

INF1600

Travail pratique 1 - **Corrigé**

Périphériques et architecture

Département de Génie Informatique et Génie Logiciel  
Polytechnique Montréal

# 1 Introduction et sommaire

Ce travail pratique a pour but de vous familiariser avec un processeur accumulateur disposant de fonctionnalités additionnelles. Les nouvelles instructions facilitent l'écriture de programmes concis et efficaces. Il sera question dans ce TP d'utiliser les instructions de branchements (permettant des boucles) et d'exploiter le registre MA pour faciliter les accès en mémoire. En vain, vous allez analyser et développer divers programmes sur [Code Machine](#) à partir des connaissances acquises au TP0 (opérations binaires, écritures et lectures d'entrées en C, etc.).

## 1.1 Remise

Voici les détails concernant la remise de ce travail pratique :

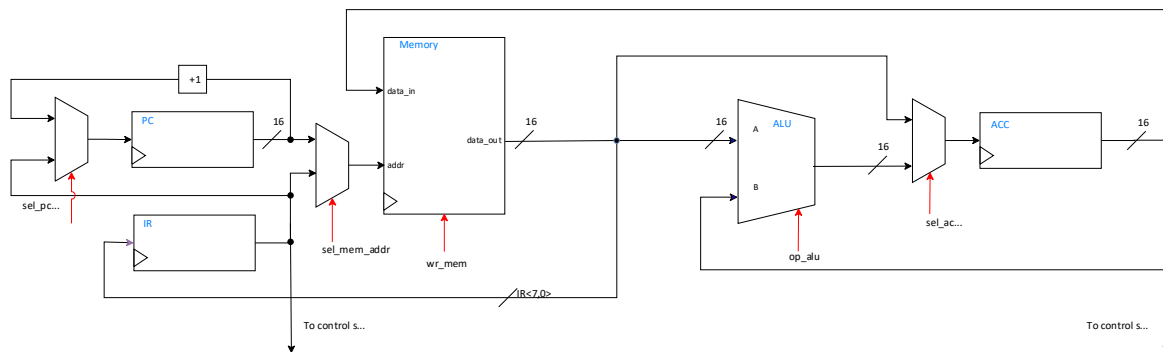
- Méthode : sur Moodle, une **seule remise par équipe**, incluant un rapport **PDF**. **Seules des équipes de deux (2) étudiants sont tolérées**, sauf avis contraire.
- Format: un dossier compressé intitulé <matricule1>-<matricule2>-<tp1\_section\_X>.**ZIP**, incluant les sources de vos programmes en C et en assembleur, de même qu'un rapport en format **.PDF**. Incluez une **page titre** où figurent les noms et matricules des deux membres de l'équipe, votre groupe de laboratoire, le nom et le sigle du cours, la date de remise et le nom de l'École. Dans une seconde page, incluez le **barème** de la section 1.2. Finalement, diverses captures d'écran **pertinentes** doivent figurer au sein de votre rapport. Celles-ci ne peuvent expliquer votre travail d'elles-mêmes ; les captures d'écran servent de support complémentaire. Une justification écrite est de mise.
- **Attention :** L'équipe de deux que vous formez pour ce TP sera **définitive** jusqu'au TP5. Il **ne sera pas possible** de changer d'équipe au cours de la session (sauf avis contraire).

## 1.2 Barème

Le travaux pratiques 1 à 5 sont notés sur 4 points chacun, pour un total de 20/20. Le TP1 est noté selon le barème suivant. Reproduisez ce tableau dans le document PDF que vous allez remettre.

<b>TP 1</b>		<b>/4,00</b>
<b>Section 2</b>		
Partie 1		/1,00
	Q1	/0,25
	Q2	/0,25
	Q3	/0,25
	Q4	/0,25
Partie 2		/0,75
	Q1 — <i>bon fonctionnement</i>	/0,10
	Q1 — <i>extensibilité</i>	/0,40
	Q1 — <i>discussion</i>	/0,25
<b>Section 3</b>		
Partie 1		/1,5
	Q1	/0,15
	Q2	/0,10
	Q3	/0,25
	Q4 — <i>bon fonctionnement</i>	/0,50
	Q4 — <i>commentaires, gestion mémoire, échange utilisateur</i>	/0,25
	Q4 — <i>extensible</i>	/0,25
Partie 2		/0,75
	Q1 — <i>bon fonctionnement</i>	/0,25
	Q1 — <i>facilement modifiable</i>	/0,50

