Agrégation d'Informatique
TP Architecture des Ordinateurs

mai 2022

Contact : sami.taktak@cnam.fr

On s'attendra à ce que les candidats traitent de manière au moins partielles les différentes parties : programmation en Python, Test unitaire, Assembleur.

Part 1 Introduction

On se propose de réaliser un simulateur de processeur RISC V 32bits. Ce simulateur sera écrit en python (version >= 3.22). L'architecture du processeur que l'on souhaite simuler est présentée figure 1.

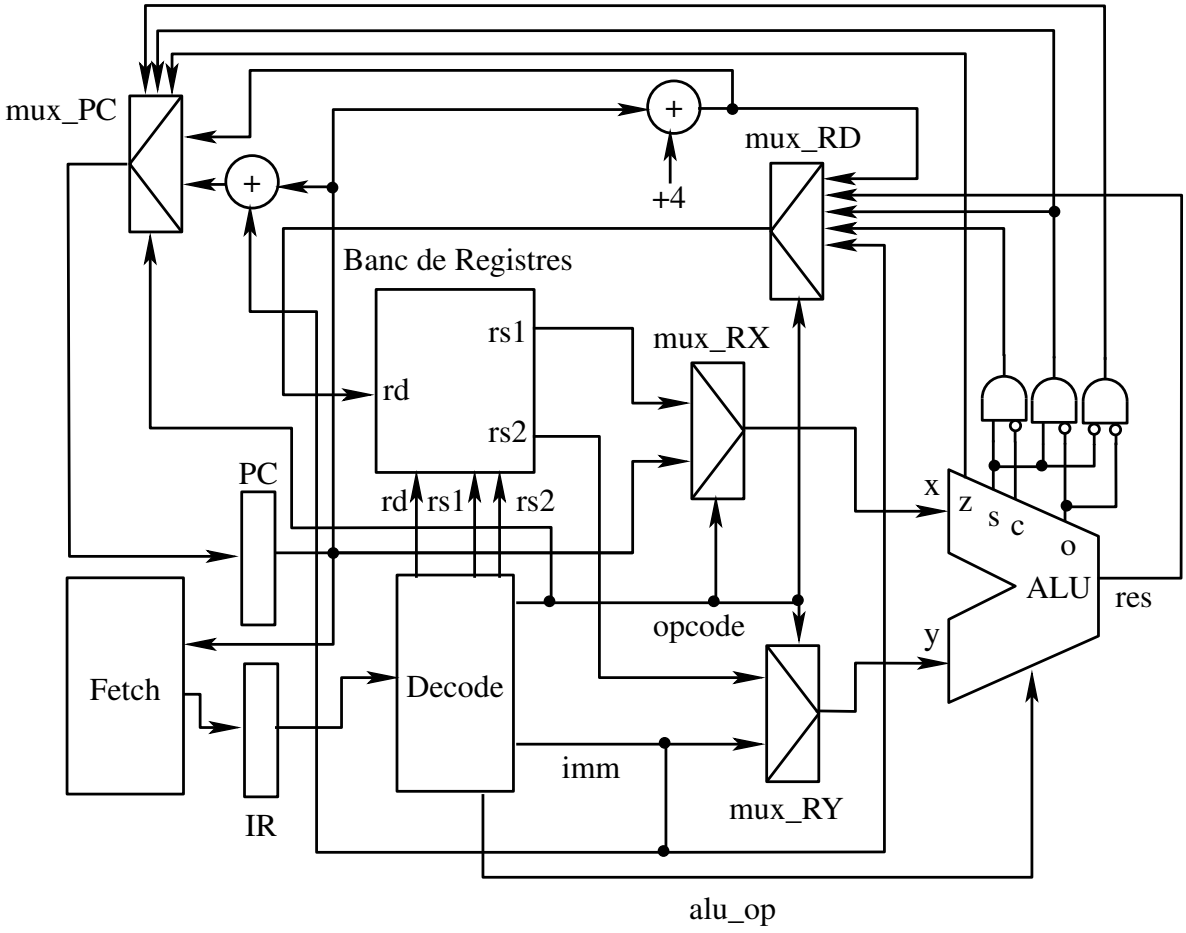

FIGURE 1 – Architecture du Processeur

Il s'agit d'un processeur non pipeline composé de 4 phases :
— la phase `Fetch`
— la phase `Decode`
— la phase `Execute`
— la phase `WriteBack`

ainsi que :

