POLYTECHNIQUE
MONTRÉAL

INF1600
Architecture des micro-ordinateurs

TP1

Groupe 02 (B2)

Soumis par:
Charles de Lafontaine - **2076524**
Geneviève Pelletier-Mc Duff - **2088742**

Le 17 février 2021

Barème de correction

TP 1			**/4,00**
Exercice 1			/1,50
	Q1	/0,50	
	Q2	/0,50	
	Q3	/0,50	
Exercice 2			/1,00
	Q1	/0,25	
	Q2	/0,25	
	Q3	/0,25	
	Q4	/0,25	
Exercice 3			/1,50
	Q1	/0,50	
	Q2	/1,00	

INF1600 - Rapport du TP1

Exercice 1 : Opérations binaires et de décalage

Pour les questions 1 à 3 de l'exercice 1, veuillez vous référer au fichier :
2088742_2076524_tp1exo1.c

Exercice 2 : Architecture simple d'un microprocesseur

Q1 : *Donnez le contenu de la mémoire qui contient le program_0 sous format hexadécimale, de l'adresse 0x0000 à l'adresse 0x000F. Vous pouvez fournir un tableau pour aider la lisibilité de votre réponse.*

En exécutant le *program_0*, il est possible d'observer le contenu de la mémoire à chacune des adresses (voir **Figure 1**). On remarque que le contenu de la mémoire est présenté en base décimale. Nous avons converti les données en mémoire en base hexadécimale et nous les avons regroupées dans le **Tableau I**, ce qui nous a permis de constater que la mémoire était composée à la fois des instructions et des nombres que nous avons mis en mémoire. Pour les instructions, l'octet le plus significatif représente l'opération, et le moins significatif, à l'adresse. Par exemple, à la case mémoire 0, le code hexadécimal 0x0307 correspond à l'opération *load* (LD) (0x0300) la case de la mémoire #7 (0x0007) dans l'accumulateur. Ainsi, les instructions qui sont présentes dans le programme se trouvent en haut de la mémoire, soit aux adresses les moins significatives, et correspondent à l'ordre d'apparition dans le code *vhdl* (voir **Tableau I**).

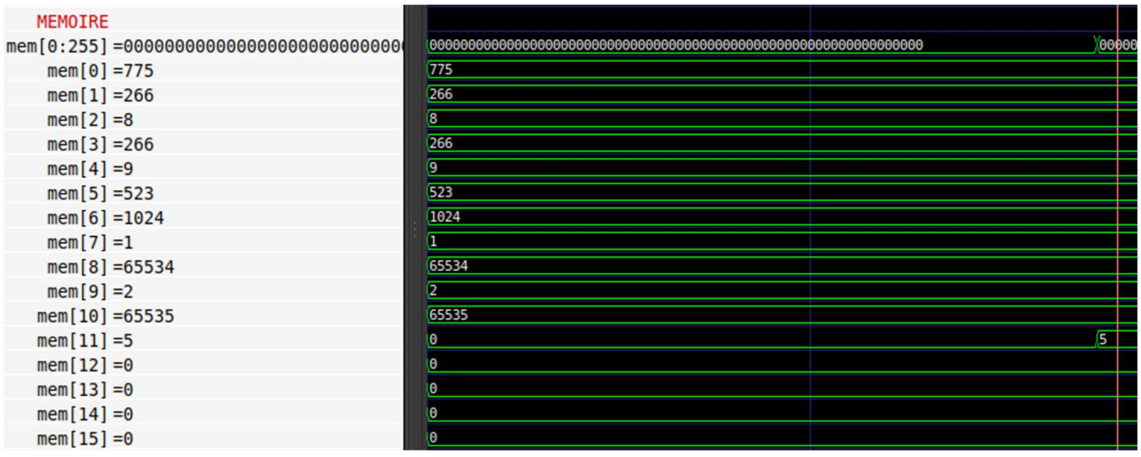

Figure 1. Capture d'écran présentant le contenu de la mémoire en base décimale pour les 16 premières adresses, soit de 0x0000 à 0x000F suite à l'exécution du *program_0*.

Tableau I. Contenu de la mémoire avant et après l'exécution du *program_0* en format hexadécimale pour les 16 premières adresses, soit de 0x0000 à 0x000F.

