Kathleen16 Mk. 1

Kamanda Aubin
Placé Louka

Nantes Université

L3 Miage Gestion
Nantes Université
2022-2023

Introduction

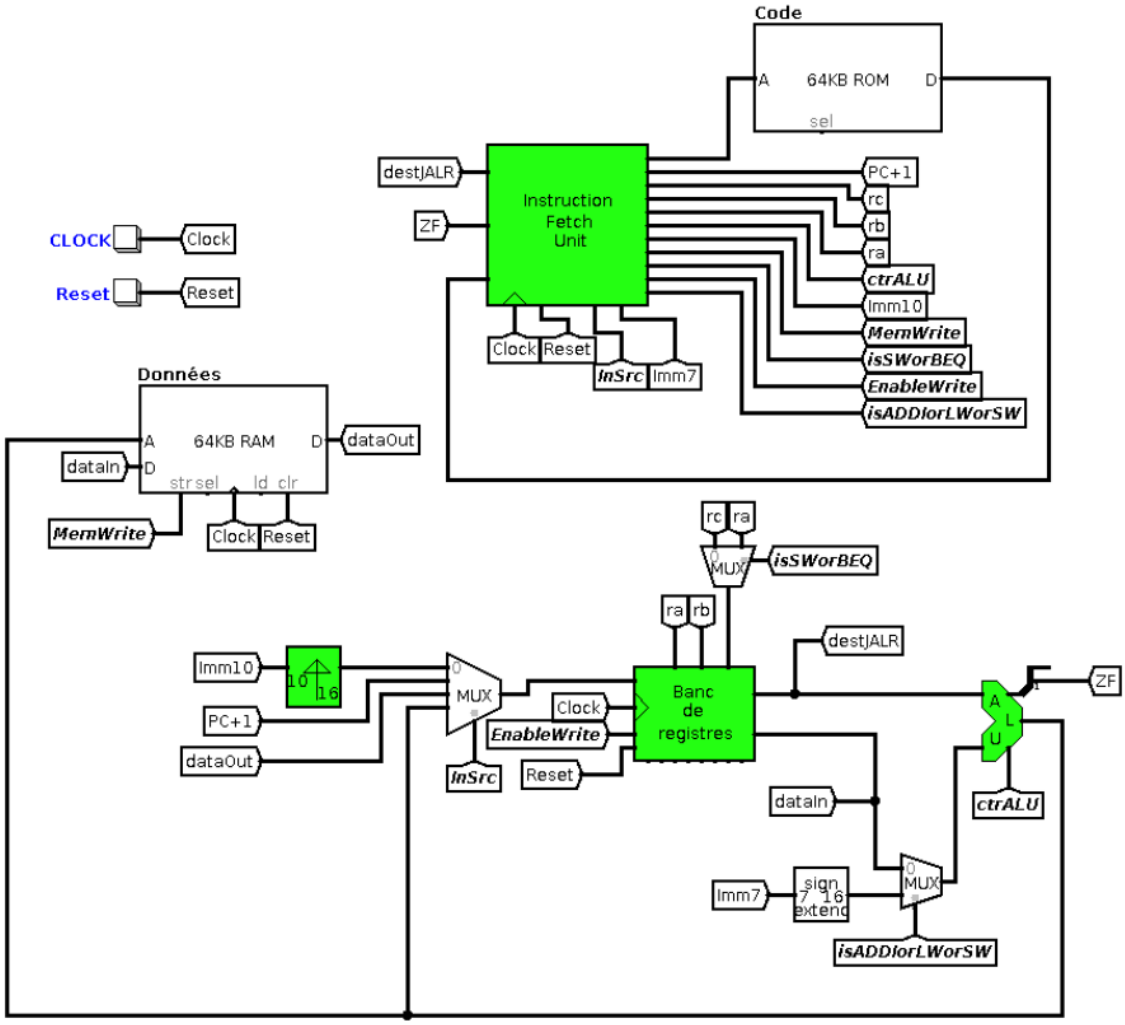
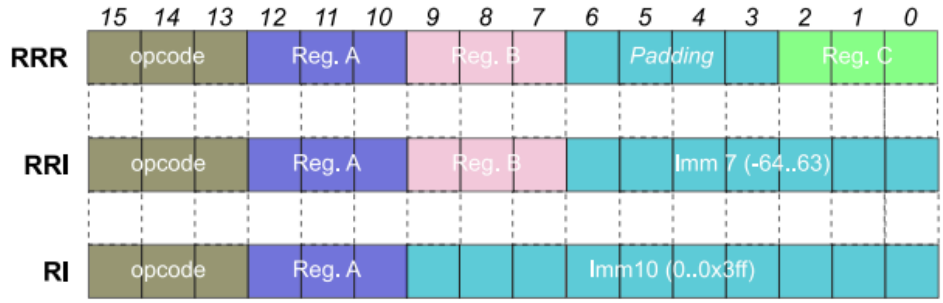


