Projet

Un processeur RiSC-16

Consignes

Le projet RiSC-16 est à réaliser en **binôme** (1 seul trinôme autorisé par groupe de TD). Le projet devra être rendu pour fin décembre/début janvier (la date sera précisée ultérieurement) sous la forme d'une archive (zip ou targz) nommé `prénom1_nom1_prénom2_nom2_risc16` contenant votre rapport et vos fichiers.

Un **barème indicatif** vous est précisé avec un total de 46 points. Votre note finale sera tronquée à 40 puis ramenée sur 20. Vous disposez de 6 points bonus, vous n'êtes donc pas tenu de tout faire. Néanmoins, nous vous suggérons de réaliser les questions dans l'ordre, le rapport difficulté/points étant croissant.

Les parties sont notés comme suit :

- Un processeur avec pipeline : 20,5 points.
- Un pipeline avec logique bypass : 9,5 points.
- Mapping Memory : 9 points.
- Un pipeline avec logique Stall et Stomp : 7 points.

Le notation tiendra compte également:

- de la concision (point trop n'en faut!) et clarté du rapport qui détaillera (1) les questions répondues et (2) les réponses apportées lorsque nécessaire (Par exemple, votre choix de segmentation de la mémoire),
- de la propreté et modularité des circuits réalisés,
- de la pertinence des programmes assembleurs (et non de leur complexité),
- de la ponctualité du rendu.

C'est à dire que des points seront retirés sur la note finale pour chaque critère ci-dessus non respecté.

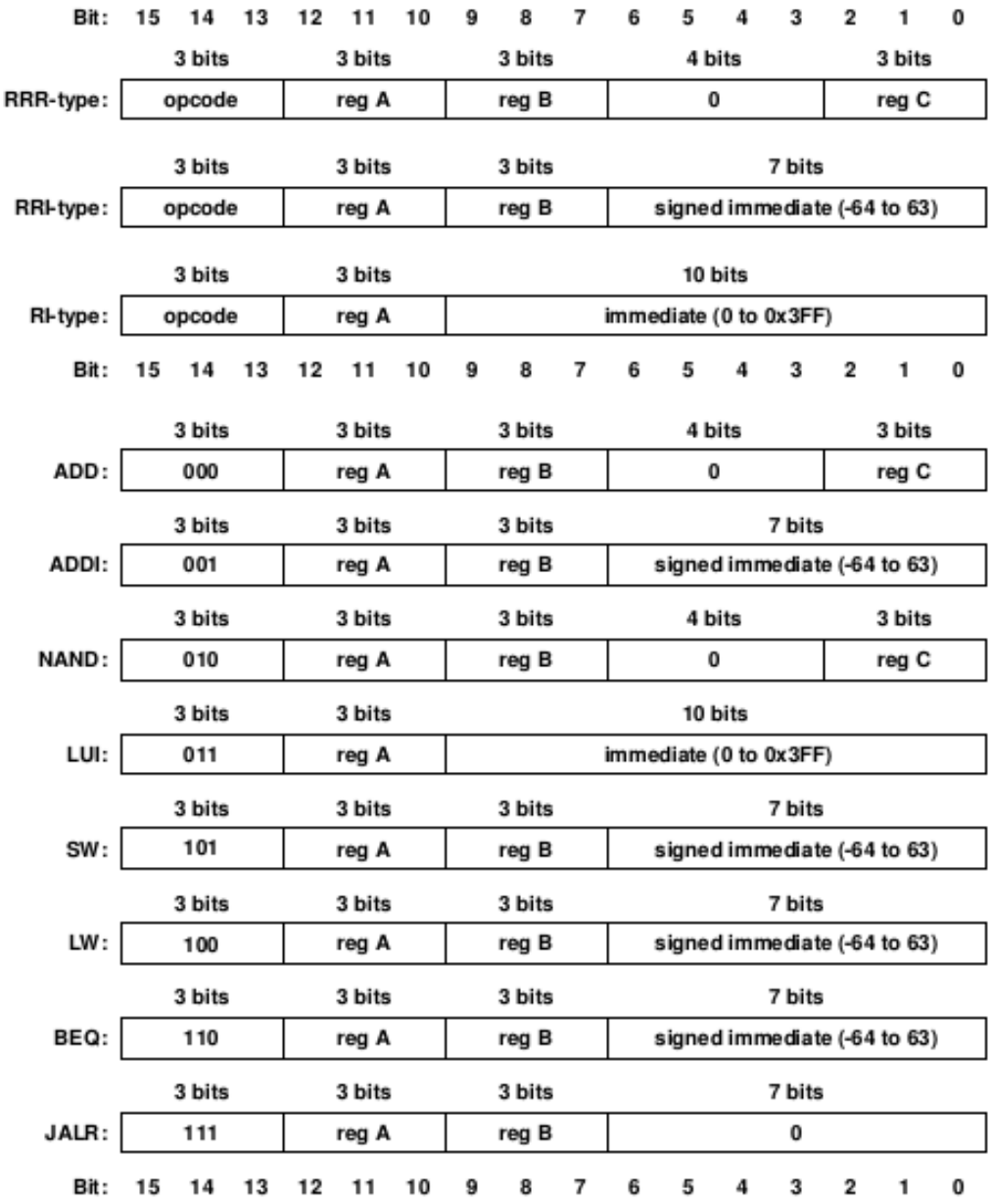
Enfin, la somme de travail demandée étant très importante, nous vous invitons à commencer au plus tôt et ainsi poser vos questions au fur et à mesure.

