# *Projet de travaux pratiques*
# Réalisation du processeur *Kathleen16 Mk. 1*

Kathleen Booth
1922–2022

Le projet est à effectuer en binôme (ou exceptionnellement, en monôme, après consultation de l'encadrant-e des séances de travaux pratiques). Le fichier *Logisim* commenté ainsi que le rapport de projet, *impérativement* au format PDF, sont à déposer sur Madoc **au plus tard le vendredi 16 décembre 2022 à 23h55**.
*Attention : en cas de doute sur la paternité des circuits produits, un entretien avec chaque membre du binôme sera programmé au début du 2e semestre, au cours duquel chacun-e devra en démontrer la compréhension et la maîtrise.*

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.

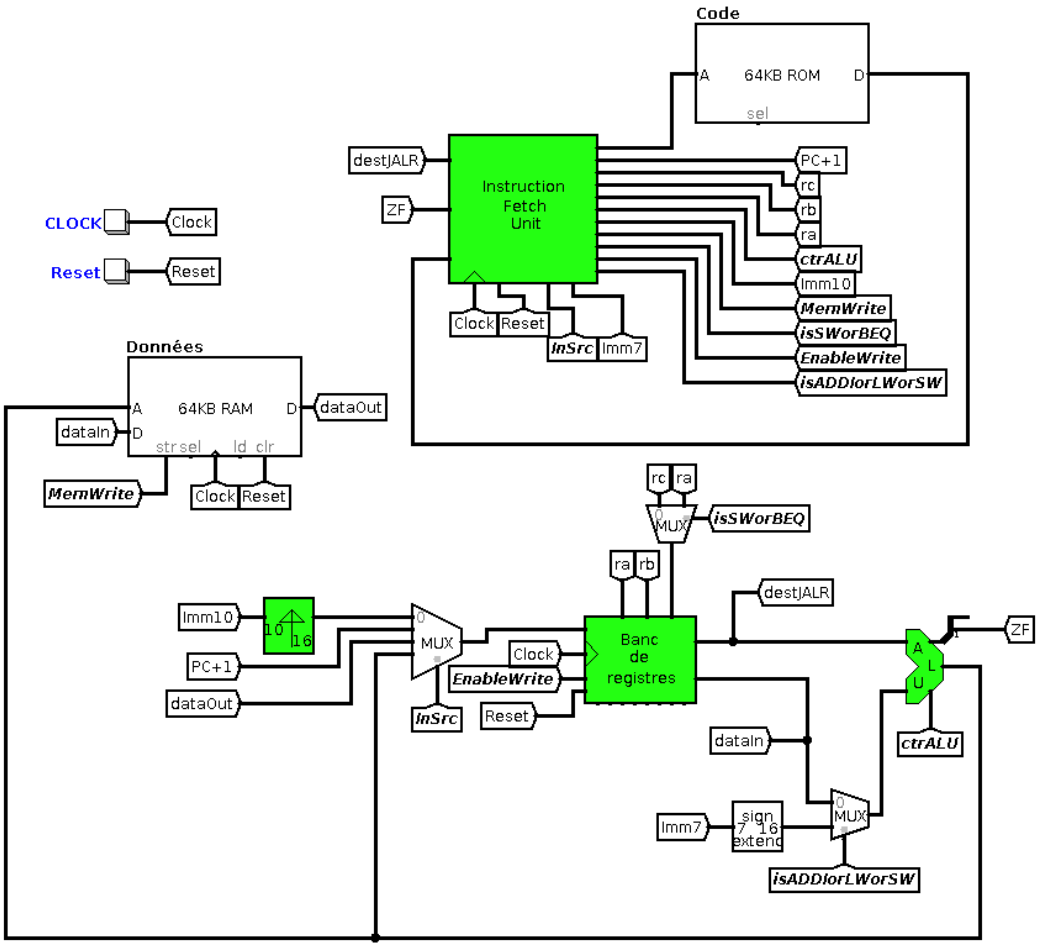
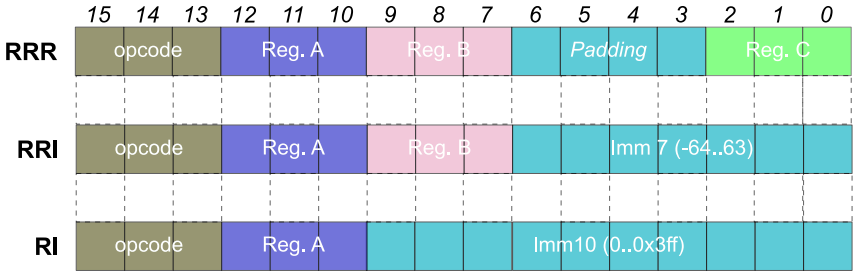



Figure 1 – Circuits du *Kathleen16 Mk. 1*.