## 2 Utilisation des instructions de branchement



**Figure 1** Architecture du processeur à accumulateur simple

Soit l'architecture du processeur accumulateur étudiée au TP0. Dans ce travail pratique, nous allons étendre le jeu d'instructions du processeur pour permettre des branchements avec et sans condition. Pour ce faire, le chemin de données entre le registre IR et le registre PC permet de modifier ce dernier au travers d'une adresse contenue dans l'adresse IR.ADR. Lorsqu'un branchement survient, PC prend la valeur précisée par IR.ADR.

**Tableau 1** Jeu d'instructions du processeur à accumulateur

Instruction	Description
add ADR	$ACC \leftarrow ACC + \text{Mémoire}[ADR]$
sub ADR	$ACC \leftarrow ACC - \text{Mémoire}[ADR]$
mul ADR	$ACC \leftarrow ACC \times \text{Mémoire}[ADR]$
st ADR	$\text{Mémoire}[ADR] \leftarrow ACC$
ld ADR	$ACC \leftarrow \text{Mémoire}[ADR]$
stop	Arrêt du programme
br ADR	$PC \leftarrow ADR$
brz ADR	$ACC = 0 ? PC \leftarrow ADR : PC \leftarrow PC + 1$
brnz ADR	$ACC \neq 0 ? PC \leftarrow ADR : PC \leftarrow PC + 1$

Le **Tableau 1** donne la liste des instructions acceptées par ce processeur. Les branchements conditionnels (brz et brnz) dépendent de la valeur courante dans l'accumulateur. Le branchement br, quant à lui, effectue un branchement inconditionnel, de sorte que PC prend toujours l'adresse contenue dans IR lors de l'exécution de cette instruction.

## 2.1 Partie 1 :

Copiez le code du fichier `partie-2-1.asm` et veuillez l'insérer dans l'éditeur de [Code Machine](#) du processeur à accumulateur **sans registre MA**. Après avoir remplacé la valeur de `a` par  $3 + \text{abs}((\text{MATRICULE\_1} + \text{MATRICULE\_2}) \% 13)$ , répondez aux questions qui suivent.

**Important :** n'oubliez pas d'inclure des captures d'écran au sein de votre rapport lorsque nécessaire.

Q0/0,00 point Indiquez votre valeur de `a` dans votre rapport.

Cette valeur dépend de  $3 + \text{abs}((\text{MATRICULE\_1} + \text{MATRICULE\_2}) \% 13)$ .

Q1/0,25 point Quelles sont les structures de contrôle utilisées au sein de ce programme ?

Nous avons des commandes à étiquettes, notamment des sauts inconditionnels (`br`), de même que des sauts conditionnels (`brnz` et `brz`). Aussi accepter : boucles conditionnelles avec `if / else`. Il faut absolument avoir un lien entre l'instruction et l'équivalence assembleur.

Q2/0,25 point Quel est le contenu en mémoire à l'adresse `0x0010` (qui diffère selon votre valeur de `data`) à la fin de l'exécution de ce programme ?

Si votre valeur de `a` est paire, un 0 apparaîtra à l'adresse `0x0010`. Dans le cas contraire (si votre valeur de `a` est impaire), un 1 apparaîtra à l'adresse `0x0010`.

Q3/0,25 point Que fait ce programme ? Répondez à la question en décrivant le principe de fonctionnement du programme en évitant de référer à son contenu ligne par ligne.

Déterminez si votre valeur de `a` est paire ou impaire et stockez le résultat dans `res`. Notion de parité présente pour obtenir tous les points.

Q4/0,25 point Si le processeur est sur 32 bits et n'utilise pas le bit le plus significatif pour représenter le signe, quel serait le résultat du programme ? Justifier votre réponse.

Le bit le plus significatif ne représente pas le signe. Ainsi, la valeur maximale pouvant être représentée sur 32 bits est  $2^{32} - 1$ , un chiffre impair. Ainsi, le résultat sera 1 dans `res`.

