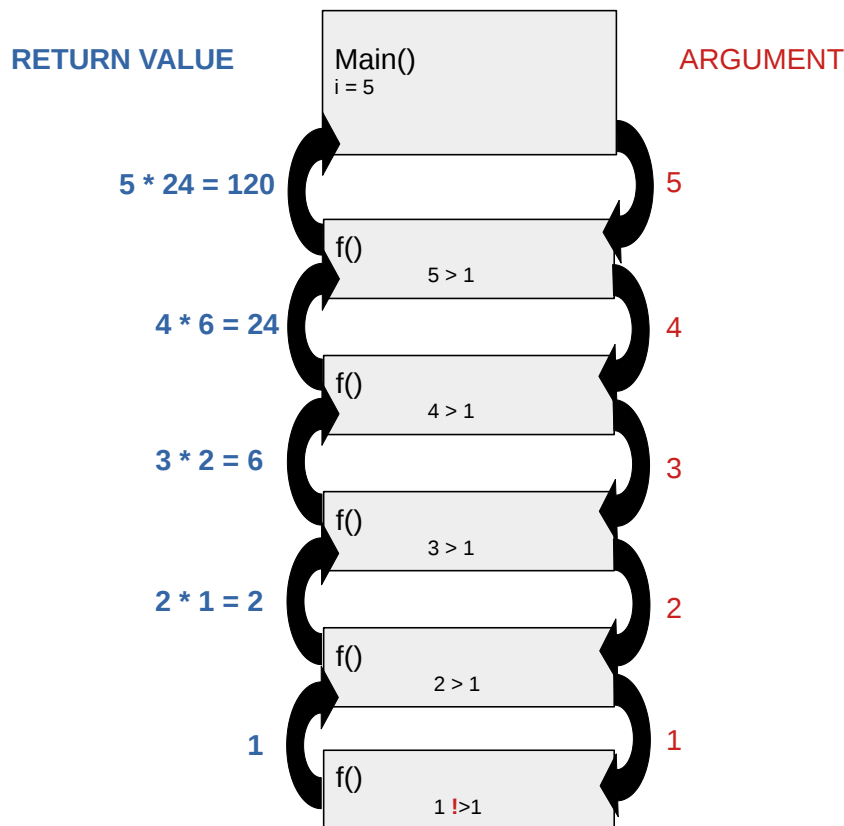


Abbildung 27: recursive call of `f()`

Anfangs wird in `main()` die Funktion `f()` mit **5 als Argument** aufgerufen. Die aufgerufene Funktion `f(5)` prüft nun, ob der Parameter größer als 1, d.h. $5 > 1$, ist. Ist dies der Fall, ruft sich die Funktion selbst wieder auf, dieses Mal jedoch mit dem **dekrementierten** Parameter als Argument. Dies wiederholt sich bis 1 bzw. 0 übergeben wird, in diesem Fall wird 1 zurückgegeben und das Ergebnis 'aufsteigend' berechnet.

4.3 c) Führen Sie das Programm im GDB aus

4.3.1 Wie viele Stack Frames werden erzeugt?

Um die Anzahl der **Stack Frames** in GDB zu ermitteln, kann man wie folgt vorgehen.

- Wir wissen:
 - Das Programm ruft sich selber rekursiv auf.
 - Dies geschieht solange der übergebene Parameter > 1 ist.
 - Andernfalls wird 1 zurückgegeben.

- Der Parameter wird innerhalb von `f()` dekrementiert und als Argument wieder an `f()` übergeben.
- Das Argument wird bei jedem rekursiven Aufruf mittels **RDI** übergeben (siehe 4.3.3).

D.h. wenn ein Bedingter Break Point so gesetzt wird, dass das Programm am Anfang der aufgerufenen Funktion `f()` anhält, genau dann wenn **RDI = 1** ist, können wir bequem die Anzahl der Stack Frames ablesen. Denn in diesem Fall können wir sicher sein, dass kein weiteres Mal `f()` rekursiv aufgerufen wird.

```
1 gef> break f if $rdi == 1
2 gef> r
```

Das Programm wird nun bei maximaler Tiefe $t = 6$, der Funktionsaufrufe angehalten. Diese Tiefe entspricht auch der Anzahl der Stack Frames, da für jeden Funktionsaufruf ein eigener Frame angelegt wird.



Abbildung 28: 1. Ausgabe von `gdb-uebung-3.c`

4.3.2 Wie ist der Inhalt dieser Stack Frames?

Die Werte der Rücksprungadressen, Frame und Stack Pointer sind aufgrund von **Address Space Layout Randomisation (ASLR)** immer unterschiedlich (bis auf die letzten 12 Bit), deshalb werden im Anschluss relative Adressen angegeben.

Main als auch `f()` allozieren für sich jedes mal 16 Byte auf dem Stack.

```
1 MOV rbp, rsp
2 SUB rsp, 0x10 // allocate 16 bytes on the heap
```

`f()` speichert dann den übergebenen Wert direkt unter dem gespeicherten Frame Pointer. Am Schluss wird das berechnete Ergebnis in `main()`, auf den Stack an Stelle `rbp-0x8` geschrieben (rote Markierung 29 in `main`).