— d'un registre 32 bits compteur de programme `PC` (*Program Counter*)
— d'un registre 32 bits d'instruction `IR` (*Instruction Register*)
— un banc de 32 registres de 32 bits.

À chaque cycle d'exécution du processeur, les quatre phases sont exécutées en séquence et les registres mis à jour en conséquence ;

1. Fetch

La phase `Fetch` est en charge de la récupération de la prochaine instruction à exécuter depuis la mémoire. L'adresse de l'instruction à récupérer est spécifiée dans le registre `PC` et l'instruction est enregistrée dans le registre `IR`.

2. Decode

La phase `Decode` est en charge du décodage de l'instruction. En fonction de l'instruction à exécuter, il va :
— sélectionner les registres sources `rs1` et `rs2`
— sélectionner le registre destination `rd` du banc de registre et configurer le multiplexeur `mux_RD` afin de sélectionner la valeur qui y sera écrite ;
— positionner l'opérateur pour la phase l'ALU sur son entrée `alu_op` ainsi que placer les valeurs des opérandes de l'ALU X et Y respectivement sur ses entrées `alu_x` et `alu_y` via les multiplexeur `mux_X` et `mux_Y` ;
— sélectionner la valeur à écrire dans `PC` pour le mettre à jour à la fin du cycle.

3. Execute

La phase `Execute` lance l'ALU pour les opérations le nécessitant. Le résultat est disponible sur la sortie `alu_z`. Les drapeaux `carry`, `overflow`, `sign` et `zero` sont positionnés en fonction du résultat de l'opération réalisée.

4. WriteBack

La phase `WriteBack` procède à la mise à jour du banc de registre et à la mise à jour de `PC`.

5. Représentation d'un mot binaire

Pour représenter les mots binaires, une structure `BitArray` est fournie. Il s'agit d'une classe python implémentant un mot binaire sous forme de liste de booléens. Cela permet d'effectuer des opérations logiques sur chacun des bits :

```
>>> from bitArray import BitArray
>>> m = BitArray(172, 8)
>>> m
BitArray([False, False, True, True, False, True, False, True])
```

`m = BitArray(172, 8)` définit un mot de 8 bits encodant la valeur entière 172. On peut voir que `m` correspond à une liste de booléens.

Chaque bit est accessible individuellement :

```
>>> m[0]
False
>>> m[7]
True
>>> m[1] = m[0] or m[7]
```

La classe `BitArray` supporte les opérations suivantes :

Opération	Type	Description
`BitArray(val, taille)`	Constructeur	Construit un `BitArray` à partir d'une valeur entière `val` et de `taille` en nombre de bits
`BitArray(bytes)`	Constructeur	Construit un `BitArray` à partir d'une valeur spécifiée sous forme de chaîne de bytes contenant 0, 1 ou X (pour non définie)
`a + b`	Concaténation	Concatène deux `BitArray`
`a[i]`	Copie	Permet de récupérer la valeur du bit `i`
`a[i] =`	Affectation	Permet d'affecter une valeur booléenne au bit `i`
`printa`	Affichage	Affiche la valeur du mot binaire
`len(a)`	Longueur	Renvoie la longueur du mot binaire `a`
`a.to_int()`	Conversion	Convertie le `BitArray()` en valeur entière signée
`a.to_uint8()`	Conversion	Convertie le `BitArray()` en valeur entière non signée
`a.to_bytes()`	Conversion	Convertie le `BitArray()` en bytes

La classe `BitArray` est décrite dans le fichier `bitArray.py`.

