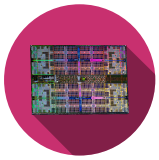
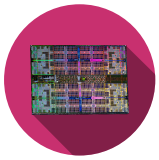


RISC-V ISA

Solutions pour étudiants
Labor Architecture des ordinateurs

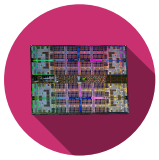
Contenu



1 Objectifs

Ce laboratoire est divisé en plusieurs blocs réalisés sur plusieurs semaines. Le but est de se familiariser avec le langage assembleur du RISC-V.

- La partie Chapitre 2 du laboratoire porte sur les instructions individuelles.
- La partie Chapitre 3 du laboratoire nous amène à la programmation impérative, respectivement aux boucles et aux fonctions.
- La dernière partie Chapitre 4 du laboratoire est consacrée au travail sur le compilateur : l'assemblage et le désassemblage d'un programme (passage d'un langage haut-niveau à du code machine et inversement).



2 Instructions

Dans cette première partie du laboratoire, nous allons travailler avec l'interpréteur RISC-V sur le site https://course.hevs.io/car/riscv-interpreter/, voir illustration Fig. 1. Celui-ci permet aux registres d'écrire en mémoire et d'exécuter le code étape par étape. Ces outils en ligne vous aideront à résoudre et à contrôler les tâches.

Seules les instructions listées sur cette page sous **Supported Instructions** peuvent être utilisées. Les pseudo-instructions ne sont pas prises en charge.

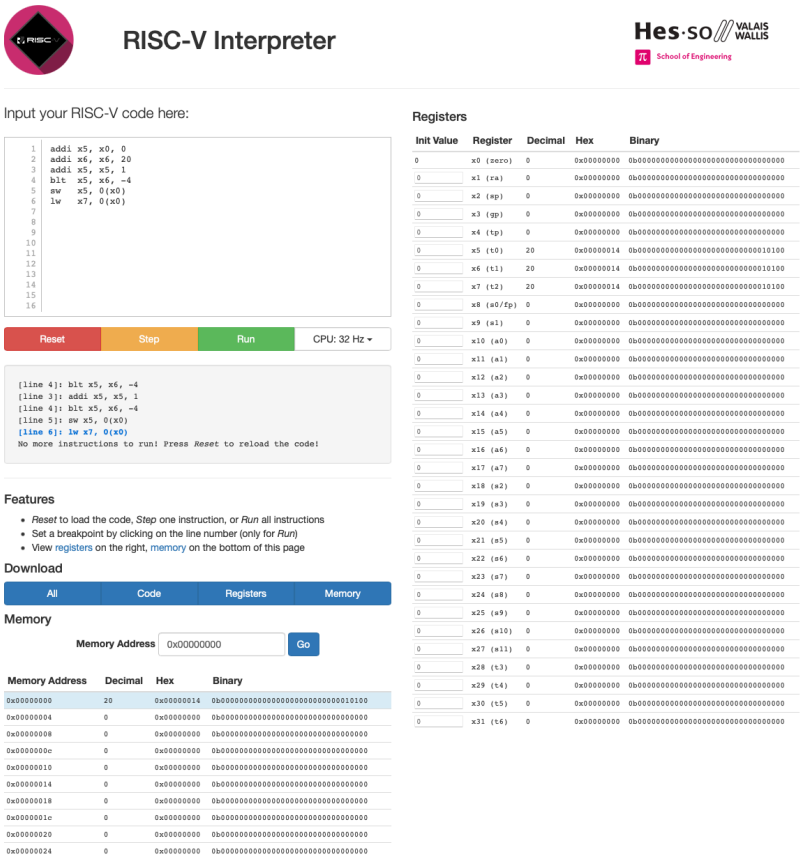
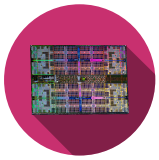


Fig. 1. – Interpréteur RISC-V en ligne

Attention aux types des variables !

- Le type **int** est considéré de taille 32 bits signé.
- Le type **unsigned int** est considéré de taille 32 bits non-signé.
- Si il est suivi d'un nombre (ex: **int16_t**), cela signifie que la variable est sur x bits (ici 16). Si précédé d'un **u**, il est non-signé.

uint8_t est donc un byte non-signé, tandis que **int8_t** est un byte signé.



