Sorbonne Université – Licence Informatique

LU3IN029 : Architecture des ordinateurs

Année 2024-2025

Sujets des TD et TME
Semaines 1 à 8

Responsable d'UE : Emmanuelle Encrenaz
Intervenants : Noé Amiot, Pirouz Bazargan-Sabet, Adrien Cassagne, Emmanuelle Encrenaz, Daniela Genius, Xunyue Hu, Paul Kling, Quentin Meunier, River Moreira-Ferreira, Franck Wajsbürt

TD1 : Représentation binaire, algèbre de Boole et entiers naturels

Objectif(s)

- ✯ Premières expressions logiques ; présentation du multiplexeur
- ✯ Premiers circuits
- ✯ Représentation des entiers naturels

Exercice(s)

Exercice 1 – Tables de vérité

Dans tout le TD, nous notons . le ET logique et + le OU logique. La notation surlignée correspond à la négation logique : par exemple $\overline{a}$ correspond à NON(a). Ces notations sont utilisées dans le cours.

Question 1

Donnez la table de vérité correpondant à l'équation booléenne à trois entrées suivante :
$s = b + \overline{b}.a.c$

Question 2

L'expression ci-dessus vous semble-t-elle simplifiable ? Justifiez votre réponse.

Question 3

Soit la table de vérité suivante (sortie s), rappelez comment on construit la forme normale disjonctive d'une fonction à partir de sa table, avant de la donner pour la sortie s :

a	b	c	s
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Quelle est l'intuition derrière cette forme normale ?

Question 4

Montrez à l'aide d'une table de vérité qu'en logique booléenne, on a :
a + (b.c) = (a+b).(a+c)
Développer le membre droit de l'égalité ci-dessus puis simplifier, en justifiant, pour aboutir au membre gauche de l'égalité.

Question 5

Donner la table de vérité de la fonction XOR .

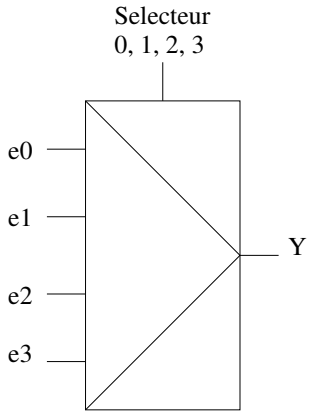

FIGURE 1 – Schéma d'un multiplexeur à 4 entrées et un sélecteur

Question 6
À partir de la table de vérité de la fonction `XOR` , donnez la forme normale disjonctive cette fonction.

Question 7
En utilisant les règles de De Morgan, déterminez une expression de la fonction $\overline{(a \texttt{ XOR } b)}$.
Que permet de tester cette fonction ?

Exercice 2 – Multiplexeurs

Un multiplexeur à n entrées est un circuit à n entrées plus une entrée (appelée 'sélecteur') permettant de désigner la sortie voulue : un multiplexeur permet de mettre en sortie une seule des n entrées, celle désignée par le 'sélecteur'.

Autrement dit, un multiplexeur à n entrées ($n > 1$) est un circuit ayant en entrée un ensemble de bits $e_0, e_1, \cdots, e_{n-1}$ et un sélecteur utilisé pour coder un numéro d'entrée compris entre 0 et $n-1$. Le rôle de ce circuit est de produire en sortie la **valeur** de l'entrée dont le numéro est donné par le sélecteur.

Par exemple, dans la figure 1, si la valeur du sélecteur est :
— 0, la sortie Y du multiplexeur est égale à la valeur de l'entrée e_0 (y = 0 si e_0 = 0, y = 1 si e_0 = 1).
— 1, la sortie Y du multiplexeur est égale à la valeur de l'entrée e_1 (y = 0 si e_1 = 0, y = 1 si e_1 = 1).
— 2, la sortie Y du multiplexeur est égale à la valeur de l'entrée e_2 (y = 0 si e_2 = 0, y = 1 si e_2 = 1).
— 3, la sortie Y du multiplexeur est égale à la valeur de l'entrée e_3 (y = 0 si e_3 = 0, y = 1 si e_3 = 1).

Question 1 – Multiplexeur à 2 entrées

Un multiplexeur à deux entrées est un circuit ayant en entrée deux bits e_0 et e_1 et un sélecteur v sur 1 bit permettant de choisir la première entrée lorsque v vaut 0 ou la deuxième lorsque v vaut 1. Ce circuit a été présenté en cours.

Donnez la table de vérité d'un multiplexeur à 2 entrées en exprimant simplement la valeur de la sortie y ($y = e_0$ ou $y = e_1$) en fonction de la valeur du sélecteur v.

Donnez la forme normale disjonctive de ce multiplexeur.

Question 2 – Multiplexeur à 4 entrées

On souhaite maintenant donner l'expression booléenne correspondant à un multiplexeur à 4 entrées. Puisque l'on est en logique booléenne, la valeur du sélecteur, comprise entre 0 et 3, est codée en binaire à l'aide deux bits notés i_0 et i_1, tel que le $i_1 i_0$ soit le codage "entiers naturels" du numéro de l'entrée à sélectionner : i_0 est le bit de poids faible du sélecteur et la **valeur v** du sélecteur est donnée par **v**$= \sum_{k=0}^{n-1} i_k.2^k$.

Remplissez la table ci-dessous en indiquant dans la 3ème colonne la valeur **v** du sélecteur encodée par les deux bits d'entréee puis indiquer dans la 4ème colonne la valeur de sortie de y.

i_1	i_0	**v**	s
0	0		
0	1		
1	0		
1	1		

En déduire une expression algébrique [1] de y.

Exercice 3 – Représentation des entiers naturels

Question 1

Remplissez le tableau suivant qui pourra vous servir pour la suite de ce TD pour les conversions.

base 10	base 2	base 16
0		
1		
2		
3		
4		
5		
6		
7		

base 10	base 2	base 16
8		
9		
10		
11		
12		
13		
14		
15		

Question 2

Combien de valeurs différentes peut-on coder dans un mot binaire de longueur n ?

Quel est l'intervalle des valeurs représentables dans un mot de n bits avec le codage "entiers naturels" ?

Combien faut-il de bits au minimum dans un mot pour qu'il puisse encoder les valeurs suivantes (selon le codage "entiers naturels") : 127_d et 32_d ?

Question 3

Combien de valeurs différentes peut-on coder dans un mot hexadécimal de longueur n ?

Quel est l'intervalle des valeurs représentables dans un mot de n chiffres hexadécimaux avec le codage "entiers naturels" ?