1 Architecture RiSC-16

1.1 Jeu d'instructions

Le processeur RiSC-16 est une machine 16 bits à 8 registres. Seul le registre 0 a pour convention de contenir la constante 0, les autres pouvant être librement utilisés. RiSC-16 possède 8 instructions différentes selon 3 formats (RRR, RRI, RI) tel qu'illustré ci-dessous.

FORMATS:



INSTRUCTIONS:

Le tableau suivant détaille les opérations réalisées par chaque instruction.

Mnemonic	Name and Format	Opcode (binary)	Assembly Format	Action
add	Add RRR-type	000	add rA, rB, rC	Add contents of **regB** with **regC**, store result in **regA**.
addi	Add Immediate RRI-type	001	addi rA, rB, imm	Add contents of **regB** with **imm**, store result in **regA**.
nand	Nand RRR-type	010	nand rA, rB, rC	Nand contents of **regB** with **regC**, store results in **regA**.
lui	Load Upper Immediate RI-type	011	lui rA, imm	Place the 10 ten bits of the 16-bit **imm** into the 10 ten bits of **regA**, setting the bottom 6 bits of **regA** to zero.
sw	Store Word RRI-type	101	sw rA, rB, imm	Store value from **regA** into memory. Memory address is formed by adding **imm** with contents of **regB**.
lw	Load Word RRI-type	100	lw rA, rB, imm	Load value from memory into **regA**. Memory address is formed by adding **imm** with contents of **regB**.
beq	Branch If Equal RRI-type	110	beq rA, rB, imm	If the contents of **regA** and **regB** are the same, branch to the address PC+1+**imm**, where PC is the address of the beq instruction.
jalr	Jump And Link Register RRI-type	111	jalr rA, rB	Branch to the address in **regB**. Store PC+1 into **regA**, where PC is the address of the jalr instruction.

1.2 Assembleur

Un assembleur (en C) vous sera fourni. Il permet de transformer du code assembleur RiSC-16 en binaire (format pour ROM Logisim).

Une ligne d'assembleur est formattée comme suit:

```
label:<whitespace>opcode<whitespace>field0, field1, field2<whitespace># comments
```

Exemple de programme comptant jusqu'à 10:

```
          addi r2, r0, 10     # r2 = 10
          addi r1, r0, 0      # r1 = 0
loop:     beq r1, r2, endloop # while r1 != r2
          addi r1, r1, 1      # r1 += 1
          beq r0, r0, loop    # loop
endloop:  halt                # end
```

Aux 8 instructions de base, nous ajoutons les 6 pseudo-instructions (compilés en utilisant les instructions de base) ci-dessous:

Assembly-Code Format	Meaning
`nop`	do nothing
`halt`	stop machine & print state
`lli regA, immed`	R[regA] <- R[regA] + (immed & 0x3f)
`movi regA, immed`	R[regA] <- immed
`.fill immed`	initialized data with value *immed*
`.space immed`	zero-filled data array of size *immed*