FIGURE 1 – Circuits du *Kathleen16 Mk. 1.*

Le Kathleen16 Mk. 1 est un processeur 16 bits nommé ainsi en l'honneur de Kathleen Booth, l'auteure du premier langage d'assemblage, décédée le 29 octobre 2022.
Le processeur Kathleen16 possède huit registres de 16 bits, $r0, $r1, . . .$r7. Comme sur le processeur MIPS, le registre $r0 contient toujours la valeur 0, quelle que soit la valeur que l'on puisse tenter d'y stocker. Le Kathleen16 utilise une architecture de Harvard : les instructions sont stockées dans une ROM ; les données sont stockées dans une RAM séparée. L'adresse de l'instruction courante se trouve dans le registre PC. L'adressage de la RAM et de la ROM se fait par mots de 16 bits.

Les instructions sont stockées en mémoire sur 16 bits suivant trois types de formats :



La table 1 présente les huit instructions natives du processeur.

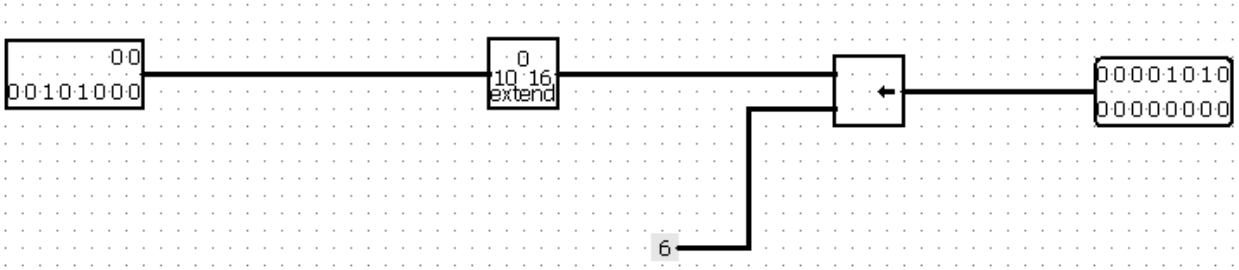
TABLE 1 – Format des instructions natives du *Kathleen16*.

Instruction	Opcode	Format	Action
`add $rd, $rs1, $rs2`	000	RRR	$rd ← $rs1 + $rs2
`addi $rd, $rs, Imm7`	001	RRI	$rd ← $rs + Imm7
`nand $rd, $rs1, $rs2`	010	RRR	$rd ← $rs1 ∧ $rs2
`lui $rd, Imm10`	011	RI	$rd ← (Imm10 << 6)
`sw $rs1, $rs2, Imm7`	100	RRI	RAM[$rs2 + Imm7] ← $rs1
`lw $rd, $rs, Imm7`	101	RRI	$rd ← RAM[$rs + Imm7]
`beq $rs1, $rs2, Imm7`	110	RRI	saut à PC+1+Imm7 si $rs1 == $rs2
`jalr $rd, $rs`	111	RRI	$rd ← PC + 1 ; saut à $rs