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 | $\texttt{\$rd} \leftarrow \texttt{\$rs1} + \texttt{\$rs2}$ |
| `addi $rd, $rs, Imm7` | 001 | RRI | $\texttt{\$rd} \leftarrow \texttt{\$rs} + \texttt{Imm7}$ |
| `nand $rd, $rs1, $rs2` | 010 | RRR | $\texttt{\$rd} \leftarrow \overline{\texttt{\$rs1} \wedge \texttt{\$rs2}}$ |
| `lui $rd, Imm10` | 011 | RI | $\texttt{\$rd} \leftarrow (\texttt{Imm10} \ll 6)$ |
| `sw $rs1, $rs2, Imm7` | 100 | RRI | $\text{RAM}[\texttt{\$rs2} + \texttt{Imm7}] \leftarrow \texttt{\$rs1}$ |
| `lw $rd, $rs, Imm7` | 101 | RRI | $\texttt{\$rd} \leftarrow \text{RAM}[\texttt{\$rs} + \texttt{Imm7}]$ |
| `beq $rs1, $rs2, Imm7` | 110 | RRI | saut à PC+1+Imm7 si $\texttt{\$rs1} == \texttt{\$rs2}$ |
| `jalr $rd, $rs` | 111 | RRI | $\texttt{\$rd} \leftarrow \text{PC} + 1$ ; saut à $\texttt{\$rs}$ |

# 1 Construction du *Kathleen16*

On souhaite implémenter le *Kathleen16* dans *Logisim* en se basant sur le circuit de la figure 1. Il y a donc essentiellement quatre circuits à produire :

- l'extenseur $10\uparrow16$ ;
- L'ALU (unité arithmétique et logique) ;
- Le banc de registres ;
- L'*Instruction Fetch Unit*.

## L'extenseur $10\uparrow16$

Ce module reçoit en entrée une valeur $v$ sur 10 bits et retourne une valeur sur 16 bits obtenue en décalant $v$ de 6 positions vers la gauche. L'extenseur $10\uparrow16$ est utilisé lors de l'implémentation de l'instruction `lui`.

**Exemple :** si l'entrée est le nombre `0b1010010111`, la sortie sera `0b1010010111000000`.

## L'Unité Arithmétique Logique (UAL)

L'UAL reçoit deux valeurs $v_1$ et $v_2$ en entrée. Elle peut effectuer trois calculs différents dessus en fonction du signal de contrôle `ctrALU` :

- $v_1 + v_2$ ;
- $v_1 - v_2$ ;
- $\overline{v_1 \cdot v_2}$ (nand).

L'UAL retourne le résultat du calcul et deux indicateurs : `ZF` et `CF`.

### Le banc de registres

Le banc de registres contient les huit registres de 16 bits du *Kathleen16*. Il possède sept entrées :
- `ra` : le numéro du registre devant, potentiellement, recevoir une nouvelle valeur ;
- `rb` : le numéro du premier registre dont on souhaite connaître la valeur ;
- `rc` ou `ra` : le numéro du deuxième registre dont on souhaite connaitre la valeur ;
- Entrée : une valeur sur 16 bits devant, potentiellement, être stockée dans le registre de numéro `ra` ;
- `Clock` : l'horloge associée à chacun des registres ;
- `Reset` : un signal pour forcer à zéro tous les registres de façon asynchrone ;
- `EnableWrite` : un signal pour autoriser l'écriture dans le registre de numéro `ra`.

En retour, le banc de registres retourne les valeurs se trouvant dans les registres du numéro `rb` et `rc` ou `ra` (en fonction du signal `isSWorBEQ`).

### L'Instruction Fetch Unit (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. Il contient aussi le **contrôleur** déterminant la valeur des signaux utilisés dans les différents chemins de données (`ctrALU`, …). L'IFU possède 5 entrées :
- `destJALR` : l'adresse d'un saut si l'instruction courante est `jalr` ;
- `ZF` : l'indicateur du dernier calcul effectué par l'UAL ;
- `Clock` : l'horloge pour piloter le registre `PC` ;
- `Reset` : un signal pour forcer à 0 le registre `PC` de manière asynchrone ;
- Instruction : l'instruction courante extraite de la ROM ;

L'IFU possède aussi 13 sorties :
- `InSrc`, `ctrALU`, `MemWrite`, `isSWorBEQ`, `EnableWrite`, `isADDIorLWorSW` : les signaux déterminés à partir de l'instruction courante pour piloter les données ;
- `Imm7`, `Imm10`, `ra`, `rb`, `rc` : les différents champs de l'instruction courante ;
- `PC` : l'adresse de l'instruction courante ;
- `PC+1` : l'adresse de l'instruction suivant l'instruction courante.

## 2 Programmation du *Kathleen16*

On souhaite désormais utiliser la machine développée dans la section précédente pour exécuter du code écrit dans son langage d'assemblage.

### Un programme simple

On veut calculer la somme des entiers multiples de 3 inférieurs à 60 :

```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
```

Assembler ce programme de façon à obtenir le code binaire chargeable dans la mémoire ROM du *Kathleen16*. On présentera le code binaire avec les explications adéquates dans le rapport du projet.
On pourra tester le code binaire produit indépendamment de la réalisation dans Logisim du processeur en utilisant le simulateur simk16.py disponible sur madoc.

