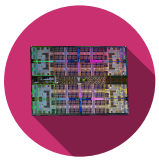


RISC-V ISA

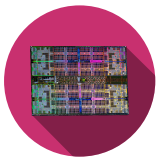
Labor Computerarchitektur

Inhalt

1 Ziel

Dieser Labore ist in mehrere Blöcke unterteilt welche während verschiedenen Wochen durchgeführt wird. Das Ziel ist es mit sich mit der Assembler Sprache des Reduced Instruction Set Computer (RISC-V) auseinanderzusetzten.

- Der Teil Abschnitt 2 des Labors betrachtet einzelne Instruktionen.
- Der Teil Abschnitt 3 des Labors führt uns zur imperative Programmierung resp. Schleifen und Funktionen.
- Im letzten Teil Abschnitt 4 des Labors wird die Arbeit des Kompilers betrachtet : das Assembly und Disassembly eines Programms (Wechsel von High-Level-Code zu Maschinencode und umgekehrt).



2 Instruktionen

In diesem ersten Teil des Labors werden wir mit dem RISC-V Interpreter auf der Webseite https://course.hevs.io/car/riscv-interpreter/ arbeiten, siehe Abbildung Abbildung 1. Dieser erlaubt es Register, Speicher zu beschreiben sowie den Code Schritt für Schritt auszuführen. Diese Onlinetools wird ihnen helfen Aufgaben zu lösen und zu kontrollieren.

Es dürfen nur Anweisungen verwendet werden, die auf dieser Seite unter **Supported Instructions** aufgelistet sind. Pseudo-Anweisungen werden nicht unterstützt.

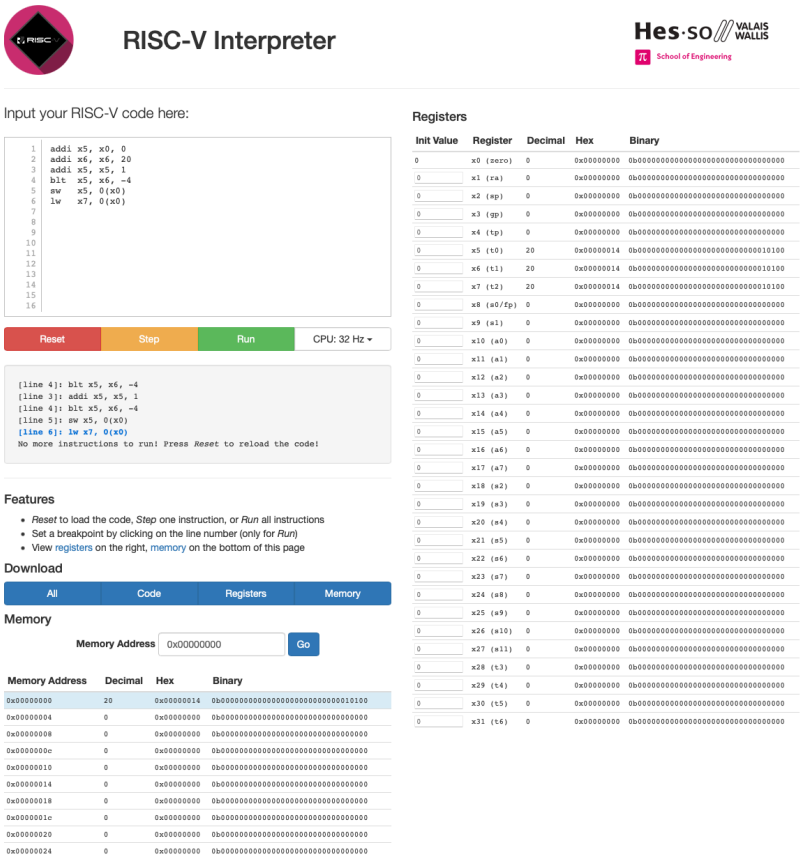
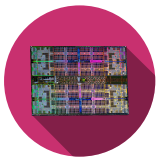


Abbildung 1 - Online RISC-V Interpreter

Achten Sie sich auf die Typen der Variablen!

- Der Typ **int** wird als vorzeichenbehaftete 32-Bit-Größe betrachtet.
- Der Typ **unsigned int** wird als unsignierter 32-Bit-Typ betrachtet.
- Wenn eine Zahl dahinter steht (z. B. **int16_t**), bedeutet dies, dass die Variable x-Bit lang ist (hier 16). Wenn ein **u** vorangestellt ist, ist er unsigniert.

uint8_t ist also ein vorzeichenloses Byte, während **int8_t** ein vorzeichenbehaftetes Byte ist.



2.1 Konstanten (Immediate)

Schreibe den RV32i Assemblercode für folgende Instruktionen:

a)

```
int a = 10;
int b = 0;
a = a + 4;
b = a - 12;

int i = 0;
int x = 2032;
int y = -78;
```

b)

```
int a = 0xABCDE123;

int b = 0xFEEDA987;
```

2.2 Grundrechenarten

Schreibe den RV32i Assemblercode für folgende Instruktionen:

a)

```
int b = 1;
int c = 2;
int d = 5;
a = b + c - d;
```

b)

```
int b = -1;
int c = 2;
int d = -78;
a = b + c - d;
```

c)

```
int b = -12;
int c = 2023;
int d = 22;
a = b + c - d;
```

2.3 Speicherzugriff

Schreibe den RV32i Assemblercode für folgende Instruktionen. Nehmen Sie an, dass das Array an der Speicheradresse liegt, die durch das Register **a0** angegeben ist:

a)

```
# We have an array of int, i.e.,
multiple ints one after the other in
memory such as [int0][int1][int2] ...
# The notation mem[x] means getting the
x th element of that array

int a = mem[4];

int b = mem[5];
```