Question 4

Complétez le tableau suivant :

1. dans l'algèbre de Boole

Mot binaire de 8 bits qui encode la valeur d	Nombre d en décimal	Mot hexadécimal de 8 bits (2 chiffres) qui encode la valeur d
	10	
0b0000 0010		
		0x10
	57	
	127	
		0x17
		0x5B
0b0010 1001		
0b1010 1010		

Exercice 4 – Exercices de conversion

Conversion binaire $\rightarrow$ hexadécimal
- $1\ 1101_b =$
- $1001\ 1000\ 0011\ 1100_b =$

Conversion hexadécimal $\rightarrow$ binaire
- $75_h =$
- $1A_h =$
- $34DF_h =$

Conversion binaire $\rightarrow$ décimal
- $1001\ 0110_b =$
- $1100\ 0110_b =$

Conversion décimal $\rightarrow$ binaire. Pour ces conversions, vous utiliserez et l'approche par divisions successives et celle par tableau de puissances.
- $57_d =$
- $1272_d =$

TME1 : Réalisation d'un multiplexeur et d'un comparateur

Objectif(s)

- ✯ Prise en main de Logisim
- ✯ Réalisation de multiplexeurs
- ✯ Écriture d'un comparateur de deux nombres naturels

Exercice(s)

Exercice 1 – Prise en main de Logisim

Lancez Logisim en tapant dans un terminal la commande "`logisim &`".
Une documentation (en anglais) est disponible en passant par le menu Help de Logisim. L'interface avec quelques éléments est présentée sur la figure 1.

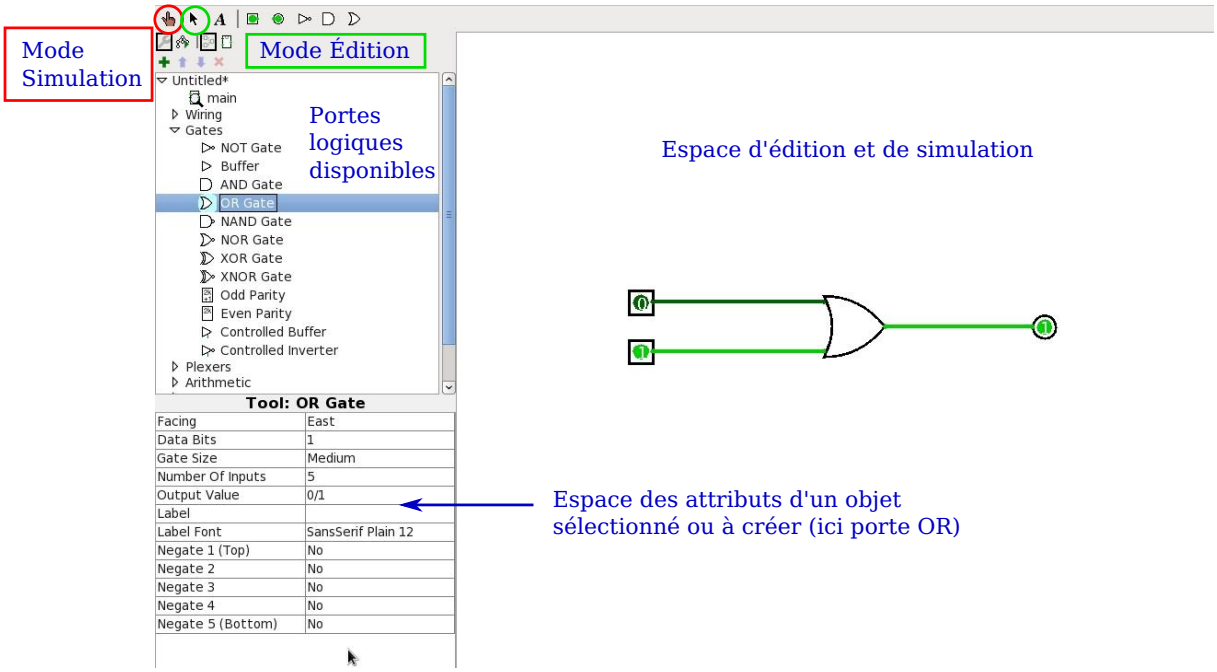

FIGURE 1 – Interface de logisim avec un exemple de circuit simple composé d'une porte OR

Note : pour tous les exercices, il vous est demandé d'utiliser uniquement des entrées, des fils et des sorties ayant une taille de 1 bit (`Data Bits=1`). Une entrée sur n bits sera réalisée à l'aide de n entrées sur 1 bit.

Question 1

Construisez un circuit qui réalise la fonction f = (a ET NON b). Pour information, la couleur des fils dans logisim vous donne des indications :

— vert foncé : valeur 0 transitant sur le fil de largeur 1

— vert clair : valeur 1 transitant sur le fil de largeur 1
— noir : indique une nappe de fils (plusieurs bits)
— bleu : aucune valeur (pas de connexion)
— rouge : erreur
— orange : problème de compatibilité entre la largeur d'une nappe de fils et le port d'un circuit ou d'une porte

Sélectionnez une porte AND et ajustez le nombre des entrées pour qu'il n'y en ait que 2. La figure 2 vous montre où changer le nombre d'entrées.

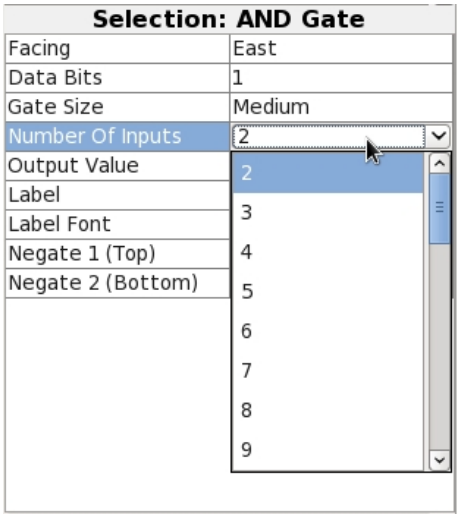

FIGURE 2 – Sélection du nombre d'entrées d'un AND

Remarque : pour chaque entrée ou sortie ajoutée, veillez à ce que le champ `Three-state?` *soit à la valeur* `No`*, et vérifiez son orientation (champ* `Facing`*).*

Attention : pour ajouter une sortie à votre circuit, il ne faut pas utiliser l'objet `probe` *du menu* `Wiring` *mais l'objet* `Pin` *et changer la valeur du champ* `Output?` *à Yes. Vous pouvez aussi utiliser l'icône circulaire verte de la barre en haut.*

Étiquetez les deux entrées de votre circuit en indiquant leur nom dans le champ `Label` des attributs des entrées. La figure 3 vous montre où est ce champ.