Contenu de la mémoire	**Mémoire à l'initiation du program_0**	**Mémoire à la fin de l'exécution du program_0**	**Instructions équivalentes dans le programme**
mem[0x0000]	0x0307		(ld, 7)
mem[0x0001]	0x010A		(mul, 10)
mem[0x0002]	0x0008		(add, 8)
mem[0x0003]	0x010A		(mul, 10)
mem[0x0004]	0x0009		(add, 9)
mem[0x0005]	0x020B		(st, 11)
mem[0x0006]	0x0400		(stop, 0)
mem[0x0007]	0x0001		+1
mem[0x0008]	0xFFFE		int16_to_uint16(-2)
mem[0x0009]	0x0002		+2
mem[0x000A]	0xFFFF		int16_to_uint16(-1)
mem[0x000B]	0x0000	0x0005	**Avant :** int16_to_uint16(0) **Après :** résultat de l'opération mathématique
mem[0x000C]	0x0000		others => 0
mem[0x000D]	0x0000		others => 0
mem[0x000E]	0x0000		others => 0
mem[0x000F]	0x0000		others => 0

Q2 : *Quelle valeur contient l'accumulateur (ACC) à la fin de l'exécution du programme program_0 ? Justifiez votre réponse par une capture d'écran de GTKwave et un calcul. À quelle adresse dans la mémoire principale (mem) cette valeur est-elle stockée ?*

L'accumulateur (*ACC*) contient la valeur 5 à la fin de l'exécution du programme *program_0* comme il est possible de constater à la **Figure 2**.

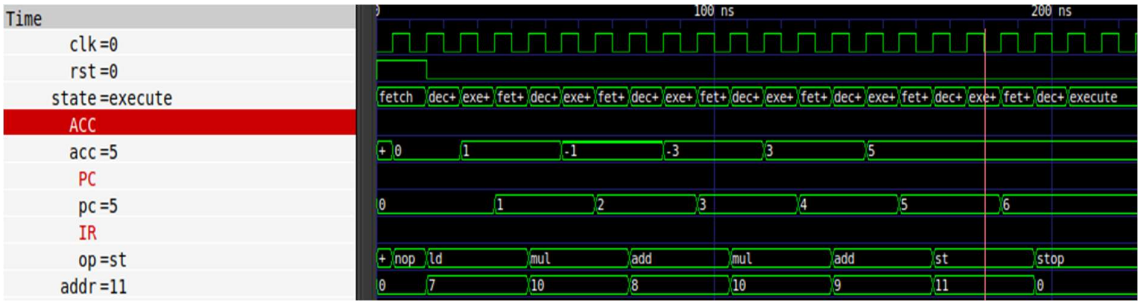

Figure 2. Capture d'écran de la simulation du *program_0* avec *GTKwave* qui présente le contenu de l'accumulateur (*ACC*) à la fin de l'exécution.

Cette valeur sera stockée à l'adresse mémoire 11 (ou 0x000B) grâce à la commande *store* (ST) qui prend le contenu de l'accumulateur et l'inscrit à une case mémoire indiquée. Il est possible de valider la valeur présente dans l'accumulateur en réalisant les mêmes étapes de calcul que le programme, soit :

Instructions réalisées par le programme 0 :

Ld	Mem[7]	=	1
MUL	Mem[10]	=	65535
ADD	Mem[8]	=	65534
MUL	Mem[10]	=	65535
ADD	Mem[9]	=	2
ST	Mem[11]	=	RESULT
STOP			

Ces dernières correspondent à ce calcul :

1) $Mem[7] * Mem[10] = Résultat$
2) $Résultat + Mem[8] = Résultat_2$
3) $Résultat_2 * Mem[10] = Résultat_3$
4) $Résultat_3 + Mem[9] = Résultat_4$
5) Mettre le $Résultat_4$ dans la mémoire à l'adresse 11 (0x000B)

Puis, nous pouvons remplacer par le contenu des cases mémoires :

6) $1 * 65534\ (correspond\ à\ -1) = -1$
7) $-1 + 65534\ (correspond\ à\ -2) = -3$
8) $-3 * 65534\ (correspond\ à\ -1) = 3$

9) $3 + 2 = 5$

10) Stocker le 5 dans la mémoire à l'adresse 11 (0x000B)

Soit le calcul suivant : $(((1 * (-1)) + (-2)) * -1) + 2 = 5$

Veuillez noter que comme il est possible de voir au **Tableau I**, la case mémoire 10 qui a le nombre 65535 correspond en fait à la valeur -1 puisqu'il est considéré comme étant un int16, soit une variable signée. Donc, en complément à deux, la valeur hexadécimale 0xFFFF correspond à -1. Par le même principe, la valeur 65534, ou 0xFFFE correspond à -2. Cela a été possible grâce aux lignes de code 19 et 21 pour convertir un type signé (*int16*) en un type normalement non signé (*uint16*).

