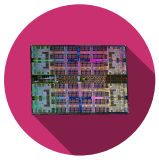
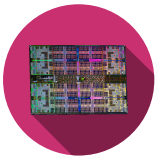


RISC-V ISA

Studentenlösungen
Labor Computerarchitektur

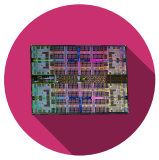
Inhalt



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 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.

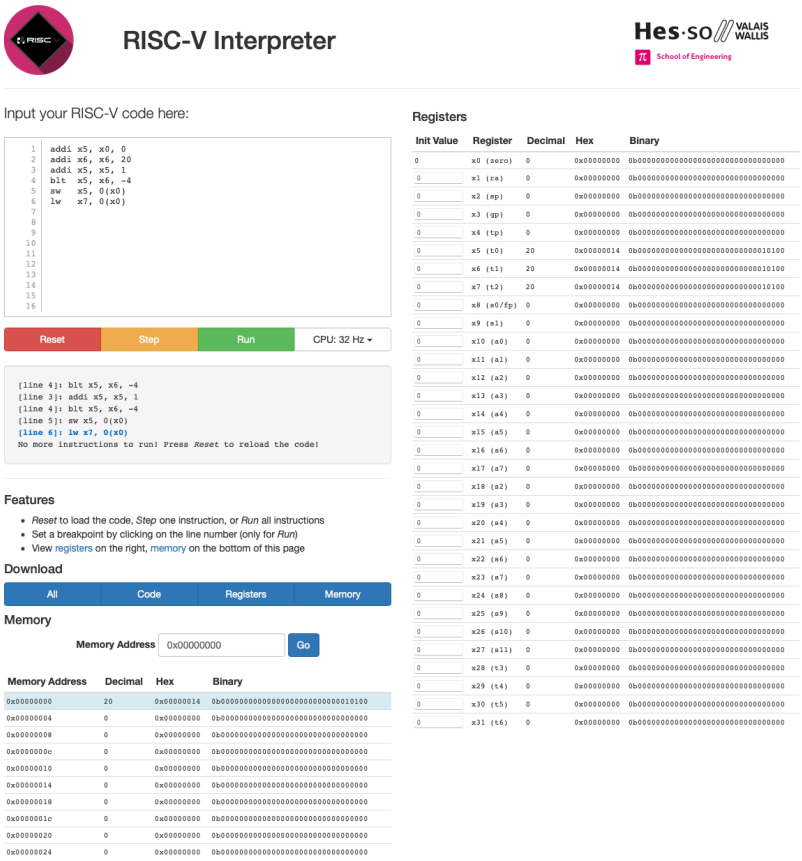
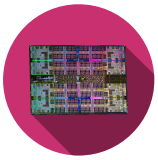


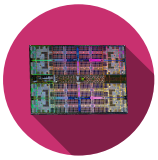
Abbildung 1: Online RISC-V Interpreter



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;
```

2.1.1.1 Lösungsunterstützung

a)

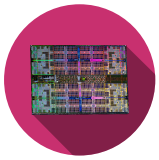
```
# s0 = a, s1 = b
addi s0, zero, 10   # int a = 10
addi s1, zero, 0    # int b = 0
addi s0, s0, 4      # a = a + 4
# addi used also for negative numbers
 # no 'subi' opcode exists
addi s1, s0, -12    # b = a - 12

# s4 = i, s5 = x, s6 = y
 # I values ranging from -2048 to 2047
 #  can be used directly
addi s4, zero, 10   # int i = 0
addi s5, zero, 2032 # int x = 2032
addi s6, zero, -78  # int y = -78
```

b)

```
# int a = 0xABCDE123;
# s2 = a
 # too big for addi => use 'lui'
lui  s2, 0xABCDE     # s2 = 0xABCDE000
addi s2, s2, 0x123   # s2= 0x ABCDE123

# int b = 0xFEEDA987;
# s3 = b
#   F    E    E    D    A    9    8    7
#
1111_1110_1110_1101_1010_1001_1000_0111
# +---------------------+ +------------
+
# +-- upper 20 bits        +-- lower 12

# Since bit 11 is 1 (neg. val) it is
sign-extended and adds 0xfffff (-1) in
the upper 20 bits
 # the upper imm must be therefore +1
# upper 20 bits 0xFEEDA+1 = 0xFEEDB
lui  s3, 0xFEEDB      # s3 = 0xFEEDB000
# lower 12 bits 0x987
addi s3, s3, 0x987    # s3 = 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;
```

2.2.1.1 Lösungsunterstützung

a)

```
# a = b + c - d;
# s0 = a, s1 = b, s2 = c,
s3 = d. t0 = t

# b = 1, c = 2, d = 5
addi s1, zero, 1
addi s2, zero, 2
addi s3, zero, 5
# t = b + c
add t0, s1, s2
# a = t - c
add s0, t0, s3

# t0 = 0x00000003
# s0 = 0x00000008
# s1 = 0x00000001
# s2 = 0x00000002
# s3 = 0x00000005
```

b)

```
# b = -1, c = 2, d = -78
addi s1, zero, -1
addi s2, zero, 2
addi s3, zero, -78
# t = b + c
add t0, s1, s2
# a = t - c
add s0, t0, s3