2.1 Constantes (Immediate)

Écrire le code assembleur RV32i pour les instructions suivantes :

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 Guide étudiants

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 Calculs de base

Écrire le code assembleur RV32i pour les instructions suivantes :

a)

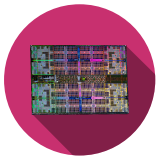
```
int b = 1;
int c = 2;
int d = 5;
a = b + c - d;
```

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 Guide étudiants

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 Accès mémoire

Écrire le code assembleur RV32i pour les instructions suivantes :

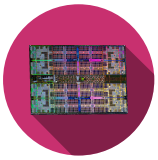
a)

```
# 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 Guide étudiants



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)

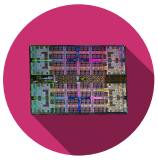
```
# mem[5] = 42;
# s7 = a

# 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 Algorithmes

Écrire le code assembleur RV32i pour les algorithmes suivants, en utilisant uniquement les instructions de base sans *loops*. *conditionals* ainsi que *branches* :

1. Dans un système, il peut être nécessaire d'attendre sans rien faire, sans modifier l'état actuel du système (pas d'accès mémoire, pas de modification des registres). Cette opération est communément appelée **NOP** (**NO Op**eration).
   1. Proposez une instruction pour ce faire.
   2. Testez qu'aucun registre ni valeur mémoire n'est effectivement modifié.
2. Calculer les 10 premiers nombres de Fibonacci. Enregistrer chaque nombre dans un registre différent.

2.4.1.1 Guide étudiants

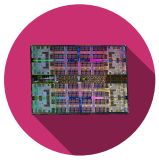


NOP)

```
# 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 Programmation impérative

3.1 Branching

3.1.1 If / else

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

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

3.1.2 Switch case

```c
int a, b;

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

3.1.2.1 Guide étudiants

If / else)

```asm
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:
  # ...
```

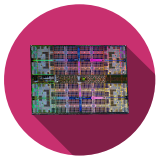
Switch case)

```asm
# 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

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

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

// B
int a = 10;
```



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

3.2.1.1 Guide étudiants

```
// 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

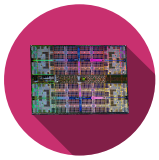
a)

```
unsigned int a = 0, i;

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;
}
```

3.2.2.1 Guide étudiants

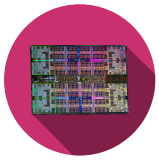


a)

```
# 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 Fonctions

Bien qu'un programmeur seul puisse agencer son code comme il l'entend, certaines conventions ont été établies pour permettre l'interopérabilité de plusieurs sources de code entre elles.

La fonction en appelant une autre est nommée **caller** et la fonction appelée nommée **callee**. Elles doivent être synchronisées sur la façon dont sont passés les arguments, quels registres doivent être conservés ...

Les notions suivantes sont les règles essentielles:

- Les registres **a0** à **a7** permettent de passer des arguments. Si plus sont nécessaires, ils sont passés sur le stack.
- Le résultat est placé dans le registre **a0**.
- Le **caller** enregistre l'addresse de retour (PC + 4) générée par l'instruction **jal** dans le registre **ra**.
- Le **callee** ne doit pas réécrire l'adresse de retour, le stack ou encore les registres sauvés sXX. Si elles sont utilisées, l'espace doit être réservé sur le stack et ces informations doivent être stockées puis restaurées.

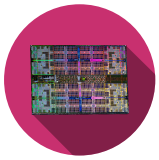
a)

```
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 Guide étudiants

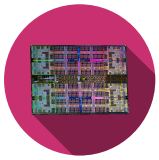


a)

```
# 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 Algorithmes

3.4.1 Fibonacci par récursivité

La notion de récursivité implique qu'une fonction se rappelle elle-même pour calculer un résultat, de la même manière qu'elle appellerait une autre fonction.

Le but est de calculer la suite de fibonacci d'un nombre entier non-signé selon l'algorithme suivant:

```
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));
  }
}
```

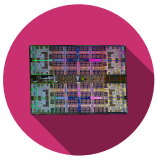
- Implémenter la fonction donnée avec RV32I
- Tester et valider la fonction
- Quelles sont les valeurs n supportées ?