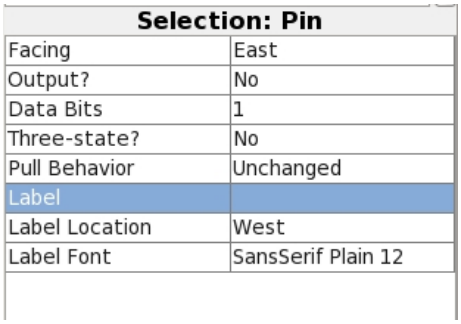

FIGURE 3 – Labelisation d'un AND

Simulez le circuit et comparez le résultat obtenu avec la table de vérité de la fonction f. Pour cela, mettez vous en mode simulation et faites varier les valeurs en entrée (en cliquant sur les entrées) pour couvrir l'ensemble des configurations possibles.

Étiquetez les deux entrées. Simulez le circuit et comparez le résultat obtenu avec la table de vérité de la fonction f.

Question 2

À l'aide de l'expression trouvée en TD, complétez votre circuit pour réaliser la fonction $f = a \oplus b$, sans utiliser la

porte XOR de logisim. Nommez `mon_xor` le circuit construit et sauvegardez le projet contenant le circuit sous le nom TME1.

Exercice 2 – Construction d'un multiplexeur à 2 entrées

Comme vu en TD, un multiplexeur est un circuit à plusieurs entrées (notées e_0, e_1, ...) avec un sélecteur dont la taille est fonction du nombre des entrées. Le rôle du sélecteur est de coder un numéro d'entrée. Le multiplexeur doit alors produire en sortie la valeur de cette entrée. Dans le cas d'un circuit à 2 entrées numérotées 0 et 1, on prend donc un sélecteur i sur 1 bit. La sortie y du circuit peut alors s'écrire :

— y = e_0 ssi i = 0
— y = e_1 ssi i = 1

Question 1

Rappelez l'expression booléenne du multiplexeur tel que décrit ci-dessus et vu en TD puis construisez un circuit qui réalise la fonction y. Pour cela, ajoutez un circuit à votre projet (Project $>$ Add Circuit...) que vous nommerez `mux2`. Vérifiez votre circuit par simulation.

Exercice 3 – Construction d'un multiplexeur à 2 entrées sur 4 bits

Question 1

On souhaite maintenant construire un multiplexeur à 2 entrées A=$a_3a_2a_1a_0$ et B=$b_3b_2b_1b_0$ sur 4 bits **UNIQUEMENT** à partir de multiplexeurs 2 entrées. La sortie S=$s_3s_2s_1s_0$ est aussi sur 4 bits (et prend soit la valeur de A, soit la valeur de B), et il y a un bit de sélection i.

Vous utiliserez comme multiplexeur à 2 entrées sur 1 bit le circuit que vous avez réalisé à la question précédente. Pour cela, cliquez sur la petite boite à côté de `mux2` et ajoutez le circuit dans la partie édition. La figure 4 vous montre comment faire. Le circuit `mux2` a comme interface : 3 entrées repérables par les points bleus sur le contour et une sortie repérable par le point rouge. Placer le curseur de la souris au-dessus d'une entrée ou sortie permet de voir son nom.

Vous pouvez aussi voir sur cette figure que le projet TME1 contient différents circuits à réaliser pour ce TP. Vous devrez créer un projet (extension de fichier .circ) pour chaque séance de TP utilisant Logisim, qui contiendra l'ensemble des réponses aux questions du TME. Pour éditer un autre circuit dans le projet, double-cliquez dessus.

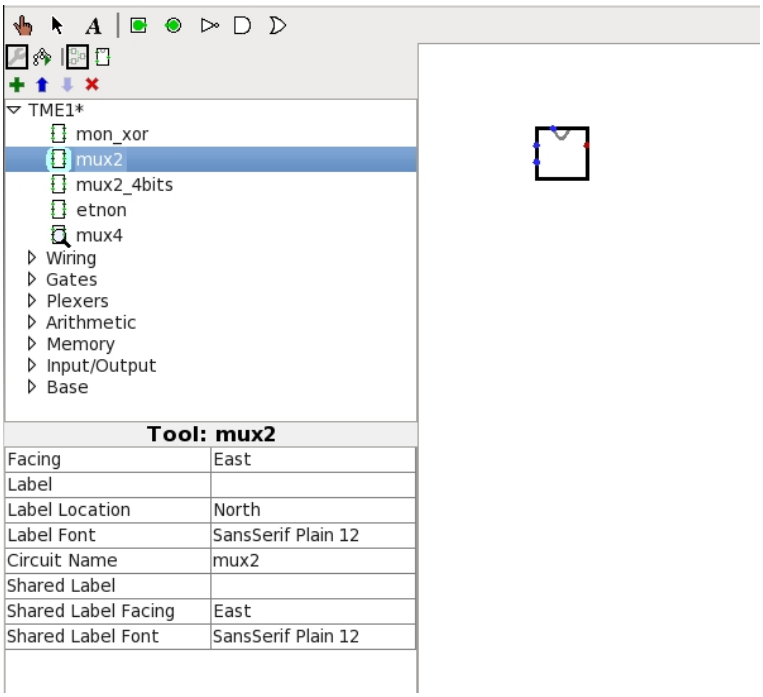

FIGURE 4 – Utilisation du circuit `mux2` depuis l'édition du circuit `mux4`

Exercice 4 – Construction d'un multiplexeur à 4 entrées

On considère maintenant un multiplexeur à 4 entrées, chacune sur 1 bit, et une sortie comme vu en TD. On rappelle que le sélecteur encode sur 2 bits le numéro de l'entrée sélectionnée (valeur comprise entre 0 et 3).

Question 1

Construisez un multiplexeur à 4 entrées 1 bits **UNIQUEMENT** à partir de multiplexeurs 2 entrées : pour cela manipuler (c-a-d ré-écrire différemment) l'expression booléenne du multiplexeur 4 entrées e_0, e_1, e_2, e_3 pour y voir apparaître (plusieurs fois) celle d'un multiplexeur 2 entrées.

Question 2

Comment généraliser la construction de multiplexeurs à 2^n entrées à partir de multiplexeurs à 2 entrées ? Expliquez la solution pour un multiplexeur à 8 entrées puis à 16 entrées et généraliser.

Exercice 5 – Comparateur d'entiers naturels sur 3 bits

On souhaite réaliser un circuit qui prend en entrée deux entiers naturels A et B codés sur 3 bits ($a_2a_1a_0$ et $b_2b_1b_0$) et qui indique `1` en sortie si B est strictement plus grand que A, `0` sinon.

Question 1

En utilisant des opérateurs de comparaison ($>, =$) bit à bit, donnez une expression booléenne qui vaut `1` si et seulement si B est plus grand que A.