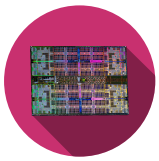
b)

```
mem[5] = 42;
```

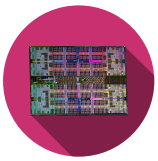
2.4 Algorithmen

Schreibe den RV32i Assemblercode für folgende Algorithmen, benutzte nur die Grundinstruktionen ohne *loops*, *conditionals* sowie *branches*:

1. In einem System kann es notwendig sein, abzuwarten, ohne etwas zu tun, ohne den aktuellen Zustand des Systems zu verändern (kein Speicherzugriff, keine Registeränderung). Diese Operation wird gemeinhin als **NOP** (**NO Op**eration) bezeichnet.



    1. Schlagen Sie eine Anweisung dafür vor.
    2. Testen Sie, ob tatsächlich keine Register oder Speicherwerte verändert werden.
2. Berechne die ersten 10 Fibonacci Nummern. Speichern Sie jede Zahl in einem anderen Register.



3 Imperatives Programmieren

3.1 Branching

3.1.1 If / else

```
int a = 1, b = 2, c;

if(a == b) {
  c = 1;
}
else {
  c = 0;
}
```

3.1.2 Switch case

```
int a, b;

switch(b) {
  case 0:
    a = 17;
    break;
  default:
    a = 99;
}
```

3.2 Loops

3.2.1 While / Do While

```
// A
int a = 10;

do{a = a - 1;}
while(a != 0);

// B
int a = 10;

while(a >= 0)
{a = a - 1;}
```

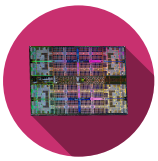
3.2.2 For

a)

```
unsigned int a = 0, i;
// mem array is at address saved in
register a0
for(i = 0; i < mem[0]; i = i + 1) {
  a = a + 2;
}
```

b)

```
// An array of 10 bytes
uint8_t myArray[10] = ...
// ...

// Let say myArray[0] is at the address
saved in register s0.
// Arrays are contiguous in memory:
myArray = [el0][el1][el2]...[elN]

int i;

for(i = 0; i < 10; i = i + 1) {
  myArray[i] = myArray[i] - 5;
}
```



3.3 Funktionen

Obwohl ein einzelner Programmierer seinen Code nach Belieben anordnen kann, wurden bestimmte Konventionen festgelegt, um die Interoperabilität verschiedener Codequellen untereinander zu ermöglichen.

Die Funktion die eine andere aufruft heisst **caller** und die aufgerufene Funktion heisst **callee**. Sie müssen synchronisiert sein, dies Bedeutet wie Argumente übergeben werden, welche Register beibehalten werden müssen ...

Die folgenden Konzepte sind die wichtigsten Regeln:

- In den Registern **a0** bis **a7** können Argumente übergeben werden. Wenn mehr benötigt werden, werden sie auf dem Stack übergeben.
- Das Ergebnis wird im Register **a0** zurückgegeben.
- Der **caller** speichert die durch die Anweisung **jal** erzeugte Rücksendeadresse (PC + 4) im Register **ra**.
- Der **callee** darf die Rücksprungadresse, den Stack oder die gespeicherten Register sXX nicht neu schreiben. Falls sie benutzt werden, muss der Platz auf dem Stack reserviert und diese Informationen gespeichert und anschliessend wiederhergestellt werden.

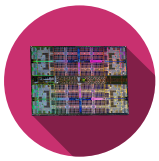
Der **Stack Pointer / sp / x2** hat im RISC-V Interpreter standardmässig den Wert 0. Sie müssen ihn manuell auf einen ausreichend hohen Wert initialisieren, um zu verhindern, dass der reservierte Speicherbereich für das Programm berührt wird. 2'000 ist in der Regel ausreichend für die folgenden Übungen.

a)

```
doNothing();

...

void doNothing() {
  return;
}
```

b)

```
int a = 1, b;
b = callA(a);

...

// Functions can be
// optimized at will

int callA(int v1) {
  v1 = v1 * 2;
  return callB(v1);
}

int callB(int v1) {
  v1 = v1 + 12;
  return v1;
}
```



3.4 Algorithmen

3.4.1 Quadratzahl

Eine quadrierte Zahl ist nichts anderes als die Multiplikation der genannten Zahl mit sich selbst. Hier arbeiten wir nur mit dem Befehlssatz RV32I:

- Schlagen Sie eine Funktion vor, die $n * n$ durch eine Additionsschleife berechnen kann.
- Testen Sie mit n = 5, 10 und 20.
- Verallgemeinern Sie die Funktion so, dass Sie ihr zwei Parameter liefern können, indem Sie $i * j$ berechnen.
- Testen Sie mit i = 5, j = 100. Testen Sie mit i = 100, j = 5. Was stellen Sie fest?
- Optimieren Sie die Funktion so, dass die Reihenfolge der Operanden keinen Einfluss auf die Rechenzeit hat.

3.4.2 Fibonacci

Die Fibonacci-Folge ist eine Folge von Ganzzahlen, in der jeder Term die Summe der beiden vorhergehenden ist. Sie beginnt mit **`0, 1, 1, 2, 3, 5, 8, 13 ...`**.