## 2.2 Partie 2 :

Avec l'éditeur d'assembleur de [Code Machine](#) pour le processeur à accumulateur **sans registre MA**, écrivez un programme qui permet de trouver le maximum de deux valeurs **disjointes, de même signe**, `a` et `b`. Dans le cas positif, vous pouvez assumer que `a` et `b` se situent toujours entre 1 et 99. Dans le cas négatif, vous pouvez assumer que `a` et `b` se situent toujours entre -99 et -1.

Le maximum doit être stocké dans la variable `res` dans la section `data` à la fin de l'exécution du programme. Par exemple, pour `a=-10` et `b=-13`, le programme retournera `res=-10`. Pour `a=10` et `b=13`, le programme retournera `res=13`. Le choix de `a` et de `b` sont à votre discrétion, mais nous devrions pouvoir les changer aisément (toujours en respectant les critères mentionnés ci-haut) pour obtenir un nouveau maximum (0,10 point pour le fonctionnement statique du programme et 0,40 point pour son extensibilité). Discutez de votre approche et des limites qui s'imposent sur le choix du processeur (0,25 point).

**Approche statique :** vous savez d'avance le signe de `a` et de `b`, vous faites une boucle en décrémentant ou en incrémentant `a` et `b` selon leur signe jusqu'à ce qu'une valeur atteigne 0. Dans le cas où `b` atteint 0 avant `a`, sa valeur sera stockée dans `res` pour une valeur négative. Pour un signe positif, ce sera la valeur de `a` qui sera stockée dans `res`.

**Approche dynamique :** vous n'estimez pas le signe de  $a$  et  $b$ . Vous avez donc 2 possibilités à tester, soit  $a$  positif et  $b$  positif ou  $a$  négatif et  $b$  négatif. Sur Code Machine, la valeur de  $a$  et de  $b$  se doit d'être petite pour résoudre ce problème, sinon vos instructions vont vite dépasser 512 (maximum représentable sur Code Machine). Vous devriez ainsi avoir 2 directives, dont la première peut consister à estimer que  $a$  et  $b$  ont tous deux le signe négatif. Vous faites un branchement répétitif incrémental 98 fois; si aucune des valeurs n'a atteint 0, vous savez logiquement que vos deux valeurs sont positives. Vous faites ainsi un second branchement répétitif décremental 98 fois ; la première valeur qui atteint 0 n'est pas le maximum.

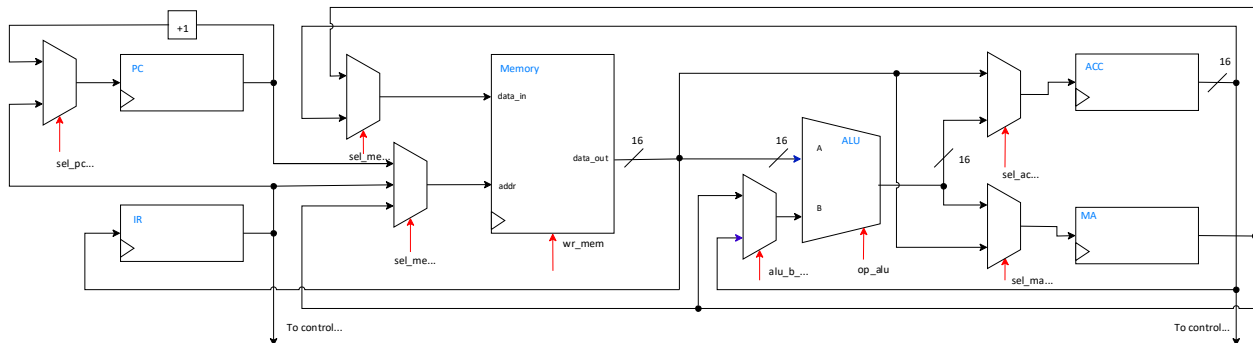
**Approche attendue tirée du meilleur groupe (matricules 1908977 et 2154216) :**

```
.text
ld nombre1
sub nombre2
st diffence1
ld nombre2
sub nombre1
st diffence2
iteration:
ld diffence1
sub one
st diffence1
brz nombre1Max
ld diffence2
sub one
st diffence2
brz nombre2Max
br iteration
nombre1Max:
ld nombre1
st res
br fin
nombre2Max:
ld nombre2
st res
br fin
fin:
stop
.data
nombre1:-73
nombre2:-70
diffence1:0
diffence2:0
one:1
res:0
```

### 3 Ajout du registre MA au processeur à accumulateur

Le processeur accumulateur a été enrichi d'un registre supplémentaire, le registre MA. Ce registre facilite les accès en mémoire. Il sert à garder l'adresse de l'espace en mémoire que le programme désire accéder.