# t0 = 0x00000001
# s0 = 0xfffffb3
# s1 = 0xffffffff
# s2 = 0x00000002
# s3 = 0x00000fb2
```

c)

```
# b = -12, c = 2023, d =
22
addi s1, zero, -12
addi s2, zero, 2023
addi s3, zero, 22
# t = b + c
add t0, s1, s2
# a = t - c
add s0, t0, s3

# t0 = 0x000007db
# s0 = 0x000007f1
# s1 = 0xfffffff4
# s2 = 0x000007e7
# s3 = 0x00000012
```

2.3 Speicherzugriff

Schreibe den RV32i Assemblercode für folgende Instruktionen:

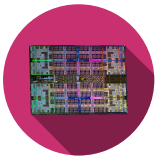
a)

```
# 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;
```

2.3.1.1 Lösungsunterstützung



a)

```
# int a = mem[4];
# s7 = data at 4 * 4 bytes
lw s7, 16(zero)

# Maybe you did 'lw 4(zero)' instead of 'lw 16(zero)', thus why the next exercise.

# int a = mem[5];
# s7 = data at 5 * 4 bytes
lw s7, 20(zero)

# If you try to use 'lw s7, 5(zero)', the instruction fails because the fetch is not aligned in memory.
# Indeed, the memory is composed of 8 bits locations like:
# [B3][B2][B1][B0]
# [B7][B6][B5][B4]
# ...
# Loading 'lw s7, 5(zero)' means trying to load [B8][B7][B6][B5] <= misaligned

# On the other hand, using 'lb' / 'lbu' / 'sb' who handle only 1 Byte can be used on any address.
# Using 'lh' / 'lhu' / 'sh' who handle 2 Bytes must be used within the same 32 bits locations, i.e. [(x+1)*4 - 1]...[x*4].
```

b)

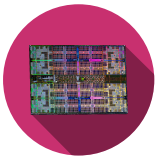
```
# mem[5] = 42;
# s7 = a

# t3 = 42
addi t3, zero, 42
# data value at mem[20] = 42
sw t3, 20(zero)

# Same as before, we cannot store unaligned in memory.

# Why ? Higher languages don't have those limitations.
# But since our processor works with 32 bits, each clock cycle can only output one 32 bits value from memory.

# E.g. with 'lw t0, 1(zero)' on our memory:
# [B3][B2][B1][B0]
# [B7][B6][B5][B4]
# ...
# We would need to:
# - Load B3..B0 in t0:  t0 = [B3'B2'B1'B0]
# - Shift t0 right:     t0 = [00'B3'B2'B1]
# - Load B4 in t1:      t1 = [00'00'00'B4]
# - Shift t1 left:      t1 = [B4'00'00'00]
# - OR t0 and t1:       t0 = [B4'B3'B2'B1]
```

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.

2.4.1.1 Lösungsunterstützung

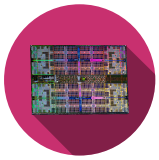


NOP)

```
# With I
addi x0, x0, 0
andi x0, x0, 0
...
# With R
add x0, x0, x0
and x0, x0, 0
...
# With J
jal x0, 4
```

Fibonacci)

```
# 10 fibonacci numbers
#  calculate  the  first  10  Fibonacci
numbers
# without loops or conditionals

# set up initial values of fib(0) and
fib(1)
addi s1, s0, 0    # fib(0) = 0
addi s2, s0, 1    # fib(1) = 1

# calculate fib(2) = fib(1) + fib(0)
add s3, s2, s1

# calculate fib(3) = fib(2) + fib(1)
add s4, s3, s2

# calculate fib(4) = fib(3) + fib(2)
add s5, s4, s3

# calculate fib(5) = fib(4) + fib(3)
add s6, s5, s4

# calculate fib(6) = fib(5) + fib(4)
add s7, s6, s5

# calculate fib(7) = fib(6) + fib(5)
add s8, s7, s6

# calculate fib(8) = fib(7) + fib(6)
add s9, s8, s7

# calculate fib(9) = fib(8) + fib(7)
add s10, s9, s8
```



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;
}
```

3.1.2.1 Lösungsunterstützung

If / else)

```
addi s0, zero, 1 # int a = 1
addi s1, zero, 2 # int b = 2

# if(a == b)
# since we don't have 'branch if equal',
revert the logic
test:
  # not equal -> goto if_nequ (imm = 12)
  bne s0, s1, if_nequ
# a == b
equal:
  addi s2, zero, 1 # c = 1
  jal end # goto end (imm = 8)
# a != b
if_nequ:
  addi s2, zero, 0  # c = 0

end:
  # ...
```

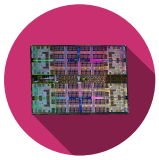
Switch case)

```
# a = s0, b = s1

# if(b == 0)
bne s1, zero, not0 # imm = 12

# b == 0
li s0, 17
jal end # imm = 8

# b != 0 (others)
not0:
  li s0, 99

end:
# ...
```

3.2 Loops

3.2.1 While / Do While

```
// A
int a = 10;

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



```c
// B
int a = 10;

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

3.2.1.1 Lösungsunterstützung

```asm
// A : do-while
addi s0, zero, 10 # int a = 10;
while_entry:
  addi s0, s0, -1 # a--
  bne zero, s0 while_entry # imm = -4

// B : while
addi s0, zero, 10 # int a = 10;
jal while_test # imm = 8
while_entry:
  addi s0, s0, -1 # a--
while_test:
  bge zero, s0, while_entry # imm = -4
```