Grandes étapes

I. Construction du Kathleen16

a) L'extenseur 10↑16



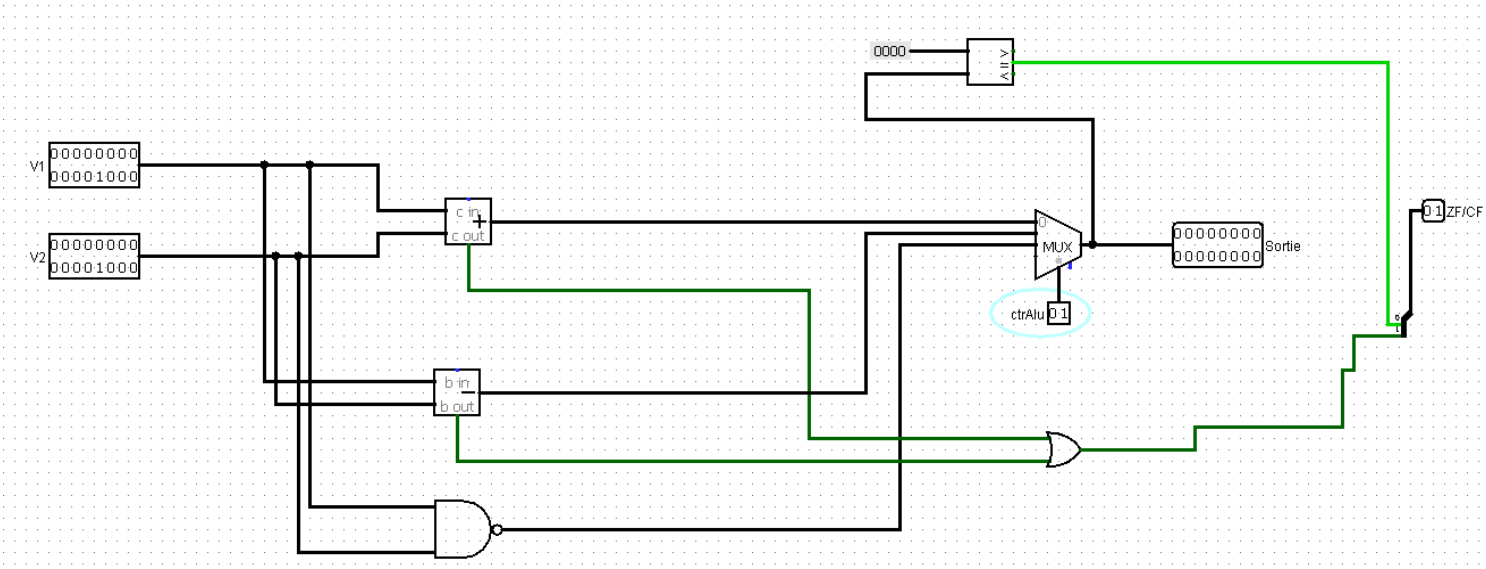
L'extenseur va prendre en entrée une valeur sur 10 bits (ici, 00.0010.1000), il va tout d'abord étendre la valeur sur 16 bits (0000.0000.0010.1000) puis décale cette valeur sur 16 bits de 6 positions vers la gauche (0000.1010.0000.0000).

b) L'Unité Arithmétique Logique (UAL)

L'UAL va recevoir deux valeurs en entrée et en fonction de la valeur de ctrALU :