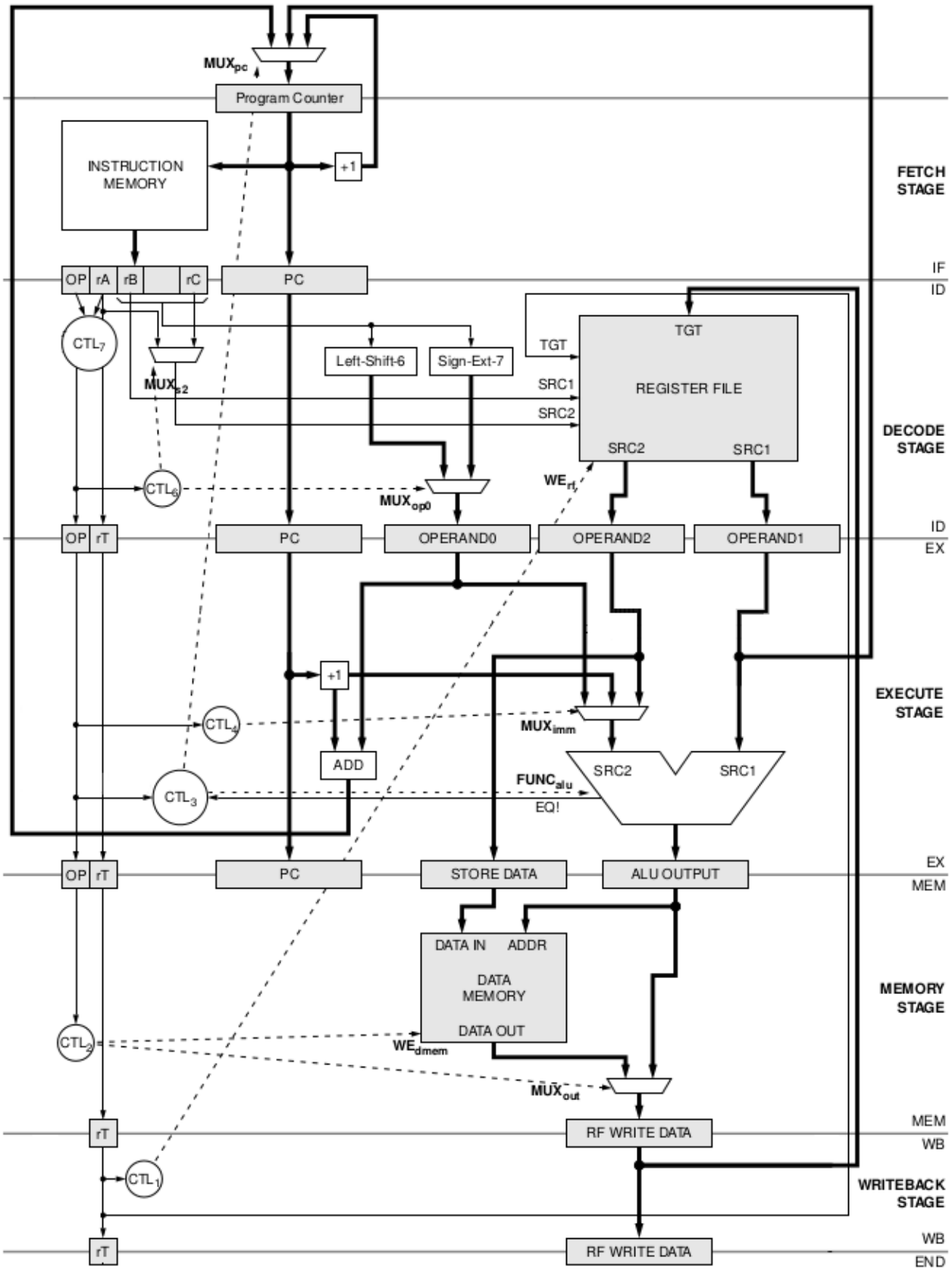
Ces pseudo-instructions ont les équivalences suivantes:

- **nop**: $add\ r0, r0, 0$
- **halt**: $jalr\ r0, r0$
- **lli**: $addi\ rA, rA, imm6$ avec $imm6 = imm\ \&\ 0x3f$ soit les 6 bits de poids faible.
- **movi**: *lui* et *lli*, soit 10 bits de poids fort (*lui*) et 6 bits de poids faible (*lli*).
- **.fill**: initialise une donnée à *immed*.
- **.space**: alloue *immed* donnée(s) (initialisée(s) à 0).