### Définition de pseudo-instructions

Le processeur *Kathleen16* ne possède que 8 instructions. Elles sont cependant suffisantes pour écrire de nombreux programmes non triviaux. Afin de faciliter le développement, un assembleur propose en général des *pseudo-instructions* utilisables lors de l'écriture d'un code en langage d'assemblage mais non supportées directement par le processeur. Par exemple, le *Kathleen16* n'a pas d'instruction `halt` pour arrêter l'exécution d'un programme. Comme on le voit dans l'exemple ci-dessus, il est possible de l'implémenter avec l'instruction `beq $r0, $r0, -1`. De même, on pourrait définir la pseudo-instruction `li $r, v`, qui charge la constante `v` dans le registre `$r` en utilisant l'instruction `addi` :

`li $r, v` → `addi $r, $r0, v`

Cette approche est cependant limitée car `v` a alors une représentation limitée à 7 bits alors qu'un registre peut en contenir 16.

1. Proposer une implémentation de la pseudo-instruction `li` permettant de stocker une valeur de 16 bits dans un registre en utilisant une ou plusieurs instructions natives du *Kathleen16* ;
2. Donner le développement en instructions natives des pseudo-instructions suivantes :

| Pseudo-instruction | Signification |
|---|---|
| `and $ra, $rb, $rc` | `$ra` ← `$rb` ∧ `$rc` |
| `nop` | Instruction qui ne fait rien |
| `not $ra, $rb` | `$ra` ← $\overline{\texttt{\$rb}}$ |
| `or $ra, $rb, $rc` | `$ra` ← `$rb` ∨ `$rc` |
| `sub $ra, $rb, $rc` | `$ra` ← `$rb` − `$rc` |
| `xor $ra, $rb, $rc` | `$ra` ← `$rb` ⊕ `$rc` |

# 3 Pour aller plus loin

*Le travail demandé dans cette section ne pourra faire l'objet que d'un bonus dans la notation et s'adresse aux étudiants et étudiantes particulièrement motivé-e-s et intéressé-e-s par ce projet.*

On veut désormais exploiter au maximum le potentiel du processeur *Kathleen16* en offrant la possibilité d'écrire des fonctions. Pour cela, on doit définir un protocole d'appel d'une fonction, qui est partie intégrante de l'ABI. On fait le choix suivant :

- La pile est située dans le module de RAM à partir de l'adresse `0xffc0`. Comme il est généralement d'usage, elle croît vers les adresses plus basses ;
- Le registre `$r7` est réservé pour accéder à la pile et indique en permanence l'adresse de la dernière case occupée ;
- La fonction appelante empile les paramètres de la fonction appelée en les considérant de la gauche vers la droite ;
- Lors de l'entrée dans une fonction, le registre `$r6` contient l'adresse de retour ;
- La fonction appelée a la charge de supprimer de la pile les paramètres qui lui sont passés ;

— Le résultat d'une fonction est passé sur la pile.

Afin de simplifier l'écriture de fonctions en langage d'assemblage, on introduit les pseudo-instructions suivantes :

| Pseudo-instruction | Signification |
|---|---|
| `call label` | Stocke dans `$r6` l'adresse de retour et saute à l'instruction suivant l'étiquette `label` |
| `halt` | Arrête le programme |
| `jr $r` | Saute à l'adresse contenue dans le registre `$r` |
| `push v` | Empile la constante `v` de 16 bits sur la pile |
| `push $r` | Empile la valeur du registre `$r` sur la pile |
| `pop $r` | Retire le mot de 16 bits sur le haut de la pile et le stocke dans le registre `$r` |
| `ret` | Saute à l'adresse contenue dans le registre `$r6` |

Proposer une implémentation en termes des instructions natives du *Kathleen16* des pseudo-instructions du tableau ci-dessus.

Le *Kathleen16* possède une instruction d'addition. Il est facile de définir une pseudo-instruction pour faire une soustraction (voir la section précédente). On désire fournir le code pour faire une multiplication entre deux opérandes signés.

L'algorithme décrit par le code C ci-dessous suit la méthode utilisée « à la main » en déterminant d'abord le signe du résultat puis en effectuant les calculs sur la valeur absolue des opérandes :

```c
int16_t multiply(int16_t a, int16_t b)
{
  uint16_t ua = (uint16_t)a;
  uint16_t ub = (uint16_t)b;

  uint16_t sign = 0;
  if (ua & (1<<15)) { // a<0 ?
    sign = 1;
    ua = ~ua + 1;
  }
  if (ub & (1<<15)) { // b<0 ?
    sign = sign ^ 1;
    ub = ~ub + 1;
  }

  uint16_t res = 0;
  uint16_t mask = 1;
  while (mask != (1<<15)) {
    if ((ub & mask) != 0) {
      res = res + ua;
    }
    ua = ua + ua;
    mask = mask + mask;
  }
  if (sign) {
    return (int16_t)(~res + 1);
  } else {
    return (int16_t)res;
  }
}
```

Écrire le code en langage d'assemblage *Kathleen16* implémentant la fonction `multiply`. Tester cette fonction avec des opérandes positifs et négatifs.