Q3 : *Le programme calcule une expression mathématique connue, qu'elle est-elle ?*

En utilisant les paramètres écrits en commentaires aux lignes 7 à 11, il est possible de déduire l'expression mathématique. En effet, le programme 0 permet de calculer l'expression suivante :

$$(ax + b)x + c = y$$

où : a = Mem[7], b = Mem[8]; c = Mem[9], x = Mem[10], y = Mem[11]

En développant l'expression, on trouve la formule suivante qui correspond à l'équation quadratique de deuxième degré.

Formule : $y = ax^2 + bx + c$

Confirmation que nous obtenons bel et bien la bonne valeur à la case mémoire 11 avec la formule mathématique:

$$Mem[11] = Mem[7] * (Mem[10])^2 + Mem[8] * Mem[10] + Mem[9]$$
$$Mem[11] = 1 * (-1)^2 + (-2) * (-1) + 2$$
$$Mem[11] = 1 * (-1)^2 + (-2) * (-1) + 2$$
$$Mem[11] = 5$$

Le résultat est exact puisqu'il correspond au contenu de la case mémoire 11 à la fin de l'exécution du programme, ce qui confirme l'exactitude de la formule.

Q4 : *Modifiez le contenu de program_1 dans acc_proc_programs.vhd afin d'implémenter un programme qui calcule les six (6) premières valeurs d'une suite numérique S(n), soit S(0) à S(5), et les stocke séquentiellement en mémoire. Les deux premières valeurs S(0) = 1 et S(1) = 2 sont déjà présentes en mémoire. On admettra que S(n+2) = 1 + S(n+1) + S(n). Attention, en ajoutant de nouvelles instructions, la position en mémoire de ces données peut changer.*

Nous pouvons voir à la **Figure 3** notre programme permettant de calculer et stocker séquentiellement les 6 premiers termes de la suite spécifiée. Nous avons testé notre programme en lançant la simulation sur *GTKwave*, ce qui nous a permis de confirmer son bon fonctionnement (voir **Figure 4**). Les six premières valeurs de la suite – soit { 1, 2, 4, 7, 12, 20 } – sont bien stockées séquentiellement en mémoire à partir de la case 23.

```
-- Programme utilise pour Q4
constant program_1 : memtype := (
   -- ECRIRE VOTRE PROGRAMME ICI
   -- UNE LIGNE PAR INSTRUCTION
   +1,                      -- 0
   +2,                      -- 1
   to_uint16((ld, 0)),      -- 2
   to_uint16((st, 23)),     -- 3 : élément S(0) de la suite, stocké à la case 23 en mémoire
   to_uint16((ld, 1)),      -- 4
   to_uint16((st, 24)),     -- 5 : élément S(1) de la suite, stocké à la case 24 en mémoire
   to_uint16((ld, 0)),      -- 6
   to_uint16((add, 0)),     -- 7
   to_uint16((add, 1)),     -- 8
   to_uint16((st, 25)),     -- 9 : élément S(2) de la suite, stocké à la case 25 en mémoire
   to_uint16((ld, 0)),      -- 10
   to_uint16((add, 1)),     -- 11
   to_uint16((add, 25)),    -- 12
   to_uint16((st, 26)),     -- 13 : élément S(3) de la suite, stocké à la case 26 en mémoire
   to_uint16((ld, 0)),      -- 14
   to_uint16((add, 25)),    -- 15
   to_uint16((add, 26)),    -- 16
   to_uint16((st, 27)),     -- 17 : élément S(4) de la suite, stocké à la case 27 en mémoire
   to_uint16((ld, 0)),      -- 18
   to_uint16((add, 26)),    -- 19
   to_uint16((add, 27)),    -- 20
   to_uint16((st, 28)),     -- 21 : élément S(5) de la suite, stocké à la case 28 en mémoire
   to_uint16((stop, 0)),    -- FIN
   others => 0
);
```

Figure 3. Capture d'écran du *program_1* de l'exercice 2 de la question 2 permettant le calcul des 6 premiers termes de la suite suivante : *S(n+2) = 1 + S(n+1) + S(n) où S(0) = 1 et S(1) = 2.* Veuillez vous référer au fichier : *2088742_2076524_tp1exo2.vhd*