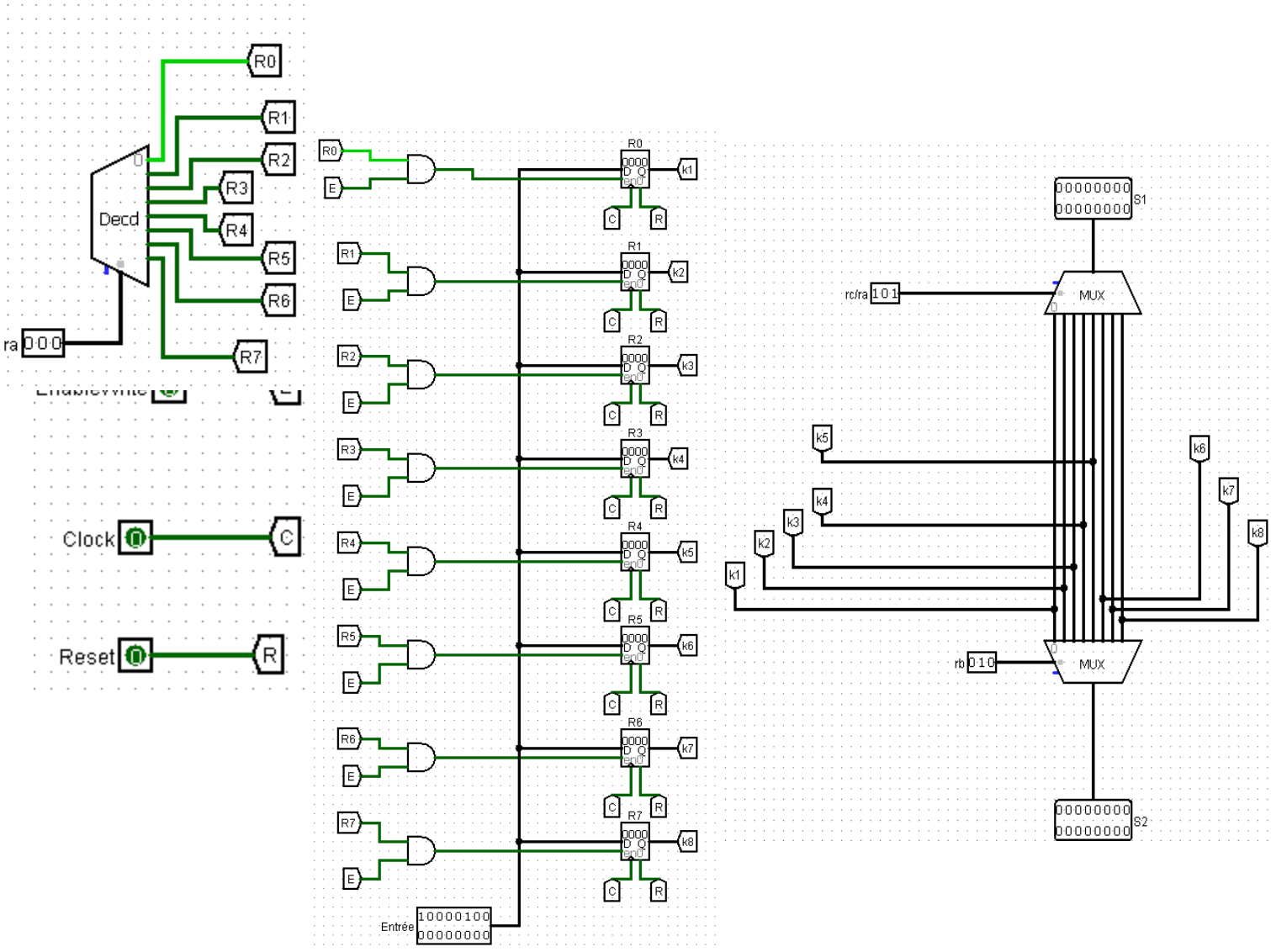
- 00 → addition via le adder (v1 + v2)
- 01 → soustraction via le substractor (v1 - v2)
- 10 → et-non via la porte NAND ($\overline{v_1 \cdot v_2}$)

Le multiplexer laisser sortir la valeur du résultat en fonction de l'opération.
Les indicateurs ZF (Zero Flag) ou CF (Carry Flag) sont si le résultat est égal à 0 ou si la dernière retenue dépasse 16 bits.