Un processeur avec pipeline

Il s'agit de réaliser ici un processeur RiSC-16 avec pipeline simple. Comme MIPS, ce pipeline est découpé en 5 étages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (MEM) et Write Back (WB). Seule une instruction peut se trouver à un étage et chaque étage dure un cycle d'horloge. Des buffers (registres) intermédiaires sont ajoutés entre chaque étage (IF/ID, ID/EX, EX/MEM, MEM/WB et WB/End) pour stocker à les résultats intermédiaires nécessaires aux étages suivants.

La figure ci-dessous décrit notre première version du pipeline. Les différents modules de contrôle (CTL_x) sont décrits à la suite.



CTL$_1$: Ce module contrôle le *write-enable* du fichier de registre. Si une donnée doit être écrite dans un registre, elle vient du registre MEM/WB. CTL1 doit uniquement lire rT:

- si rT est 000, write-enable (WE) est mis à 0;
- sinon, il est mis à 1.

CTL$_2$: Ce module contrôle le *write-enable* du *data memory* et l'operation de MUX$_{\text{out}}$. La seule

entrée de ce module est l'*opcode*.

- si l'instruction est une instruction d'écriture en mémoire (SW), alors le *write-enable* du *data memory* est activé.
- si on lit la mémoire (LW) il doit indiquer à MUX_{out} de choisir *Data out*; sinon, il choisit *Alu output*.

CTL$_3$: Ce module contrôle l'opération de l'ALU et de MUX_{pc}. Il décide cela en utilisant l'opcode et la valeur EQ! produite par l'ALU.

- l'opcode suffit à determiner FUNC_{alu} (qui sera un Nand, une addition ou la transmission de OPERAND0 pour LUI).
- MUX_{pc} est déterminé par EQ! et l'opcode:
  - si l'instruction est BEQ et que EQ! indique que les deux opérandes sont égaux alors MUX_{pc} doit choisir l'entrée PC+1+OPERAND0 calculée par le add dans l'*execute stage*,
  - si l'instruction est JALR, MUX_{pc} choisit la valeur de OPERAND1 produite dans l'*execute stage*,
  - sinon (autre instruction ou instruction BEQ avec EQ! indiquant que les deux opérandes sont différentes) MUX_{pc} choisit l'entré PC+1 calculé par le module *+1* du *fetch stage*.

CTL$_4$: Ce module contrôle l'operation de MUX_{imm} le multiplexer qui determine l'entrée src2 de l'ALU. La seule entrée de ce module est l'*opcode*.

- si l'instruction est JALR alors on choisit la valeur PC+1 venant de la boite *+1* de l'*execute stage*, pour pouvoir écrire PC+1 dans le registre.
- si l'intruction est ADDI, LUI, LW ou SW alors il sélectionne l'immédiat venant de OPERAND0,
- sinon, si l'intruction est ADD, NAND ou BEQ, il sélectionne OPERAND2 (correspondant à un registre).

CTL$_6$: Ce module contrôle l'opération de MUX_{op0} et de MUX_{s2}. Il n'utilise que l'*opcode* pour définir les actions de ces deux multiplexeurs. Il indique à MUX_{s2} s'il faut choisir entre le champ rA ou rC pour le deuxième registre opérand (src2):

- si l'instruction est ADD ou NAND alors on choisit rC,
- sinon on choisit rA.

Il indique à MUX_{op0} ce qu'il doit envoyer à OPERAND0:

- si l'instruction est LUI alors on choisis Left-Shift 6 (les 10 bits de l'immédiat suivis de 6 bits à zéro) ,
- sinon (pour ADDI, LW, SW, BEQ, et JALR) on choisit Sign-Ext 7 (l'immédiat signé sur 7 bits).

CTL$_7$: Ce module lit op sans la modifier et produit rT en fonction de op et rA.

- si op est SW ou BEQ rT vaut 000,
- sinon rT est rA.