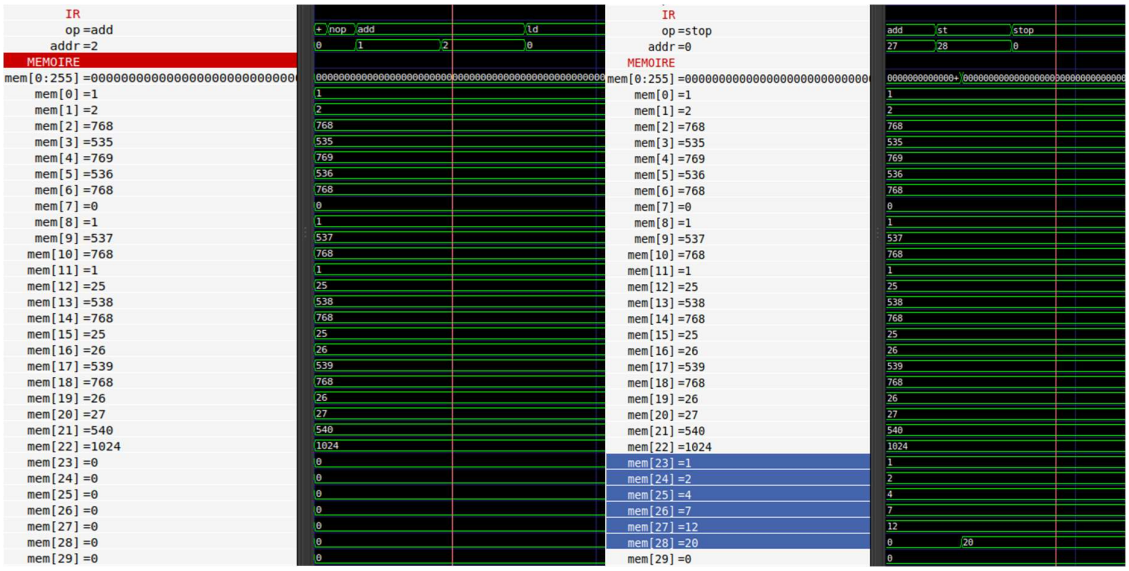

Figure 4. Captures d'écran présentant la simulation avec le logiciel *GTKwave* du *program_1* au début (gauche) et à la fin (droite). Il est possible de constater la présence en mémoire des six premiers termes de la suite qui sont surlignés en bleu.

Exercice 3 : Modes d'adressage et branchements

Q1 : *Donnez le contenu du registre ACC quand le processeur a terminé d'exécuter program_0. Expliquez brièvement ce que fait le programme.*

Le programme permet de décrémenter par 1 le chiffre 2 (où tout nombre placé dans la case mémoire 7) jusqu'à l'obtention de 0. Plus précisément, la première commande est un *load* qui permet de mettre le contenu de la case mémoire 7 dans l'accumulateur. Puis, la seconde commande permet d'incrémenter le compteur de programme (PC) à la valeur 3. Ainsi, cela saute l'instruction de la case mémoire 2, soit un *load* de la case mémoire 8. Puis, à l'adresse mémoire 3, l'opération effectuée est un *brz* qui permet de changer la valeur du compteur de programme à 5 si le contenu de l'accumulateur est nul. Étant donné que ce n'est pas le cas ($ACC = 2$), on continue l'exécution en ajoutant le contenu de la case mémoire 9 à l'accumulateur, donc 2 + (-1) est effectué ($ACC = 1$). Ensuite, la commande *brnz* permet de changer la valeur du compteur de programme pour faire une boucle si et seulement si la valeur de l'accumulateur est différente de 0. Puisque c'est le cas ($ACC = 1 \neq 0$), on remet le compteur de programme à 1 et on recommence les opérations. Donc, l'instruction associée à la case 1 est la commande *br* qui permet de sauter la valeur 2 du compteur de programme. À l'instruction 3, nous avons l'opération *brz* qui, mais puisque la valeur de *ACC* n'est pas nulle, on continue à l'instruction 4. Puis, nous additionnons la valeur de la case 9, soit -1, à celle de l'accumulateur pour un total de 0. Étant donné que l'accumulateur a pour valeur 0, la condition de l'opération *brnz* ($ACC \neq 0$) n'est plus remplie, alors le programme va arrêter avec la commande *stop*. En conclusion, ce programme a permis de faire un décompte de type {2→1→0}dans l'accumulateur en utilisant des boucles. La **Figure 5** présente la simulation de *program_0* avec le logiciel *GTKwave*.