c) Le banc de registres

En fonction de de l'entrée ra, à l'aide d' un décodeur on va choisir l'entrée du registre (000 → R0, 001 → R1, 010 → R2,.... R7). Les entrées EnableWrite et les registres R0 à R7 sont reliés par une porte AND afin d'autoriser l'écriture dans le registre de numéro ra. L'horloge (Clock) noté C est relié à tous les registres ainsi que le signal pour forcer à 0 tous les registres (Reset) noté R. On a également deux sortie S1 ou S2 en fonction de la valeur de rc/ra ou rb



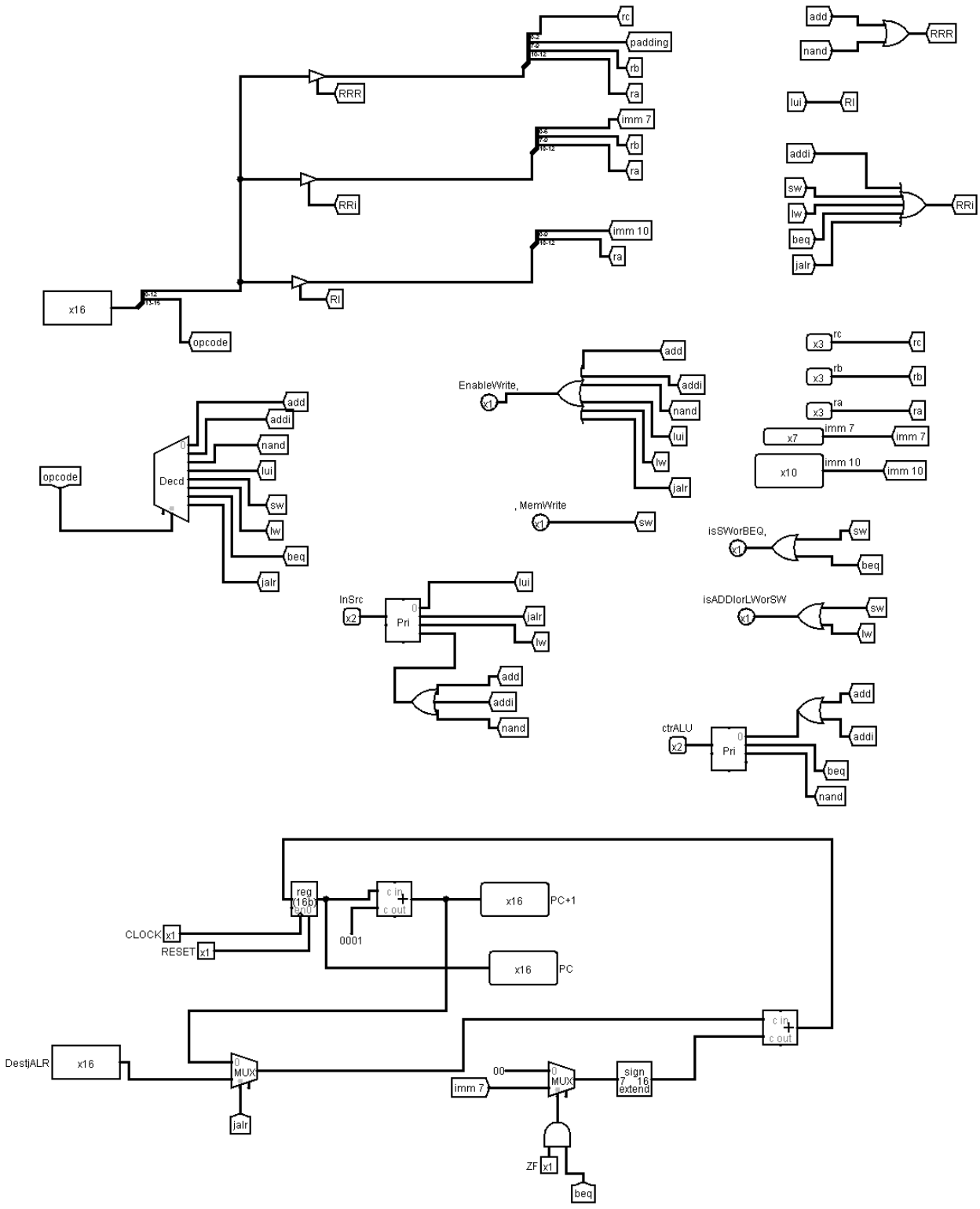
d) L' IFU

Ce circuit est responsable de la mise à jour du registre PC (Program Counter) et de l'extraction et du découpage de l'instruction courante à partir de la ROM.

