

Sichere Programmierung

Projekt 2

Julian Sobott
(76511)

David Sugar
(76050)

Inhaltsverzeichnis

1	Zu Aufgabe 1: C-for-Schleifen in Assembler	3
1.1	a) Analyse des C-Codes	3
1.2	b) Ausführen des Programms	3
1.3	c) Analyse des Assembler-Codes	4
1.4	d) Ausführen des Programms in gdb	5
2	Zu Aufgabe 2: Stackframes	10
2.1	a) Analyse des C-Codes	10
2.2	b) Ausführen des Programms	11
2.3	c) Stackframes	11
2.3.1	Reihenfolge der Funktionsaufrufe	11
2.3.2	Analyse der Stackframes	12
3	Zu Aufgabe 3	18
3.1	a) Analysieren Sie den in der Datei enthaltenen Source Code	18
3.1.1	Implementierung	18
3.1.2	Aufruf	19
3.2	b) Kompilieren Sie den C Code und führen Sie das Programm aus	19
3.2.1	Kompilieren	19
3.2.2	Ausführen	19
3.3	c) Führen Sie das Programm im GDB aus	20
3.3.1	Wie viele Stack Frames werden erzeugt?	20
3.3.2	Wie ist der Inhalt dieser Stack Frames?	20
3.3.3	Wie wird die Parameterübergabe in Assembler umgesetzt?	20
4	Zu Aufgabe 5: Binäre Suche	21
4.1	a) Analyse des C-Codes	21

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

1.1 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 Integern verwendet wird. In diesem Fall spielt die Formatierung aber keine Rolle.

1.2 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

1.3 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>).

1.4 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_dtors_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      : 0x000000000000402004 → 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 $i \leq 0x13$).