Remarque. ⚠ *Lors de l'affichage d'un `BitArray`, la liste de booléens est affichée de gauche à droite en commençant par l'indice 0 comme pour toute liste python. Lors de l'appel à `print()`, la valeur du `BitArray` est affichée sous forme de mot binaire avec le bit de poids faible à droite. De plus un espace est inséré tous les 8 bits en partant de la droite.*

L'ordre des bits est donc inversé à l'affichage, mais l'indice d'un bit d'un `BitArray` correspond au poids de ce bit dans le mot binaire.

Cette remarque est aussi valable pour les valeurs de type bytes, où la valeur d'indice zéro apparaît en premier en partant de la gauche comme pour les listes.

```
>>> m = BitArray(172,12)
>>> m
BitArray([False, False, True, True, False, True, False, True, False, False, False, False])
>>> print(m)
0000 10101100
>>> m.to_bytes()
b'001101010000'
```

Question 1. *Afin de prendre en main cette structure `BitArray()` écrivez les tests unitaires pour les fonctions suivantes de la classe `BitArray` :*

- *Création d'un `BitArray` à partir d'une valeur entière ou d'une valeur bytes ;*
- *Accès individuel aux bits en lecture et écriture*
- *Accès à une séquence de bits*
- *Concaténation de deux `BitArray`*
- *Appelle à la fonction `len()` sur un `BitArray()`*
- *Conversion d'un `BitArray` en entier ou en bytes*

Les tests unitaires seront à écrire dans le fichier `test_bitArray.py`.

Une introduction aux tests unitaires en python est fournie en annexe A.

6. Les Registres

- **Registre Simple**

Les registres sont implémentés par la classe `Registre`. La classe `Registre` permet de définir des registres de n bits, n étant un paramètre du constructeur d'un registre. La valeur d'un registre est de type `BitArray`.

```
>>> from registre import Registre
>>> r = Registre(32)
>>> print(r)
XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX
```

Comme on peut le voir, les registres n'ont pas de valeur initiale. La lecture et l'écriture d'un registre se font via les méthodes `lecture()` et `ecriture()` :

```
>>> r.ecriture(BitArray(123,32))
>>> r.lecture()
BitArray([True, True, False, True, True, True, True, False, False, False, False, False, False, False,
    False, False, False, False, False, False, False, False, False, False, False, False, False, False,
    False, False, False, False])
>>> print(r)
00000000 00000000 00000000 01111011
```

Question 2. *Écrire, dans le fichier `test_registre.py`, 2 à 3 tests unitaires sur les opérations suivantes de la classe `Registre` :*

- *Création d'un registre*
- *Écriture dans un registre*
- *Lecture dans un registre*

- **Banc de Registre**

Le banc de registre est implémenté par la classe `BancRegistre`. Un banc de registre est caractérisé par son nombre de registres ainsi que la taille de ses registres. Le registre 0 est "collé" à 0, ce qui signifie qu'il renvoie toujours la valeur 0 et qu'une écriture n'y a pas d'effet.

```
>>> from banc_reg import BancRegistre
>>> banc = BancRegistre(4, 8)
>>> print(banc)
R0: 00000000
R1: XXXXXXXX
R2: XXXXXXXX
R3: XXXXXXXX
```

Question 3. *Testez la classe `BancRegistre` en complétant le fichier `test_banc_reg.py` avec quelques tests judicieusement choisis.*

Part 2 Réalisation du Simulateur

1. Le jeu d'instructions

On va chercher à implémenter un sous ensemble des instructions RISC V. Pour rappel, les formats des instructions RISC V sont donnés table 1

Le jeu d'instructions supporté par notre simulateur est présenté dans la table 2.

Si le nombre d'instructions peut paraître important de prime abord, un grand nombre de ces instructions peuvent être réalisées avec un petit nombre d'opérations arithmétiques et logiques.

En effet, une opération de l'ALU entre deux opérandes sera toujours effectuée de la même façon quelque soit la source de l'opérande (banc de registre, immédiat, PC, ...).