3.2.2 For

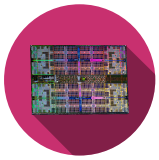
a)

```c
unsigned int a = 0, i;

for(i = 0; i < mem[0]; i = i + 1) {
  a = a + 2;
}
```

b)

```c
// 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;
}
```

3.2.2.1 Lösungsunterstützung

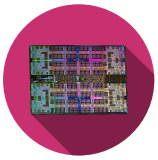


a)

```
# a is s0, i is s1, mem[0] = s2
lw s2, 0(x0) # loop target
li s1, 0 # i
li s0, 0 # a

jal for_test # imm = 8

for_do:
  # a = a + 2
  addi s0, s0, 2
  # MUST be at the end of for
  addi s1, s1, 1
# Do not put the 'addi s1, s1, 1' right
after the for_do label. The for loop
index changes AFTER the iteration

for_test:
  bltu  s1, s2, for_do # imm = -4
```

b)

```
# myArray[0] is at addr. s0 in memory
# myArray[1] is at addr. s0 + 1 in mem...
# Works because we have bytes (i.e. 8bits
values)

# i is s1, target is t0
addi s1, zero, 0  # i = 0
addi t0, zero, 10 # target = 10

for_test:
  bge s1, t0, end # i > 10, done

  add t1, s0, s1   # i + base array addr.
  lbu t2, 0(t1)    # load mem[myArray[i]]
  addi t2, t2, -5 # do -5
  sb t2, 0(t1)    # store back variable
  addi s1, s1, 1  # i = i + 1
  # j is pseudo for 'jal x0, label'
  j for_test

end:
# ...
```



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.

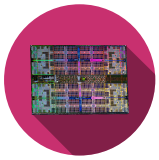
a)

```
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;
}
```

3.3.1.1 Lösungsunterstützung

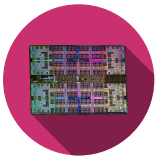


a)

```
# jumps to function
jal ra, doNothing

# ...

doNothing:
  # return, no modification of vars.
  jalr zero, ra, 0 # or pseudo jr ra
```

b)

```
# a is s0, b is s1
li s0, 1  # a = 1
mv a0, s0 # copy into a0 as func argument
jal ra, callA
mv s1, a0 # b = result

# ...

callA:
  # save context (ra => 4 stack spaces)
  addi sp, sp, -4
  sw ra, 0(sp)

  # do v1 *= 2
  sll a0, a0, 1
  # callB with v1 in a0
  jal ra, callB

  # No need to copy return value since
it is already in a0 by convention

  # restore context
  lw ra, 0(sp)
  addi sp, sp, 4
  jr ra

callB:
 # No context saving since we don't touch
sX registers, nor call other functions

  # do v1 += 12
  addi a0, a0, 12
  jr ra
```



3.4 Algorithmen

3.4.1 Fibonacci durch Rekursion

Der Begriff Rekursivität impliziert, dass eine Funktion sich selbst aufruft, um ein Ergebnis zu berechnen, so wie sie auch eine andere Funktion aufrufen würde.

Das Ziel ist es, die Fibonacci-Folge einer unsignierten ganzen Zahl nach dem folgenden Algorithmus zu berechnen:

```
unsigned int fibbonacci(unsigned int n){
  if(n == 0){
    return 0;
  } else if(n == 1) {
    return 1;
  } else {
    return (fibbonacci(n-1) + fibbonacci(n-2));
  }
}
```

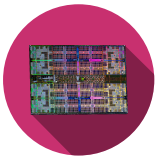
- Implementieren Sie die gegebene Funktion mit RV32I
- Testen und Validieren der Funktion
- Welche n-Werte werden unterstützt?

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.

3.4.1.1 Lösungsunterstützung

```
# s0 is n
li s0, 15

mv a0, s0   # Pass argument n in a0 to fibo(n)
jal fibo    # Return value from fibo(n) in a0
mv s1, a0   # Save return value

# ...

fibo:
  # end function if n == 0
  beq a0, zero, fibo_end
  # end function if n == 1
  addi t0, a0, -1 # t0 = n-1
  beq t0, zero, fibo_end

  # save context (ra, n in a0) => 2*4 Bytes
  addi sp, sp, -8
  sw a0, 0(sp)
  sw ra, 4(sp)
```



```
  # Fibo(n-1)
  addi a0, a0, -1 # n-1
  jal fibo        # fibo(n-1)

  # Fibo(n-2)
  lw t0, 0(sp)    # Retrieve original n
  sw a0, 0(sp)    # Save fibo(n-1) in mem (we won't need the original n anymore)
  addi a0, t0, -2 # n-2
  jal fibo        # fibo(n-2)

  # Add fibo(n-1) and fibo(n-2)
  lw t0, 0(sp)    # Get result of fibo(n-1) from stack
  add a0, a0, t0  # add fibo(n-1) and fibo(n-2)

  # restore context
  lw ra, 4(sp)
  addi sp, sp, 8

fibo_end:
  jr ra
```

3.4.2 Quadratzahl

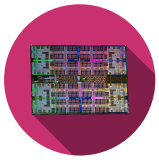
Eine quadrierte Zahl ist nichts anderes als die Multiplikation der genannten Zahl mit sich selbst. Hier arbeiten wir nur mit dem Befehlssatz RV32I:

- 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.