3.4.2.1 Iteration

Das Ziel ist es, den N'ten Term der Fibonacci-Folge einer nicht signierten Ganzzahl mit einer Schleife zu berechnen. Kodieren Sie den Algorithmus in Form einer Funktion gemäß dem folgenden Rezept:

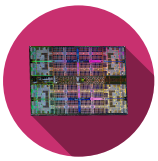
```
unsigned int n = 5; // Fibonacci term to calculate
unsigned int fib1 = 0; // First Fibonacci number
unsigned int fib2 = 1; // Second Fibonacci number

for (unsigned int i = 0; i < n; i++) {
    unsigned int res = fib1 + fib2; // Calculate the next Fibonacci number
    fib1 = fib2; // Update the first number
    fib2 = res; // Update the second number
}
// The loop will run n times, and at the end, fib1 will hold the nth Fibonacci number
```

3.4.2.2 Rekursion

Das Konzept der Rekursion bedeutet, dass eine Funktion sich selbst aufruft, um ein Ergebnis zu berechnen, so wie sie eine andere Funktion aufrufen würde.

Das Ziel ist es, die Fibonacci-Folge einer nicht signierten Ganzzahl gemäß folgendem Algorithmus zu berechnen:

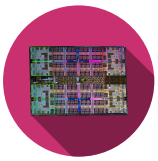
```
unsigned int fibonacci(unsigned int n){
  if(n == 0){return 0;}
  else if(n == 1) {return 1;}
  else {
    unsigned int a1 = fibonacci(n-1);
    unsigned int a2 = fibonacci(n-2);
    unsigned int a3 = a1 + a2;
    return a3;
  }
}
```



- Implementieren Sie die gegebene Funktion mit RV32I.
- Testen und Validieren der Funktion.
- Welche Werte n werden unterstützt?

Rezept

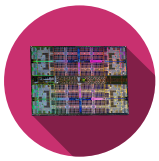
```
fibonacci:
  ... # return function if n == 0

  ... # return function if n == 1

  # we will call a function, so we need to save context on stack
    # ra must be saved, else we won't be able to exit this function properly
    # n (a0) too, since we need it afterward and A registers are saved by caller
    #   => reserve 2*4 Bytes on stack pointer and save registers
  ...

  # fibonacci(n-1)
  a0 = a0 - 1
  jal ra, fibonacci

  # a0 now contains fibonacci(n-1)
    # save it on stack (you may reserve one more slot on stack)
  ...

  # calculate fibonacci(n-2)
    # since we called a function, n (a0) may have been overwritten.
    # we need to read the original N back from stack to calculate n-2
  ...
    # fibonacci(n-2)
  a0 = a0 - 2
  jal ra, fibonacci

  # Sum up the results
    # fibo(n-2) is in a0
    # fibo(n-1) is saved on stack, so read it back
  ...
  ... # calculate fibonacci(n-1) + fibonacci(n-2)
  ... # put it in a0 for return value

  # restore context
  ... # restore ra from stack
  ... # restore stack pointer: from 2 to 3 * 4 bytes, depending how you managed stack

  jalr x0, 0(ra) # finish
```



Optionale Aufgabe: Speicherung

Es ist möglich, die Berechnung stark zu beschleunigen, indem man die Speicherung verwendet. Das Prinzip besteht darin, Speicherplatz zu reservieren und die bereits bekannten Ergebnisse für jedes berechnete n zu speichern und diesen Wert direkt wieder zu verwenden.

- Implementieren Sie die gegebene Funktion. RV32IM wird hier empfohlen.
- Testen und Validieren der Funktion.
- Vergleichen Sie die Algorithmen.



4 Assembler / Disassembler

4.1 Programme Ripes

In diesem Labor können Sie weiter den online Interpreter - Abbildung 1 benutzten. Desweiteren steht Ihnen auch der Program **Ripes** zu Verfügung, siehe Tabelle 1.

Er unterstützt das gesamte RV32I-Set, Labels sowie Pseudo-Anweisungen.

Dieses müssen Sie zuerst herunterladen und konfigurieren. Oder wenn Sie die Online Version (https://ripes.me/) verwenden, müssen Sie diese nur konfigurieren. Allerdings verlieren Sie dabei die Möglichkeit, C-Code zu schreiben.

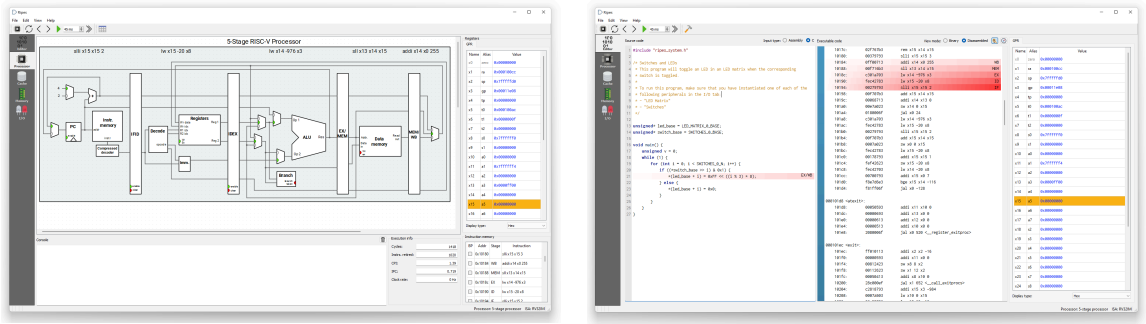


Tabelle 1 - Ripes Grafische Entwickleroberfläche

Auf Labor-PCs befindet sich unter **C:/eda/RiscV** Ripes und der unten beschriebene **gcc** Compiler.