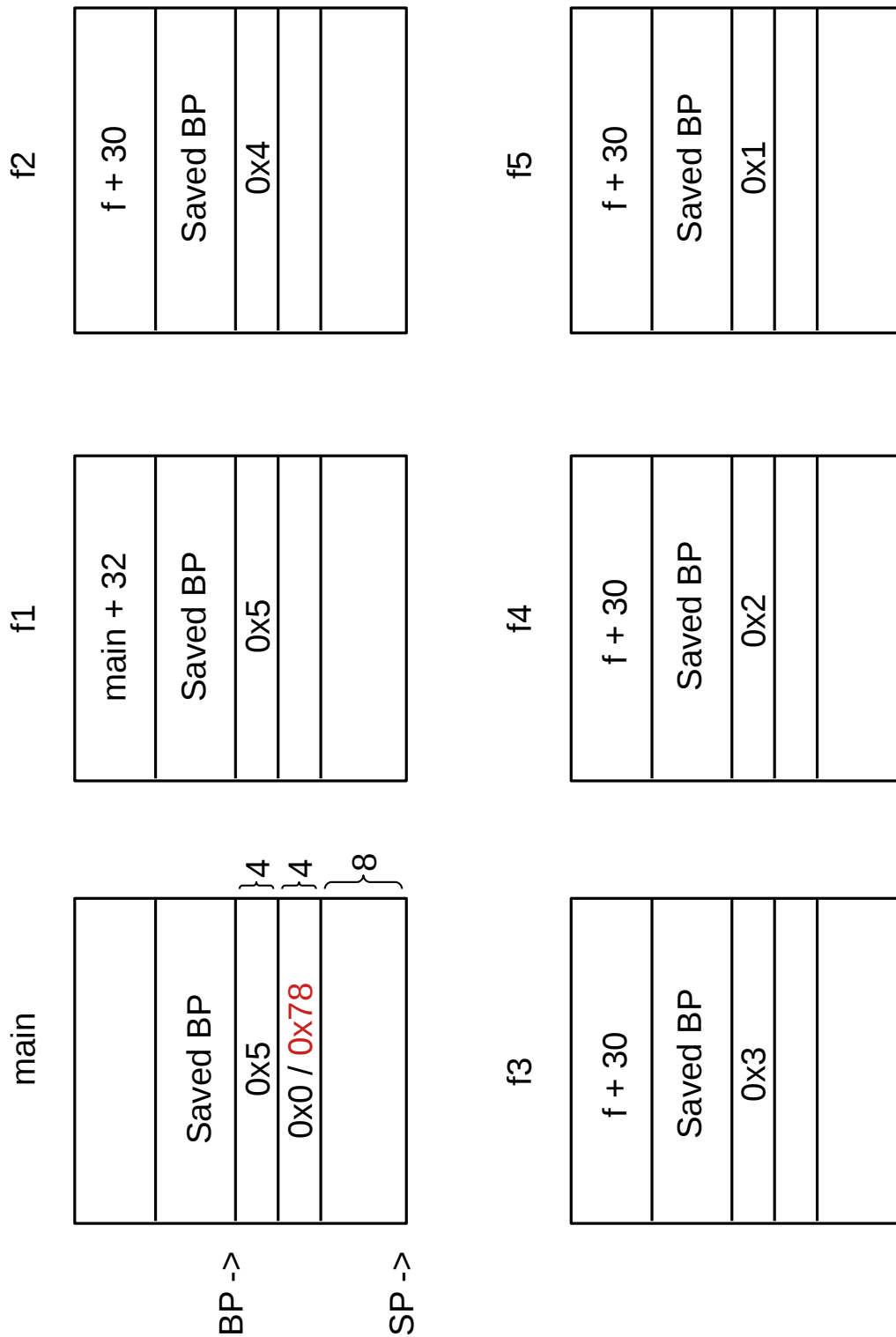


Abbildung 29: Stack Frames aller Funktionsaufrufe für $i = 5$

4.3.3 Wie wird die Parameterübergabe in Assembler umgesetzt?

Für die Übergabe von Parametern an Subroutinen muss unter x86 eine Fallunterscheidung gemacht werden. Je nachdem, ob es sich um Programme für ältere 32-Bit Prozessoren handelt oder um Programme für neuere 64-Bit Prozessoren, werden verschiedene sog. **Calling Conventions** (Aufruf Konventionen) verwendet. Diese Konventionen dienen dazu einen Ablauf zu definieren, sodass unabhängig vom Autor des Codes darauf vertraut werden kann, dass Abläufe wie z.B. ein Unterprogrammaufruf **immer** auf die selbe Weise durchgeführt werden.

32-Bit

Ein Unterprogrammaufruf kann in folgende Schritte untergliedert werden.

1. Zuerst müssen die **Caller Saved Register**, falls nötig, auf dem Stack gespeichert werden, da die **aufgerufene** Funktion für diese Register keine Garantie übernimmt, dass diese nicht überschrieben werden. Die Register sind: ebx, ecx, edx, r10, r11.
2. Danach müssen die Parameter in **umgekehrter Reihenfolge** auf den Stack gepushed werden. Aufgrund der Funktionsweise des Stacks ist dann der Erste Parameter der Subroutine direkt angrenzend an die gespeicherte Rücksprungadresse, die im nächsten Schritt auf dem Stack hinterlegt wird.
3. Nun wird mit **call** der Unterprogrammaufruf durchgeführt. Dabei wird die **Rücksprungadresse** (die Adresse des auf call folgenden Befehlswortes) auf den Stack gepushed und ein Sprung zum ersten Befehl des Unterprogramms, markiert durch das angegebene **Label**, gesprungen.
4. Die ersten Befehle des Unterprogramms bilden einen sog. **Function Prologue**. Hier werden als aller erstes die Werte der **Callee Save Register**, falls die entsprechenden Register benötigt werden, sowie der **Base Pointer (bp)** auf den Stack gepushed. Danach wird der Aktuelle Wert des **Stack-Pointers (sp)** genutzt um den für den derzeitigen **Stack-Frame** verantwortlichen bp zu initialisieren, indem der Wert von sp in bp verschoben wird. Danach wird der sp mit **SUB** verringert um Speicher auf dem Stack für lokale Variablen zu allozieren.
5. Am Ende der Subroutine wird dann der sog. **Function Epilogue** ausgeführt. Als erstes wird der Wert von bp wieder in sp verschoben, wodurch der allozierte Speicherplatz auf dem Stack wieder freigegeben wird. Danach werden die im Epilogue gepushten Werte, in sinngemäß umgekehrter Reihenfolge in die jeweiligen Register gepoppt. Der Rückgabewert wird in das **A-Register** verschoben und schlussendlich mit dem **RET** Befehl das Unterprogramm wieder verlassen, indem zur gespeicherten Rücksprungadresse gesprungen wird.