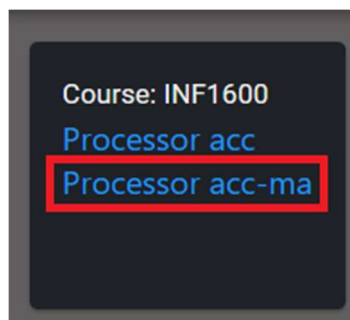
Soit la nouvelle architecture du processeur à accumulateur avec registre MA:



**Figure 2** Processeur à accumulateur avec registre MA

La valeur du registre MA peut être enregistrée en mémoire, mais elle peut aussi être utilisée pour préciser une adresse en mémoire que le processeur peut accéder en lecture ou en écriture. Le registre MA sert à implémenter l'adressage indirect puisqu'il permet d'écrire un programme où les adresses espaces mémoires ne sont plus donnés en absolu dans les instructions, mais sont évalués dynamiquement durant l'exécution de celui-ci.

Le processeur à accumulateur avec registre MA est accessible sur [Code Machine](#). Sur la fenêtre d'accueil de l'outil, sélectionnez l'option « Processor acc-ma » :



Le lien mène à la nouvelle architecture du processeur accumulateur, telle qu'illustrée à la **Figure 2**. La syntaxe de programmation de ce processeur est identique à celle du processeur à accumulateur simple et le jeu d'instructions présenté à la **Section 2** demeure valide pour cette architecture. Ainsi, ce processeur peut exécuter n'importe quel programme compatible avec la première version du processeur à accumulateur. De plus, le jeu d'instructions de ce processeur est étendu aux instructions données au **Tableau 2**. L'instruction sub permet de soustraire à la valeur présente dans l'accumulateur une valeur en mémoire. Les instructions shl et shr implémentent un décalage à gauche et à droite, respectivement. Finalement, le reste des instructions données au **Tableau 2** permettent de manipuler le registre MA de différentes façons.



Tableau 2 - Jeu d'instructions étendu du processeur à accumulateur

Instruction	Description
sub ADR	$ACC \leftarrow ACC - \text{Mémoire}[ADR]$
shl	$ACC \leftarrow ACC \ll 1$
shr	$ACC \leftarrow ACC \gg 1$
adda ADR	$MA \leftarrow MA + \text{Mémoire}[ADR]$
suba ADR	$MA \leftarrow MA - \text{Mémoire}[ADR]$
addx	$ACC \leftarrow ACC + \text{Mémoire}[MA]$
subx	$ACC \leftarrow ACC - \text{Mémoire}[MA]$
lda ADR	$MA \leftarrow \text{Mémoire}[ADR]$
sta ADR	$\text{Mémoire}[ADR] \leftarrow MA$
ldi	$ACC \leftarrow \text{Mémoire}[MA]$
sti	$\text{Mémoire}[MA] \leftarrow ACC$

### 3.1 Partie 1

Copiez le code du fichier partie-3-1.asm et veuillez l'insérer dans l'éditeur de [Code Machine](#) du processeur à accumulateur avec registre MA. Après avoir remplacé la valeur de data par  $3 + |\sum_{i=1}^7 X_i - \sum_{j=1}^7 Y_j|$  si  $X_i$  désigne le  $i^{\text{ème}}$  chiffre du numéro de matricule de l'étudiant 1 de matricule  $X$  et si  $Y_j$  désigne le  $j^{\text{ème}}$  chiffre du numéro de matricule de l'étudiant 2 de matricule  $Y$ , répondez aux questions qui suivent.

**Important :** n'oubliez pas d'inclure des captures d'écran au sein de votre rapport lorsque nécessaire.

Q0/0,00 point Indiquez votre valeur de data dans votre rapport.

Cette valeur dépend de  $3 + |\sum_{i=1}^7 X_i - \sum_{j=1}^7 Y_j|$  si  $X_i$  désigne le  $i^{\text{ème}}$  chiffre du numéro de matricule de l'étudiant 1 de matricule  $X$  et si  $Y_j$  désigne le  $j^{\text{ème}}$  chiffre du numéro de matricule de l'étudiant 2 de matricule  $Y$ .