Ce sont les multiplexeurs `mux_X` et `mux_Y` présentés ci-dessous qui sont responsables de la sélection de la valeur de chaque opérande.

TABLE 1 – Format des instructions RISC V

Format	31 25	24 20	19 15	14 12	11 7	6 0
R	funct7	rs2	rs1	funct3	rd	opcode
I	imm[11 : 0]		rs1	funct3	rd	opcode
S	imm[11 : 5]	rs2	rs1	funct3	imm[4 : 0]	opcode
B	imm[12|10 : 5]	rs2	rs1	funct3	imm[4 : 1|11]	opcode
U	imm[31 : 12]				rd	opcode
J	imm[20|10 : 1|11|19 : 12]				rd	opcode

De même, les branchements conditionnels peuvent être réalisés grâce à une soustraction et la décision de branchement se fait sur la valeur des drapeaux `zero`, `sign`, `carry` et `overflow`.

Question 4. *Donnez les conditions de branchement sous forme de fonction booléenne en fonction des valeurs de `zero`, `sign`, `carry` et `overflow` pour les instructions `beq`, `bne`, `blt`, `bge`, `bltu` et `bgeu`. Tous les drapeaux ne sont pas forcément nécessaires.*

Le même raisonnement s'applique pour les instructions `slt`, `sltu`.

2. L'Unité Arithmétique et Logique (ALU)

L'ALU est le premier composant de notre processeur auquel on va s'intéresser. Comme on l'a vu ci-dessus, l'ALU procède de la même façon que les opérandes proviennent du banc de registre où d'un immédiat. On ne distinguera donc pas la source des valeurs des opérandes dans l'ALU.

Les opérations à implémenter sont donc :
- les opérations arithmétiques : `add`, `sub`
- les opérations logiques : `and`, `or`, `xor`
- les opération de décalage : `sll`, `srl`, `sra`

L'ALU est implémentée par la classe `Alu` du fichier `aly.py`. L'instanciation de l'ALU se fait de la façon suivante :

```
>>> from alu import Alu
>>> alu = Alu()
```

L'exécution de l'ALU se fait en appelant la méthode `eval(op, x, y)` où `op` est l'opération à réaliser, `x` et `y` les opérandes.

Le résultat est disponible dans l'attribue `res`, de même que les drapeaux `carry`, `overflow`, `sign` et `zero`.

On va maintenant implémenter ces différentes opérations dans l'ALU.

• Encodage de l'opérateur

Sur la table 2, on peut voir que les codes *funct3* et *funct7* sont identiques pour une même opération avec ou sans immédiat. Par exemple :
- `add` et `addi` ont la même valeur pour *funct3* : 0x0. Seul `opcode` est différent mais ces opérations ne sont pas différenciées par l'alu ;
- `sub` se différencie de `add` par la valeur du bit 6 de `funct7`, de même que `srl` et `sra` ;
- `xor` et `sll` se différencient par la valeur de `funct3`.

Conclusion :
- l'`opcode` n'est pas significatif pour l'ALU ;
- `funct3` permet de différencier les opérations entre elles ;
- le bit 6 de `funct7` permet de différentier des variations de certains opérateurs.