64-Bit

Ein Unterprogrammaufruf unter 64-Bit läuft wie oben beschrieben ab, mit einer Änderung. Die Argumente werden hier mithilfe von Registern übergeben. Das erste Argument wird über **RDI** und die weiteren mithilfe von **RSI**, **RDX**, **RCX**, **r8**, **r9** übergeben. Sollten mehr Argumente übergeben werden müssen, dann werden diese wie oben beschrieben mittels Stack übergeben.

```
1 // ##### PROLOGUE #####
2 PUSH    rbp           // save rbp on the stack
3 MOV     rbp, rsp
4 SUB     rsp, 0x10     // allocate 16 byte on the stack
5
6 ...
7
8 // ##### EPILOGUE #####
9 MOV     rsp, rbp      // deallocate memory
10 POP     rbp          // restore old base pointer
11 RET                      // return to calling function
```

Abbildung 30: Typischer Funktionsprolog und -epilog

Programmbeispiel

Ein Beispiel für einen Funktionsaufruf mit Argumentübergabe ist folgende Stelle.

```
mov     edi, eax           Parameter(i:= 5)
call    0x401112 <f>
mov     DWORD PTR [rbp-0x8], eax Rückgabewert
```

Abbildung 31: 0. Ausgabe von gdb-uebung-3.c

Hier wird, bevor `f()` aufgerufen wird, der Wert von `i` (5), welcher sich in `EAX` befindet mittels `MOV` nach `EDI` verschoben. In `f()` selber wird dann als erstes Platz für lokale Variablen und Parameter geschaffen, indem Speicher alloziert wird.

```
0x401116 <f+4>          sub     rsp, 0x10
0x40111a <f+8>          mov     DWORD PTR [rbp-0x4], edi
```

Abbildung 32: 2. Ausgabe von gdb-uebung-3.c

Danach wird das übergebene Argument von `EDI`, für die weitere Nutzung, nach `rbp-0x4` verschoben. Dies spiegelt einen typischen 64-Bit Funktionsaufruf unter x86 wieder.

5 Zu Aufgabe 4

5.1 a) Analysieren Sie den in der Datei enthaltenen Source Code

5.1.1 Analyse

Das gegebene Programm soll die Summe mehrere Fließkommazahlen mittels einer Schleife berechnen. Dabei wird ebenfalls ein Float als Laufvariable *i* in der Schleife verwendet. Die Schleife zählt dabei, in jeder Iteration, zur Summe den aktuellen Wert von *i* hinzu. Danach wird ein Zähler namens *anzahl* inkrementiert und zu *i* der Wert 0.01 hinzugezählt. Die Schleife wird solange ausgeführt, wie *i* den Wert 1000.03 nicht übersteigt.

```
1 #include <stdio.h>
2
3 int main() {
4     int anzahl;
5     double summe;
6     double i;
7
8     summe = 0, anzahl = 0;
9     for (i = 1000; i <= 1000.03; i += .01) {
10         summe += i;
11         anzahl++;
12     }
13     printf("Summe: %f, Anzahl: %d\n", summe, anzahl);
14
15     return 0;
16 }
```

Am Ende wird dann noch der in Zeile 13 zu sehende, formatierte String ausgegeben. Aufgrund der Benutzung einer Fließkommazahl, ist eine gewisse Ungenauigkeit mit einzurechnen, da Fließkommazahlen nur annäherungsweise in Computern dargestellt werden können.

5.1.2 Floating Point Numbers

Grundlegend unterscheidet man in **32-Bit Floating point Zahlen** (float) und **64-Bit (double precision) Floating Point Zahlen** (double). Der **IEEE 754 32-bit floating-point standard** ist z.B. wie folgt definiert.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s	biased exponent								fraction																						

Abbildung 33: 0. Ausgabe von gdb-uebung-4.c

Dabei wird die Zahl intern in drei Teilbereichen dargestellt. **S** gibt das Vorzeichen an, d.h. 0 bedeutet positiv und 1 negativ.

Die Zahl selber wird normalisiert abgespeichert, d.h. die Fließkommazahl wird so verschoben, dass nur eine 1 vor dem Komma steht. Damit man die Zahl später wieder rekonstruieren kann, wird der Exponent um den verschoben wurde plus 127 mit abgespeichert. Man spricht dann auch vom **biased exponent**, da zu ihm ein Bias hinzugefügt wurde. Aufgrund der Tatsache, dass durch diese Normalisierung immer eine einzige 1 vor dem Komma steht, wird diese in der internen Darstellung nicht mit angegeben, dadurch steht ein weiteres Bit für die **fraction** zur Verfügung. Ein floating point Zahl f kann demnach wie folgt angegeben werden.

$$f = (-1)^S * 1.fraction * 2^{E-127}$$

Abbildung 34: IEEE 32-bit floating point

Beispiel

$$-0.125_{10} = -0.001_2 = (-1)^1 * 1.0 * 2^{-3} = (-1)^1 * 1.0 * 2^{-3+127-127} = (-1)^1 * 1.0 * 2^{124-127}$$

Abbildung 35: Beispiel anhand von -0.125

Daraus ergibt sich:

S	E	fraction
1	01111100	000000000000000000000000

5.2 b) Kompilieren Sie den C Code und führen Sie das Programm aus

5.2.1 Kompilieren

Das Kompilieren des Quellcodes innerhalb von `gdb-uebung-4.c` kann mittels des folgenden Befehls auf der Kommandozeile ausgeführt werden.

```
1 $ gcc gdb-uebung-4.c -o gdb-uebung-4
```

Der in diesem Fall genutzte Compiler heißt **gcc** (GNU compiler collection). Dabei ist `gdb-uebung-3.c` der Name der Quelldatei. Mittels **-o gdb-uebung-3** wird der gewünschte Name, der zu erstellenden Programmdatei, angegeben.

5.2.2 Ausführen

Das Programm kann nun auf der Kommandozeile ausgeführt werden.

```

1 $ ./gdb-uebung-4
2 Summe: 3000.030029, Anzahl: 3

```

Das Ergebnis überrascht jedoch zuerst. Eigentlich hätte es zu einem weiteren Durchlaufen der Schleife kommen sollen und auch das Ergebnis weicht um 0.000029 vom erwarteten Zwischenergebnis 3000.03 ab.