Dieser Algorithmus hat die Komplexität $O(n)$. Es ist möglich, ihn auf eine Komplexität von $O(\log_2(n))$ zu reduzieren, indem man Additionen und Shifts clever einsetzt. Dieser heisst **fast multiplication**:

- Schlagen Sie eine optimierte Funktion vor.

3.4.2.1 Lösungsunterstützung

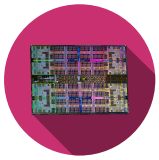


Non optimized adds $a + a \Rightarrow b$ times.

```
# Worse O(n_b)
mulFunct: # mulFunct(int a, int b)
  # Prepare loop
  mv t0, a0
  addi a1, a1, -1

  mul_beg:
    bgeu zero, a1, mul_end # if 1 > b
    add a0, a0, t0
    addi a1, a1, -1
    j mul_beg

  mul_end:
    jr ra

# But ... it does not work with a1 = 0,
can you fix the algo. ?
```

Optimized adds $a + a \Rightarrow b$ times if $b < a$ else $b + b \Rightarrow a$ times.

```
# Better O(n_min[a,b])
sfmulFunct: # sfmulFunct(int a, int b)
  # save context (ra)
  addi sp, sp, -4
  sw ra, 0(sp)

  # check which is bigger
  bltu a1, a0, do_sfmul # if b < a, do
  # else swap them
  mv a2, a0
  mv a0, a1
  mv a1, a2

  # Multiply
  do_sfmul:
    jal ra, mulFunct

  # restore context
  lw ra, 0(sp)
  addi sp, sp, 4
  jr ra
```



4 Assembler / Disassembler

4.1 Program Ripes

In diesem Labor können Sie weiter den online Interpreter - Abbildung 1 benutzten. Desweiteren steht Ihnen auch der Program **Ripes** zu Verfügung, siehe Abbildung 2.

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.

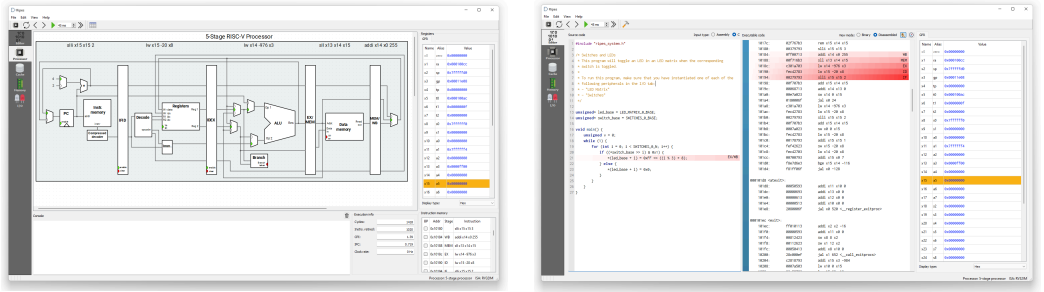


Abbildung 2: Ripes Grafische Entwickleroberfläche

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. 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 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**. Hierzu müssen sie die Datei **`/riscv64-unknown-elf-gcc-8.3.0.exe`** auswählen.

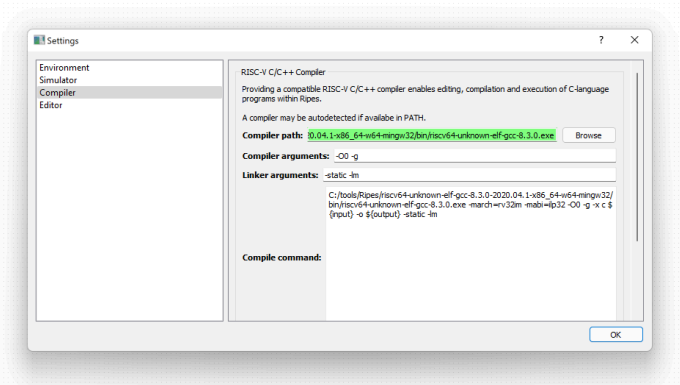


Abbildung 3: Ripes Toolchain Einstellungen

4.1.2 Setup Ripes

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

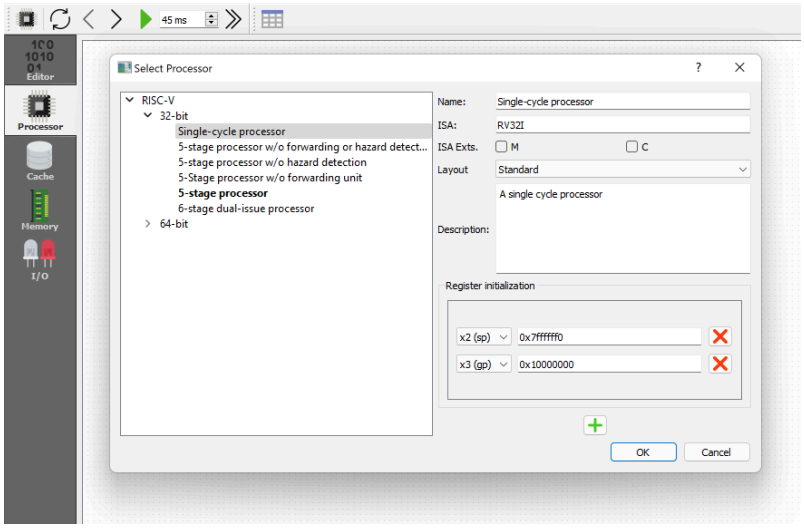


Abbildung 4: Ripes Prozessor Einstellungen

4.2 Analyse 1: Speicherverwaltung in Ripes

Ripes ermöglicht die Auswahl mehrerer Prozessoren (siehe Abbildung Abbildung 4). Obwohl der Single-Cycle Prozessor zwei verschiedene Speicherchips für Befehle und Daten trägt :