Question 2

Quel opérateur logique permet de coder l'expression $b_i = a_i$?
Trouvez une expression logique qui code la comparaison $b_i > a_i$.

Question 3

Construisez un circuit qui réalise la comparaison de A et B.

TD2 : Entiers naturels et relatifs, addition et soustraction

Objectif(s)

- ✯ Représentation et additions d'entiers naturels et relatifs
- ✯ Réalisation d'un opérateur d'addition

Exercice(s)

Exercice 1 – Opérations arithmétiques avec des naturels

Question 1

Soit l'addition suivante de deux entiers naturels, quelle doit être la taille minimale du mot stockant le résultat ?

$$\begin{array}{rr} & 5B_h \\ + & 17_h \\ \hline = & \end{array}$$

Question 2

Même question pour la somme des entiers naturels 71_h et $B5_h$.

Question 3

Faites les opérations sur les mots suivants en indiquant s'il y a ou non dépassement de capacité sur entiers naturels. Les deux premières opérations sont sur 8 bits et la dernière sur 16 bits.
0x2B + 0x95 =
0xC8 + 0x6D =
0x8B57 + 0xA34F =

Question 4

Enoncez une règle permettant de détecter un dépassement de capacité lors de l'addition d'entiers naturels sur n bits.

Exercice 2 – Construction d'un additionneur n bits

Question 1

Combien faut-il d'entrées et de sorties pour réaliser un étage d'une addition ?

Question 2

Donnez la table de vérité d'un additionneur 1 bit.

Question 3

Donnez une expression booléenne pour cout et s.

Question 4
Comment réaliser un additionneur n bits à partir de n additionneurs 1 bit ?

Exercice 3 – Représentation des entiers relatifs

La représentation binaire des entiers relatifs est utilisée sur des écritures de nombres de **longueur donnée** (nombres écrits couramment sur 8, 16, 32 ou 64 bits). Dans une telle écriture on utilise le bit de poids fort (bit le plus à gauche) du nombre pour stocker la valeur du coefficient négatif (-2^{n-1} sur n bits).

Question 1
Les première et troisième colonnes du tableau ci-dessous contiennent tous les mots binaires de 3 bits. Donnez pour chaque mot la valeur décimale de son interprétation selon le codage *entiers relatifs en complément à 2*.

Mot binaire de 3 bits	Valeur en décimal selon codage entiers relatifs	Mot binaire de 3 bits	Valeur en décimal selon codage entiers relatifs
0b000		0b100	
0b001		0b101	
0b010		0b110	
0b011		0b111	

Quel est l'intervalle des valeurs représentables sur un mot de n bits par le codage *entiers relatifs* ?
Quand peut-on avoir un dépassement de capacité lors d'une addition de deux nombres relatifs et quand est-on sur qu'il n'y aura pas de dépassement de capacité ?

Question 2
Chacun des mots de 16 bits suivants contient un entier relatif codé en complément à 2. Indiquez si celui-ci est positif ou négatif, puis calculez le mot contenant l'opposé, et écrivez-le en base 2 et en base 16.

Mot de 16 bits Écriture base 16	signe	Mot de 16 bits Écriture base 2	Opposé = CÀ2 Écriture base 2	Opposé = CÀ2 Écriture base 16
0x0B24				
0xABCD				
0xFFFF				
0x5A72				
0x0072				

Exercice 4 – Opérations arithmétiques

Soit un additionneur 8 bits, muni en plus de la sortie résultat sur 8 bits, de deux sorties `Cout` et `Ov`, telles que `Cout` = $\texttt{Cout}_{n-1} = \texttt{Cout}_7$ et `Ov` = $\texttt{Cout}_{n-1} \oplus \texttt{Cout}_{n-2} = \texttt{Cout}_7 \oplus \texttt{Cout}_6$.

Question 1
Soit l'addition binaire suivante : $0b01101001 + 0b10000000 = 0b11101001$. Quelle opération réalise-t-on en décimal dans le cas où les opérandes et le résultat sont interprétés comme des entiers naturels ? Le résultat théorique de l'opération est-il codable sur 8 bits ? Quelles sont les valeurs des bits `Cout` et `Ov` ? Même question dans le cas où les opérandes et le résultat sont interprétés comme des entiers relatifs en complément à 2.

Question 2
Soit l'opération $0xE2 + 0x9C$. Calculer le résultat. Quelle opération effectue-t-on si les opérandes sont interprétés des entiers naturels ? Le résultat est-il valide ? Mêmes questions si les opérandes sont vus comme des relatifs. Quelles sont les valeurs des bits `Cout` et `Ov` ?

Question 3
Soit l'opération sur entiers naturels $136 + 250$. Quels sont les valeurs des mots en entrée et en sortie de l'additionneur (en binaire ou en hexa) ? Le résultat est-il valide, et pourquoi ? Si maintenant les mêmes valeurs en entrée de l'additionneur sont interprétées comme des entiers relatifs, que vaut le résultat ? Est-il valide ?

Question 4
Trouver des valeurs d'opérandes en entrée de l'additionneur qui produisent un résultat valide s'ils sont interprétés comme entiers naturels et invalide s'ils sont interprétés comme entiers relatifs.

Exercice 5 – Extension d'entiers naturels et relatifs

Question 1
Soit le mot hexadécimal 0xFF sur 8 bits. Donnez sa valeur en décimal dans le cas d'un codage entier naturel et dans le cas d'un codage entier relatif (en complément à 2). Étendez ce nombre sur 32 bits. Obtenez-vous la même chose si l'on considère que c'est un entier naturel ou un entier relatif ? En déduire les règles d'extension d'un entier naturel et d'un entier relatif.

TME2 : Réalisation d'opérateurs sur entiers naturels et relatifs

Objectif(s)

- ✯ Réalisation d'un additionneur 4 bits à partir de l'additionneur 1 bit vu en TD
- ✯ Réalisation d'une unité arithmétique et logique permettant l'addition et la soustraction de deux entiers de 4 bits
- ✯ Réalisation d'un opérateur de décalage

Exercice(s)

Exercice 1 – Additionneur 4 bits

Question 1

Sous Logisim, construisez un additionneur 1 bit, ayant les entrées A, B et Cin (sur un bit), et les sorties S et Cout. Vérifiez son fonctionnement par simulation (en testant toutes les combinaisons possibles pour les valeurs en entrée).

Question 2

En répliquant le schéma de l'additionneur 1 bit créé précédemment, construisez un additionneur 4 bits. L'additionneur devra disposer de l'interface suivante : deux entrées A (A3 A2 A1 A0) et B (B3 B2 B1 B0) sur 4 bits, une sortie S (S3 S2 S1 S0) sur 4 bits et une sortie Cout sur un bit. Pour cette question, il vous est demandé de ne pas utiliser d'instances de l'additionneur 1 bit, mais uniquement des portes logiques de base.