TABLE 2 – Liste des instructions RISC V supportées

Inst	opcode	funct3	funct7	format	Opération	Effet
add	1100110	0x0	0x00	R	add rd,rs1,rs2	$rd = rs1 + rs2$
sub	1100110	0x0	0x20	R	sub rd,rs1,rs2	$rd = rs1 - rs2$
xor	1100110	0x4	0x00	R	xor rd,rs1,rs2	$rd = rs1 \oplus rs2$
or	1100110	0x6	0x00	R	or rd,rs1,rs2	$rd = rs1 \vee rs2$
and	1100110	0x7	0x00	R	and rd,rs1,rs2	$rd = rs1 \wedge rs2$
slt	1100110	0x2	0x00	R	slt rd,rs1,rs2	$rd = (rs1 < rs2)?1 : 0$
sltu	1100110	0x3	0x00	R	sltu rd,rs1,rs2	$rd = (rs1 < rs2)?1 : 0$
sll	1100110	0x1	0x00	R	sll rd,rs1,rs2	$rd = rs1 << rs2$
srl	1100110	0x5	0x00	R	srl rd,rs1,rs2	$rd = rs1 >> rs2$
sra	1100110	0x5	0x20	R	sra rd,rs1,rs2	$rd = rs1 << rs2$
addi	1100100	0x0		I	addi rd,rs1,imm	$rd = rs1 + imm$
xori	1100100	0x4		I	xori rd,rs1,imm	$rd = rs1 \oplus imm$
ori	1100100	0x6		I	ori rd,rs1,imm	$rd = rs1 \vee imm$
andi	1100100	0x7		I	andi rd,rs1,imm	$rd = rs1 \wedge imm$
slti	1100100	0x2		I	slti rd,rs1,imm	$rd = (rs1 < immm)?1 : 0$
sltiu	1100100	0x3		I	sltiu rd,rs1,imm	$rd = (rs1 < imm)?1 : 0$
slli	1100100	0x1	0x00	I	slli rd,rs1,imm	$rd = rs1 << imm[0:4]$
srli	1100100	0x5	0x00	I	srli rd,rs1,imm	$rd = rs1 >> imm[0:4]$
srai	1100100	0x5	0x20	I	srai rd,rs1,imm	$rd = rs1 >> imm[0:4]$
beq	1100011	0x0		B	beq rs1,rs2,imm	if $(rs1 == rs2)$ $PC+ = imm$
bne	1100011	0x1		B	bne rs1,rs2,imm	if $(rs1 \neq rs2)$ $PC+ = imm$
blt	1100011	0x4		B	blt rs1,rs2,imm	if $(rs1 < rs2)$ $PC+ = imm$
bge	1100011	0x5		B	bge rs1,rs2,imm	if $(rs1 \geq rs2)$ $PC+ = imm$
bltu	1100011	0x6		B	bltu rs1,rs2,imm	if $(rs1 < rs2)$ $PC+ = imm$
bgeu	1100011	0x7		B	bgeu rs1,rs2,imm	if $(rs1 \geq rs2)$ $PC+ = imm$
jal	1111011			J	jal rd,imm	$rd = PC + 4; PC+ = imm$
jalr	1110011	0x0		I	jalr rd,rs1,imm	$rd = PC + 4; PC = rs1 + imm$
lui	1110110			U	lui rd,imm	$rd = imm << 12$
auipc	1110100			U	auipc rd,imm	$rd = PC + (imm << 12)$

Les opérations supportées par l'ALU seront donc encodées par :
— les 3 bits `funct3` suivis du bit 6 de *funct7*, si *funct7* existe
exemple : `sub: 0001`
— les 3 bits `funct3` suivis de 0 si *funct7* n'est pas présent : exemple : `xori: 1000`
Remarque. *Pour `ssli`, `srli` et `srai`, bien qu'il n'y ait pas de champs `funct7` dans le format I, les bits 5 à 11 de l'immédiat jouent le rôle de `funct7`.*
Les différents codes des opérateurs de l'ALU sont résumés table 3

TABLE 3 – Encodage de l'opérateur de l'ALU sur 4 bits

Opération	code
add	0000
sub	0001
xor	1000
or	1100
and	1110
sll	0010
srl	1010
sra	1011

- **l'Additionneur**

Nous allons maintenant procéder à l'implémentation de l'opérateur addition dans l'ALU.

Question 5.
— *Rappelez le circuit logique d'un additionneur 1 bits à deux opérandes ainsi que l'expression de ses sorties `s` et $\texttt{c}_{\texttt{out}}$ en fonction de ses entrées `a`, `b` et $\texttt{c}_{\texttt{in}}$*
— *Implémentez l'opérateur d'addition de l'ALU dans la méthode `add()` de la classe `Alu` dans le fichier `alu.py`*
— *Testez votre additionneur en écrivant des tests unitaire dans le fichier `test_alu.py`. Expliquez le choix de votre jeu de tests.*

- **La soustraction**

Nous allons maintenant réaliser l'opérateur de soustraction. Pour réaliser cet opérateur, on peut remarquer que :

$$a - b = a + (-b)$$

En prenant en compte ce résultat et les spécificités de la représentation en complément à deux, on peut réaliser l'opérateur de soustraction en réutilisant l'additionneur réalisé ci-dessus.