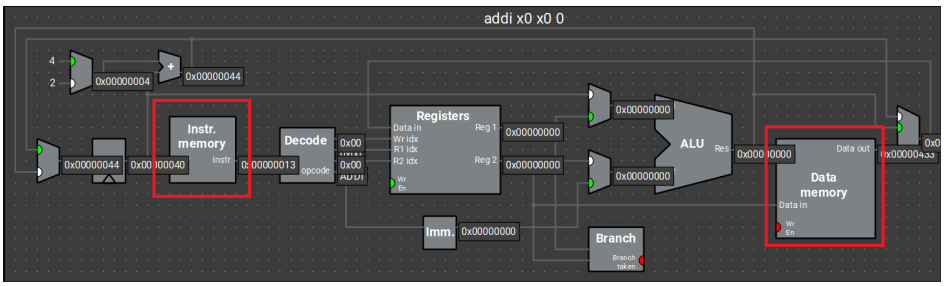
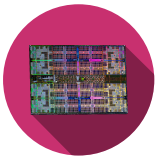


Abbildung 5: Single-Cycle Prozessor Speicher



Der Speicher wird als gemeinsam betrachtet, ebenso wie generell auf einem physischen Chip. 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.

```
# 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
```

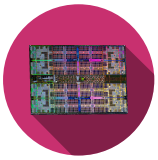
Die Schleife soll 0 an der Adresse 0, 1 an der Adresse 4, 2 an der Adresse 8 ... speichern. Aber in der vierten Iteration der Schleife wird die Anweisung `sw s0, 0(s1)` als `sw 3, 0(0x0C)` ausgeführt, wodurch die gleiche Anweisung durch 0x00000003 ersetzt wird, was einer `lb x0, 0(x0)`-Anweisung entspricht:

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 6: 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: Main Funktion

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 = 0;
}
```

```
addi sp sp -32 # stack for 8 items
sw s0 28 sp     # saves s0 in stack[0]
as it will use it and is a callee saved
register
addi s0 sp 32  # s0 points to stack[-1]
sw x0 -20 s0   # stack[4] = 0
addi x0 x0 0   # nop
lw s0 28 sp    # s0 = stack[0]
addi sp sp 32  # unstack 8 items
jalr x0 x1 0   # return
```

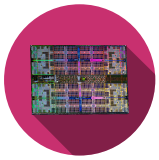
b)

```
void main() {
  while(1) {
    int a = 0;
  }
}
```

```
addi sp sp -32 # stack for 8 items
sw s0 28 sp     # saves s0 in stack[0]
as it will use it and is a callee saved
register
addi s0 sp 32  # s0 points to stack[-1]
sw x0 -20 s0   # stack[4] = 0
jal x0 -4      # jal to code
```

Was ist in den zwei Assembler-Codes beinhaltet. Wiso sind diese verschieden ?

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).



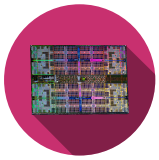
4.4 HEIRV-32

Den RISC-V Assembler und Disassembler welcher im Labo benutzt wird kann im Repository unter **`isa/heirv32-asm/`** für Windows (**`HEIRV32-ASM_1.2.0_windows_x86_64.exe`**) & Linux amd64 (**`HEIRV32-ASM_1.2.0_linux_x86_64`**) sowie macOS arm64 (**`HEIRV32-ASM_1.2.0_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`** ⇨ Assemblerdatei **`*.s`**

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

```
$cd isa/heirv32-asm
$./HEIRV32-ASM_1.2.0_macos_aarch64

usage: HEIRV32-ASM
  Assembler/Disassembler to support HEIRV32 ISA,
   See help below for usage : it auto-detects if the file is binary like (=> will
disassemble) or contains ASM instructions (=> will assemble).
  If no switches are given, will open a GUI to select the file and run with '-t = 'phb''
argument.

  Usage:
  heirv32 -f <ASMfile> [-t="phb"] [-i="HEIRV32" | "RV32I"]
  heirv32 -s <string> [(-t="hb" -of <outputPathName>)] [-i="HEIRV32" | "RV32I"]
  heirv32 -g [-t="phb"] [-i="HEIRV32" | "RV32I"]
  heirv32 -i <ISA>
  heirv32 -h | --help
  heirv32 [-g -t="phb" -i="HEIRV32"]

  Options:
  -h --help     Show this screen.
  -f            Input ASM file.
  -s            Input string.
  -g            Open GUI to select file to convert.
  -i             ISA to use ('HEIRV32' or 'RV32I' supported), default 'HEIRV32'. Can be
used alone to log the ISA specs.