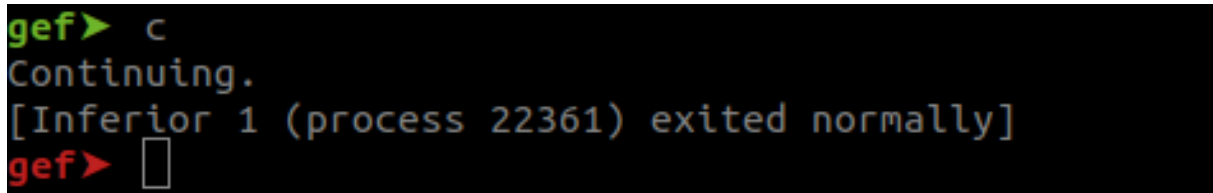
```

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.

2 Zu Aufgabe 2: Stackframes

2.1 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.

2.2 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.

2.3 c) Stackframes

2.3.1 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`).

2.3.2 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.

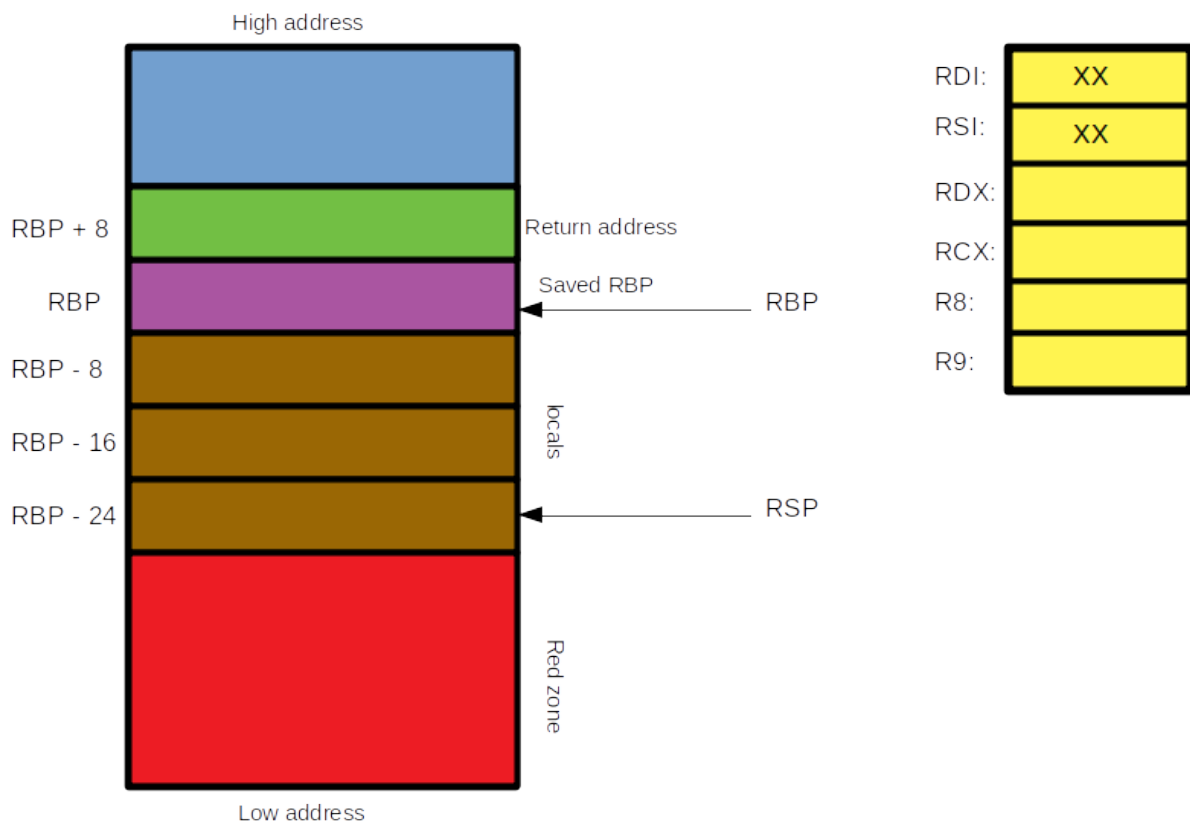


Abbildung 19: 11. Ausgabe von gdb-uebung-2.c

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.

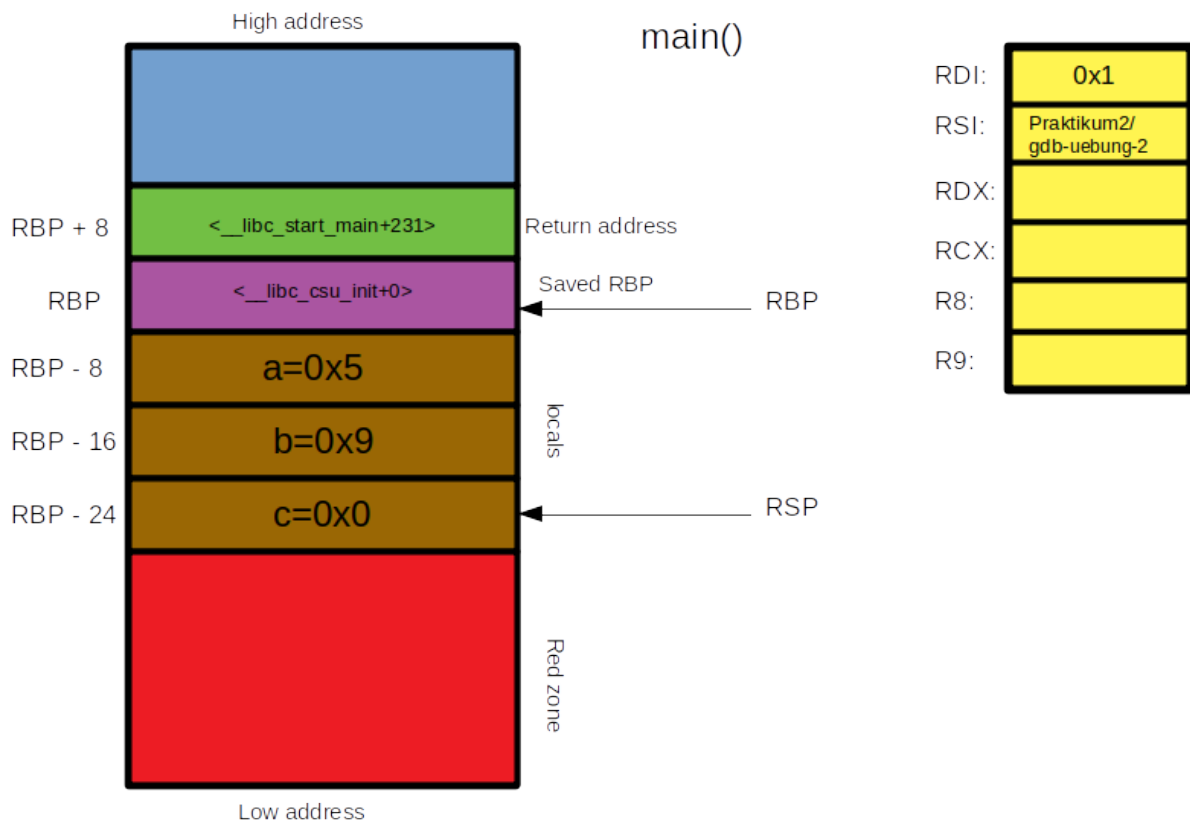


Abbildung 20: 0. Ausgabe von gdb-uebung-2.c

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.