Questions

La modélisation dans Logisim de ce processeur devra être incrémentale et modulaire. On définira donc **1 module par étage**. Dans le reste du document, la définition d'un module correspond à **la définition de ses entrées et de ses sorties** et l'implémentation à **la réalisation du circuit**.

1. **Étage IF**
    1. (1 point) Définissez le module IF.
    2. (1 point) Implémentez-le en utilisant une ROM pour *instruction memory*.
    3. Testez-le.
2. **Étage ID**
    1. (1 point) Définissez le module ID.
    2. (1 point) Définissez puis implémentez le module *register file* contenant les 8 registres (le registre 0 contient la constante 0).
    3. (1 point) Définissez puis implémentez le module CTL_7.
    4. (1 point) Définissez puis implémentez le module CTL_6.
    5. (1 point) Implémentez le module ID.
    6. Testez-le.
3. **Étage EX**
    1. (1 point) Définissez le module EX.
    2. (1 point) Définissez puis implémentez le module CTL_3.
    3. (1 point) Définissez puis implémentez l'ALU.
    4. (1 point) Définissez puis implémentez le module CTL_4.
    5. (1 point) Implémentez le module EX.
    6. Testez-le.
4. **Étage MEM**
    1. (1 point) Définissez le module MEM.
    2. (1 point) Définissez puis implémentez le module CTL_2.
    3. (1 point) Implémentez le module MEM en utilisant une RAM pour *data memory* (Attention au mode de fonctionnement choisi pour la RAM).
    4. Testez-le.
5. **Étage WB**
    1. (0.5 points) Définissez le module WB.
    2. (1 point) Définissez puis implémentez le module CTL_1.
    3. (0.5 points) Implémentez le module WB.
    4. Testez-le.
6. **Pipeline**
    1. (1 point) Implémentez le pipeline simple final en réutilisant les modules des étages.
    2. Testez-le.
7. **Assembleur RiSC-16**

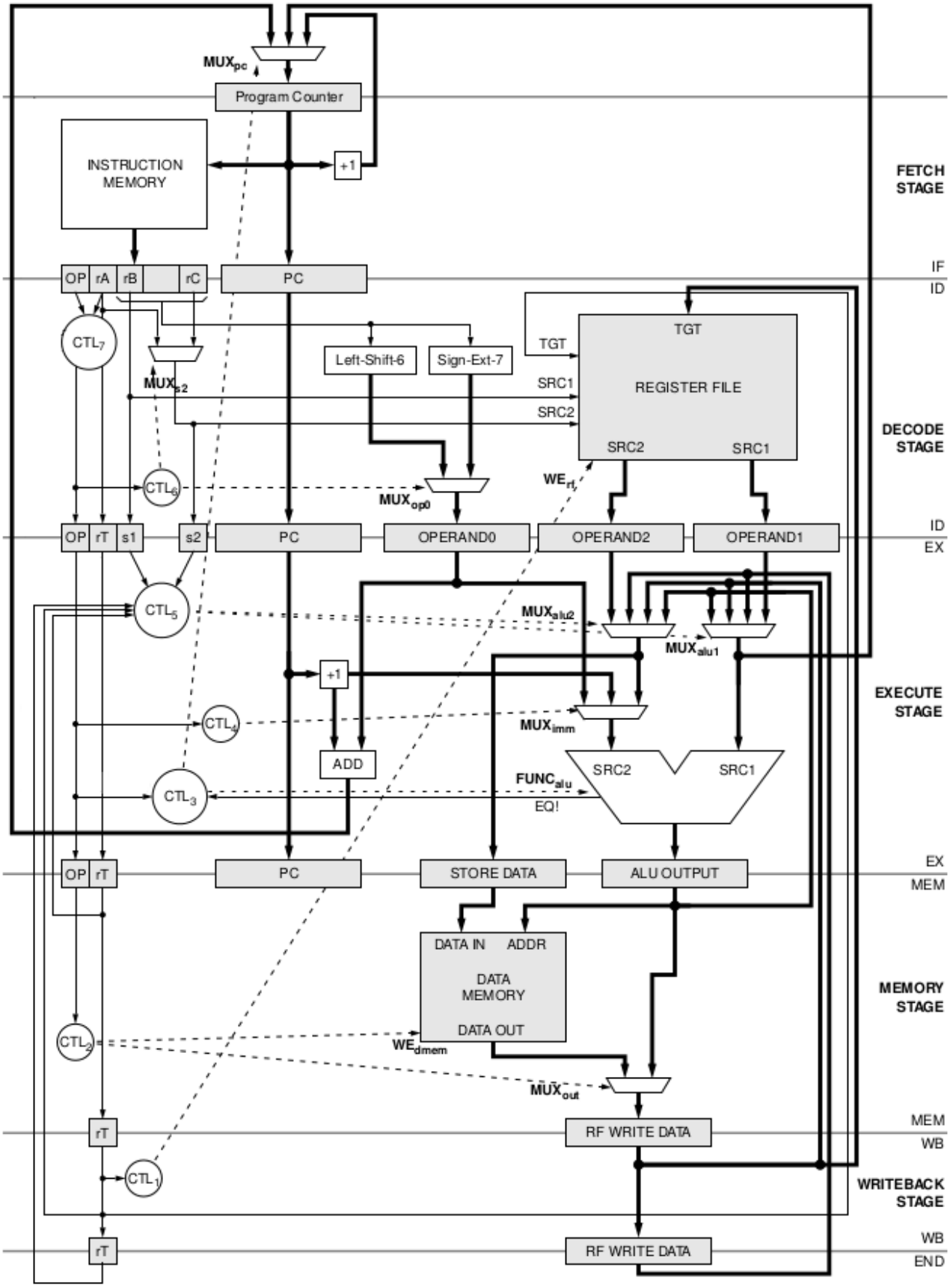
    Les réponses aux questions ci-après sont à joindre au rapport final.

```
lui r2, 0x8000
lw r1, r2, 1
```