  -t           Output type ('p' - print, 'h' - BRAM file, 'b' - bin File), default 'p'.
  -of          Output file path whe convertin an input string ('-s'), if 'h' and/or 'b'
are used for '-t'
```



4.4.1 Benutzung `HEIRV32-asm`

Abhänging vom Eingangsfiles wird automatisch assembliert oder disassembliert.

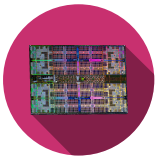
4.4.1.1 Ligne decommande (CLI)

Assembler ⇨ Binär

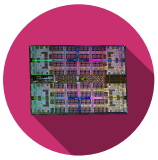
```
$HEIRV32-ASM_1.2.0_macos_aarch64 -f ./tassembly.s -t p

Generating file with output type p

File: ./tests/tassembly.s
Conversion ongoing
 * Using the ASSEMBLER
 * Found a value in code : 1500 at address 88 - do not let code try to run it !
 * Found a value in code : 1 at address 92 - do not let code try to run it !
 * Found a value in code : 255 at address 96 - do not let code try to run it !
 ** Found data / instructions: 25

------ p : Printing instructions list and write to file ------
 * Output file: _syn.txt
0x0000: addi x2 x0 5      => 0b00000000'01010000'00000001'00010011 => 0x00500113
0x0004: addi x3 x0 12     => 0b00000000'11000000'00000001'10010011 => 0x00c00193
0x0008: addi x7 x3 -9     => 0b11111111'01110001'10000011'10010011 => 0xff718393
0x000c: or   x4 x7 x2     => 0b00000000'00100011'11100010'00110011 => 0x0023e233
0x0010: and  x5 x3 x4     => 0b00000000'01000001'11110010'10110011 => 0x0041f2b3
0x0014: add  x5 x5 x4     => 0b00000000'01000010'10000010'10110011 => 0x004282b3
0x0018: beq  x5 x7 end    => 0b00000010'01110010'10001000'01100011 => 0x02728863
0x001c: slt  x4 x3 x4     => 0b00000000'01000001'10100010'00110011 => 0x0041a233
0x0020: beq  x4 x0 around => 0b00000000'00000010'00000100'01100011 => 0x00020463
0x0024: addi x5 x0 0      => 0b00000000'00000000'00000010'10010011 => 0x00000293
0x0028: slt  x4 x7 x2     => 0b00000000'00100011'10100010'00110011 => 0x0023a233
0x002c: add  x7 x4 x5     => 0b00000000'01010010'00000011'10110011 => 0x005203b3
0x0030: sub  x7 x7 x2     => 0b01000000'00100011'10000011'10110011 => 0x402383b3
0x0034: sw   x7 84(x3)    => 0b00000100'01110001'10101010'00100011 => 0x0471aa23
0x0038: lw   x2 88(x0)    => 0b00000101'10000000'00100001'00000011 => 0x05802103
0x003c: add  x9 x2 x5     => 0b00000000'01010001'00000100'10110011 => 0x005104b3
0x0040: jal  x3 end       => 0b00000000'10000000'00000001'11101111 => 0x008001ef
0x0044: addi x2 x0 1      => 0b00000000'00010000'00000001'00010011 => 0x00100113
0x0048: add  x2 x2 x9     => 0b00000000'10010001'00000001'00110011 => 0x00910133
0x004c: sw   x2 0x20(x3)  => 0b00000010'00100001'10100000'00100011 => 0x0221a023
0x0050: beq  x2 x2 main   => 0b11111010'00100001'00001000'11100011 => 0xfa2108e3
0x0054: jal  x3 main      => 0b11111010'11011111'11110001'11101111 => 0xfadff1ef
0x0058: 1500              => 0b00000000'00000000'00000101'11011100 => 0x000005dc
0x005c: 0b1               => 0b00000000'00000000'00000000'00000001 => 0x00000001
0x0060: 0xff              => 0b00000000'00000000'00000000'11111111 => 0x000000ff
```

4.4.1.2 Binär zu Assembler

```
$HEIRV32-ASM_1.2.0_macos_aarch64 -f ./tassembly.txt -t p

Generating file with output type p

File: ./tests/tassembly.txt
Conversion ongoing
 * Using the DISASSEMBLER
```



```
 * Finding instructions
  ** 00000000000000000000010111011100 no match - assuming value
  ** 00000000000000000000000000000001 no match - assuming value
  ** 00000000000000000000000011111111 no match - assuming value
 * Recomposing rd, rs and immediates
 * Adding labels
['addi x2 x0 5', 'addi x3 x0 12', 'addi x7 x3 -9', 'or x4 x7 x2', 'and x5 x3 x4', 'add
x5 x5 x4', 'beq x5 x7 48', 'slt x4 x3 x4', 'beq x4 x0 8', 'addi x5 x0 0', 'slt x4 x7
x2', 'add x7 x4 x5', 'sub x7 x7 x2', 'sw x7 84(x3)', 'lw x2 88(x0)', 'add x9 x2 x5', 'jal
x3 8', 'addi x2 x0 1', 'add x2 x2 x9', 'sw x2 32(x3)', 'beq x2 x2 -80', 'jal x3 -84',
'(value) 1500', '(value) 1', '(value) 255']
 Modifying beq x2 x2 -80 with label 0 - instr nb. 20
 Modifying jal x3 -84 with label 0 - instr nb. 21
 Modifying beq x4 x0 8 with label 1 - instr nb. 8
 Modifying beq x5 x7 48 with label 2 - instr nb. 6
 Modifying jal x3 8 with label 2 - instr nb. 16
 * Recomposed labels / instructions: 28