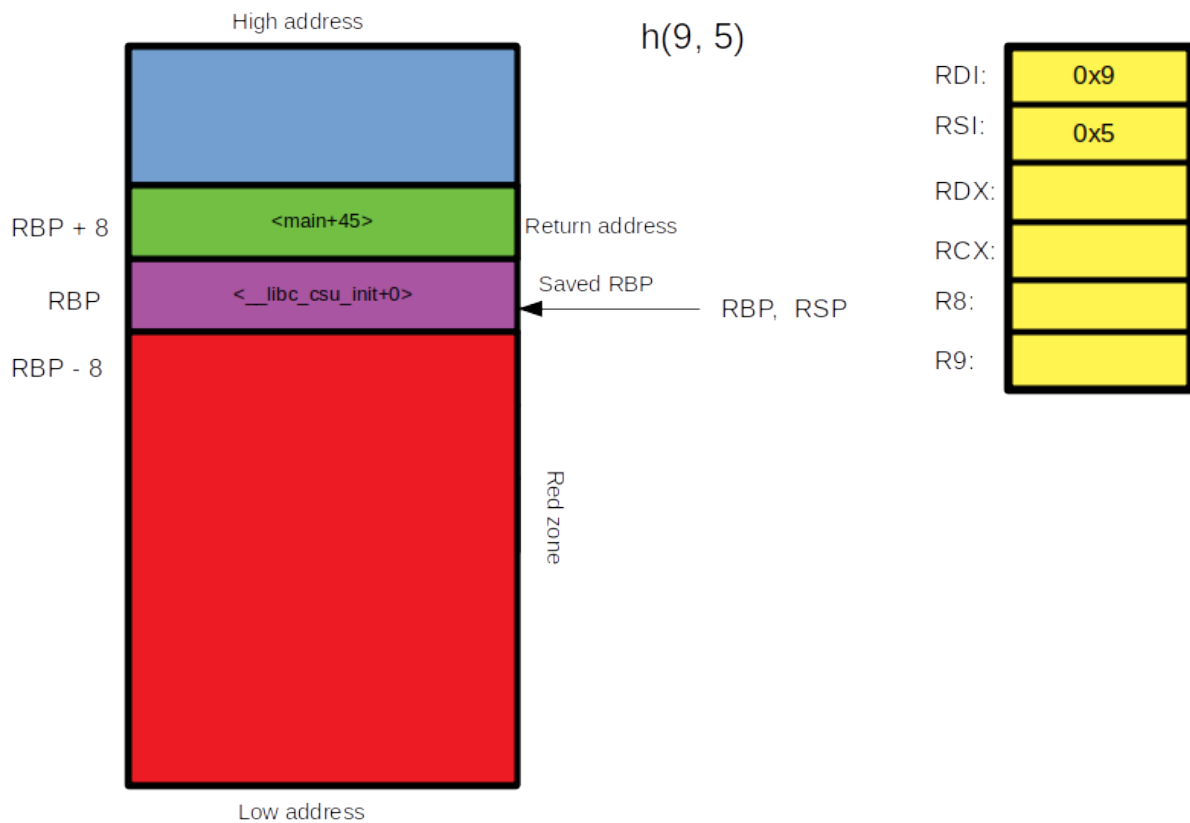


Abbildung 21: 1. Ausgabe von gdb-uebung-2.c

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.

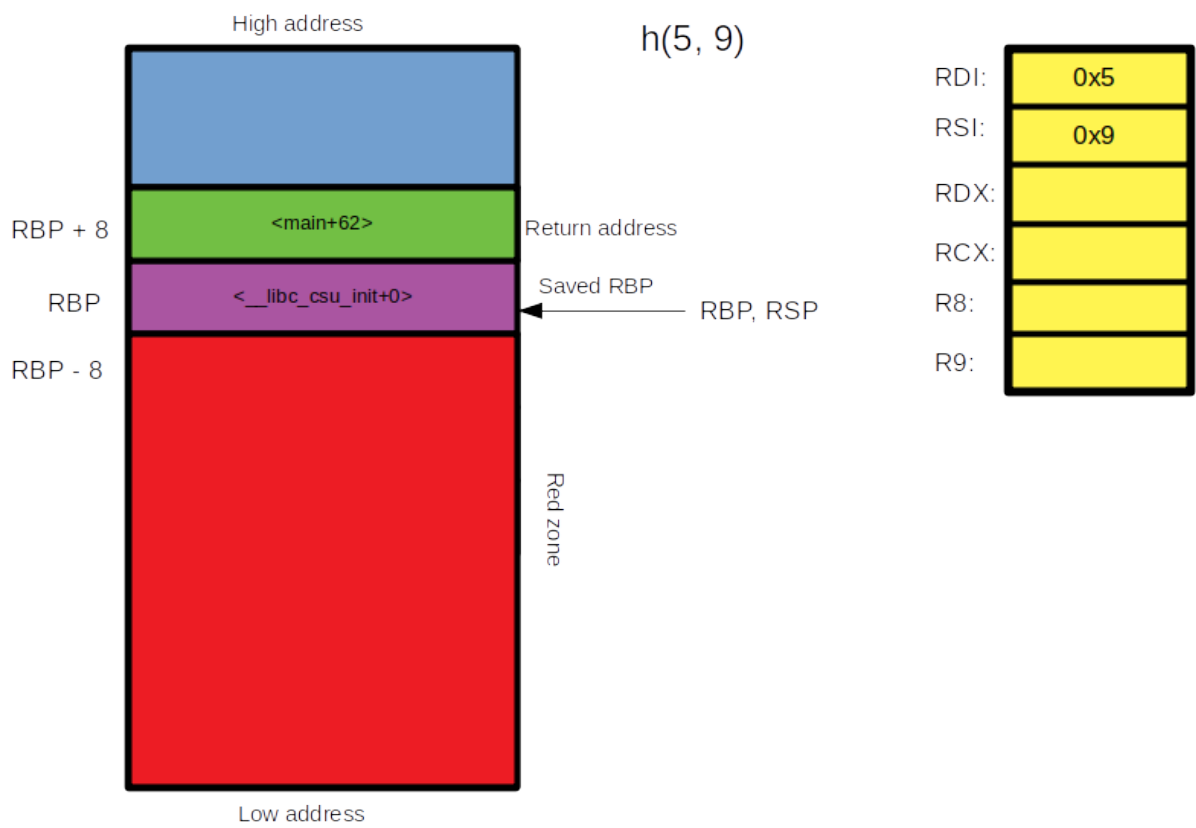


Abbildung 22: 2. Ausgabe von gdb-uebung-2.c

Zweiter Aufruf der Funktion `h` mit umgedrehten Paramtern un höherer Rücksprungadresse.

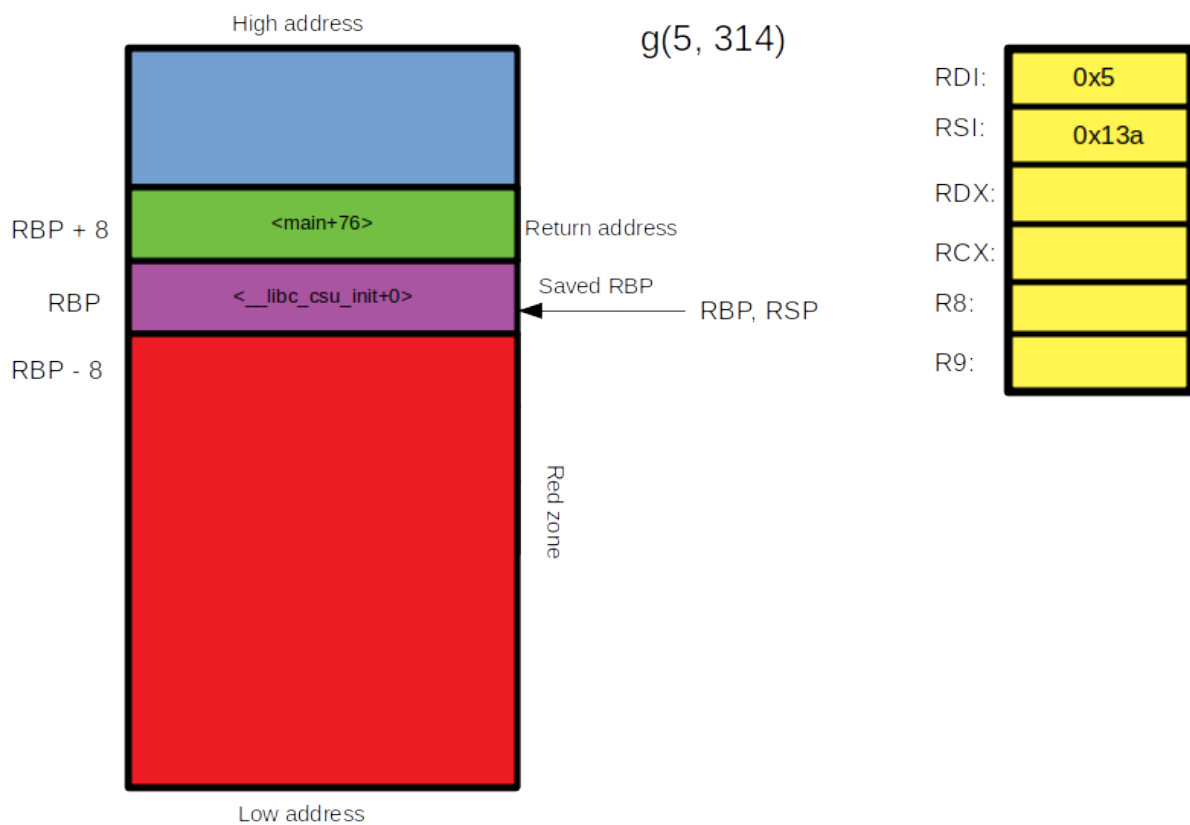


Abbildung 23: 3. Ausgabe von gdb-uebung-2.c

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

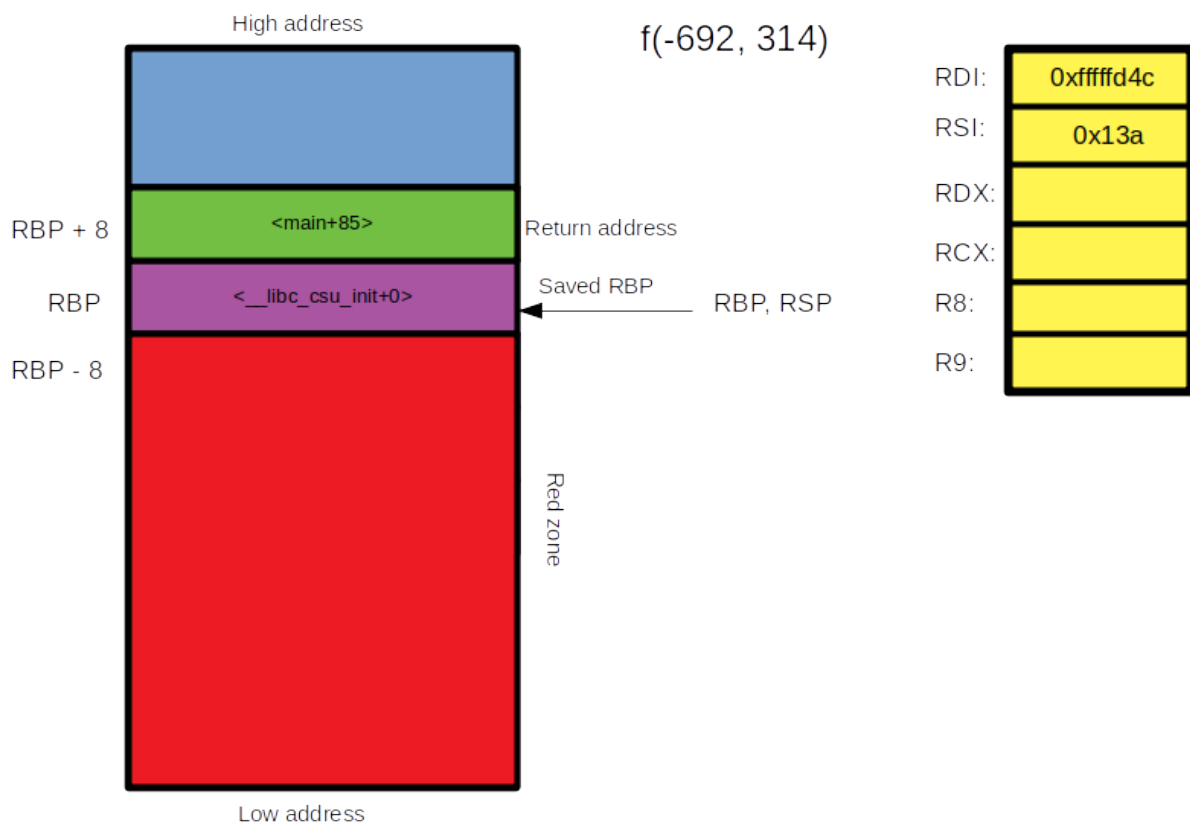


Abbildung 24: 4. Ausgabe von gdb-uebung-2.c

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

3 Zu Aufgabe 3

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