1. (0.5 points) Dans le programme assembleur ci-dessus, quel problème peut se poser ? Modifier ce programme pour le rendre correct.
2. (2 points) Écrivez un programme assembleur couvrant les différentes opérations et illustrant le bon fonctionnement de votre implémentation. Vous serez attentif au problème exposé précédemment.

Un pipeline avec logique bypass

La logique *bypass* permet de transférer le résultat d'un étage au cycle i à un étage précédent au cycle $i + 1$ et ainsi ne pas avoir à attendre qu'une instruction sorte du pipeline pour réutiliser son résultat. La figure ci-dessous reprend la première version du pipeline en y ajoutant plusieurs *bypass*, soit EX $\rightarrow$ EX, MEM $\rightarrow$ EX et WB $\rightarrow$ EX.



Les modifications principales à apporter au circuit pour implémenter le bypass sont:

CTL$_5$: C'est le module principal qui gère le data forwarding notament en controlant l'opération de MUX$_{\text{alu1}}$ et MUX$_{\text{alu2}}$. Ces deux multiplexers ont la responsabilité de faire remonter les données depuis les registres pipelines vers l'execute stage. Ce module compare les registres s1 et s2 du execute stage aux registres rT des intructions précédentes (dans les 3 étages suivants):

- Si l'une des 3 instructions précédentes écrit dans *si*, MUX_{alui} doit séléctionner l'entrée correspondante au forward de la sortie correspondante.
- sinon, MUX_{alui} séléctionne l'entrée correspondante à *srci*

MUX_{s2}: La sortie de MUX_{s2} (gérée par CTL_6) est dédoublée pour pouvoir être déplacée dans le pipeline en même temps que l'éxécution correspondante. Cela simplifie la tâche de **CTL_5** qui peut donc accéder facilement au *s2* de l'instruction quand elle est dans l'execute stage.

Questions

1. **Étage WB**
   1. (1 point) Modifiez la définition du module WB en conséquence.
   2. (1 point) Modifiez l'implémentation.
2. **Étage MEM**
   1. (1 point) Modifiez la définition du module MEM en conséquence.
   2. (1 point) Modifiez l'implémentation.
3. **Étage EX**
   1. (1 point) Modifiez la définition du module EX en conséquence.
   2. (1 point) Définissez puis implémentez le module CTL_5.
   3. (1 point) Modifiez l'implémentation du module EX.
4. **Pipeline**
   1. (1 point) Modifiez l'implémentation du pipeline.
   2. Testez-le.
5. **Assembleur RiSC-16**

   Les réponses aux questions ci-après sont à joindre au rapport final.

```
lui r2, 10
beq r1, r2, label
addi r2, r2, 1
```

   1. (0.5 points) Dans le programme assembleur ci-dessus, quel problème peut se poser ? Modifier ce programme pour le rendre correct.
   2. (1 point) Écrivez (ou modifiez) un programme assembleur couvrant les différentes opérations et illustrant le bon fonctionnement de votre implémentation.