On est parti sur la première entrée sur 16 bits qui prend les instructions en suivant les trois types de format RRR, RRI, RI et on a divisé ces 16 bits selon les bits que chaque instructions utilise. Dans le cas où l'opcode reste le même sur tous les formats, il utilise les 3 premier bits sur 16 bits; les 13 bits qui restent vont être divisés sur chaque instruction selon le type de format et chaque format on lui affecte les instructions qu'il utilise.

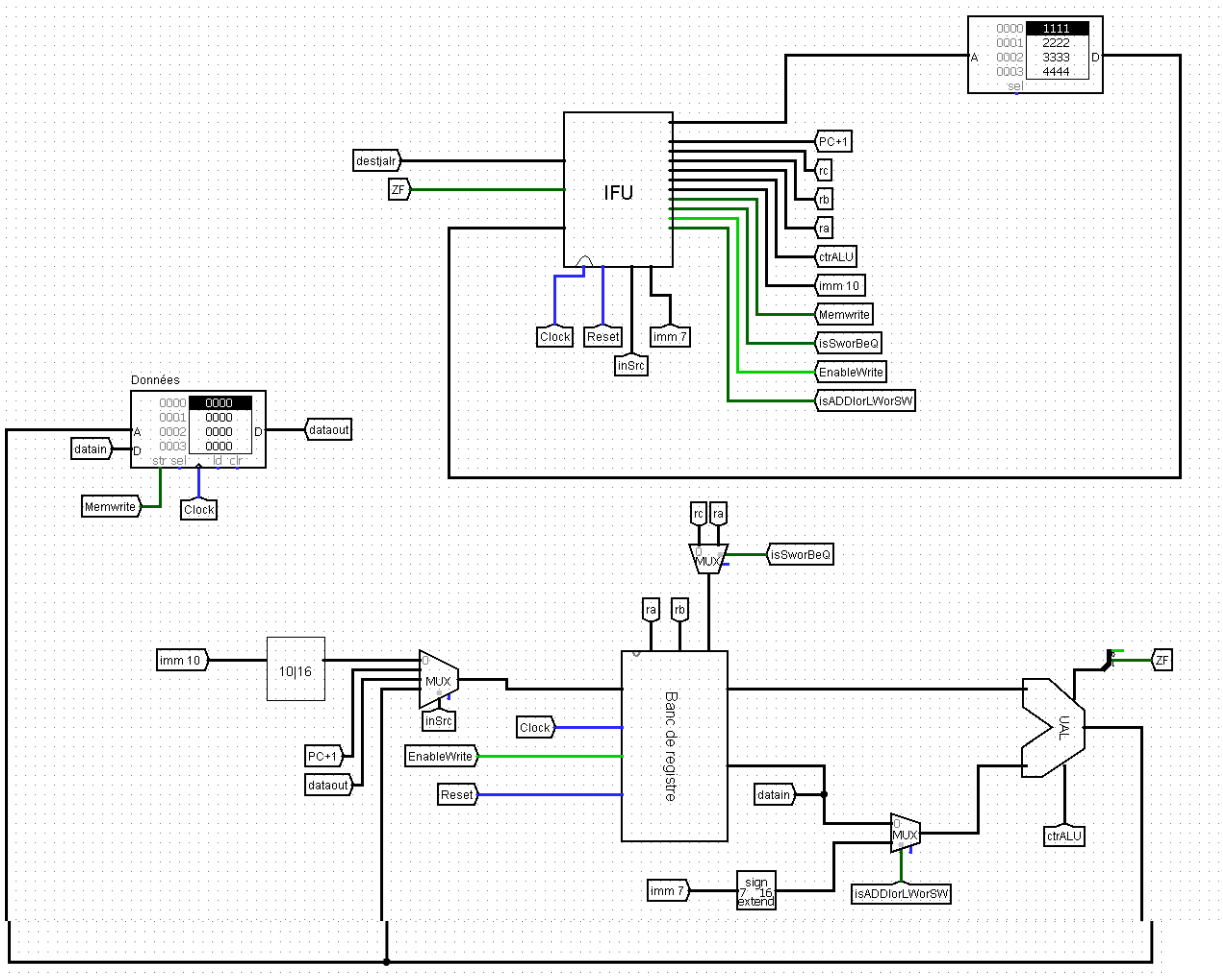
Les entrées destJALR, ZF, Clock, Reset sont utilisées dans la partie pc (programme counter) où on a un registre PC, et un registre PC+1 qui prend la sortie du PC courant avec la valeur constante 1. L'entrée destJALR va être stockée dans le registre PC quand on a un ZF, et si non les registres vont stockés l'addition de destJALR et l'imm 7 passé en 16 bits signé.