Tabellarisiert haben die Variablen zum n'ten Schleifendurchlauf folgende Zustände.

n	summe	i	anzahl
0	0	1000	0
1	1000.0	1000.01	1
2	2000.01	1000.02	2
3	3000.03	1000.03	3
4	4000.06	1000.04	4

5.3 c) Analysieren Sie das Programm mit dem GDB

5.3.1 Berechnet die Schleife das korrekte Ergebnis?

Nein, wie oben beschrieben weicht das Ergebnis von dem erwarteten Ergebnis ab.

5.3.2 Welche Werte nehmen die Variablen bei der Ausführung des Programms an?

Das das Programm mit Floatingpoint-Zahlen rechnet gilt es als erstes einige Befehle zu erläutern, die das Programm in diesem Fall nutzt und die nur auf diesem Datentyp operieren.

Floating Point Instruktionen

xmm0-15 - Floating Point Register

movss <dest>, <src> - Kopiert einen 32-Bit Quell-Operanden in einen 32-Bit Ziel-Operanden.

movsd <dest>, <src> - Kopiert einen 64-Bit Quell-Operanden in einen 64-Bit Ziel-Operanden.

cvtss2sd <dest>, <src> - Wandelt einen 32-Bit FP Quell-Operanden in einen 64-Bit FP Zahl um und kopiert sie nach <dest>.

cvtss2sd <dest>, <src> - Wandelt einen 64-Bit FP Quell-Operanden in einen 32-Bit FP Zahl um und kopiert sie nach <dest>.

addss <dest>, <src> - <32-Bit dest> := <32-Bit dest> + <32-Bit src>

addsd <dest>, <src> - <64-Bit dest> := <64-Bit dest> + <32-Bit src>

ucomisd <RXsrc>, <src> - Vergleich der beiden 64-Bit FP Register <RXsrc> und <src>. Ein Operand kann auch eine Speicheradresse sein.

Analyse

Um die Werte über die verschiedenen Schleifendurchläufe hinweg verfolgen zu können, muss als ersten ein Breakpoint an den Kopf der Schleife gesetzt werden. Dafür gilt es den Kopf der Schleife mittels **GDB** zu ermitteln.

Da sich der komplette Programmcode in **main()** befindet, wird diese Funktion auch disassembliert.

```
1  gef> disass main
2
3      <+0>:    push    rbp
4      <+1>:    mov     rbp, rsp
5      <+4>:    sub     rsp, 0x10
6      <+8>:    pxor    xmm0, xmm0
7      <+12>:   movss   DWORD PTR [rbp-0x8], xmm0
8      <+17>:   mov     DWORD PTR [rbp-0xc], 0x0
9      <+24>:   movss   xmm0, DWORD PTR [rip+0x116]          # 0x780
10     <+32>:   movss   DWORD PTR [rbp-0x4], xmm0
11     <+37>:   jmp     0x69e <main+84>
12     <+39>:   movss   xmm0, DWORD PTR [rbp-0x8]
13     <+44>:   addss   xmm0, DWORD PTR [rbp-0x4]
14     <+49>:   movss   DWORD PTR [rbp-0x8], xmm0
15     <+54>:   add     DWORD PTR [rbp-0xc], 0x1
16     <+58>:   cvtss2sd xmm0, DWORD PTR [rbp-0x4]
17     <+63>:   movsd   xmm1, QWORD PTR [rip+0xf7]          # 0x788
18     <+71>:   addsd   xmm0, xmm1
19     <+75>:   cvtsd2ss xmm2, xmm0
20     <+79>:   movss   DWORD PTR [rbp-0x4], xmm2
21     <+84>:   cvtss2sd xmm0, DWORD PTR [rbp-0x4]
22     <+89>:   movsd   xmm1, QWORD PTR [rip+0xe5]          # 0x790
23     <+97>:   ucomisd  xmm1, xmm0
24     <+101>:  jae     0x671 <main+39>
25     <+103>:  cvtss2sd xmm0, DWORD PTR [rbp-0x8]
26     <+108>:  mov     eax, DWORD PTR [rbp-0xc]
27     <+111>:  mov     esi, eax
28     <+113>:  lea     rdi, [rip+0xa6]          # 0x768
29     <+120>:  mov     eax, 0x1
30     <+125>:  call    0x520 <printf@plt>
31     <+130>:  mov     eax, 0x0
32     <+135>:  leave
33     <+136>:  ret
```

Danach muss die Abfolge von Instruktionen, die nun erscheint, auf Instruktionsblöcke hin abgesucht werden, die auf genau so einen Kopf schließen lassen. Gleichzeitig können einzelne Speicheradressen und Register, durch Analyse genau dieser Adressen, den passenden Variablen im Quelltext zugeordnet werden. Dies hilft, den Code besser verstehen und vergleichen zu können.

Bei `<main + 12>` wird der 32-Bit FP Wert 0 aus Register **xmm0** an Adress **rbp-0x8** copiert. Da es nur ein FP Variable im Quelltext gibt, die mit 0 initialisiert wird, fällt

die Zuordnung in diesem Fall leicht. Es muss sich bei der Speicheradress `rbp-0x8` um die Variable **summe** handeln.

```
1 <+12>:  movss  DWORD PTR [rbp-0x8],xmm0      ; summe := 0
```

Ähnlich verhält es sich hier mit der darauf folgenden Zeile. Da bis auf eine Ausnahme mit FP Zahlen gerechnet wird, muss es sich bei Instruktion `<main + 17>` um die Initialisierung der Integer-Variable **anzahl** mit 0 handeln.

```
1 <+17>:  mov     DWORD PTR [rbp-0xc],0x0      ; anzahl := 0
```

Daraufhin wird der Wert an Adresse **rip+0x116** in `xmm0` kopiert und danach gleich wieder an Adresse **rbp-0x4** geschrieben. Da mit `rip+0x116` die Adresse relativ zum Instruktions-Pointer angegeben wird, kann davon ausgegangen werden, dass der Wert in der schreibgeschützten **Text-Section** befindet. Damit muss es sich hierbei um eine Konstante handeln. Die Konstanten, die in dem gegebenen Programm vorkommen sind 1000, 1000.03 und 0.01.

```
1 <+24>:  movss  xmm0,DWORD PTR [rip+0x116]
2 <+32>:  movss  DWORD PTR [rbp-0x4],xmm0
```

Um den exakten Wert zu erhalten und damit eine Zuordnung vornehmen zu können gilt es die Speicheradresse zu inspizieren. Dabei wird die **f Option** gewählt, da es sich mit hoher Wahrscheinlichkeit um ein FP Zahl an der zu inspizierenden Adresse handelt.

```
1 gef> x/f 0x780
2 0x780:  1000
```

Und tatsächlich scheint es sich bei den Instruktionen um die Initialisierung von **i** mit dem Wert 1000.0 zu handeln.