3.1.1 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, abgebildet in ?? 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.

3.1.2 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.

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

3.2.1 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.

3.2.2 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 scheint auf den ersten Blick also richtig implementiert. Um nachzuvollziehen, wie das Ergebnis zustande kommt, kann der folgende Graph betrachtet werden.

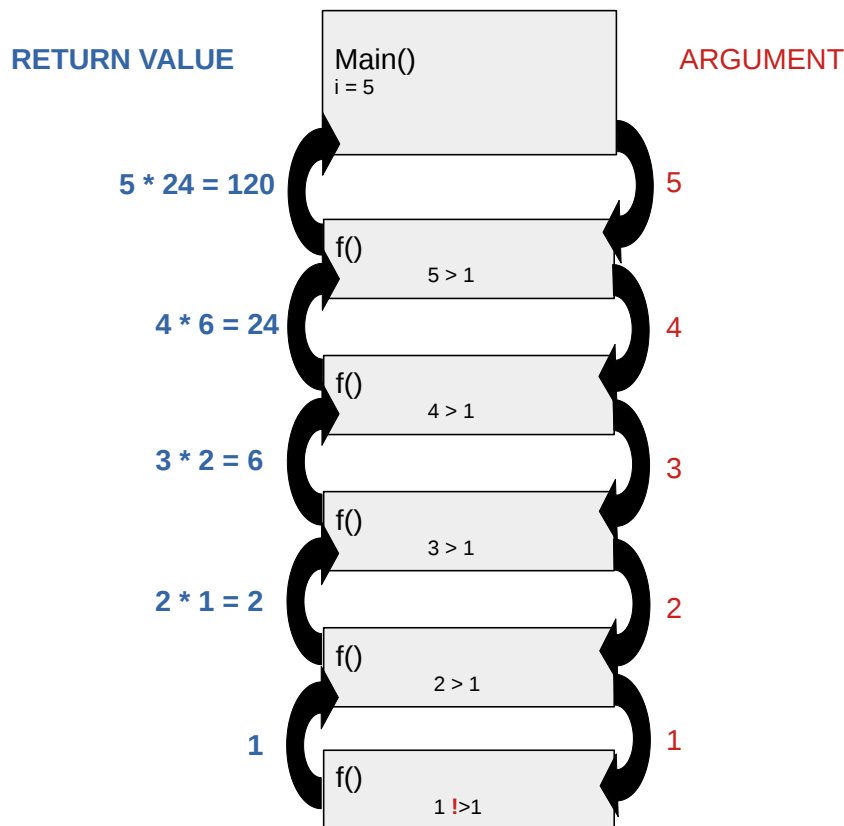


Abbildung 27: recursive call of `f()`

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.

