ti23_assignment_12_Alabrsh_Panov_Zeitler

1a) Implemented function copy_asm in assembly by using given template → siehe src (copy_asm.s)

1b) Compiled C kernal copy_c from previous task using optimization flag -02. → siehe src (copy_c.o)
Disassembled compiler-generated machine code. → siehe src (copy_c.dis):

```
copy_c.o:     file format elf64-littleaarch64

SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 copy_c.c
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l    d  .note.GNU-stack 0000000000000000 .note.GNU-stack
0000000000000000 l    d  .eh_frame 0000000000000000 .eh_frame
0000000000000000 l    d  .comment 0000000000000000 .comment
0000000000000000 g     F .text  0000000000000024 copy_c
```

- als erstes wird Symboltabelle ausgegeben (wegen –syms)
- Aufbau der Tabelle: Wert des Symbols | gesetzte Flag (l-lokal, g-global) | Sektion, mit der Symbol verknüpft ist (Bsp: .data) | Alignment oder die Größe des Symbols | Symbolname

```
Disassembly of section .text:

0000000000000000 <copy_c>:
   0:   a9401807        ldp     x7, x6, [x0]
   4:   a9411005        ldp     x5, x4, [x0, #16]
   8:   a9420803        ldp     x3, x2, [x0, #32]
   c:   f9401800        ldr     x0, [x0, #48]
  10:   a9001827        stp     x7, x6, [x1]
  14:   a9011025        stp     x5, x4, [x1, #16]
  18:   a9020823        stp     x3, x2, [x1, #32]
  1c:   f9001820        str     x0, [x1, #48]
  20:   d65f03c0        ret
```

→ LDR, LDP, STR, STP Funktion erklärt in Code als Kommentar: copy_asm.s

Explained obtained assembly code:
- Beginn: Pointer *i_a in x0 und *i_b in x1 gespeichert
- alle 7 Werte von Array a werden aus dem Stack geladen → Adresse steht in x0
- es gibt Offset, damit richtigen Elemente (Stelle im Array) geladen werden
- wenn LDP benutzt, dann wird Offset angepasst (2 * 8Byte)
- ganz links stehen Adressen der kompletten Instruktion (Bsp: 0: doer 4:) und daneben stehen die Instruktionen als Hexadezimalwert
- rechts wird Instruktion als geschriebene Operation angezeigt (Bsp: ldr) → wie von Disassembler interpret.

- Werte stehen in x0 und x2 bis x7 → x1 nicht mit einbezogen, weil Pointer i_b* darin liegt
- Werte werden in Register gespeichert/geladen mit Ladeoption LDR/LDP
  → diese Werte werden dann in Array b (x1) gespeichert/kopiert mit Speicheroption STR/STP
- am Ende springt man zum Ruckgabepunkt
- der Rückgabewert (ret) ist der Pointer auf dem Array a (wird von C++-Driver ignoriert, da als Rückgabewert void definiert ist)

(→genauere Erklärung zum Code allgemein, siehe: copy_asm.s)

1c) Implemented function copy_asm_loop by using loop to copy the seven values. → siehe src (copy_asm.s)

2a) Implemented function add_asm in assembly by using given template → siehe src (add_asm.s)

2b) Compiled C kernal add_c from previous task using optimization flag -02. → siehe src (add_c.o)
Disassembled compiler-generated machine code. → siehe src (add_c.dis):

→ Beschreibung des Aufbaus von SYMBOL TABLE und DISASSEBLY siehe 1b)

```
Disassembly of section .text:

0000000000000000 <add_c>:
   0:   b4000120        cbz     x0, 24 <add_c+0x24>
   4:   d2800004        mov     x4, #0x0                // #0
   8:   f8647825        ldr     x5, [x1, x4, lsl #3]
   c:   f8647846        ldr     x6, [x2, x4, lsl #3]
  10:   8b0600a5        add     x5, x5, x6
  14:   f8247865        str     x5, [x3, x4, lsl #3]
  18:   91000484        add     x4, x4, #0x1
  1c:   eb04001f        cmp     x0, x4
  20:   54ffff41        b.ne    8 <add_c+0x8>  // b.any
  24:   d65f03c0        ret
```

Explained obtained assembly code:
- Anzahl der Werte im Array steht in x0
- Pointer *a_i steht in x1 und *i_b steht in x2 und Pointer *o_c steht in x3
- mit cbz wird geprüft, ob Anzahl der Werte Null ist
  →wenn Ja, dann wird an die 24. Stelle im Programm gesprungen (zu ret), bei dem Programm beendet wird und keine Elemente kopiert werden (alternativ: cmp-compare)
  →wenn Nein, dann wird in x4 neue Laufvariable inizialisiert, da Schleife beginnt / genutzt wird
- bei Beginn der Schleife: es wird je ein Element aus Array a und Array b ausgelesen
- Offset wird mit Left Shift und der Laufvariable angegeben (→ gilt auch für weiteren Werte)
- Werte werden Addiert und in x5 geladen (x5 wird danach nichtmehr benutzt und im nächsten Schleifendurchlauf wird nächster Wert von a wieder hineingeladen)
- nach addieren der Werte werden diese in Array c gespeichert und Offset wird wieder in Left Shift angegeb.
- dann prüft man ob Schleife wieder ausgeführt werden soll, dabei wird Laufvariable erhöht
  → Laufvariable wird dann mit Anzahl der Elemente in x0 verglichen
  → Wenn Werte ungleich, dann wird wieder zum Schleifenbeginn gesprungen
  → Passiert solange bis Anzahl der Elemente mit Wert der Laufvariable gleich ist
- Am Ende springt man wieder zum Rückgabepunkt (ret)

Aufgabenbearbeitung:

Aufgabe 1 → Christian, Cora, Rahaf
Aufgabe 2 → Christian, Cora, Rahaf