4.1.1 Installation Ripes

1. Laden Sie die letzte Release Version des Programmes für Ihre Platform herunter unter dem Link: https://github.com/mortbopet/Ripes/releases.
2. *(OPT) Laden Sie die folgende Version der RISC-V -GNU-Toolchain für Ihre Platform unter folgendem Link herunter: https://github.com/sifive/freedom-tools/releases/tag/v2020.04.0-Toolchain.Only*
3. Entpacken Sie die beide *(OPT) und kopieren Sie den RISC-V -GNU-Toolchain Ordner in den Ripes Ordner.*
4. Starten Sie Ripes und konfigurieren Sie die Kompiler Einstellungen unter: **Edit** → **Settings**. *(OPT)Hierzu müssen sie die Datei* **`/riscv64-unknown-elf-gcc-8.3.0.exe`** *auswählen.*

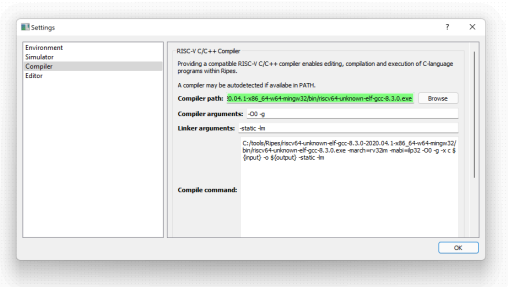
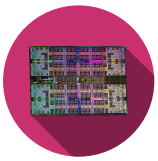


Abbildung 2 - Ripes Toolchain Einstellungen



4.1.2 Setup Ripes

1. Wählen Sie in den Prozessor Einstellungen den RISC-V → 32-bit → Single-Cycle Processor ohne ISA Extentions aus.

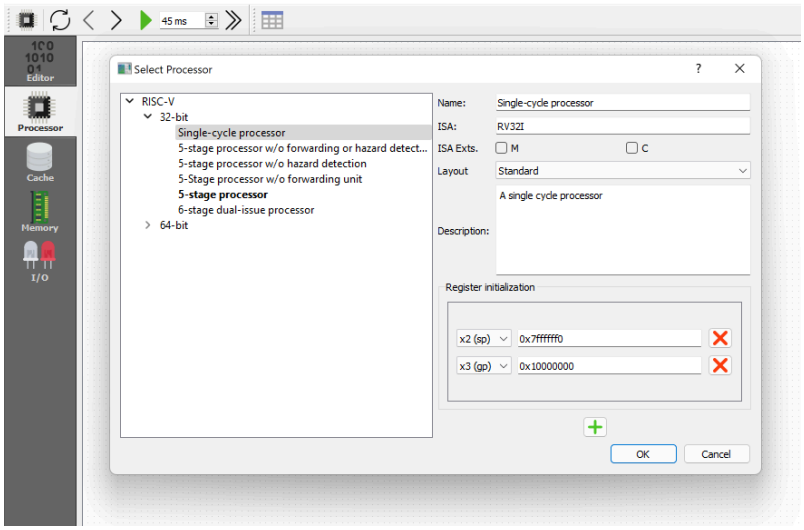
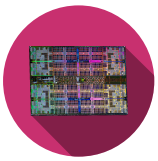


Abbildung 3 - Ripes Prozessor Einstellungen

Benutzen Sie Ripes, um Ihre Labore und auch Ihre Übungsreihen zu erstellen und testen. Es kann wie ein echter Prozessor reagieren und Ihnen Verhaltensweisen zeigen, die Sie vielleicht nicht erwarten würden (z. B. mit dem Umschreiben des eigenen Programms, indem er L-S-Anweisungen falsch handhabt - Abschnitt 4.2).



4.2 Analyse 1: Speicherverwaltung in Ripes

Ripes ermöglicht die Auswahl mehrerer Prozessoren (siehe Abbildung 3). Obwohl der Single-Cycle Prozessor zwei verschiedene Speicherchips für Befehle und Daten trägt, der Speicher wird als gemeinsam betrachtet, ebenso wie generell auf einem physischen Chip :

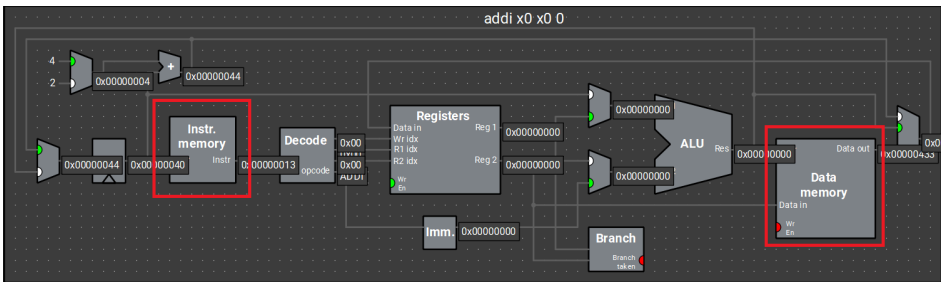


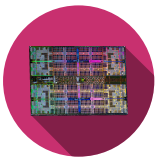
Abbildung 4 - Single-Cycle Prozessor Speicher

Das bedeutet, dass es möglich ist, **versehentlich Ihren eigenen Code zu schreiben**.

