Allgemeiner Hinweis: Für viele der nachfolgenden Aufgaben ist es von Vorteil, den in der Vorlesung gezeigten Simulator zu verwenden. Relevante Links finden Sie im TUWEL. Sämtliche Aufgaben können aber auch ohne Verwendung des Simulators gelöst werden.

Weiters finden Sie im TUWEL zu vielen Aufgaben Assemblycode. Dieser Code beinhaltet einige Testfälle oder hilft Ihnen dabei, den Simulator in den gewünschten Zustand zu bringen (z.B. Speicherinitialisierung). Kopieren Sie dazu die bereitgestellten Assembly Files einfach in den Editor vom Simulator.

Wichtig: Die Testfälle sollen Ihnen bei der Lösung der Aufgaben helfen. Sie decken womöglich nicht alle Fälle ab und positive Testergebnisse bedeuten nicht, dass Sie automatisch 100% auf Ihre Tafelleistung bekommen.

Aufgabe 1: RISC-V – Programm-Analyse

Gegeben ist der folgende RISC-V-Programmcode inklusive initialer RAM-Belegung. Nehmen Sie an, dass sich das System Datenwörter in little-endian[1] speichert und lädt. *Hinweis:* Verwenden Sie den bereitgestellten Quellcode von TUWEL. Dieser initialisiert den Speicher für Sie. Verwenden Sie den `ebreak` Befehl, um einen Breakpoint zu setzen.

```
1  addi t0, x0, 2
2  lui t1, 0x80000
3  addi t2, x0, 4
4  loop:
5      beq  t2, x0, end
6      addi t2, t2, -1
7      addi t0, t0, 2
8      lh   t3, 0(t0)
9      sub  t4, t1, t3
10     blt  x0, t4, loop
11     mv   t1, t3
12     j    loop
13 end:
```

RAM-Adresse	+1	+0
⋮	⋮	⋮
0x00000002	0000 0000	1000 0000
0x00000004	1110 0000	0100 1001
0x00000006	0000 0000	0000 1011
0x00000008	1110 1101	1000 0000
0x0000000A	0000 0000	0010 0011
0x0000000C	1000 0000	0000 1111
⋮	⋮	⋮
0x7FFFFFFE	0000 0000	0000 0001
0x80000000	0000 0000	0001 0000
0x80000002	0000 0000	0001 0001
0x80000004	0000 0000	0000 0001
0x80000006	0000 0000	0010 0000
⋮	⋮	⋮

a) Geben Sie die Zugriffe auf den Arbeitsspeicher in der Codezeile `8` an? Für jeden Zugriff soll die Addresse und der resultierende Wert im Register `t3` (als Dezimalzahl) angegeben werden.

	Addresse	Wert im Register nach Ausführung
0		
1		
2		
3		

[1] `https://de.wikipedia.org/wiki/Byte-Reihenfolge`

b) Tragen Sie die Registerinhalte von `t1` und `t3` direkt vor jeder Ausführung der Codezeile 5 in die Tabelle ein. Sobald das Programm terminiert, lassen Sie nachfolgende Tabellenzeilen frei. Führende Nullen können weggelassen werden. Interpretieren Sie die Inhalte als Zahl im Zweierkomplement. Falls der Inhalt eines Registers nicht bestimmbar ist, merken Sie das in dem zugehörigen Feld an.

`loop`	Register `t1`	Register `t3`
0		
1		
2		
3		
4		

c) Welche mathematische Funktion realisiert dieses Programm? Wird diese immer korrekt berechnet? *Hinweis*: Überlegen Sie sich für die zweite Frage auch, was passieren würde, wenn das Programm über andere Teile des oben gezeigten Speicherbereichs operieren würde.

Aufgabe 2: RISC-V – Assemblieren eines Programmes

Übersetzen Sie die nachfolgenden fünf RISC-V-Instruktionen A–E in Maschienencode der kompatibel mit der RISC-V Architektur ist. Bei den Instruktionen handelt es sich um korrekte RISC-V-Instruktionen.

Instruktion A: `lui x5, 0xF`

Instruktion B: `add x5, x5, x10`

Instruktion C: `lw x6, 8(x5)`

Instruktion D: `addi x6, x6, 1`

Instruktion E: `sw x6, 8(x5)`

a) Geben Sie für jede Instruktion das zugehörige Instruktionsformat an. Weiters, bestimmen Sie die Werte für alle Formatfelder der Instruktionen. Nutzen Sie dazu die „RISC-V Instruction Set Summary“[2] die von dem Lehrbuch zur Verfügung gestellt wird. Auch für Formatfelder welche auf mehrere Abschnitte aufgeteilt sind soll nur der logische Wert als Zahl angegeben werden. *Hinweis:* Orientieren Sie sich am Beispiel.