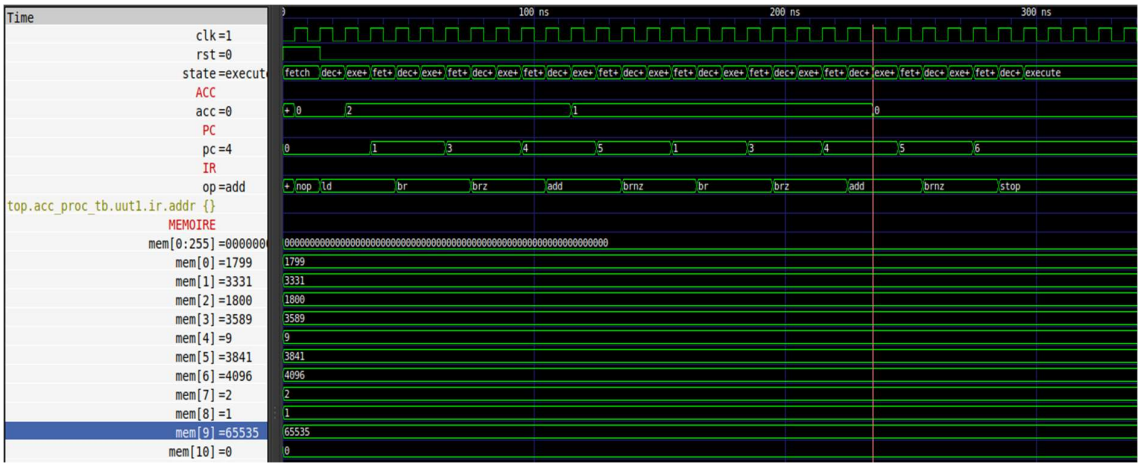

Figure 5. Capture d'écran de la simulation du *program_0* avec *GTKwave* qui présente le comportement et le contenu de la mémoire, de l'accumulateur et du compteur de programme durant la simulation.

Q2 : *Modifiez le contenu de program_1 dans acc_proc_programs.vhd afin d'implémenter un programme qui calcule les six (6) premières valeurs d'une suite numérique S(n), soit S(0) à S(5), et les stocke séquentiellement en mémoire. Les deux premières valeurs S(0) = 1 et S(1) = 2 sont déjà présentes en mémoire. On admettra que S(n+2) = 1 + S(n+1) + S(n). Comparez le nombre d'instructions nécessaires de ce programme par rapport au nombre d'instructions obtenus à l'exercice 2.*

Pour le *program_1*, nous avons décidé de stocker les termes de la suite dans les cases mémoire de 1 à 6. À la case 0, nous avons mis la valeur +1 qui sera ajoutée à chacun des calculs d'un terme. Pour les deux premiers termes (aux cases 1 et 2), nous les avons ajoutés directement en mémoire puisqu'ils étaient donnés dans l'énoncé. Puis, nous avons mis la valeur -1 aux cases mémoires qui vont contenir les autres termes de la suite. À la case 7, nous avons mis le nombre d'itérations voulues au programme, correspondant au nombre de termes de la suite auquel on soustrait deux. Puis, nous pouvons voir en mémoire les codes hexadécimaux correspondant à chacune des instructions qui ont été réalisées dans le programme. Il est possible de voir ce résultat à la **Figure 6**. Nous pouvons voir que le résultat obtenu est exact puisque les cases mémoires 1 à 6 contiennent les 6 premiers termes de la suite dans la simulation avec *GTKwave* (voir **Figure 7**).

```
-- Programme utilise pour Q3
constant program_1 : memtype := (
   -- ajouter votre programme ici, une ligne par instruction
   +1,                    -- 0
   +1,                    -- 1 : élément S(0) de la suite
   +2,                    -- 2 : élément S(1) de la suite
   int16_to_uint16(-1),   -- 3 : élément S(2) de la suite
   int16_to_uint16(-1),   -- 4 : élément S(3) de la suite
   int16_to_uint16(-1),   -- 5 : élément S(4) de la suite
   int16_to_uint16(-1),   -- 6 : élément S(5) de la suite
   +4,                    -- 7 : nombre d'itérations de la suite
   to_uint16((lda, 0)),   -- 8
   to_uint16((ld, 0)),    -- 9
   to_uint16((addx, 0)),  -- 10
   to_uint16((adda, 0)),  -- 11
   to_uint16((addx, 0)),  -- 12
   to_uint16((adda, 0)),  -- 13
   to_uint16((sti, 0)),   -- 14
   to_uint16((suba, 0)),  -- 15
   to_uint16((ld, 7)),    -- 16
   to_uint16((sub, 0)),   -- 17
   to_uint16((st, 7)),    -- 18
   to_uint16((brnz, 9)),  -- 19
   to_uint16((stop, 0)),  -- 20
   others => 0   --  FIN
);
```