Question 3

Testez votre additionneur 4 bits par simulation. Combien d'opérations votre additionneur est-il capable de réaliser ? Proposer un ensemble de tests qui permettent de tester son bon fonctionnement sans pour autant couvrir toutes les configurations des valeurs d'entrées possibles.

Exercice 2 – Additionneur-soustracteur 4 bits

Question 1

Pour cet exercice, vous devez repartir du schéma de l'additionneur 4 bits de l'exercice précédent, que vous copierez dans un nouveau circuit. Il ne faut pas instancier un (ou plusieurs) additionneur(s) 4 bits dans l'additionneur/soustracteur, mais editer directement le circuit de l'additionneur.

Vous devez construire une ALU (unité arithmétique et logique), nommée ALU v1, pouvant effectuer soit une addition, soit une soustraction d'opérandes entiers sur 4 bits. Lors d'une addition, les opérandes pourront être considérés comme des entiers naturels ou relatifs. Lors d'une soustraction, les opérandes seront considérés comme des entiers relatifs.

L'ALU v1 à réaliser aura deux entrées X (sur 4 bits) et Y (sur 4 bits) qui seront les opérandes, une entrée Aluop (sur 1 bit) qui permet de selectionner le type d'opération à effectuer ($0 \rightarrow$ addition, $1 \rightarrow$ soustraction), une sortie S sur 4 bits et une sortie Cout correspondant à la retenue sortante.

Conseils :

— Pour effectuer plus efficacement vos simulations, votre instance finale devra avoir l'interface de la figure 1.
— Vous pouvez utiliser des portes logiques de base (comme pour l'exercice précédent) ou des instances de l'additionneur 1 bit ; en revanche vous veillerez dans tous les cas à placer les éléments de manière à avoir un schéma lisible : regroupement des entrées et des sorties du circuit, minimisation des croisements de fils, etc.

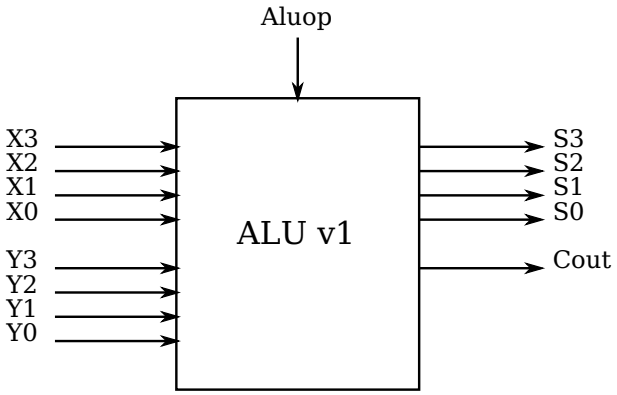

FIGURE 1 – Interface de l'ALU v1

— Vous pourrez aussi avoir besoin des générateurs des constantes logiques 0 et 1 (figure 2) disponibles dans Logisim.

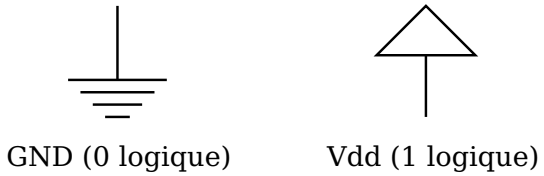

FIGURE 2 – Schéma du zéro logique (GND) et du 1 logique (Vdd)

Question 2

Vérifiez que votre ALU v1 fonctionne correctement en effectuant par simulation un ensemble d'opérations couvrant les différents cas possibles.

Question 3

Modifiez votre ALU v1 en ALU v2 pour que celle-ci génère en sortie les signaux suivants (nommez-les) :
— un signal OV qui vaut 1 lorsque l'opération génère un dépassement de capacité sur entiers relatifs
— un signal ZF qui vaut 1 lorsque le résultat de l'opération est nul
— un signal SF qui vaut 1 lorsque le résultat est négatif
— un signal CF qui correspond à la retenue sortante du dernier rang

L'interface de l'ALU v2 est alors celle représentée sur la figure 3.

Question 4

Vérifiez le bon fonctionnement de votre nouvelle ALU par simulation, en proposant un ensemble d'opérations bien choisies permettant d'illustrer les différents cas d'opération.
Combien de cas d'addition sur entiers relatifs est il nécessaire de tester pour couvrir tous les cas (signe des opérandes + dépassement ou non de capacité) ?
Comparez les résultats obtenus avec ceux obtenus par un calcul à la main.

Question 5

Pour réaliser des additions sur entiers naturels, on peut utiliser l'additionneur construit : comment peut on détecter un dépassement de capacité dans ce cas ?
Illustrez votre réponse avec 2 exemples d'addition, un avec depassement et un sans dépassement.

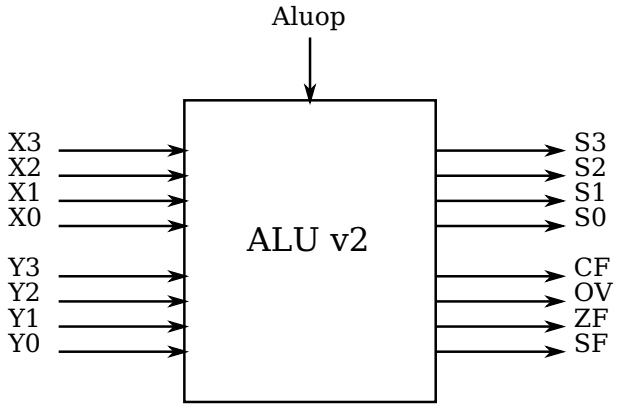

FIGURE 3 – Interface de l'ALU v2

Pour réaliser des soustractions sur entiers naturels, on peut aussi utiliser le soustracteur construit. Cherchez quel drapeau (et quelle valeur pour ce drapeau) permet de détecter un dépassement de capacité lors d'une soustraction sur entiers naturels.
Indication : la soustraction n'est pas définie si l'opérande que l'on soustrait est le plus grand des 2 (le résultat n'est pas dans l'intervalle des valeurs représentable, il est négatif !).

Exercice 3 – Décalage à gauche

On souhaite réaliser un opérateur qui effectue un décalage à gauche de 0, 1, 2 ou 3 bits d'un nombre codé sur 4 bits. Cet opérateur aura en entrée un entier codé sur 4 bits $a_3a_2a_1a_0$ et un entier codé sur 2 bits d_1d_0 indiquant le nombre de bits à décaler. On notera $s_3s_2s_1s_0$ la sortie du décaleur.