Mapping Memory

On souhaite ajouter la gestion de périphériques. La solution *mapping memory* consiste à statiquement associer une plage d'adresses à un périphérique. L'espace d'adressage est segmenté entre les différentes entrées/sorties qu'on désire supporter. Les entrées/sorties sont réalisées dans l'étage MEM.

Vous devrez **au minimum** ajouter les périphériques suivants (en plus de la RAM):

1. TTY

2. Keyboard

Et de manière **faculative** l'un des (ou les) périphériques suivants:

- Video Card (LED Matrix dans Logisim)
- Joystick

Chaque périphérique facultatif rapporte *1 point* pour l'implémentation et le code assembleur testable, soit *2 points* maximum.

Questions

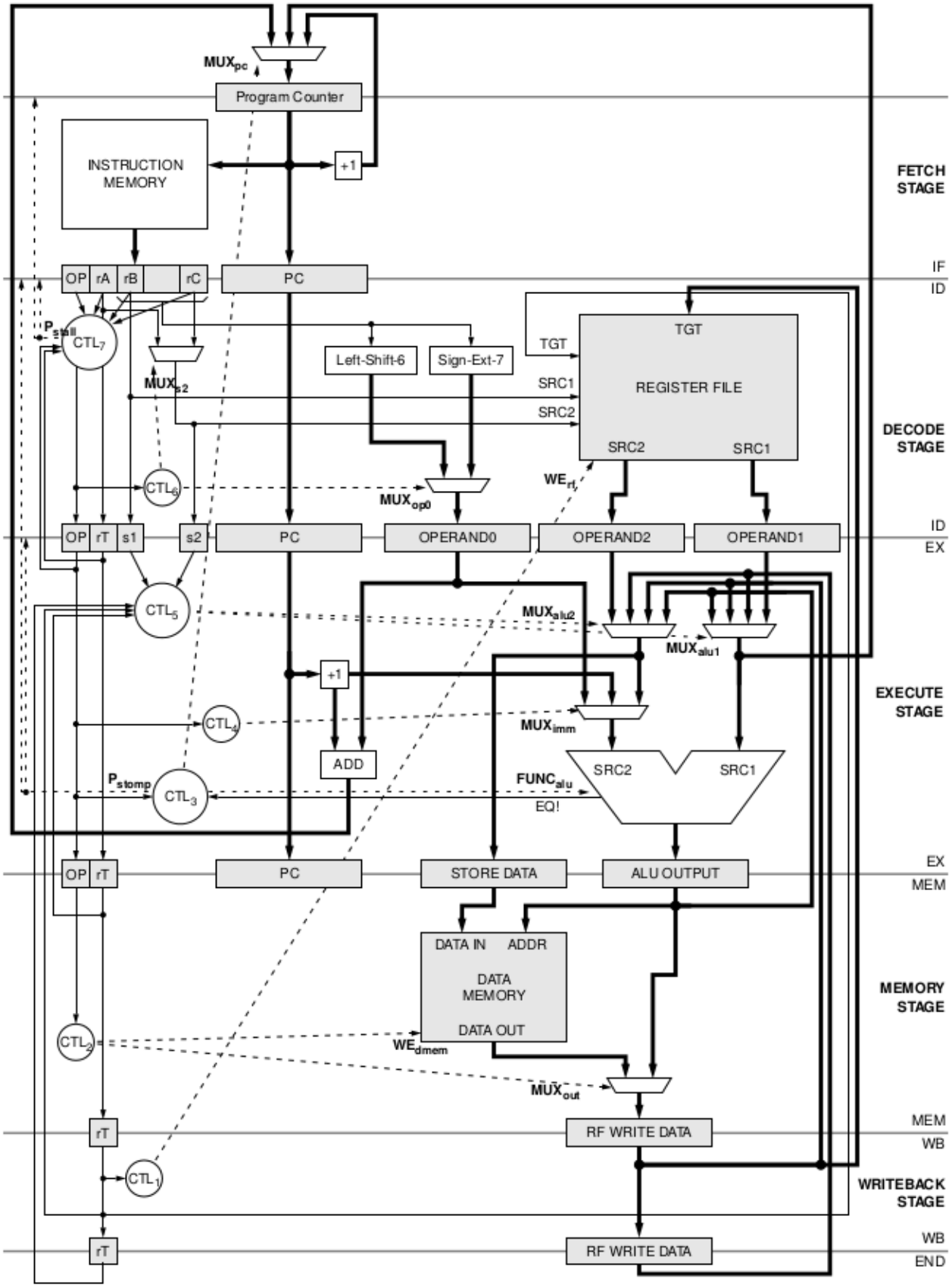
1. **Étage MEM**
    1. (1 point) Modifiez la définition et l'implémentation du module MEM pour externaliser la RAM.
2. **Memory Mapping**
    1. (1 point) Proposez une segmentation cohérente de l'espace d'adressage en fonction des périphériques que vous avez choisi de supporter.
    2. (1 point) Définissez et implémentez le module *memory mapping* en conséquence.
    3. Testez-le.
3. **Pipeline**
    1. (2 points) Modifiez le pipeline pour ajouter le module *memory mapping* et les périphériques choisis.
    2. Testez-le.
4. **Assembleur RiSC-16**