------ p : Printing instructions list ------
lb0:
addi x2 x0 5
addi x3 x0 12
addi x7 x3 -9
or x4 x7 x2
and x5 x3 x4
add x5 x5 x4
beq x5 x7 lb2
slt x4 x3 x4
beq x4 x0 lb1
addi x5 x0 0

lb1:
slt x4 x7 x2
add x7 x4 x5
sub x7 x7 x2
sw x7 84(x3)
lw x2 88(x0)
add x9 x2 x5
jal x3 lb2
addi x2 x0 1

lb2:
add x2 x2 x9
sw x2 32(x3)
beq x2 x2 lb0
jal x3 lb0
(value) 1500
(value) 1
(value) 255
```



4.4.1.3 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
3. Drücken auf den Knopf **`Next`**
2. Auswahl des Eingabefiles - Entweder **`*.s`**, **`*.asm`** oder **`*.bin`**

Danach finden Sie die Ausgabe im selben Ordner und im Terminal die Ausgabe des Programmes.

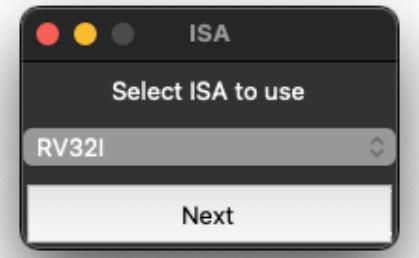


Abbildung 7: GUI des Programmes HEIRV32-ASM

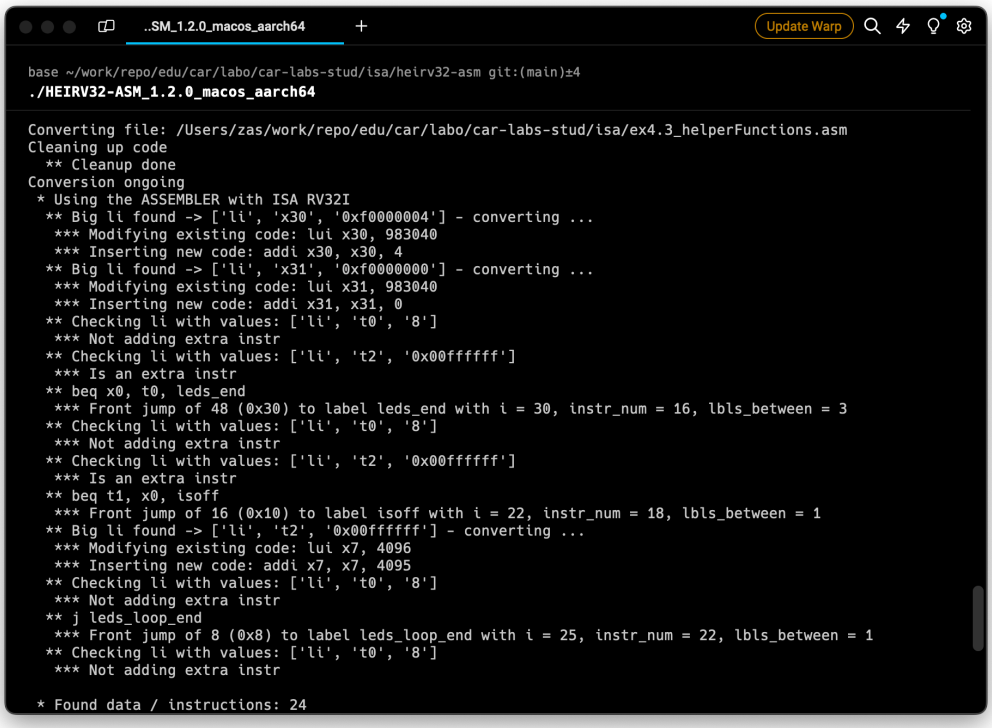
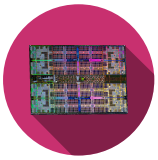


Abbildung 8: Ausgabe des Programms heirv32-asm



4.4.2 Analyse 3 : Assembly & Disassembly

- 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:

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

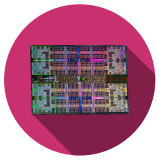
  Eine Datei **.txt** wird erstellt, die eine für Menschen lesbare hexadezimale Version der erzeugten Datei **.bin** ist.

- Nun dekompilieren Sie die Datei **.txt** auf die gleiche Weise, um eine Assemblerdatei neu zu erstellen:

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

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

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



4.5 Reverse Engineering

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:

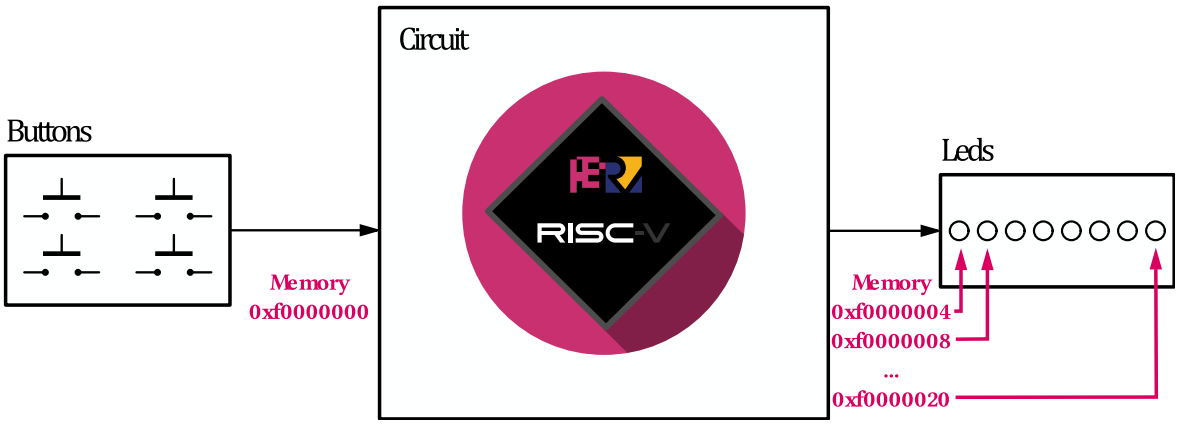
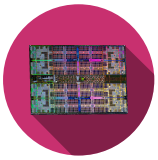


Abbildung 9: Modifizierter RISC-V-Prozessor

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:

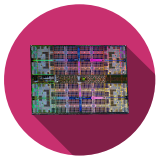