Question 6.
— *Donnez le circuit logique de l'opération de soustraction en réutilisant l'additionneur ci-dessus ;*
— *Implémentez l'opérateur de soustraction de l'ALU dans la méthode `sub()` de la classe `Alu` dans le fichier `alu.py`*
— *Testez l'opération de soustraction en écrivant des tests unitaires dans le fichier `test_alu.py`. Expliquez le choix de votre jeu de tests.*

- **Opérateurs Logiques**

Question 7.
— *Implémentez les opérateurs logiques `xor`, `or` et `and` respectivement dans les méthodes `xor()`, `or()` et `and()` de la classe `Alu` dans le fichier `alu.py`*

— *Testez ces opérateurs en écrivant des tests unitaires dans le fichier* `test_alu.py`*. Expliquez le choix de votre jeu de tests.*

- **Opérateurs de Décalage**

Question 8.
— *Implémentez les opérateurs de décalage* `sll`*,* `srl` *et* `sra` *respectivement dans les méthodes* `sll()`*,* `srl()` *et* `sra()` *de la classe* `Alu` *dans le fichier* `alu.py`
— *Testez ces opérateurs en écrivant des tests unitaires dans le fichier* `test_alu.py`*. Expliquez le choix de votre jeu de tests.*

3. Le Décodeur

Le décodeur lit l'instruction présente dans le registre `IR` et la décode ce qui permet de récupérer les différents champs d'une instruction `opcode`, `rd`, `funct3`, `rs1`, `rs2`, `funct7` et `imm`.

Les `opcode`, `rd`, `funct3`, `rs1`, `rs2`, `funct7` sont toujours décodés quelque soit le format de l'instruction présente dans `IR` et peuvent donc contenir une valeur non pertinente.

Le champs `imm` n'est décodé que si l'instruction possède un champ immédiat et il est reconstitué (les bits sont réordonnés et, lorsque le bit 0 est manquant, celui-ci est rajouté).

Ensuite, il doit déterminer l'opération que l'ALU doit effectuer.

Question 9.
— *Pour chaque instruction, déterminez l'opération que doit effectuer l'ALU ;*
— *Complétez la méthode* `calcul_alu_op()` *de la classe* `Decodeur` *dans le fichier* `decodeur.py`*;*
— *Écrivez vos tests unitaires dans le fichier* `test_decodeur.py`*.*

4. Mise à jour de `PC`

La mise à jour de `PC` se fait généralement en incrémentant `PC` de 4, chaque instruction étant codée sur 4 octets.

Cependant, lors de l'exécution d'instructions de branchement, la mise à jour de `PC` dépend de différents paramètres :

— s'il s'agit de l'instruction `jal`, la nouvelle valeur de `PC` sera la somme de la valeur courante de `PC` plus l'immédiat
— s'il s'agit de l'instruction `jalr`, la nouvelle valeur de `PC` sera la somme de la valeur de `rs1` plus l'immédiat
— s'il s'agit d'un branchement conditionnel, la nouvelle valeur de `PC` dépendra du résultat de la comparaison

Question 10.
— *Déterminez pour chacune des instructions comment sera mise à jour* `PC`
— *Complétez la fonction* `update_PC` *dans le fichier* `update_PC.py` *qui retourne la nouvelle valeur de* `PC` *en fonction de l'instruction courante, de la valeur courante de* `PC` *et, si besoin, des sorties de l'ALU. Pour réaliser cette fonctions, on dispose des mêmes opérations que pour l'ALU. On suppose que l'on dispose d'un opérateur ajoutant 4 à* `PC` *ainsi que d'un additionneur entre* `PC` *et l'immédiat. Pour cela, on pourra utiliser l'opérateur addition de python.*
— *Écrivez vos tests unitaires dans le fichier* `test_update_PC.py`*.*