3.3 c) Führen Sie das Programm im GDB aus

3.3.1 Wie viele Stack Frames werden erzeugt?

3.3.2 Wie ist der Inhalt dieser Stack Frames?

3.3.3 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 Rücksprungsadresse, die im nächsten Schritt auf dem Stack hinterlegt wird.
3. Nun wird mit **call** der Unterprogrammaufruf durchgeführt. Dabei wird die **Rücksprungsadresse** (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 wird als aller erstes der aktuelle Wert des **Base-Pointers (bp)** auf den Stack gepushed, sowie die **Callee-Saved Register** falls benötigt. Danach wird der Aktuelle Wert des **Stack-Pointers (sp)** genutzt um den für den derzeitigen **Stack-Frame** verantwortlichen bp zu initialisieren, indem 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. Hier werden falls vorhanden, die gesicherten Register gepoppt, zusammen ...

4 Zu Aufgabe 5: Binäre Suche

4.1 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    int mitte = (links + rechts) / 2;
11    printf("\nRekursionstiefe: %d", rekursionstiefe);
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", zahl, position);
36     }
37     else {
38         printf("\nZahl %d nicht gefunden\n", zahl);
39     }
40     return 0;
41 }

```

...

4.2 b) Ausführen des Programms

...

4.3 c) Ausführen des Programms in gdb

...

4.4 d) Beheben des Fehlers

...