Hier ein Beispiel: `addi x1, x0, 128`
I-Type, op: $(19)_{10}$, rd: $(1)_{10}$, funct3: $(0)_{10}$, rs1: $(0)_{10}$, imm: $(128)_{10}$

b) Codieren Sie nun die Werte der Formatfelder in der untenstehenden Tabellen. Achten Sie dabei auf die erwartete Reihenfolge der Bits. Wir empfehlen Ihnen, dass Sie die 32-Bits ähnlich zu dem unten angegebenen Beispiel einteilen um Übersicht zu behalten.

Hier ein Beispiel: `addi x1, x0, 128`
I-Type, op: $(19)_{10}$, rd: $(1)_{10}$, funct3: $(0)_{10}$, rs1: $(0)_{10}$, imm: $(128)_{10}$

31 ← 0

0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	1
imm												rs1					f3			rd					op						

[2] `https://pages.hmc.edu/harris/ddca/ddcarv/DDCArv_AppB_Harris.pdf`

c) Das Programm lädt ein Element mit einem konstanten Offset von Register `x10`. Kann diese Addresse auch nur mit Hilfe der Ladeinstrukion berechnet werden? In anderen Worten, kann man Instruktion A weglassen und die Konstante in Instruktion C verändern, sodass das neue Programm sich gleich verhält. Wenn ja, geben Sie die neue Instruktion C und ihre Codierung an. Wenn nein, geben Sie eine Erklärung ab.

Aufgabe 3: RISC-V – Disassemblieren eines Programms

Gegeben ist der nachfolgende Speicherbereich, welcher eine Programmsequenz enthält die kompatibel zur RISC-V Spezifikaiton ist.

RAM-Adresse	+3	+2	+1	+0
⋮	⋮	⋮	⋮	⋮
$Addr_A$	0000 0110	0100 0000	0000 0000	1001 0011
$Addr_B$	0000 0000	1010 0000	1000 0000	1011 0011
$Addr_C$	0000 0000	1000 0000	1010 0001	0000 0011
$Addr_D$	0000 0000	0010 0001	0001 0001	0001 0011
$Addr_E$	0000 0000	0010 0000	1010 0100	0010 0011
⋮	⋮	⋮	⋮	⋮

a) Decodieren Sie die Bitfolgen in des RISC-V Codes. Bei den Bitfolgen handelt es sich um korrekte RISC-V-Instruktionen. Nutzen Sie dazu die „RISC-V Instruction Set Summary“[3] die für das Lehrbuch zur Verfügung gestellt wird. Die RISC-V Instruktionen werden mit little-endian gespeichert. *Hinweis:* Orientieren Sie sich an dem Beispiel der letzten Aufgabe und invertieren Sie die Vorgehensweise.

Instruktion A:

Instruktion B:

Instruktion C:

Instruktion D:

Instruktion E:

b) Interpretieren Sie den Code bestehend aus den Instruktionen A–E. Beschreiben Sie das Program in Worten.

[3] `https://pages.hmc.edu/harris/ddca/ddcarv/DDCArv_AppB_Harris.pdf`

Aufgabe 4: RISC-V – Laden von 32-Bit Konstanten

Das manuelle herumhantieren mit Konstanten und Addressen ist sehr fehleranfällig. Daher stellen Assembler meistens Hilfestellungen[4] für Programmierer:innen zur Verfügung. Eines dieser Werkzeuge sind "modifier". Viele RISC-V Assembler unterstützen die `%hi` und `%lo` modifier. Diese werden in der RISC-V ABI[5] wie folgt definiert[6], wobei s den Wert der Konstante denotiert.