```
setup:
    # led is sw xx, offBy4(x30) with xx loaded as 0x00rrggbb
    li x30, 0xf0000004 # base address for leds
    li x31, 0xf0000000 # base address for buttons, one register

    # 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, 0x00ff00ff
    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
```

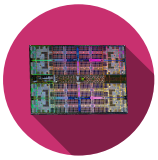
Leider ist Ihr Kollege heute abwesend und hat Ihnen keine Kopie des bereits geschriebenen Codes hinterlassen (*denken Sie daran, Git zu lernen !*). Ausserdem gab es einige Bugs:

- 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.

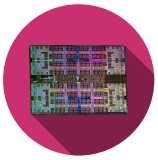


Sie erinnern sich, dass der zuletzt geschriebene Code auf Ihr Board RISC-V geflasht wurde, und da der Chip nicht gegen Rücklesen geschützt war, können Sie den Code durch Zugriff auf den JTAG-Bus extrahieren:



```
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



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:

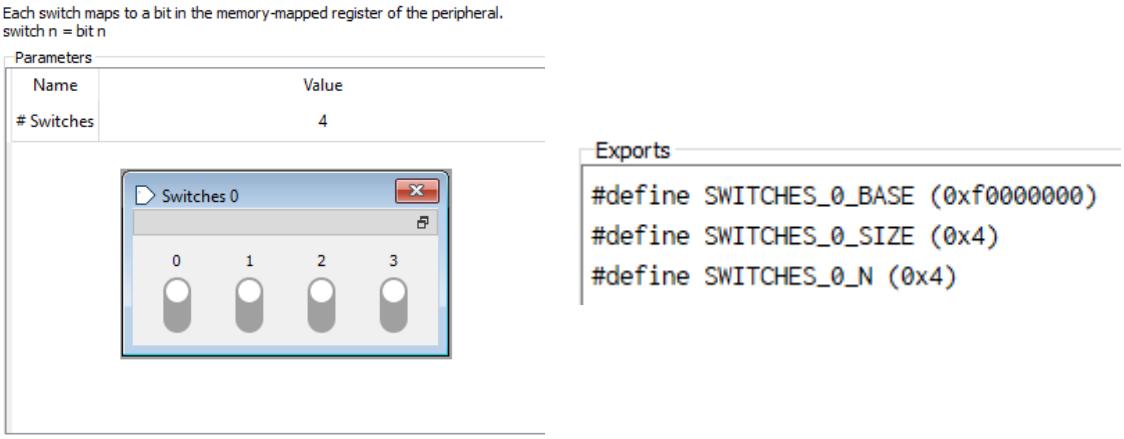


Abbildung 10: Ripes - Schalterkonfiguration

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:

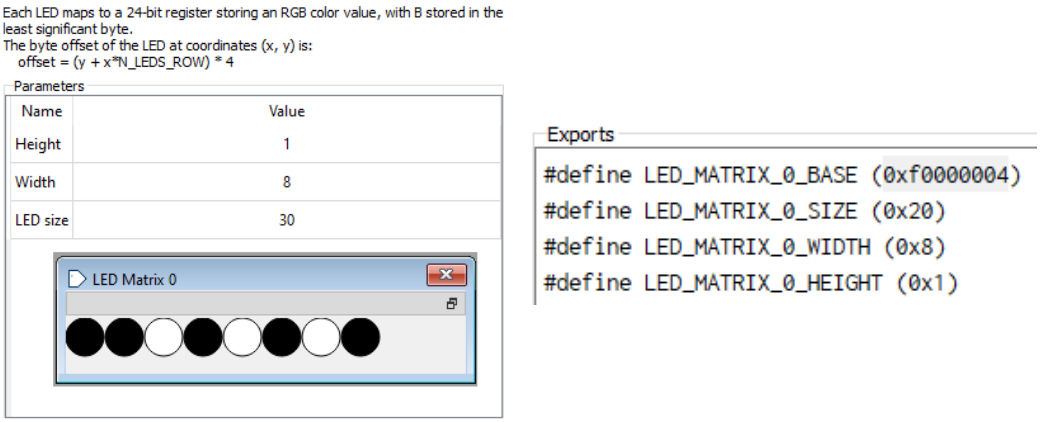
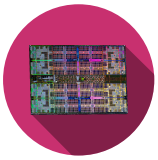


Abbildung 11: Ripes - Led-Konfiguration

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

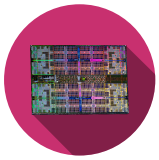
Fügen Sie zuerst die Schalter und dann die Leds hinzu, um die richtige Basisadresse der Module zu erhalten.



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 genannten Probleme (4 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 (kopiert), 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).



Bibliographie