Il est possible d'accélérer grandement le calcul en utilisant la mémoïsation. Le principe est de réserver de la place mémoire et de stocker les résultats déjà connus pour chaque n calculé et de réutiliser cette valeur directement.

- Implémenter la fonction donnée. RV32IM est conseillé ici.
- Tester et valider la fonction
- Comparer les algorithmes

3.4.1.1 Guide étudiants

```
# 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 Nombre au carré

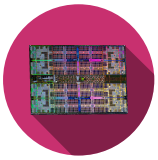
Un nombre mis au carré n'est autre que la multiplication dudit nombre par lui-même. Mais ici, nous ne travaillons qu'avec le set d'instruction RV32I:

- Proposer une fonction capable de calculer $n * n$ par une boucle d'additions.
- Tester avec n = 5, 10 et 20.
- Généraliser la fonction pour pouvoir lui fournir deux paramètres en calculant $i * j$.
- Tester avec i = 5, j = 100. Tester avec i = 100, j = 5. Que constatez-vous ?
- Optimiser la fonction pour que l'ordre des opérandes n'influe pas sur le temps de calcul.

Cet algorithme est de complexité $O(n)$. Il est possible de le réduire à une complexité de $O(\log_2(n))$ en utilisant intelligement les additions et shifts. Ce dernier s'appelle **fast multiplication**:

- Proposer une fonction optimisée.

3.4.2.1 Guide étudiants

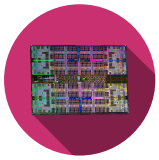


Non optimized adds $a + a \Rightarrow b$ times.

```
# 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 Setup Ripes

Dans ce laboratoire, vous pouvez continuer à utiliser l'interpréteur en ligne - Fig. 1. En outre, le programme **Ripes** est également disponible, voir l'illustration Fig. 2.

Il supporte tout le set RV32I, les labels ainsi que les pseudo-instructions.

Vous devez d'abord le télécharger et le configurer.

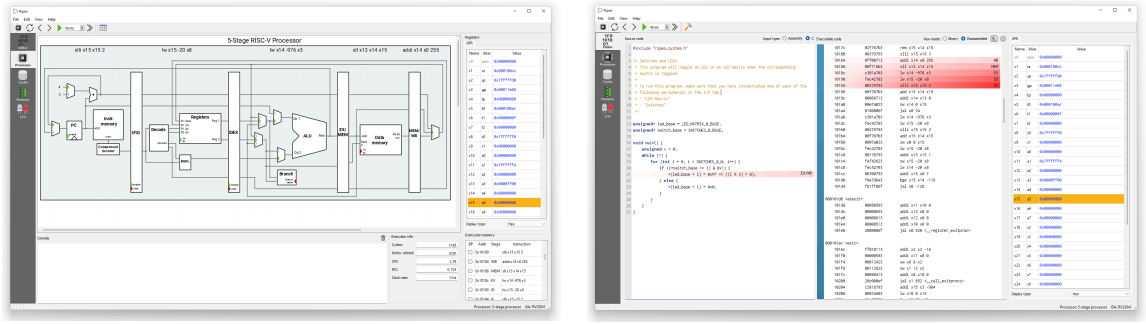


Fig. 2. – Interface graphique de développement de Ripes

> Sur les PCs de laboratoire, Ripes et le compilateur **gcc** énoncés ci-après sont trouvables sous **C:/eda/RiscV**.

1. Téléchargez la dernière version du programme pour votre plateforme en suivant le lien : https://github.com/mortbopet/Ripes/releases.
2. Téléchargez la version suivante de la chaîne d'outils RISC-V -GNU pour votre plateforme en suivant le lien : https://github.com/sifive/freedom-tools/releases/tag/v2020.04.0-Toolchain.Only
3. Décompressez les deux et copiez le dossier RISC-V -GNU-Toolchain dans le dossier Ripes.
4. Lancez Ripes et configurez les paramètres du compilateur dans : **Edit** → **Settings**. Pour cela, vous devez sélectionner le fichier **`/riscv64-unknown-elf-gcc-8.3.0.exe`**.