    Le(s) programme(s) est(sont) à joindre au rapport final.
    1. (2 points) Écrivez (ou modifiez) un (ou plusieurs) programme(s) assembleur utilisant les périphériques que vous avez choisi de supporter.

Un pipeline avec logique Stall et Stomp

La logique *Stomp* permet en cas de branchement d'évacuer du pipeline les instructions qui ne doivent pas être exécuté. La logique *Stall* permet de bloquer le pipeline en cas de dépendance de données ne pouvant être résolu avec un *bypass* dès le cycle suivant.

La figure ci-dessous reprend la version précédente du pipeline en y ajoutant les contrôles pour la logique *Stomp* et *Stall*.



IL faut apporter les modificatiosn suivantes aux CTL:

CTL$_3$: Ce module doit maintenant aussi controler les stomps. Ce module utilise l'opcode et EQ! pour décider si un STOMP doit avoir lieu.

- Si on procède à un saut (intruction Jal ou instruction BEQ et EQ! indique l'égalité) alors un STOMP doit avoir lieu. Tout les registres de IF/ID et ID/EX sont remplis par une instruction NOP.

- Sinon aucun STOMP n'a lieu.

CTL_7: Ce module va maintenant devoir aussi gérer les STALL. Si l'instruction actuellement dans l'execute stage est un LW écrivant sur un des registres que l'instruction dans decode utilise comme registre source, alors un STALL doit avoir lieu. Dans ce cas les registres PC et de la pipeline IF/ID doivent retenir leurs valeurs pendant un cycle le temps que le conflit soit réglé et un NOP est placé dans les registres ID/EX.

Questions

1. **Étage EX et Pipeline**

   1. (1 point) Modifiez l'implémentation du pipeline pour écraser les registres IF/ID et ID/EX en cas d'évènement *Stomp*.
   2. (1 point) Modifiez le module CTL_3 pour ajouter la logique *Stomp*.
   3. (1 point) Modifiez la définition et l'implémentation du module EX en conséquence.
   4. Testez-le.

2. **Étage ID et Pipeline**

   1. (1 point) Modifiez l'implémentation du pipeline pour retenir les valeurs des registres *Program Counter* et IF/ID en cas d'évènement *Stall*.
   2. (1 point) Modifiez le module CTL_7 pour ajouter la logique *Stall*.
   3. (1 point) Modifiez la définition et l'implémentation du module ID en conséquence.
   4. Testez-le.

3. **Assembleur RiSC-16**

   Le(s) programme(s) est(sont) à joindre au rapport final.

   1. (1 point) Écrivez (ou modifiez) un programme assembleur illustrant le bon fonctionnement de votre implémentation de la logique *Stall* et/ou *Stomp* (selon ce que vous avez implémenté).