Figure 6. Capture d'écran du *program_1* de l'exercice 3 de la question 2 permettant le calcul des 6 premiers termes de la suite suivante : *S(n+2) = 1 + S(n+1) + S(n) où S(0) = 1 et S(1) = 2.* Veuillez vous référer au fichier : *2088742_2076524_tp1exo3.vhd*

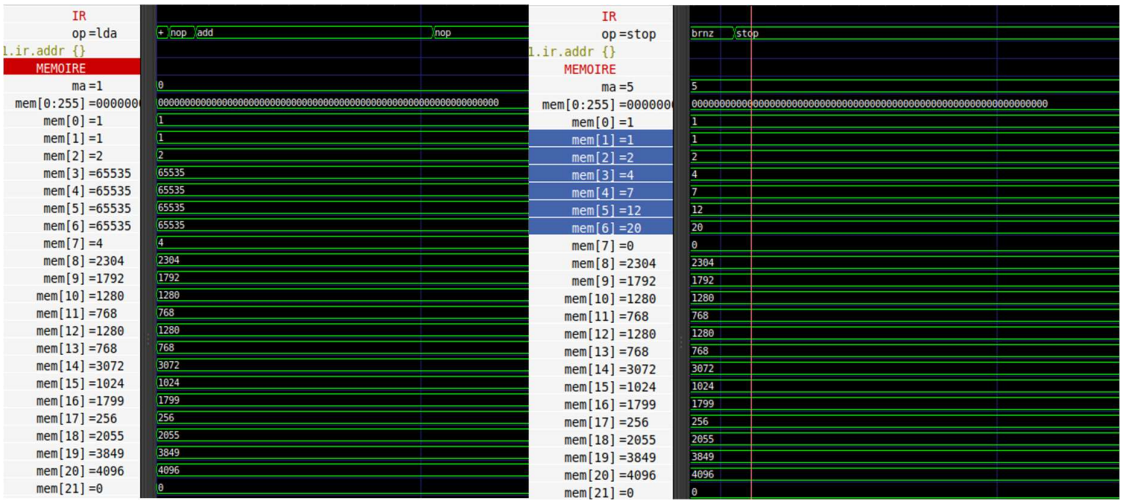

Figure 7. Captures d'écran présentant la simulation avec le logiciel *GTKwave* du *program_1* au début (gauche) et à la fin (droite). Il est possible de constater la présence en mémoire des six premiers termes de la suite qui sont surlignés en bleu.

Notre programme compte un total de 20 lignes, ce qui est un nombre similaire à celui de la question 2. Par contre, notre programme de la question 3 est beaucoup plus robuste, puisqu'il utilise des boucles et automatise le processus. Ainsi, nous pourrions facilement calculer les 100 premiers termes avec le code de la question 3. Il suffirait d'ajouter des cases équivalentes à -1 au départ pour avoir plus de mémoire pour inscrire la séquence et changer le nombre d'itérations voulu (soit : *le nombre d'itérations - 2*), en n'oubliant pas de changer conséquemment les lignes 16 et 18 (utilisant, elles aussi, le nombre d'itérations comme paramètre). En effet, avec le programme de la question 2, nous avons répété le processus 6 fois pour calculer les 6 premiers termes de la suite en copiant-collant le code et en changeant pour chacun les cases mémoires associées. Cela requiert beaucoup plus de travail puisque nous devions manuellement choisir les cases à additionner et à stocker. De plus, étant donné que nous stockions les valeurs de la suite après les opérations, nous devions nous assurer de compter le nombre d'instructions. Ainsi, l'utilisation des boucles dans la question 3 facilite grandement la modification du programme pour augmenter le nombre de termes à calculer sans avoir à penser aux cases de mémoire.