Zuordnung

- `anzahl = rbp-0xc` (32-Bit int)
- `summe = rbp-0x8` (32-Bit float)
- `i = rbp-0x4` (32-Bit float)

Mit dieser Zuordnung ist nun klar, welche Adressen im Speicher (Variablen) es zu beobachten gilt.

Nun gilt es noch den Kopf der Schleif ausfindig zu machen. Die Instruktion an Stelle `<main + 37>` springt an Adresse `<main + 84>`. Hier wird als erstes die 32-Bit FP Zahl **i** in eine 64-Bit FP Zahl umgewandelt und nach `xmm0` kopiert. Danach wird die Konstante 1000.03 in `xmm1` kopiert. `xmm0` und `xmm1` werden daraufhin verglichen und falls `1000.03 >= i` ist, wird an die Stelle `<main + 39>` gesprungen.

```
1 <+84>:  cvtss2sd xmm0,DWORD PTR [rbp-0x4]      ; xmm0 := i
2 <+89>:  movsd   xmm1,QWORD PTR [rip+0xe5]      ; xmm1 := 1000.03
3 <+97>:  ucomisd  xmm1,xmm0                      ; while 1000.03 >= i:
4 <+101>:  jae     0x671 <main+39>                  ; ...
```

Damit ist der Kopf der Schleife Gefunden. Nun kann ein Breakpoint an Stelle <main+84>, d.h. Adresse 0x69e gesetzt werden. Dazu wird als erstes ein Breakpoint an den Anfang von main() gesetzt und das Programm gestartet.

```
1 gef> break main
2 gef> r
```

Der Prozess wird nun gleich nach dem Function Prologue ?? angehalten.

```
0x55555555464a <main+0>      push    rbp
0x55555555464b <main+1>      mov     rbp, rsp
0x55555555464e <main+4>      sub     rsp, 0x10
```

Abbildung 36: 0. Ausgabe von gdb-uebung-4.c

Mit **disass main** können nun noch einmal alle Instruktionen angezeigt werden. Diesmal enthält jede Spalte jedoch zusätzlich noch die jeweilige Adresse im Speicher. Diese können wir nun Nutzen um den Breakpoint an den Schleifenkopf zu setzen. Außerdem wird noch ein Breakpoint hinter die **jae** Instruktion gesetzt, damit sich das Programm nicht sofort, nach dem letzten Schleifendurchlauf beendet. Mit **continue** wird der Prozess danach bis zum nächsten Breakpoint fortgesetzt.

```
1 gef> break *0x000055555555469e      ; address can differ
2 gef> break *0x00005555555546b1
3 gef> c
```

```
0x55555555469e <main+84>      cvtss2sd xmm0, DWORD PTR [rbp-0x4]
```

Abbildung 37: 1. Ausgabe von gdb-uebung-4.c

Um die Werte der einzelnen Variablen abzufragen, kann wieder der **x** (examine) Befehl verwendet werden. Zum 0'ten Schleifendurchlauf halten die Variablen **anzahl**, **summe**, **i** folgende Werte.

```
1 gef> x/d $rbp-0xc
2 0x7fffffffef7d4: 0      ; anzahl
3 gef> x/f $rbp-0x8
4 0x7fffffffef7d8: 0      ; summe
5 gef> x/f $rbp-0x4
6 0x7fffffffef7dc: 1000    ; i
7 gef> c
```

Nach dem 1'ten Schleifendurchlauf halten die Variablen **anzahl**, **summe**, **i** folgende Werte. Hier fällt bei **i** auf, dass nicht wie erwartet exakt 0.01 zu **i** addiert wurde, sondern eine leicht höhere Zahl. D.h. die Addition der FP Zahl 0.01 zu **i** stellt einen Defekt in dem gegebenen Programm dar. Durch diesen wurde die Variable **i** infiziert.

```
1 gef> x/d $rbp-0xc
2 0x7fffffffef7d4: 1      ; anzahl
3 gef> x/f $rbp-0x8
```



```

4 0x7ffffffffffe7d8: 1000      ; summe
5 gef> x/f $rbp-0x4
6 0x7ffffffffffe7dc: 1000.01001  ; i
7 gef> c

```

Nach dem 2'ten Schleifendurchlauf halten die Variablen **anzahl**, **summe**, **i** folgende Werte. Auch hier verschlimmert sich die Infektion, d.h. **i** weicht noch mehr vom zu erwartenden Wert ab.

```

1 gef> x/d $rbp-0xc
2 0x7ffffffffffe7d4: 2      ; anzahl
3 gef> x/f $rbp-0x8
4 0x7ffffffffffe7d8: 2000.01001  ; summe
5 gef> x/f $rbp-0x4
6 0x7ffffffffffe7dc: 1000.02002  ; i
7 gef> c

```

Nach dem 3'ten Schleifendurchlauf halten die Variablen **anzahl**, **summe**, **i** folgende Werte.

```

1 gef> x/d $rbp-0xc
2 0x7ffffffffffe7d4: 3      ; anzahl
3 gef> x/f $rbp-0x8
4 0x7ffffffffffe7d8: 3000.03003  ; summe
5 gef> x/f $rbp-0x4
6 0x7ffffffffffe7dc: 1000.03003  ; i
7 gef> c

```

Nach dem 3'ten Schleifendurchlauf kommt es zu einem Abbruch der Schleife, da die Bedingung nicht mehr erfüllt ist, d.h. $i = 1000.03003 \not\leq 1000.03$. Was wiederum zu dem Fehlverhalten des Programms führt.