Question 1

On note $<<$ l'opération de décalage à gauche (comme en C). Quel est le résultat des opérations suivantes en binaire et en décimal ? Quel est l'effet d'un décalage à gauche de i bits ?

— $0011 << 0 =$
— $0011 << 1 =$
— $0011 << 2 =$
— $0011 << 3 =$

Question 2

Donnez l'expression booléenne de chacun des bits de sortie (les s_i) en fonction des entrées. Vous pourrez vous aider d'une table de vérité si besoin.

Question 3

Réalisez le circuit sous Logisim et vérifiez son fonctionnement par simulation en testant, pour plusieurs valeurs de l'entier sur 4 bits, les différentes valeurs possibles de d_1d_0.
Vérifiez par simulation votre réponse à la question 1 de cet exercice.

TD3 : Premiers programmes assembleur

Objectif(s)

- ✯ Familiarisation avec la structure d'un programme assembleur MIPS
- ✯ Familiarisation avec les instructions arithmétiques et logiques MIPS et les appels système
- ✯ Utilisation du mémento MIPS
- ✯ Codage et décodage d'instructions
- ✯ À la fin de cette semaine, sont considérées comme acquises les notions ci-dessus et la maîtrise de votre mémento MIPS

Exercice(s)

Exercice 1 – Structure d'un programme assembleur

Question 1

1. Rappelez les sections qui composent un programme assembleur et expliquez ce que l'on trouve dans ces sections.
2. Quel est le point d'entrée d'un programme ?
3. Que doit contenir au minimum un programme ?

Exercice 2 – Jeu d'instructions MIPS : instructions arithmétiques et logiques

Question 1

Parcourez sur votre mémento toutes les opérations arithmétiques (opérations sur les entiers) du jeu d'instructions MIPS. Notez, pour chacune, son code opération ainsi que le nombre de ses opérandes sources et résultats, et la nature de ces opérandes ainsi que le format de codage. En déduire une réponse aux questions suivantes :

— Combien d'opérandes ont ces instructions ?
— Quelles sont les exceptions ?
— Déduisez-en une relation entre le type des opérandes sources et le format de codage.

Question 2

Regardez le format de codage I. Quelle est la taille du champ où est encodé l'immédiat ? Qu'en déduisez vous sur les valeurs possibles ?
Comment cet immédiat est il étendu à l'exécution ?

Question 3

À l'aide de votre mémento, donnez le codage binaire des instructions suivantes :

```
addiu $12, $18, 15
```

```
addu $12, $18, $4
```

Question 4
Parcourez sur votre mémento toutes les opérations logiques (opérations sur des vecteurs de bits) du jeu d'instructions MIPS. Notez, pour chacune, son code opération, le nombre de ses opérandes sources et résultats, et la nature de ces opérandes ainsi que le format de codage. La règle trouvée précédemment est elle correcte ? Quelle est l'exception à la règle ? Pourquoi ?

Question 5

1. Si le registre \$8 contient la valeur 0x0000000F, quelle est la valeur contenue dans le registre \$9 après l'opération `sll $9,$8,8` ?
2. Si le registre \$8 contient la valeur 0xF0000000, quelle est la valeur contenue dans le registre \$9 après l'opération `srl $9,$8,28` ?
3. Si le registre \$8 contient la valeur 0xF0000000, quelle est la valeur contenue dans le registre \$9 après l'opération `sra $9,$8,28` ?
4. Si le registre \$8 contient la valeur 0x00000036, quelle est la valeur contenue dans le registre \$9 après l'opération `andi $9,$8,0x000F` ?

Question 6
À l'aide de votre mémento, donnez le codage binaire des instructions suivantes :
```
sll $12, $18, 5
mult $8, $9
```

Question 7

1. Quelle est la particularité du registre \$0 ?
2. Donnez plusieurs instructions qui permettent de mettre la valeur 0 dans le registre \$8.
3. Donnez plusieurs instructions qui permettent de copier le contenu du registre \$10 dans le registre \$8.

Exercice 3 – Chargements d'une valeur dans un registre

Question 1

1. Donnez une instruction permettant de charger (mettre) la valeur `0x1234` dans le registre \$8.
2. Rappelez le nombre de bits du champ immédiat des instructions avec un opérande immédiat ? Comment mettre la valeur `0x12345678` dans le registre \$8 ? Donnez un exemple.
3. Les séquences suivantes sont elles équivalentes ? Que contient le registre \$8 après ces deux séquences d'instructions ?

```
xor $8, $8, $8
addiu $8, $8, 0x8765
```

```
xor $8, $8, $8
ori $8, $8, 0x8765
```

4. Comment charger la valeur -1 dans un registre ?

Question 2
Écrivez un programme complet (toutes les sections doivent apparaître) qui met respectivement les valeurs 0x34 et 34 dans les registres \$8 et \$9, et produit le résultat de l'addition de ces deux valeurs dans le registre \$10.

Exercice 4 – Appels système de lecture et d'écriture d'entiers

Question 1

Qu'est-ce qu'un appel système ? Comment réaliser un appel système ?

Question 2

1. Complétez le programme assembleur de l'exercice précédent afin d'afficher la valeur résultant de l'addition à la fin du programme (cherchez dans le mémento l'appel système permettant d'afficher un entier).
2. Modifiez votre programme pour que la valeur du premier entier soit lue au clavier (cette valeur doit ensuite être ajoutée à la valeur 34).

TME3 : Premiers programmes en assembleur

Objectif(s)

- ✯ Prise en main de Mars (lancement, interface, actions possibles, debug),
- ✯ Écriture et exécution de programmes assembleur,
- ✯ Familiarisation avec les appels système (`syscall`) de lecture et d'affichage d'une chaine de caractères ou d'un nombre entier.

Exercice(s)

Exercice 1 – Prise en main de MARS, visualisation du codage d'une instruction

Lancez Mars (commande donnée par vos enseignants ou sur le site web).
La figure 1 montre l'interface graphique de MARS.