Les sorties InSrc, ctrALU, MemWrite, isWorBEQ, EnableWrite, isADDIorLWorSW prend les sorties selon les signaux déterminés à partir de l'instruction courante pour piloter les données. Dans le cas où on prend par exemple ctrALU qui est utilisé dans le circuit de l'UAL, prend la sortie selon les instructions qu'il utilise et à quels signaux elles sont, ainsi les autres sorties c'est la même chose , pour enableWrite il ne prend pas les instructions , sw, beq parce qu'ils sont utilisé dans la RAM, et PC.


rc
padding
rb
ra
RRR
imm 7
rb
ra
RRI
imm 10
ra
RI
x16
opcode
add
nand
RRR
lui
RI
addi
sw
lw
beq
jalr
RRI
add
addi
nand
lui
sw
lw
beq
jalr
Decd
opcode
EnableWrite,
x1
add
addi
nand
lui
lw
jalr
, MemWrite
x1
sw
x3
rc
x3
rb
x3
ra
x7
imm 7
imm 7
x10
imm 10
imm 10
isSWorBEQ,
x1
sw
beq
isADDIorLWorSW
x1
sw
lw
InSrc
x2
Pri
lui
jalr
lw
add
addi
nand
ctrALU
x2
Pri
add
addi
beq
nand
reg (16b)
c in
c out
0001
x16
PC+1
x16
PC
CLOCK
x1
RESET
x1
DestjALR
x16
MUX
jalr
imm 7
00
MUX
sign extend
7 16
c in
c out
ZF
x1
beq

e) Implémentation du Kathleen16

Nous avons implémenté le processeur Kathleen16 comme représenté dans l'énoncé ainsi que la RAM avec un adressage sur 16 bits avec des cases de 16 bits, de même pour la ROM.



II. Programmation du Kathleen16

a) Un programme simple

```c
#include <stdio.h>
#include <stdint.h>

int main(void)
{
   int16_t sum = 0;
   for (int16_t i = 0; i != 60; i+=3) {
      sum = sum + i;
   }
   printf("%hd\n",sum);
}
```