5.3.3 Warum wird das falsche Ergebnis berechnet?

Computer haben ein großes Problem, wenn es um die Darstellung von Fließkommazahlen geht. Dazu kann man sich folgende Frage Stellen. Man nehme zwei Zahlen, $n = 0.0000001$ und $m = 0.0000002$. **wie viele Zahlen liegen zwischen n und m?** Die Antwort ist weniger überraschend, nämlich **unendlich viele**. Dies gilt natürlich für zwei beliebige Reelle Zahlen n und m , zeigt aber das Problem der Endlichkeit von Computern.

Das Programm nutzt z.B. 32-Bit Floating Point Zahlen. Wenn man vom IEEE 754 FP Standard ausgeht, bedeutet das, dass 23 Bit für die Darstellung einer Kommazahl zur Verfügung stehen, plus 9-Bit für Exponent und Vorzeichen. Damit können natürlich nicht unendlich viele reelle Zahlen dargestellt werden. Zahlen werden **näherungsweise** dargestellt, d.h. anstelle der eigentlichen Zahl wird die nächstmögliche Zahl verwendet, die darstellbar ist. Dabei ist die Wertigkeit der einzelnen Stellen zu berücksichtigen.

Wertigkeit der Stellen zur Basis 2

...	2^2	2^1	2^0	2^{-1}	2^{-2}	...
...	4	2	1	0.5	0.25	...

Um die näherungsweise Darstellung zu illustrieren kann von einem vereinfachten Beispiel mit Festkommazahl ausgegangen werden.

Beispiel

- Sei R 8-Bit Register mit 1 Vorkomma- und 7 Nachkommastellen ohne Vorzeichen.
- D.h. $R = V.NNNNNNN$
- Speichere 0.1 binär in Register R.
- dann ergibt sich die näherungsweise Darstellung $0.0001100_2 = 0.09375_{10}$
- d.h. die Zahl die sich darstellen lässt und am nächsten an 0.1 liegt.

Das falsche Ergebnis wird berechnet, da sich 0.01 nicht exakt mit 32-Bit darstellen lässt. Anstelle von 0.01 wird der nächstmögliche darstellbare Wert verwendet. Dieser ist jedoch um 0.00001 größer als 0.01, wodurch die Schleife zu früh terminiert, was wiederum zu dem auftretenden Fehler führt.

5.3.4 Erstellen Sie ein modifiziertes Programm, welches ein korrektes Ergebnis liefert.

In Schleifen mit FP Zahlen zu hantieren ist so gut wie immer eine schlechte Idee und sollte um jeden Preis vermieden werden. Dieses kleine Programm unterstützt diese These weiter. Trotzdem stellt die Verwendung von 64-Bit Double Precision FP Zahlen (double) anstelle von float, einen hinreichenden Bug-Fix dar.

Code

```
1 #include <stdio.h>
2
3 int main() {
4     int anzahl;
5     double summe;    // BUGFIX
6     double i;        // BUGFIX
7
8     summe = 0, anzahl = 0;
9     for (i = 1000; i <= 1000.03; i += .01) {
10         summe += i;
11         anzahl++;
12     }
13     printf("Summe: %f, Anzahl: %d\n", summe, anzahl);
14
15     return 0;
16 }
```

Ergebnis

```
1 $ ./gdb-uebung-4
2 Summe: 4000.060000, Anzahl: 4
```

6 Zu Aufgabe 5: Binäre Suche

6.1 Kurze Erklärung von binary-search

Der binary-search Algorithmus wird verwendet um ein Element in einer sortierten Liste zu finden. Binary search hat den Vorteil gegenüber Iterativer Suche, dass es $O(\log N)$ Laufzeit statt $O(N)$ hat. Der Algorithmus teilt das Feld in zwei Teile an. Danach wird geschaut, ob das in welchen der beiden Felder das zu suchende Element ist oder ob es genau in der Mitte ist. Das Einteilen ist möglich, da das Feld sortiert ist. Dies wird rekursiv auf dem neuen Teil des Feldes ausgeführt, bis das Feld Element gefunden wurde. Spätestens, wenn es nur noch das Teilfeld nur noch gröÙe 1 hat.

6.2 a) Analyse des C-Codes

gdb-uebung-5.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define MAX 8
5  int array[MAX] = {1,4,12,18,26,31,40,42};
6  int rekursionstiefe = 0;
7
8  int binarysearch(int zahl, int links, int rechts) {
9      rekursionstiefe++;
10
11      int mitte = (links + rechts) / 2;
12      if (array[mitte] == zahl)
13          return mitte;
14      if (links == rechts)
15          return -1;
16      if (array[mitte] > zahl)
17          return binarysearch(zahl, links, mitte);
18      else
19          return binarysearch(zahl, mitte, rechts);
20 }
21
22 int main(int argc, char *argv[]) {
23     int zahl, position, i;
24     if(argc < 2) {
25         printf("Benutzung: %s <zu suchende Zahl>\n", argv[0]);
26         return 1;
27     }
28     zahl = atoi(argv[1]);
29
30     for (i = 0; i < MAX; i++) {
31         printf("%4d", array[i]);
32     }
```

```

33     position = binarysearch(zahl, 0, 7);
34     if (position >= 0) {
35         printf("\nGesuchte Zahl %d an Arrayposition %d\n",
36             zahl, position);
37     }
38     else {
39         printf("\nZahl %d nicht gefunden\n", zahl);
40     }
41     return 0;
42 }

```

Das Programm besitzt drei globale Variablen. Die Größe `MAX` des arrays, ein sortiertes `array` mit bereits initialisierten Werten und ein `int`, der die `rekursionstiefe` speichert. Zusätzlich zur `main` Funktion, gibt es die `binarysearch` Funktion. Diese ist die Funktion, die durch rekursive Aufrufe von sich selbst den oben beschriebenen Algorithmus durchführt.

In der `main` Funktion wird eine Schnittstelle zum benutzer geboten. Dieser kann beim Aufruf des Programms eine Zahl als Argument mit angeben. Diese Zahl wird dann gesucht und der Index in einem formatierten String in der Konsole ausgegeben. Wurde die Zahl nicht gefunden, wird dies in einer alternativen Ausgabe ausgegeben.

6.3 b) Ausführen des Programms

Wird das Programm nun kompiliert und probeweise mit Eingaben ausgeführt erhalten wir zum Beispiel folgende Ausgaben.