Part 3 Audit de code

Dans cette partie de l'épreuve, il est demandé au candidat d'étudier un fichier source qui comporte des erreurs ou des maladresses, de qualité de code ou de fonctionnement, et il est demandé d'auditer ce fichier, c'est-à-dire :

— de comprendre et d'être capable d'expliquer le fonctionnement du code à l'oral ;

— de proposer des corrections en réécrivant certaines parties afin de corriger les erreurs ou maladresses éventuelles et de rendre le code plus clair, notamment dans une optique pédagogique ;
— de proposer des améliorations de la complexité en temps ou en mémoire, ou de la sûreté du code.

Le temps indicatif de préparation de cette partie est d'une heure.

```python
archi=32  # architecture 32 bits
nb_reg=32 # nombre de registre du banc de registres

# La mémoire est modélisée comme une liste de mot de 8b
mem = []

def init_mem(prog=None):
    for inst in prog:
        i = GenInst.parse_instruction(inst)
        mem.append(i[0:8])
        mem.append(i[8:16])
        mem.append(i[16:24])
        mem.append(i[24:32])

def fetch(addr):

    return mem[addr.to_uint()] + \
        mem[addr.to_uint()+1] + \
        mem[addr.to_uint()+2] + \
        mem[addr.to_uint()+3]

def mux_RX(opcode):
    # selection de l'opérande X de l'alu
    if opcode  in [b'1100100', b'1100011', b'1110011']:
        return banc[decodeur.rs1.to_uint()]
    elif opcode in [b'1100110', b'1110100']:
        return PC.lecture()
    else:
        print("mux_RX: opcode ({opcode}) non supporté")
        raise NotImplemented

def mux_RY(opcode, funct3):
    # selection de l'opérande Y de l'alu
    if opcode  in [b'1100110', b'1100011']:
        return banc[decodeur.rs2.to_uint()]
    if opcode  in [b'1100100']:
        if funct3.to_bytes() in [b'100', b'101']:
            return BitArray(decodeur.imm[0:5].to_uint(), archi)
        else:
            return BitArray(decodeur.imm.to_bytes(), archi)
    elif opcode in [b'1110100']:
        return BitArray(b'0', 12) + decodeur.imm
    elif opcode in [b'1110011']:
        return BitArray(decodeur.imm.to_bytes(), archi)
    else:
        print("mux_RY: opcode ({opcode}) non supporté")
        raise NotImplemented

def writeback(opcode, funct3):
    # mise à jour banc de registres
    if opcode in [b'1100110', b'1100100']:
        if funct3.to_bytes() == b'010':
            return BitArray(alu.sign and alu.overflow, archi)
        if funct3.to_bytes() == b'110':
            return  BitArray(alu.sign and not alu.carry, archi)
        else:
            return alu.res
    elif opcode in [b'1111011', b'1110011']:
        return BitArray(PC.lecture().to_int()+4, archi)
    elif opcode in [b'1100011']:
        pass
    elif opcode in [b'1110100']:
        return alu.res
    elif opcode in [b'1110110']:
        return BitArray(b'0', 12) + decodeur.imm

def cycle():
```

```
    """
    Effectue les opération d'un cycle CPU
    """

    ############## Fetch       ################################
    IR.ecriture(fetch(PC.lecture()))

    ############## Decode      ################################
    # decode l'instruction présente dans IR et configure l'ALU
    decodeur.decode(IR.lecture())
    opcode = decodeur.opcode.to_bytes()

    ############## Exec        ################################
    # Execute l'instruction
    # si opération arithmétiques:
    if decodeur.alu_op is not None:
        # selection de l'opérande X de l'alu
        x = mux_RX(opcode)

        # selection de l'opérande Y de l'alu
        y = mux_RY(opcode, decodeur.funct3)

        alu.eval(decodeur.alu_op, x, y)

    ############## Write Back ################################
    if opcode not in [b'1100011']:
        banc[decodeur.rd.to_uint()] = writeback(opcode, decodeur.funct3)

    # Mise à jour de PC
    PC.ecriture(update_PC(decodeur, PC.lecture(), alu))

if __name__ == '__main__':
    init_mem(fibo)

    # les registres CPU
    IR = Registre(archi)
    PC = Registre(archi)
    banc = BancRegistre(nb_reg, archi)

    # les differentes unité du CPU
    decodeur = Decodeur()
    alu = Alu()

    # Initialisation de PC
    PC.ecriture(BitArray(b'00000000000000000000000000000000'))

    try:
        while True:
            cycle()
    except IndexError:
        print("Fin du programme")
        print(banc)
        print(f"IR: {IR.lecture()}\t {decode_instruction(IR.lecture())}")
        print(f"PC: {PC.lecture()}")
```