Um dies zu veranschaulichen, führen die folgenden Code aus und schauen Sie wie die Instruktionen im Instruction Memory überschrieben werden (tab Memory).
Benutzen Sie den Step by Step Modus.

```
# s0 counts the number of times we really loop
addi s0, zero, 0
addi s1, zero, 0 # place in memory
# t0 is the number of times we SHOULD be looping
addi t0, zero, 10

store_loop:
  sw s0, 0(s1)      # store current loop counter in memory
  addi s1, s1, 4    # increment memory place
  addi s0, s0, 1    # increment how many times we have looped
  addi t0, t0, -1   # decrement loop counter
  blt t0, zero, end
  jal zero, store_loop

end:
  nop
```

Die Schleife soll 0 an der Adresse 0, 1 an der Adresse 4, 2 an der Adresse 8 ... speichern. Aber in der vierten Iteration der Schleife wird die Anweisung `sw s0, 0(s1)` als `sw 3, 0(0x0C)` ausgeführt, wodurch die gleiche Anweisung durch 0x00000003 ersetzt wird, was einer `lb x0, 0(x0)`-Anweisung entspricht:



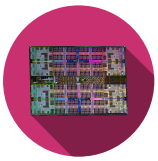
0x00000020	4276088943	111	240	223	254
0x0000001c	181347	99	196	2	0
0x00000018	4294083219	147	130	242	255
0x00000014	1311763	19	4	20	0
0x00000010	4490387	147	132	68	0
0x0000000c	3	3	0	0	0
0x00000008	2	2	0	0	0
0x00000004	1	1	0	0	0
0x00000000	0	0	0	0	0

Abbildung 5 - Erinnerung neu schreiben

Der Code wurde also verändert und entspricht nicht mehr dem ursprünglichen Zweck.

Um solche Probleme zu vermeiden, arbeiten Sie mit dem Speicher über den **stack pointer**, wie bei Funktionsaufrufen :

- Speicherplatz reservieren, indem **sp** um die Anzahl der benötigten Bytes verringert wird (für das Beispiel hätten wir $10 * 4$ Bytes).
- NUR mit dem reservierten Speicher arbeiten.



4.3 Analyse 2: Arbeit eines Compilers

Im Program Ripes Abschnitt 4.1, haben wir folgende 2 C-Codes kompiliert. Suchen Sie im generierte Code die entsprechenden Assemberbefehle. Was bedeuten die einzelnen Befehle im `main:` ?

a)

```
void main() {
  int a = 3;
}
```

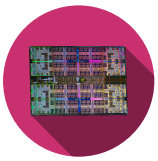
```
addi sp sp -32 # stack for 8 items
sw s0 28(sp)   # saves s0 in stack[1]
as it will use it and is a callee saved
register

addi s0 sp 32  # s0 points to start of
stack
addi a5 x0 3   # a5 = 3
sw a5 -20(s0)  # stack[5] = 3
addi x0 x0 0   # nop

lw s0 28(sp)   # restore s0
addi sp sp 32  # unstack 8 items
jalr x0 x1 0   # return
```

b)

```
void main() {
  while(1) {
    int a = 3;
  }
}
```

```
addi sp sp -32 # stack for 8 items
sw s0 28(sp)   # saves s0 in stack[1]
as it will use it and is a callee saved
register

addi s0 sp 32  # s0 points to start of
stack
addi a5 x0 3   # a5 = 3
sw a5 -20(s0)  # stack[5] = 3
jal x0 -8      # jal 2 instructions
back
```

Was enthalten die beiden Assemblercodes ? Warum ist der rechte Code, mit mehr Anweisungen, kürzer ?



4.4 HEIRV-32

Den RISC-V Assembler und Disassembler welcher im Labo benutzt wird kann im Repository unter **`isa/heirv32-asm/`** für Windows amd64 (**`HEIRV32-ASM_1.2.3_windows_x86_64.exe`**), Linux amd64 (**`HEIRV32-ASM_1.2.3_linux_x86_64`**) sowie macOS arm64 (**`HEIRV32-ASM_1.2.3_macos_aarch64`**) gefunden werden. Diese Tool erlaubt es ihnen Assemblercode eines RV32i oder HEIRV32 Prozessors in Binärcode zu verwandeln (Assembly) sowie Binärcode zurück in Assemblercode zu verwandeln (Disassembly).

- Assembly ⇨ Assemblerdatei **`*.s`** oder **`*.asm`** ⇨ Binärdatei **`*.bin`**
- Disassembly ⇨ Binärdatei **`*.bin`** oder hexadezimale **`*.txt`** ⇨ Assemblerdatei **`*.s`**

> Das Tool kann entweder in der Kommandozeile oder als GUI benutzt werden.

4.4.1 Mit der grafischen Benutzeroberfläche (GUI)

Hierfür müssen die nur die Datei ausführen und in der Oberfläche folgenden 3 Schritte durchführen:

1. Auswahl der Prozessors
   - **`RV32I`** - Kompletter RISC-V Prozessor
   - **`HEIRV32`** - Prozessor des Projektes am Ende des Semsters
2. Drücken auf den Knopf **`Next`**
3. Auswahl des Eingabefiles - Entweder **`*.s`**, **`*.asm`**, **`*.txt`** oder **`*.bin`**. Das Programm bestimmt automatisch, ob der Compiler oder Disassembler verwendet werden soll.

Danach finden Sie die generierten Dateien im gleichen Ordner wie die ausgewählte Datei. Auch bleibt ein Log-Fenster offen, das es Ihnen ermöglicht, die Schritte zu verstehen, die der (De)Compiler durchgeführt hat.