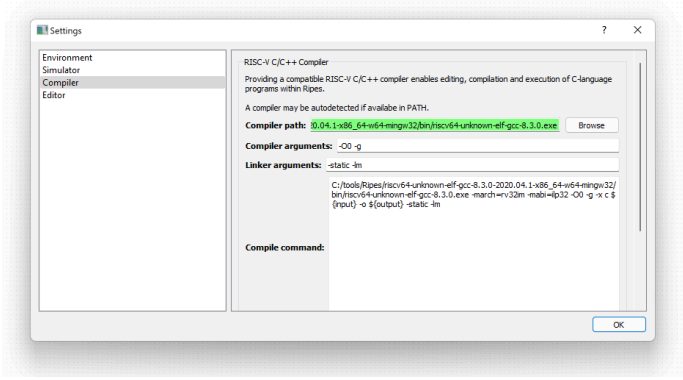


Fig. 3. – Paramètres de la chaîne d'outils de Ripes

5. Dans les paramètres du processeur, sélectionnez le processeur monocycle RISC-V → 32-bit → sans extensions ISA.

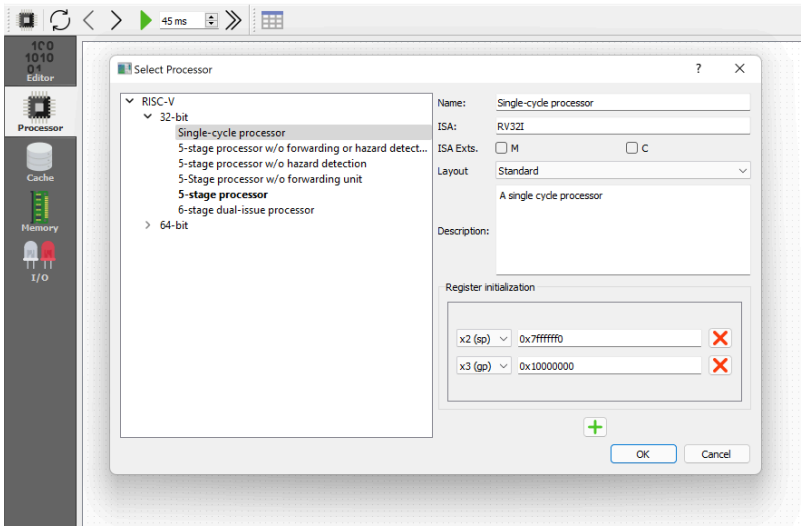


Fig. 4. – Paramètres du processeur de Ripes

4.1.1 Gestion de la mémoire dans Ripes

Ripes permet de sélectionner plusieurs processeurs (voir figure Fig. 4). Malgré que le processeur single-cycle arbore deux puces de mémoire différentes pour les instructions et les données :

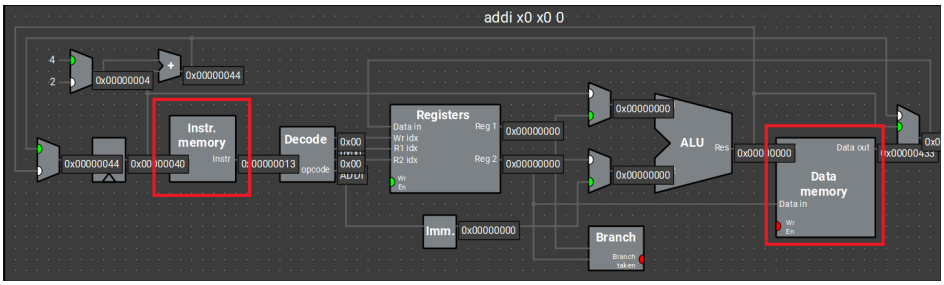
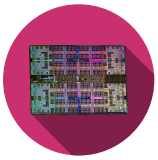


Fig. 5. – Mémoires processeur single-cycle

La mémoire est considérée comme commune, de même que généralement sur une puce physique. Cela veut dire qu'il est possible de **réécrire votre propre code par inadvertance**. Pour l'illustrer :

```
# 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
```



La boucle est censée enregistrer 0 à l'adresse 0, 1 à l'adresse 4, 2 à l'adresse 8 ... Mais lors de la quatrième itération de la loop, l'instruction `sw s0, 0(s1)` est exécutée telle que `sw 3, 0(0x0C)`, ce qui remplace cette même instruction par 0x00000003, correspondant à une instruction `lb x0, 0(x0)`:

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

Fig. 6. – Réécriture mémoire

Le code a donc été modifié et ne correspond plus au but initial. Pour éviter de tels problèmes, il faut travailler avec la mémoire au travers du **stack pointer**, comme lors d'appels de fonctions :

- Réservez de la place en mémoire en décrémentant **sp** du nombre de bytes nécessaires (pour l'exemple, nous aurions $10 * 4$ bytes)
- Travaillez avec la mémoire réservée UNIQUEMENT

4.2 Fonction main

Dans le programme Ripes montré sous Chapitre 4.1, compilez les codes C suivants et cherchez dans le code généré les instructions assembleur correspondantes. Que signifient les différentes instructions dans `main:` ?

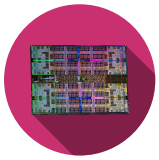
a)

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

b)

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

Utilisez Ripes pour réaliser et tester vos laboratoire mais aussi votre série d'exercices. Il est capable de réagir comme un vrai processeur, vous montrant des comportements auxquels on peut ne pas s'attendre (ex. avec la réécriture de son propre programme en gérant mal les instructions L-S - Chapitre 4.1.1).



4.3 HEIRV-32

L'assembleur et désassembleur RISC-V utilisé dans ce labo peut être trouvé dans le repo sous **`/heirv32-asm/`** pour Windows & Linux amd64 et macOS ARM64. Cet outil vous permet de transformer le code assembleur RV32i ou HEIRV32 en code binaire (Assembly) et de retransformer le code binaire en code assembleur (Disassembly).

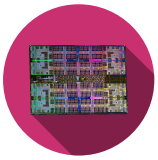
```
$ heirv32-asm --help

usage: HEIRV32-ASM
    Assembler/Disassembler to support HEIRV32 ISA,
    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), dflt 'p'.
    -of           Output file path when converting a string ('-s'), if 'h' and/or 'b'
                  are used for '-t'.
```

4.3.1 Commandes HEIRV32-asm

Assemblage ou désassemblage automatique en fonction des fichiers d'entrée.

4.3.1.1 Assembleur vers binaire

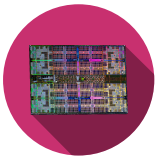
```
$HEIRV32-ASM_1.1.3_Darwin_ARM -f ./tests/tassembly.s -t p

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
```



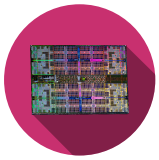
```
0x0008: addi x7 x3 -9     => 0b11111111'01110001'10000011'10010011 => 0xff718393
0x000c: or   x4 x7 x2     => 0b00000000'00100011'11100010'00110011 => 0x0023e233
0x0010: and  x5 x3 x4     => 0b00000000'01000001'11110010'10110011 => 0x0041f2b3
0x0014: add  x5 x5 x4     => 0b00000000'01000010'10000010'10110011 => 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
```

4.3.1.2 Binaire vers assembleur

```
./dist/HEIRV32-ASM_1.1.3_Darwin_ARM -f ./tests/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
```



4.3.2 Prise en main

1. Enregistrez le code suivant sous un fichier **.s**:

```
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
```

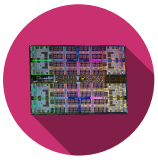
2. Compilez-le à l'aide de HEIRV32-ASM avec les sorties **print** et **bin** activées :

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

   Un fichier **.txt** est créé, qui est une version hexadécimale lisible par les humains du fichier **.bin** généré.
3. Maintenant, décompilez-le fichier **.txt** de la même manière pour recréer un fichier assembleur :

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

   Cela devrait créer un fichier **_disassembly.s**.
4. Qu'est-ce qui change entre les deux codes ? Donnez au moins 3 points.
5. A quoi peut servir l'instruction `jal ra myLabel` dans ce contexte ?



4.4 Reverse Engineering

Vous travaillez sur un projet en binôme sur un processeur RISC-V quelque peu modifié.