```

$ gcc -o gdb-uebung-5 gdb-uebung-5.c
$ ./gdb-uebung-5 1
  1   4  12  18  26  31  40  42
Rekursionstiefe: 1
Rekursionstiefe: 2
Rekursionstiefe: 3
Gesuchte Zahl 1 an Arrayposition 0
$ ./gdb-uebung-5 4
  1   4  12  18  26  31  40  42
Rekursionstiefe: 1
Rekursionstiefe: 2
Gesuchte Zahl 4 an Arrayposition 1
$ ./gdb-uebung-5 26
  1   4  12  18  26  31  40  42
Rekursionstiefe: 1
Rekursionstiefe: 2
Rekursionstiefe: 3
Gesuchte Zahl 26 an Arrayposition 4

```

Abbildung 38: 1. Ausgabe von `gdb-uebung-5.c`

Durch nachzählen kann auch schnell überprüft werden, dass die Indizes stimmen.

```

$ gcc -o gdb-uebung-5 gdb-uebung-5.c; ./gdb-uebung-5 27
  1  4 12 18 26 31 40 42
Rekursionstiefe: 1
Rekursionstiefe: 2
Rekursionstiefe: 3
Rekursionstiefe: 4
Rekursionstiefe: 5
Rekursionstiefe: 6
Rekursionstiefe: 7
Rekursionstiefe: 8
Rekursionstiefe: 9
Rekursionstiefe: 10
...
Rekursionstiefe: 174521
Rekursionstiefe: 174522
Rekursionstiefe: 174523
Rekursionstiefe: 174524
Rekursionstiefe: 174525
Rekursionstiefe: 174526
Rekursionstiefe: 174527
Rekursionstiefe: 174528
Rekursionstiefe: 174529
[1] 4366 segmentation fault (core dumped) ./gdb-uebung-5 27

```

Abbildung 39: 2. Ausgabe von gdb-uebung-5.c

Wird das Programm nun allerdings mit einem Wert aufgerufen, der nicht im Array steht, terminiert die Rekursion nie. Das Programm bricht mit einem Segmentation fault ab.

6.3.1 Segmentation fault (core dumped)

Segmentation fault heißt, der Prozess versucht auf Speicher zuzugreifen, auf den er kein Zugriff hat. In unserem Fall ist die Ursache ein **Stackoverflow**. Wie in vorherigen Aufgaben schon festgestellt, wird bei jedem Funktionsaufruf ein Stackframe auf dem Stack angelegt. Dieser Speicher wird erst wieder frei, wenn die Funktion fertig ist. Das heißt bei jedem rekursiven Funktionsaufruf wird ein neuer Stackframe auf dem dem Stack gespeichert. Der verfügbare Speicher des Stacks ist begrenzt. Ist dieser aufgebraucht (Grenze vom Stack ist an der Grenze des Heaps), und es wird versucht was auf dem Stack abzulegen, kommt es zum **Stackoverflow** was zum **segmentation fault** führt und den Prozess beendet. Dies verhindert, dass der Stack speicher im Heap überschreibt.

```

$ gcc -o gdb-uebung-5 gdb-uebung-5.c; ./gdb-uebung-5 42
1 4 12 18 26 31 40 42
Rekursionstiefe: 1
Rekursionstiefe: 2
Rekursionstiefe: 3
Rekursionstiefe: 4
Rekursionstiefe: 5
Rekursionstiefe: 6
Rekursionstiefe: 7
Rekursionstiefe: 8
Rekursionstiefe: 9
Rekursionstiefe: 10
...
Rekursionstiefe: 174521
Rekursionstiefe: 174522
Rekursionstiefe: 174523
Rekursionstiefe: 174524
Rekursionstiefe: 174525
Rekursionstiefe: 174526
[1] 4153 segmentation fault (core dumped) ./gdb-uebung-5 42

```

Abbildung 40: 3. Ausgabe von gdb-uebung-5.c

Bei einer Eingabe anderen tritt das selbe Problem auf. Die Rekursion terminiert wieder nicht. Diesmal ist die gesuchte Zahl allerdings im Array. Das heißt wir haben einen weiteren Defekt im Programm. Die Besonderheit bei dieser Zahl ist, dass sie an der letzten Stelle im Array steht. Vielleicht kann man hier mit der Fehlersuche anfangen.

6.3.2 Schlussfolgerung

- Es gab Eingaben bei denen die Ausgabe korrekt war. All diese Eingaben kamen im Array vor.
- Wenn die Zahl nicht im Array vorkommt kann es zu einem Fehler kommen. Es muss noch getestet werden, ob dieser Fehler bei jeder Zahl auftritt, die nicht im Array steht.
- Es kam zu einem Fehler obwohl die eingegebene Zahl im Array steht.
- -> Um den Fehler zu finden, kann man das Programm debuggen um den Defekt zu finden.

6.4 c) Ausführen des Programms in gdb

Es soll nun der Fehler durch debuggen des Programms in gdb gefunden werden. Hierfür wird das Programm mit debug Informationen kompiliert und in gdb gestartet.

```

$ gcc -g gdb-uebung-5.c -o gdb-uebung-5
$ gdb gdb-uebung-5

```

Abbildung 41: 1. Ausgabe von gdb-uebung-5.c

Als erstes wird das Programm mit der Eingabe 27 getestet. Diese ist nicht im Array vorhanden und es sollte -1 zurück gegeben werden. Stattdessen terminiert die Rekursion aber nicht in in der vorherigen Aufgabe festgestellt wurde.