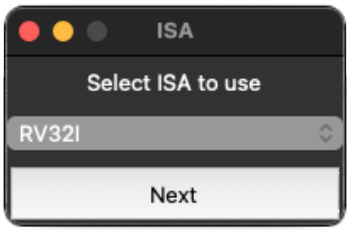


Abbildung 6 - GUI des Programmes HEIRV32-ASM

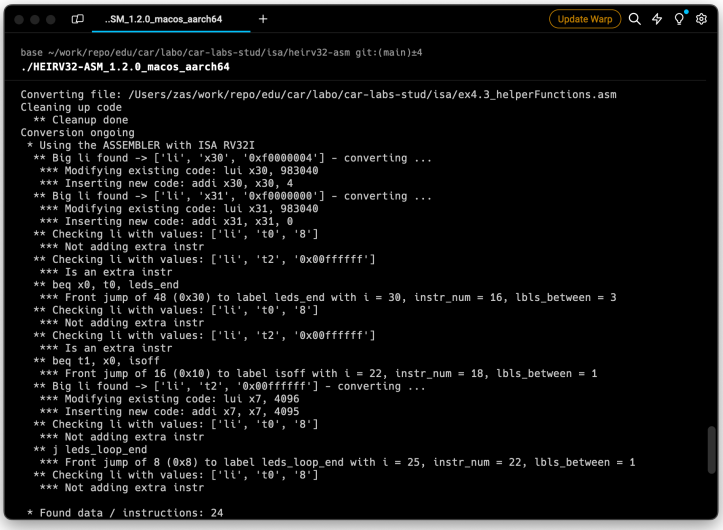
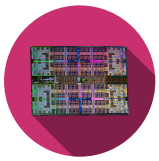


Abbildung 7 - Ausgabe des Programms heirv32-asm



4.4.2 Analyse 3 : (De)kompilierung

- Speichern Sie den folgenden Code als **.s** oder **.asm**-Datei:

```
myLabel:
  li   x20 0b11111111111
  add  x20 x20 x20
  add  x20 x20 x20
  add  x20 x20 x20
  addi x3 x0 0
begin:
  mv   x1 x0
a:
  slt  x2 x1 x20
  beq  x2 x0 subing
  addi x1 x1 1
  beq  x0 x0 a
subing:
  addi x3 x3 -1
  beq  x3 x0 test
label_wo_beq:
  addi x30 x0 0xCC
  jal  ra begin
test:
  addi x30 x0 0b00110011
  jal  begin
l6: # This is very important
# But won't always save you
  jal  ra myLabel
```

- Kompilieren Sie es mit HEIRV32-ASM, wobei die Ausgaben **print** und **bin** aktiviert sind. Um dies zu tun, verwenden Sie die GUI mit dem Set **RV32I** oder in der Kommandozeile:

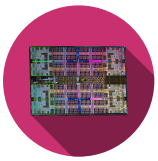
  ./dist/HEIRV32-ASM_1.1.3_xxx -i RV32I -f ./my/file/disassembly-01.s -t pb

  Eine Datei **.txt** wird erstellt, die eine für Menschen lesbare hexadezimale Version der erzeugten Datei **.bin** ist.
- Nun dekompilieren Sie die Datei **.txt** auf die gleiche Weise, um eine Assemblerdatei neu zu erstellen, entweder über die GUI mit dem Set **RV32I** oder in der Kommandozeile:

  ./dist/HEIRV32-ASM_1.1.3_xxx -i RV32I -f ./my/file/assembled-01.txt -t pb

  Dies sollte eine Datei **xxx_disassembly.s** erstellen.

1. Was ändert sich zwischen den beiden Codes ? Nennen Sie 3 Punkte.
2. Wozu kann die Anweisung **jal ra myLabel** in diesem Zusammenhang dienen ?



4.5 Reverse Engineering

Bei epileptischen Symptomen, nicht mit dem Abschnitt 4.5 fortfahren.

Die LEDs, die in Tabelle 3 gezeigt werden, blitzen schnell von weiß auf schwarz.

Sie arbeiten in einem Partnerprojekt an einem etwas modifizierten RISC-V -Prozessor.

Um sich das Leben zu vereinfachen und mit der Aussenwelt zu kommunizieren, es kann den Status von 4 Tasten die mit dem Prozessor verbundenen auslesen, indem er die Speicheradresse **0xf0000000** liest, und 8 LEDs einschalten, indem er an die Speicheradressen **0xf0000004** (led 0) bis **0xf0000020** (led 7) schreibt:

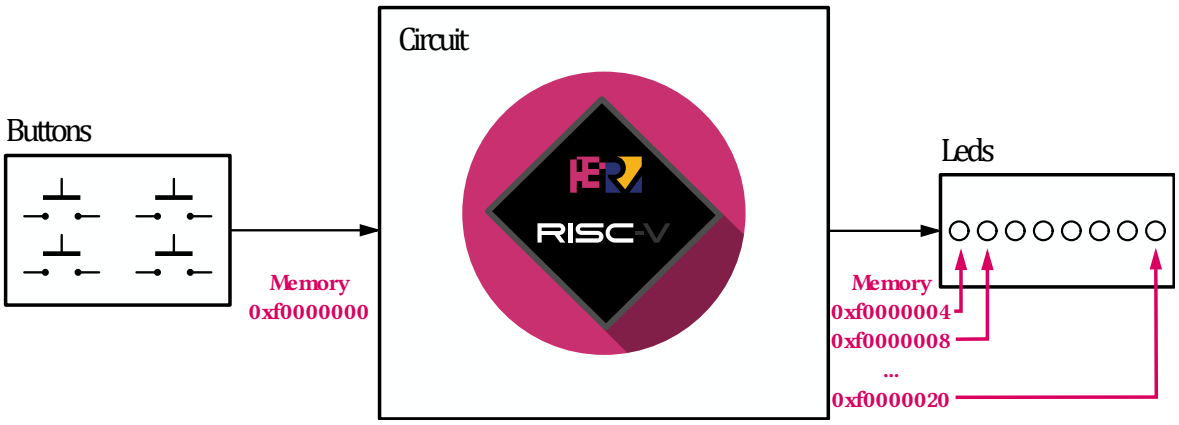


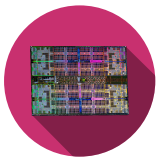
Abbildung 8 - Modifizierter RISC-V-Prozessor