Listing 1 – Simulateur RISC V

Part 4 Programmation Assembleur

On se propose dans cette partie d'écrire un programme en assembleur à faire exécuter par notre simulateur.

On va chercher à écrire un programme calculant les n premiers nombres de la suite de Fibonacci.

La suite de Fibonacci est une suite d'entiers dans laquelle chaque terme est la somme des deux termes qui le précèdent avec :

$$F_0 = 0, F_1 = 1 \text{ et } F_n = F_{n-1} + F_{n-2}$$

Question 11.

— *Écrivez un programme assembleur en utilisant les instructions définies ci-dessus qui calcule les n premiers nombres de la suite de Fibonacci. Une instruction doit être composée d'un mnémonique suivit d'un espace, suivit de ses paramètres. Les paramètres doivent être séparés entre eux par des virgules et sans espace. Par exemple :* `add 1, 2, 3` *est l'instruction effectuant l'addition des valeurs des registres R2 et R3 dans R1.*

— *Passez ce programme sous forme d'une liste d'instructions à la fonction* `init_mem()` *dans le fichier* `simv.py` *et lancez le simulateur avec la commande :*
`$ python3 simv.py`

A Tests Unitaires en Python

Les tests unitaires peuvent être réalisés en python en utilisant le module `unittest`.

L'usage est d'écrire les tests unitaires dans un fichier dont le nom sera celui du module à tester préfixé de `test_` ou avec le suffixe `_test.py`.

Les fichiers de tests seront, soit placés dans le même répertoire que les modules testés, soit dans un répertoire dédié.

Nous adopterons la convention où les noms des fichiers de tests contiennent le préfixe `test_` et sont placés dans le même répertorie que les fichiers à tester.

Un fichier de tests unitaires commence par importer le module `unittest` puis les modules nécessaires aux tests et a la structure suivante :

```
1 import unittest
2 import module_a_tester
3
4 class TestModuleAtester(unittest.TestCase):
5     def test_un(self):
6         self.assertEqual(1, 1)
```

Une classe dont le nom commence par `Test` et héritant de la classe `unittest.TestCase` est déclarée, puis une méthode de test est définie pour chaque test à réaliser.

La liste des assertions disponibles peut être obtenue directement depuis l'interpréteur python :

```
python3
Python 3.10.4 (main, Apr  2 2022, 09:04:19) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import unittest
>>> help(unittest)
Help on package unittest:

NAME
    unittest

MODULE REFERENCE
    https://docs.python.org/3.10/library/unittest.html
```

ou directement à l'adresse `https://docs.python.org/3.10/library/unittest.html`

Pour lancer une série de tests, on exécutera la commande suivante depuis le répertoire contenant le fichier `test_alu.py` :

```
$ python3 -m unittest test_alu.py
Test addition

op:  0000
```

```
x:   00000000 00000000 00000000 00000101
y:   00000000 00000000 00000000 00000111
res: 00000000 00000000 00000000 00001100
carry: False overflow: False zero: False sign: False
⋮

.
----------------------------------------------------------------------
Ran 8 tests in 0.003s

OK
```