En effet, pour se simplifier la vie et communiquer avec le monde extérieur, ce dernier peut lire l'état de 4 boutons reliés au processeur en lisant l'adresse mémoire **0xf0000000**, et allumer 8 leds en écrivant aux adresses mémoire **0xf0000004** (led 0) à **0xf0000020** (led 7):

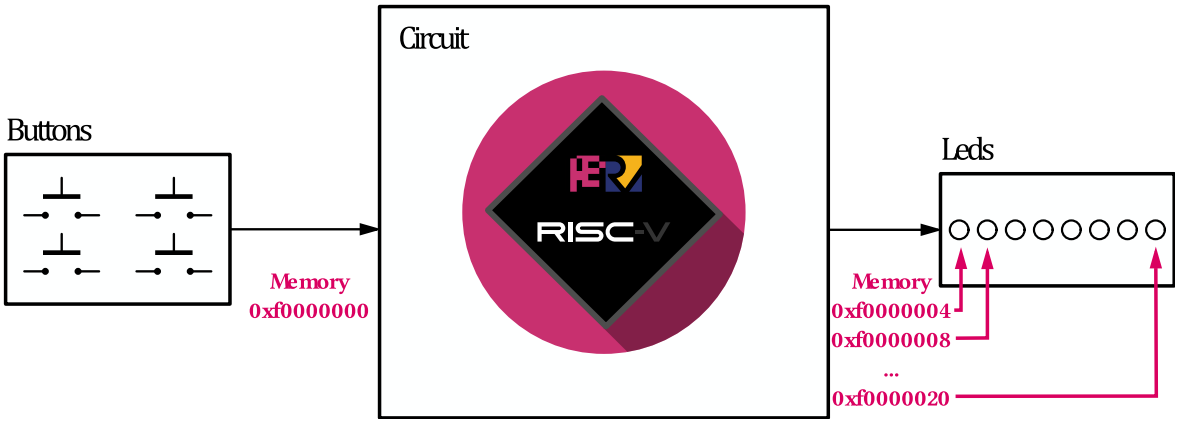
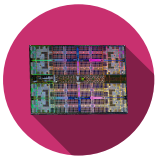


Fig. 7. – Processeur RISC-V modifié

Le premier laboratoire consistait à faire clignoter les leds selon les boutons appuyés:

- Aucun bouton: les 8 leds sont allumées selon le pattern 0xAA
- Le bouton 0 est à ‹ 1 ›: les 8 leds clignotent à une fréquence raisonnable (le clignotement est visible)

Pour simplifier l'accès au bouton et aux leds, le code suivant vous était donné:



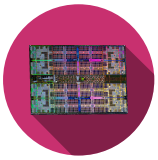
```
setup:
    # led is sw xx, offBy4(x30) with xx loaded as 0x00rrggbb
    li x30, 0xf0000004 # base address for leds
    li x31, 0xf0000000 # base address for buttons, one register

    # 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
```

Malheureusement, votre collègue est absent aujourd'hui et ne vous a pas laissé une copie du code déjà écrit (*pensez à apprendre Git !*). En plus, il y'avait quelques bugs:

- Aucun bouton: les leds s'allument selon le pattern 0x2A
- Le bouton 0 est à ‹ 1 ›: seules 7 leds clignotent, et à une fréquence qui semble bien trop élevée
- Un des autres bouton est appuyé: même comportement que si le bouton 0 était appuyé

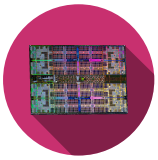
Un problème matériel peut être exclu.