4.5.1 Gegeben

Das erste Labor bestand darin, die Leds entsprechend der gedrückten Tasten blinken zu lassen:

- Keine Taste: Die 8 LEDs werden nach dem Muster `0xAA` eingeschaltet
- Der Knopf 0 ist auf `'1'`: Alle 8 LEDs blinken in einer angemessenen Frequenz (das Blinken ist sichtbar)

Um den Zugriff auf die Taste und die Leds zu vereinfachen wurde Ihnen der folgende Code gegeben:

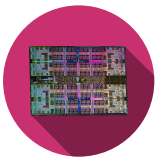


```
setup:
    # led is sw xx, offBy4(x30) with xx loaded as 0x00rrggbb
    li x30, 0xf0000004 # base address for leds
    li x31, 0xf0000000 # base address for buttons, one register

    # DO NOT MODIFY x30 (t5) and X31 (t6) !!!
    # MUST BE THE FIRST THING IN YOUR PROGRAM, BEFORE MAIN



get_btns: # return buttons value in a0
  lw a0, 0(x31)
  jr ra



set_leds: # pass a 8 bits which if bit(n) = '1', led(n) is on
  li t0, 8   # loop the leds
  mv t3, x30 # mem position
  addi t3, t3, 28 # leds display is reversed, so stock reverted

  leds_loop:
    beq x0, t0, leds_end
    andi t1, a0, 1
    # if 1, lights corresponding led
    beq t1, x0, isoff

  ison:
    li t2, 0x00ffffff
    j leds_loop_end

  isoff:
    mv t2, x0

  leds_loop_end:
    sw t2, 0(t3) # save led value
    srli a0, a0, 1  # shift right leds value
    addi t0, t0, -1 # decrement loop
    addi t3, t3, -4  # add memory pos
    j leds_loop

  leds_end:
    jr ra
```

Leider ist Ihr Kollege heute abwesend und hat Ihnen keine Kopie des bereits geschriebenen Codes hinterlassen (*denken Sie daran, Git zu lernen !*). Ausserdem gab es einige Bugs:

- Keine Tasten: die Leds leuchten nach dem Muster **`0x2A`** auf
- Der Knopf 0 ist auf **`'1'`**: nur 7 LEDs blinken, und das mit einer Frequenz, die viel zu hoch zu sein scheint.
- Eine der anderen Schaltflächen wird gedrückt: Dasselbe Verhalten, als wenn die Schaltfläche 0 gedrückt wird

Ein Hardwareproblem kann ausgeschlossen werden.

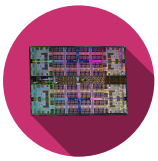


4.5.2 Dekompilierung

Sie erinnern sich, dass der zuletzt geschriebene Code auf Ihr Board RISC-V geflasht wurde, und da der Chip nicht gegen Rücklesen geschützt war, können Sie den Code durch Zugriff auf den JTAG-Bus extrahieren:

```
f0000f37
004f0f13
f0000fb7
000f8f93
00000413
040000ef
02a00463
00900663
fff48493
ff1ff06f
06400493
00800663
00000413
0140006f
0bf00413
00c0006f
02a00413
00000493
00040513
010000ef
fc5ff06f
000fa503
00008067
00800293
000f0e13
01ce0e13
02500863
00157313
00030863
010003b7
fff38393
0080006f
00000393
007e2023
00155513
fff28293
ffce0e13
fd5ff06f
00008067
```

1. Dekompilieren Sie Ihren Code
2. Identifizieren und trennen Sie den gegebenen Code vom Rest
3. Die Verwendung von gegebenen Funktionen verstehen



4.5.3 Simulierung

Um das System zu simulieren, öffnen Sie unter Ripes die Registerkarte I/O:

- Doppelklicken Sie auf Switches, wählen Sie das erstellte Widget aus und konfigurieren Sie es so, wie es ist:

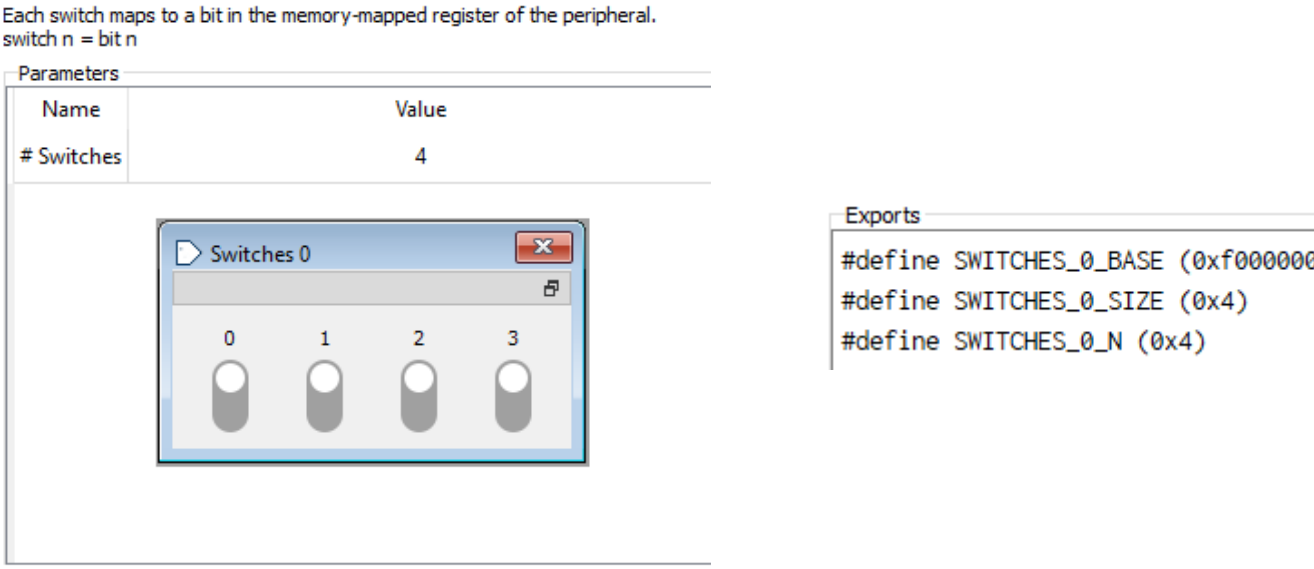


Tabelle 2 - Ripes - Schalterkonfiguration

Stellen Sie sicher, dass **Exports** die gleichen Informationen enthält wie auf dem Bild, das nach der Einstellung des Moduls gegeben wurde.

- Doppelklicken Sie auf LED Matrix, wählen Sie das erstellte Widget aus und konfigurieren Sie es so, wie es ist:

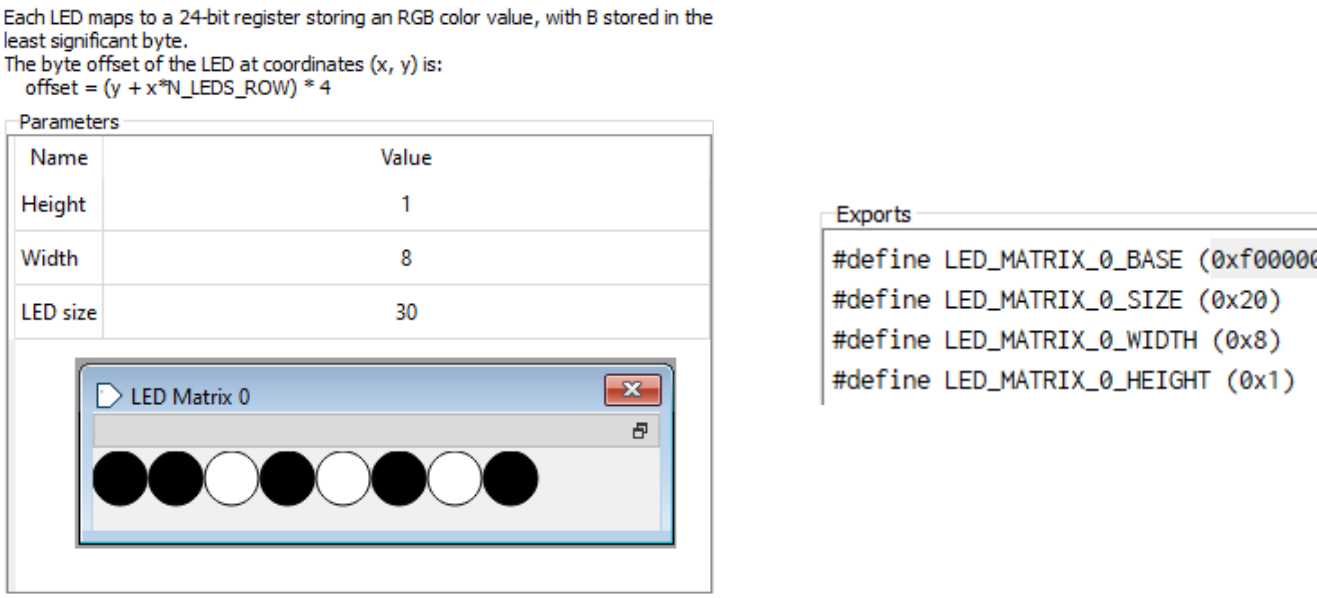
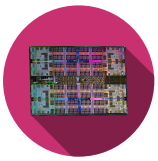
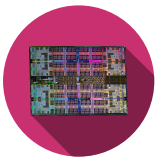


Tabelle 3 - Ripes - Led-Konfiguration

Stellen Sie sicher, dass **Exports** die gleichen Informationen enthält wie auf dem Bild, das nach der Einstellung des Moduls gegeben wurde.



Fügen Sie zuerst die Schalter und dann die Leds hinzu, um die richtige Basisadresse der Module zu erhalten.



4.5.4 Fehlerbehebung

1. Laden Sie Ihren dekompiliert Assemblercode in **Editor**.
2. Drücken Sie F8 (oder das Symbol >>) und stellen Sie fest, dass der Schaltkreis gemäss den genannten Problemen funktioniert, indem Sie unter **I/O** mit den Schaltern interagieren.
3. Fixieren Sie die **4** genannten **Probleme**
4. Testen Sie Ihre Rennstrecke auf Ripes

Da die Clock nicht genau eingestellt werden kann, wird die Blinkfrequenz der Leds ohne Berechnung ermittelt. Korrigieren Sie den Code, um etwa 2 Hz zu erhalten.

Was Sie gerade getan haben, ähnelt dem Hardware-Hacking: Ein Code wird gedumpt (vom Ziel abgerufen), analysiert, verändert und dann wieder in ein System eingespeist, um es nach Lust und Laune anzupassen.

Natürlich ist es nicht immer so einfach, einen Code zu kopieren und neu einzuspeisen. Es sind teilweise exotische Methoden entstanden (siehe den Kamikaze-Hack der Xbox 360).