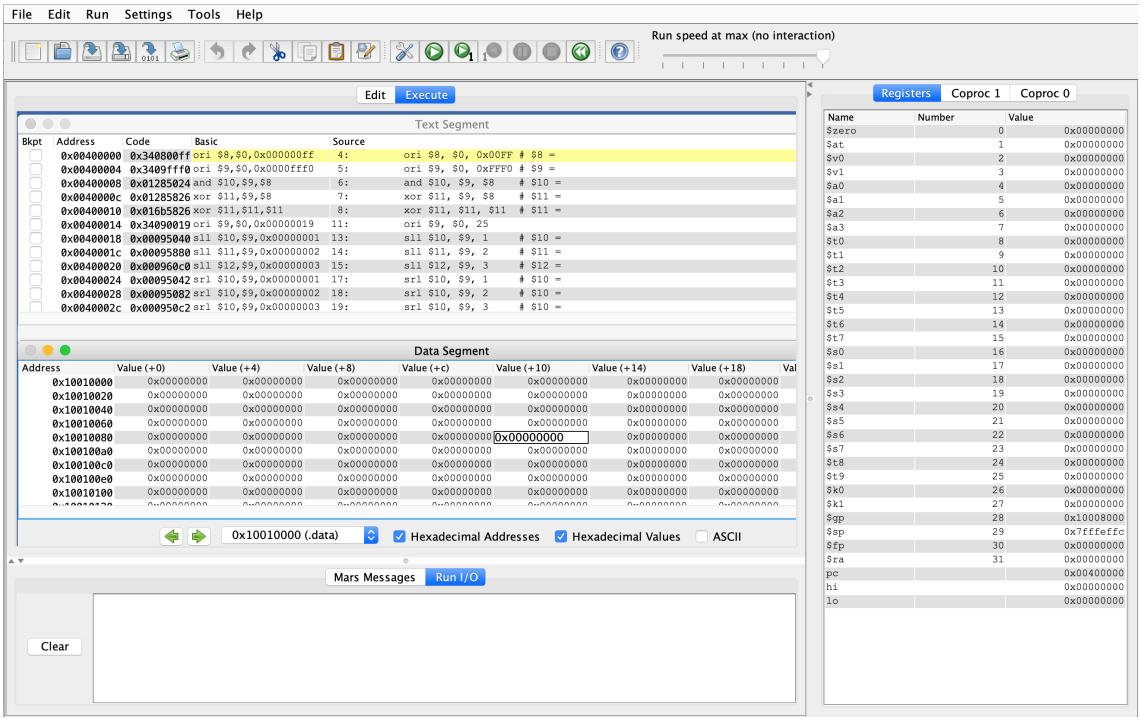

FIGURE 1 – MARS

Sur la droite de l'interface, vous pouvez apercevoir le contenu des registres généraux (`$0,$1,..., hi, lo`), ainsi que des registres spéciaux comme le compteur de programme (pc) etc.
Au centre de l'interface on peut voir deux onglets : **Edit** et **Execute**.

- L'onglet **Edit** permet d'éditer et de modifier un code assembleur. C'est là qu'apparait le code assembleur après chargement d'un fichier.
- L'onglet **Execute** contient 2 parties (cliquez sur l'onglet pour voir) : **Text Segment** et **Data Segment**.
  - **Text Segment** contient le code assemblé : codage des instructions, adresses où elles sont implantées.

— **Data Segment** contient les données (variables globales, pile, ...).

Au bas de l'interface, vous avez les onglets **Mars Messages** et **Run I/O**. Les messages d'erreur du simulateur seront affichés dans **Mars Messages**. Les sorties du simulateur, en particulier les affichages effectués par le programme, se feront dans **Run I/O**.

En haut de l'interface, vous avez les différentes actions possibles dans les menus **File, Edit, ...**. Des raccourcis sont disponibles sous la forme de petites icônes juste en dessous des menus. A droite de ces icônes se trouve un curseur réglable qui se nomme **Run speed** qui permet de régler la vitesse d'exécution des instructions.

IMPORTANT : Dans le menu Settings :

1. décochez l'utilisation des pseudo-instructions (*"Permit extended (pseudo) instructions and formats"*) : vous devrez toujours utiliser uniquement les instructions de base dans votre programme.
2. cochez l'affichage des valeurs et des adresses en hexadécimal (*"Values (resp. Addresses) displayed in hexadecimal"*).

Question 1

Où trouver dans le simulateur les instructions, directives et appels système supportés par celui-ci ?

Question 2

1. Dans Mars, ouvrez un nouveau fichier (menu File, New) et écrivez-y l'instruction `addi $12, $18, 33`.
2. La saisie de cette instruction modifie-t-elle les informations se trouvant dans l'onglet Execute ?
3. Sauvegardez votre fichier sous le nom test.s. Cette action modifie-t-elle les informations se trouvant dans l'onglet Execute ?
4. Assemblez votre programme avec la commande Assemble du menu Run. Que se passe-t-il ? Que lit-on dans **Mars Messages** ?
5. Où pouvez vous trouver le codage hexadécimal de cette instruction ? Quel est-il ? À quelle adresse est implantée cette instruction ?

Question 3

1. Sur votre feuille, donnez le codage en binaire de l'instruction `addi $12, $18, 33`. Inversez (i.e., prenez la négation de) la valeur du bit n°29 (appelé aussi 30ème bit : les bits sont numérotés de droite à gauche, le bit le plus à droite est le bit n°0, ou 1er bit).
2. À l'aide de votre mémento, déterminez l'instruction correspondant à ce codage binaire. Donnez le codage hexadécimal correspondant.
3. Saisissez cette instruction après l'instruction précédente. Vérifiez que votre réponse à la question précédente est correcte en regardant, dans l'onglet `Execute`, la valeur hexadécimale associée à l'instruction que vous venez de saisir.
4. Exécutez cette série d'instructions. Que remarquez-vous ?

Question 4

Modifiez l'instruction `addu $0, $18, $12` afin de générer une erreur de syntaxe. Vous pouvez par exemple mettre 12 au lieu de $12. Que se passe-t-il ?

Exercice 2 – Exécution de programme

Le but de cet exercice est d'exécuter des programmes assembleur avec Mars et d'apprendre comment manipuler Mars pour lancer un programme, l'exécuter pas à pas, l'exécuter sans arrêt, revenir en arrière dans l'exécution.

Question 1

1. Dans l'éditeur de Mars, écrivez un programme assembleur qui met dans le registre \$8 la valeur *décimale* 137, affiche cette valeur avec l'appel système d'affichage d'un entier (affichage en décimal) avant de quitter. Sauvegardez le fichier avec l'extension `.s` (extension classique des fichiers assembleur). Aidez-vous du mémento pour savoir comment utiliser l'appel système permettant d'afficher un entier.
2. Chargez le programme dans Mars et assemblez-le. Si la fenêtre Text Segment ne s'affiche pas, c'est qu'il y a une erreur de syntaxe. Corrigez l'erreur et recommencez l'opération de chargement et de correction de syntaxe jusqu'à ce que votre programme puisse être assemblé.
3. Exécutez le programme pas à pas en observant le contenu des différents registres après l'exécution de chaque instruction.
4. Réinitialisez le simulateur (double flèche vers la gauche).
5. Ré-exécutez le programme mais sans arrêt cette fois.
6. Réinitialisez le simulateur (double flèche vers la gauche).
7. Diminuez la vitesse d'exécution et ré-exécutez le programme. Au cours du programme faites pause après le premier syscall. Revenez en arrière d'une instruction et modifiez la valeur du registre \$8 (dans la zone d'édition des registres, double-cliquez sur le registre \$8 et entrez la valeur 1). Relancez le programme. Que se passe-t-il ?