Vous vous souvenez que le dernier code écrit avait été flashé sur votre board RISC-V et, vu que le chip n'avait pas été protégé contre la relecture, vous arrivez à l'extraire par accès au bus JTAG, ce qui vous a donné:



```
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. Décompilez votre code
2. Identifiez et séparez le code donné du reste
3. Comprenez l'utilisation des fonctions données



Pour simuler le système, sous Ripes, ouvrez l'onglet I/O:

- Double-cliquez sur Switches, sélectionnez le widget créé et configurez-le tel que:

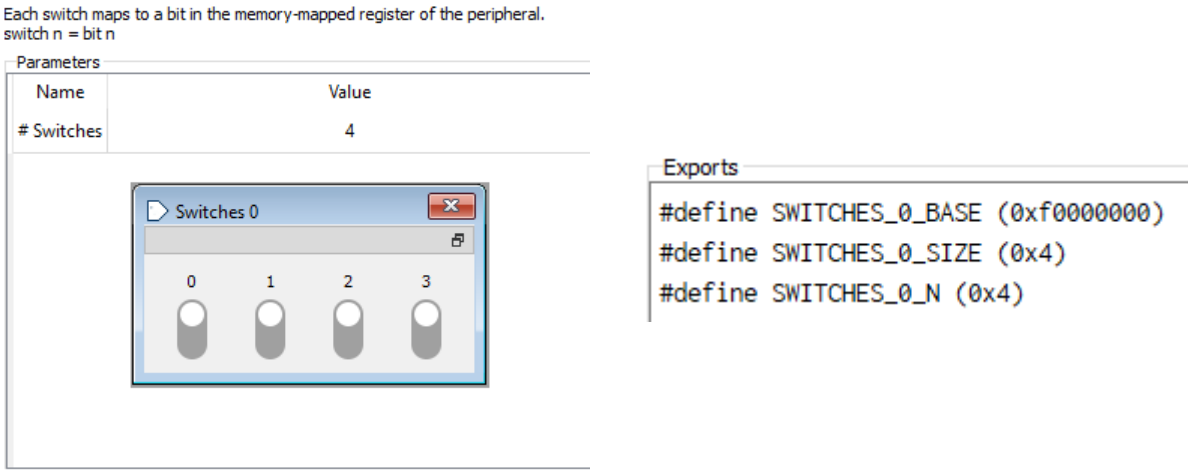


Fig. 8. – Ripes - configuration des interrupteurs

Contrôlez que **Exports** contienne les mêmes informations que sur l'image donnée après réglage du module.

- Double-cliquez sur LED Matrix, sélectionnez le widget créé et configurez-le tel que:

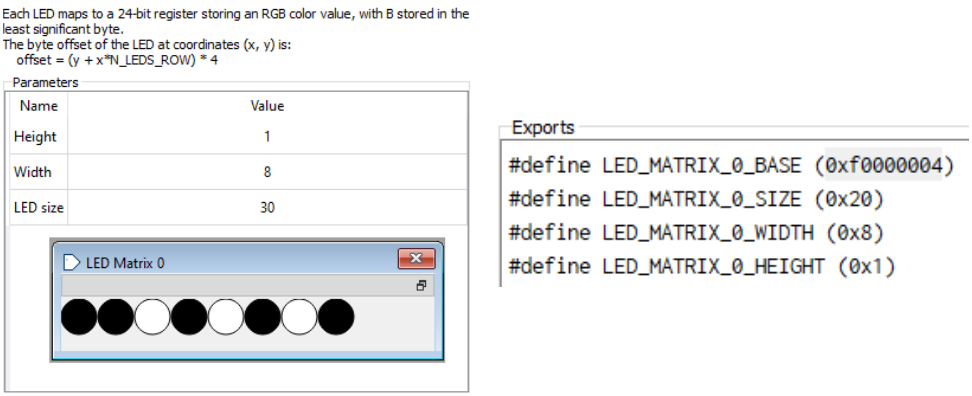
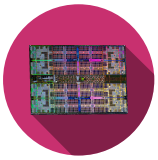


Fig. 9. – Ripes - configuration des leds

Contrôlez que **Exports** contienne les mêmes informations que sur l'image donnée après réglage du module.

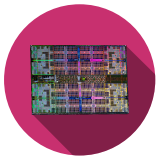
Ajoutez d'abord les switchs puis les leds pour obtenir la bonne addresse de base des modules.



1. Chargez votre code assembleur décompilé dans **Editor**
2. Appuyez sur F8 (ou l'icône >>) et constatez que le circuit fonctionne selon les problèmes énoncés en interagissant sous **I/O** avec les interrupteurs.
3. Fixez les problèmes cités (4 problèmes)
4. Testez votre circuit sur Ripes

La clock ne pouvant être réglée de manière précise, la fréquence de clignotement des leds se fait sans calcul. Corriger le code pour obtenir environ 2 Hz.

Ce que vous venez d'accomplir s'apparente à du hardware hacking : un code est dumpé (copié), analysé, modifié, puis réinjecté dans un système pour l'adapter à son envie. Bien sûr, il n'est pas toujours si simple de copier et réinjecter un code. Des méthodes parfois exotiques ont vu le jour (voir le Kamikaze Hack de la Xbox 360).



Bibliographie