Modifier	Berechnung	Ergebnis
%hi	$(s + 0\text{x}800) >> 12$	least-significant 20 Bits
%lo	s	least-significant 12 Bits

Hier ein Beispiel, wie diese Tabelle zu lesen ist:

```
1 addi a0, a0, %lo(0xF00F)
```

Aus der Tabelle entnehmen wir, dass der `%lo` modifier zur Berechnung die Identitätsfunktion verwendet. Daher ist das Zwischenergebnis weiterhin `0xF00F`. Von diesem sind aber nur die untersten 12-Bit relevant (vgl. mit Ergebnis). Daher ist das Endergebnis der Bitvector `0x00F`. Dieser wird nun in `addi` eingesetzt, was der folgenden Instruktion entspricht.

```
1 addi a0, a0, 0x00F
```

a) Erzeugt die unten stehende Instruktionssequenz mit der oben stehenden Definition die korrekte Konstante (`0x100874`) im Register `a0`?

```
1 lui  a0, %hi(0x100874)
2 addi a0, a0, %lo(0x100874)
```

b) Ist diese Berechnung korrekt für alle möglichen 32-bit Konstanten? Falls die Berechnung nicht immer korrekt ist, geben Sie ein Gegenbeispiel an. Falls die Berechnung korrekt ist, begründen Sie wie diese Lösung alle möglichen Konstanten abdeckt. Sie müssen hier keinen formalen Beweis präsentieren.

[4] Falls unauflösbare Addressen/Symbole verwendet werden wird diese Hilfestellung auch verwendet um wichtige Informationen (Relocations) an den Linker weiterzugeben.

[5] `https://github.com/riscv-non-isa/riscv-elf-psabi-doc/releases/tag/v1.0`, Sektion 8.4.4.

[6] Eigentlich werden in der ABI die dazugehörenden Relocations definiert.

Aufgabe 5: RISC-V – GGT Berechnung

Schreiben Sie ein RISC-V-Programm zur Berechnung des größten gemeinsamen Teilers von `a0` und `a1`. Dabei können Sie davon ausgehen, dass `a0` $\geq$ `a1` und `a0` ≥ 0 gilt. Der größte gemeinsame Teiler soll nach Programmausführung im Register `a0` stehen. *Hinweis:* Sie können für die Berechnung den euklidischen Algorithmus verwenden.

Beispiel:

`a0` vor Programmausführung:	$(15)_{10}$
`a1` vor Programmausführung:	$(12)_{10}$
`a0` nach Programmausführung:	$(3)_{10}$

Aufgabe 6: RISC-V – Paritätsbit

Ein Paritätsbit kann zur Erkennung von Datenwortfehlern in einer Folge von Bits verwendet werden. Das Paritätsbit einer Folge von Bits dient als Ergänzungsbit, um die Anzahl der mit 1 belegten Bits (inklusive Paritätsbit) der Folge auf eine gerade oder ungerade Zahl zu ergänzen. Man spricht von gerader Parität, wenn die Anzahl der mit 1 belegten Bits in der Folge gerade ist, andernfalls von ungerader Parität.

Schreiben Sie ein RISC-V-Programm für die Berechnung der ungeraden Parität eines Datenworts. Das Programm liest das Halbwort (16-Bit) von der Speicheradresse $(38A04)_{16}$ ein und legt es in das Register `t2` ab. Schreiben Sie einen Funktionsaufruf mittels `jal` zu einem Label names `nr_of_ones` zur Berechnung der Anzahl der mit 1 belegten Bits. Nehmen Sie an, dass die Paritätsberechnung **nicht** weiß, welche Register die Funktion `nr_of_ones` verwendet. Nutzen Sie die in der LVA präsentierte *Calling Konventionen* und denken Sie an das Stack Pointer *4-word alignment*. Das Paritätsbit soll am Ende des Programms im LSB in Register `a0` abgelegt werden.

Beispiel 1:	Wert an Adresse $(38A04)_{16}$:	00011010
	Paritätsbit:	0
	`a0` nach Programmausführung:	00000000 00000000 00000000 00000000
Beispiel 2:	Wert an der Adresse $(38A04)_{16}$:	01110111
	Paritätsbit:	1
	`a0` nach Programmausführung:	00000000 00000000 00000000 00000001

Aufgabe 7: RISC-V – Hamming-Distanz

Schreiben Sie ein RISC-V-Programm zur Berechnung der Hamming-Distanz zwischen zwei binären Zeichenketten mit 32-Bit Länge.
Die Hamming-Distanz ist ein Maß für die Unterschiedlichkeit von Zeichenketten. Der Wert der Hamming-Distanz entspricht dabei der Anzahl der unterschiedlichen Stellen.

Beispiel: 1011$\underline{0}$ und 1010$\underline{0}$ → Hamming-Distanz = 1
$\underline{1}$011$\underline{0}$ und $\underline{1}$111$\underline{1}$ → Hamming-Distanz = 2

Register `a0` beinhaltet die erste binäre Zeichenkette, Register `a1` die zweite. Register `a0` soll schließlich den errechneten Wert der Hamming-Distanz als Zweierkomplementzahl beinhalten.