```
gef> b main
Breakpoint 1 at 0x741: file gdb-uebung-5.c, line 24.
gef> b binarysearch
Breakpoint 2 at 0x69b: file gdb-uebung-5.c, line 9.
gef> r 27
```

Abbildung 42: 2. Ausgabe von gdb-uebung-5.c

Als erstes wird anhand der Werte die in `links`, `rechts`, `mitte` stehen, versucht den Fehler zu finden. Der Grund ist, das die Rekursion beenden müsste, wenn `links == rechts` ist. Dieser Fall tritt nicht ein also wird geschaut was statt dessen eintritt. Hierfür wird eine Tabelle erstellt, für die in jeder Rekursionsstufe die Werte der Variablen eingetragen werden (nach der berechnung von `mitte`). Um die Werte schnell rauslesen zu können, wird das Programm bis nach der berechnung mit `n (next)` ausgeführt. Hier wird ein Breakpoint gesetzt und es kann mit `c (continue)` bis zu diesem Breakpoint ausgeführt werden.

```
11      int mitte = (links + rechts) / 2;
      // zahl=0x1b, mitte=0x3
→ 12      if (array[mitte] == zahl)
13          return mitte;
14      if (links == rechts)
15          return -1;
16      if (array[mitte] > zahl)
17          return binarysearch(zahl, links, mitte);

[#0] Id 1, Name: "gdb-uebung-5", stopped, reason: SINGLE STEP

[#0] 0x555555546be → binarysearch(zahl=0x1b, links=0x0, rechts=0x7)
[#1] 0x555555547d3 → main(argc=0x2, argv=0x7ffffffdda8)

12      if (array[mitte] == zahl)
gef> b *$rip
Breakpoint 3 at 0x555555546be: file gdb-uebung-5.c, line 12.
```

Abbildung 43: 3. Ausgabe von gdb-uebung-5.c

gefzeigt jetzt die jeweiligen Werte der Variablen an den entsprechenden Stellen an. `mitte` direkt im source code über der nächsten Abfrage. `links` und `rechts` im trace wo die Funktionsaufrufe mit den jeweiligen Werten angezeigt werden. Der Wert `array[mitte]` kann händisch aus dem Array ausgelesen werden oder in gdb mit `x/x array + mitte`

Rekursionsstufe	links	mitte	rechts	array[mitte]	zahl
1	0x0	0x3	0x7	0x12	0x1b
2	0x3	0x5	0x7	0x1f	0x1b
3	0x3	0x4	0x5	0x1a	0x1b
4	0x4	0x4	0x5	0x1a	0x1b
5	0x4	0x4	0x5	0x1a	0x1b

Da zwischen die Werte im 4. und 5. Schritt gleich sind kann gestoppt werden. Ab hier werden sich die Funktion immer mit denselben Werten wieder selbst aufrufen. Das heißt die Rekursion terminiert nicht, da hier der Wert **rechts** immer eins größer ist als **links**. Aus der Tabelle kann jetzt auch leicht rausgelesen werden, welcher Zweig der if-Bedingung genommen wurde um die Funktion erneut aufzurufen. Da `array[mitte] < zahl` muss es der else Zweig sein. Es könnte also sein, das an dieser Stelle ein Fehler vorliegt. Schaut man sich den Aufruf an sieht man, das der jetzige **mitte** Wert der neue **links** Wert im nächsten Funktionsaufruf ist. Der **rechts** Wert wird übernommen. Also kann der Fehler daran liegen, wie die Grenzen übergeben werden. Schaut man sich die Berechnung für den Wert der **mitte** an stellt man fest, das hier ein int durch zwei geteilt wird und wider in einem int gespeichert wird. Das heißt bei geraden Werten, ist die Division Verlustfrei allerdings bei ungeraden Werten wird immer etwas abgeschnitten. Dadurch entsteht das Problem, es passieren kann, dass **links** nie gleich **rechts** ist. Z.B gilt im Moment folgende Rechnung: $(5 + 4)/2 = 4$ Um eine Terminierung zu erzeugen müsste allerdings gelten: $(5 + 4)/2 = 5$. Um diesen Wert um eins zu erhöhen, kann man im Aufruf der Funktion den Wert von **mitte** um eins erhöhen. Damit verhindert man diesen Sonderfall.

Zusammenfassung:

- Das Problem ist die Berechnung der **mitte**
- Dieses Problem wirkt sich beim Aufruf der Funktion so aus, dass bestimmte Werte nie erreicht werden
- Dadurch terminiert die Rekursion nicht
- Das Programm bricht ab, wenn versucht wird die Funktion aufzurufen es aber keinen Platz mehr auf dem Stack gibt

6.5 d) Beheben des Fehlers

Um den Fehler zu beheben, muss in Zeile 19, im else Teil also `mitte + 1` statt `mitte` übergeben werden.

gdb-uebung-5-bugfix.c

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define MAX 8
5  int array[MAX] = {1,4,12,18,26,31,40,42};
6  int rekursionstiefe = 0;
7
8  int binarysearch(int zahl, int links, int rechts) {
9      rekursionstiefe++;
10
11     int mitte = (links + rechts) / 2;
12     if (array[mitte] == zahl)
13         return mitte;
```

```

14     if (links == rechts)
15         return -1;
16     if (array[mitte] > zahl)
17         return binarysearch(zahl, links, mitte);
18     else
19         return binarysearch(zahl, mitte + 1, rechts);
20 }
21
22 int main(int argc, char *argv[]) {
23     int zahl, position, i;
24     if(argc < 2) {
25         printf("Benutzung: %s <zu suchende Zahl>\n", argv[0]);
26         return 1;
27     }
28     zahl = atoi(argv[1]);
29
30     for (i = 0; i < MAX; i++) {
31         printf("%4d", array[i]);
32     }
33     position = binarysearch(zahl, 0, 7);
34     if (position >= 0) {
35         printf("\nGesuchte Zahl %d an Arrayposition %d\n",
36             zahl, position);
37     }
38     else {
39         printf("\nZahl %d nicht gefunden\n", zahl);
40     }
41     return 0;
42 }

```