Q1/0,15 point Quelles sont les structures de contrôle utilisées au sein de ce programme ?

Nous avons des commandes à étiquettes, notamment des sauts inconditionnels (br), de même que des sauts conditionnels (brnz et brz). Aussi accepter : boucles conditionnelles avec if / else. Il faut absolument avoir un lien entre l'instruction et l'équivalence assembleur.

Q2/0,10 point Quel est le contenu en mémoire à l'adresse 0x002A (qui diffère selon votre valeur de data) à la fin de l'exécution de ce programme ?

Devrait être 0 ou 1.

Q3/0,25 point Que fait ce programme ? Répondez à la question en décrivant le principe de fonctionnement du programme en évitant de référer à son contenu ligne par ligne.

**Indice :** Pour quel(s) valeur(s) de data ne trouve-t-on pas un 1 à l'adresse 0x002A ?

Le programme permet de **masquer** le bit le moins significatif de n'importe quel nombre placé dans data, à condition que ce nombre soit représenté correctement au nombre de bits du processeurs. Par exemple, si mon data vaut 0xFF et que j'ai une architecture sur 16 bits, je dois faire du *padding* pour m'assurer du bon fonctionnement. Tous les points si vous aviez dit une application concrète du programme, et non une explication linéaire (ligne par ligne).

**Alternative possible relevée par quelques étudiants :** le programme permet aussi de savoir la parité de data, à condition que data soit au moins de valeur  $\frac{N}{2}$  où  $N$  correspond au nombre de bits du processeur. Pour un processeur 8 bits et data = 0b1000 0001, le programme retournerait 1 (impair). Pour data = 0b1111 0000, le programme retournerait 0 (pair).

Q4/1 point Écrivez votre version en langage **C** du programme. Ce dernier devrait compiler et s'exécuter sans erreur, et ce, avec le même comportement et le même résultat que le code en assembleur fourni (0,5 point). Votre programme devrait inclure des commentaires, un échange (printf) avec l'utilisateur et une gestion intelligente de la mémoire qui devrait être libérée à la fin du programme (aucune fuite de mémoire) (0,25 point). Enfin, votre programme se doit d'être facilement extensible et modifiable (notamment en ce qui concerne la variable data) (0,25 point).

**Exemple typique :**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

int getSize(int data) {
    int length = 0;
    while (data != 0) {
        data >>= 1;
        length += 1;
    }
    return --length;
}

int main() {
    int data = 0xFF;
    int dataLength = getSize(data);
    printf("res=%d", data << dataLength >> dataLength);
    return 0;
}
```

## 3.2 Partie 2

Q1/0,75 point À l'aide du jeu d'instructions étendu du processeur à accumulateur avec registre MA, écrivez un programme **assembleur** qui permet de calculer le  $7 + \text{abs}(\text{MATRICULE\_1} + \text{MATRICULE\_2}) \% 13$  terme de la suite de [Fibonacci](#). Le programme doit être le plus court possible et il ne doit seulement garder en mémoire que les trois derniers termes calculés (valeur courante de la suite, celle précédente et celle qui précède la précédente) (0,25 point). À noter que 0,50 point est accordé à cette question si le code fourni peut être **facilement modifié** pour calculer le dernier  $M$  terme voulu de la suite de Fibonacci, et ce, en changeant simplement la valeur **d'une variable** dans la partie .data du programme. **Attention au plagiat.**

**Devrait utiliser MA.**

**Approche attendue tirée du meilleur groupe (matricules 1908977 et 2154216) :**

**À noter que cette approche pourrait encore être réduite/simplifiée, mais le but premier n'était pas l'optimisation, mais plutôt la fonctionnalité.**

```
.text
ld nbTermsFibo
sub one
brz firstFibo
sub one
brz secAndThirdFibo
sub one
brz secAndThirdFibo
st nbTermsFibo
br iteration
firstFibo:
ld zero
st presentValue
br end
secAndThirdFibo:
ld one
st presentValue
br end
iteration:
lda lastValue
adda presentValue
ld lastValue
st secondLastValue
ld presentValue
st lastValue
sta presentValue
ld nbTermsFibo
sub one
st nbTermsFibo
brz end
br iteration
end:
stop
.data
nbTermsFibo:6
presentValue:1
lastValue:1
secondLastValue:0
zero:0
one:1
```