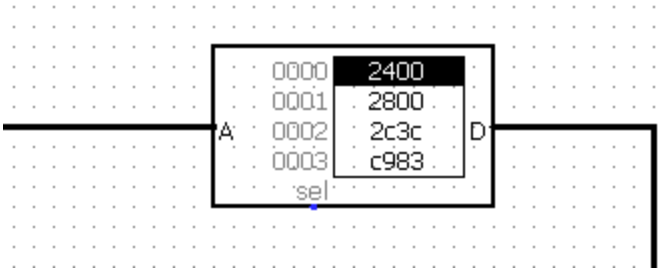
On propose le code *Kathleen16* suivant :

```
  ;; Langage d'assemblage Kathleen16
  ;; Somme des entiers multiples de 3 inférieurs à 60

  addi $r1, $r0, 0        ; sum = 0
  addi $r2, $r0, 0        ; i = 0

  addi $r3, $r0, 60
for:
  beq $r2, $r3, endfor
do:
  add $r1, $r1, $r2
  addi $r2, $r2, 3
  beq $r0, $r0, for       ; b for
endfor:
  beq $r0, $r0, endfor     ; halt
```

Programme assembler, puis charger dans la mémoire ROM du Kathleen16 :



Code binaire chargeable du Kathleen16:

Code du Kathleen16	Format de l'instruction	Code binaire chargeable	Valeur en hexadécimal
addi $r1, $r0, 0	RRI	0010010000000000	2400
addi $r2, $r0, 0	RRI	0010100000000000	2800
addi $r3, $r0, 60	RRI	0010110000111100	2C3C
for:	----------	----------	----------
beq $r2, $r3, endfor	RRI	1100100110000011	C983
do:	----------	----------	----------
add $r1, $r1, $r2	RRR	0000010010000010	0482
addi $r2, $r2, 3	RRI	0010100100000011	2903
beq $r0, $r0, for	RRI	1100000001111100	C07C
endfor:	----------	----------	----------
beq $r0, $r0, endfor	RRI	1100000001111111	C07F

b) Définition de pseudo-instructions

```
li $r, v                ⇒   lui $r, v
-------------------------------------------------------------------------------|
and $ra, $rb, $rc       ⇒   add $t0, $zero, $rb                                |
                            add $t1, $zero, $rc                                |
                            nand $t2, $t0, $t1                                 |
                            addi $ra, $t3, 0                                   |
-------------------------------------------------------------------------------|
nop                     ⇒   and $r0, $r0, $r0                                  |
-------------------------------------------------------------------------------|
not $ra, $rb            ⇒   add $t0, $zero, $rb                                |
                            nand $t0, $t0, $zero                               |
                            addi $ra, $t0, 0                                   |
-------------------------------------------------------------------------------|
or $ra, $rb, $rc        ⇒   nand $rd, $ra, $rb                                 |
                            lui $re, 1                                         |
                            add $rd, $re, $rd                                  |
-------------------------------------------------------------------------------|
sub $ra, $rb, $rc       ⇒   addi $rd, $rb, -$rc                                |
                            nand $rd, $rd, $rd                                 |
                            lui $re, 1                                         |
-------------------------------------------------------------------------------|
xor $ra, $rb, $rc       ⇒   nand $rd, $rd, $rc                                 |
                            lui $re, 1                                         |
                            add $rd, $re, $rd                                  |
```

Pseudo-instruction	Signification
`and $ra, $rb, $rc`	$\$ra \leftarrow \$rb \wedge \$rc$
`nop`	Instruction qui ne fait rien
`not $ra, $rb`	$\$ra \leftarrow \overline{\$rb}$
`or $ra, $rb, $rc`	$\$ra \leftarrow \$rb \vee \$rc$
`sub $ra, $rb, $rc`	$\$ra \leftarrow \$rb - \$rc$
`xor $ra, $rb, $rc`	$\$ra \leftarrow \$rb \oplus \$rc$

III. Conclusion

Tout au long de ce projet nous avons pu mettre en application nos connaissances théoriques du langage d'assembleur mais aussi de la modélisation Logisim.
Nous ressortons de ce projet avec beaucoup plus de compétences qu'avant celui-ci.
Nous savons maintenant comment implémenter un processeur ainsi que son fonctionnement interne.

Kamanda Aubin

&

Placé Louka