Question 2

Reprenez le programme de la question précédente en mettant cette fois dans \$8 la valeur 65537 = $2^{16} + 1$. Rappelez, avant de modifier votre programme, le nombre de bits affectés au stockage d'une valeur immédiate dans le format I (instructions avec un opérande de type immédiat) du jeu d'instructions MIPS. Exécutez votre programme et vérifiez le contenu de \$8.

Exercice 3 – Instructions de multiplication et de division

Les instructions de multiplication et division sont un peu particulières. Elles ont deux opérandes sources mais pas d'opérande explicite de destination. La raison est que la multiplication de deux valeurs sur 32 bits donne un résultat sur 64 bits (il faut donc 2 registres) et la division a deux résultats : le quotient et le reste. C'est pourquoi ces instructions n'ont pas d'opérande de destination explicite, elles produisent leur résultat dans deux registres spéciaux, les registres `lo` et `hi`.

L'instruction `div ri, rj` effectue la division de `ri` par `rj`. Elle met le quotient dans le registre `lo` et le reste dans le registre `hi`.

L'instruction `mult ri, rj` effectue le produit de `ri` par `rj`. Elle met les 32 bits de poids fort du résultat (sur 64 bits) dans le registre `hi` (high) et les 32 bits de poids faible dans le registre `lo` (low).

Il est possible de copier le contenu du registre spécial `lo` dans un registre général `$r` avec l'instruction `mflo $r`. De même, l'instruction `mfhi $r` permet de copier le contenu du registre `hi` dans le registre `$r`. Ces deux instructions sont les deux seules instructions permettant de récupérer les valeurs contenues dans les registres `lo` et `hi`, c'est-à-dire le résultat des instructions `mult` et `div`.

Question 1

1. Écrivez un programme assembleur qui charge les valeurs (décimales) 84 et 10 dans les registres \$9 et \$10, puis effectue la division de 84 par 10, récupère le quotient et le reste dans les registres \$11 et \$12 et les affiche.
2. Assemblez le programme et exécutez le pas à pas en regardant le contenu des différents registres tout au long de l'exécution, notamment les registres `lo` et `hi`.
3. Complétez le programme pour qu'il reconstruise la valeur 84 dans le registre \$13 à partir du quotient et du reste. Le programme doit afficher le résultat recalculé.
4. Assemblez le programme et exécutez-le pas à pas en regardant le contenu des différents registres tout au long de l'exécution.

Question 2

1. Modifiez votre programme pour que les deux entiers de départ soient lus au clavier.
2. Testez le programme pour différentes valeurs et vérifiez qu'il est correct (la valeur reconstruite est bien égale à la valeur initiale).

Exercice 4 – Instructions logiques, décalages et comparaisons

Soit code suivant :

```
.data
.text
        # Operations logiques
        ori $8, $0, 0x00FF # $8 =
        ori $9, $0, 0xFFF0 # $9 =
        and $10, $9, $8    # $10 =
        xor $11, $9, $8    # $11 =
        xor $11, $11, $11  # $11 =

        # Decalages
        ori $9, $0, 25

        sll $10, $9, 1     # $10 =
        sll $11, $9, 2     # $11 =
        sll $12, $9, 3     # $12 =

        srl $10, $9, 1     # $10 =
        srl $10, $9, 2     # $10 =
        srl $10, $9, 3     # $10 =

        addi $9, $0, -25

        srl  $10, $9, 1    # $10 =
        srl  $11, $9, 2    # $11 =

        sra  $12, $9, 1    # $12 =
        sra  $13, $9, 2    # $13 =
        sra  $14, $9, 3    # $14 =

        # Comparaisons
        ori $9, $0, 2
        ori $8, $0, 4
        slt $11, $8, $9    # $11 =
        slt $12, $9, $8    # $12 =

        ori $2, $0, 10
        syscall
```

Sans exécuter le code, donnez les valeurs contenues dans les registres après l'exécution de chaque ligne.

Copiez et exécutez le code, regardez le contenu des registres après l'exécution de chaque instruction et vérifiez vos réponses à la question précédente. En cas d'erreur, vous devez comprendre votre erreur ! Vous pouvez demander l'affichage des valeurs en décimal avant l'exécution des opérations de décalage et de comparaison.

Qu'en déduisez vous sur les opérations de décalage à gauche et à droite ?

Exercice 5 – Mélange des octets d'un mot

Donner un programme qui charge dans le registre `$3` la valeur 0xAABBCCDD et affiche cette valeur en hexadécimal (appel système numéro 34). Testez votre programme.

Ensuite, ajouter une suite d'instructions comportant des décalages et opérations logiques permettant d'obtenir à partir d'un mot de 4 octets $\circ_3\circ_2\circ_1\circ_0$ dans `$3` le mot $\circ_0\circ_2\circ_3\circ_1$ dans le registre `$5`. Afficher à la fin la valeur obtenue.

Testez votre programme sur la valeur initiale puis modifiez celle-ci pour vérifier que votre code fonctionne sur n'importe quel mot de 32 bits.

TD4 : Variables globales et instructions mémoire

Objectif(s)

- ✯ Familiarisation avec la déclaration de variables globales/réservation de mots mémoire.
- ✯ Familiarisation avec les instructions d'accès mémoire MIPS
- ✯ Tableaux et chaines de caractères
- ✯ Appels système de lecture/écriture de chaines de caractères

Exercice(s)

Exercice 1 – Déclaration de variables globales et réservation de mots mémoire

Question 1

1. Rappelez les sections qui composent un programme assembleur et ce que l'on trouve dans ces sections.
2. Que deviennent ces différentes sections à l'exécution ? Où sont elles implantées ?

Question 2

1. Expliquez, à l'aide de votre mémento, les différentes directives de déclaration de variables globales/réservation de zones mémoire.
2. Déclarez une variable globale de type entier et initialisée à 10 en décimal.
3. Déclarez une variable globale de type entier non initialisée.
4. Déclarez une zone mémoire de 10 octets non initialisée.
5. Déclarez une variable globale contenant la chaine de caractères "toto".
6. Que représentent les étiquettes que l'on utilise pour déclarer les variables globales ?
7. Déclarez 3 entiers consécutifs de telle sorte que l'adresse du premier soit `tab`. Quelle est l'adresse du 2ème entier ?
8. Rappelez comment se calcule l'adresse d'implantation d'une déclaration puis donnez les adresses d'implantation de toutes les déclarations précédentes.

Exercice 2 – Instructions MIPS de lecture/écriture en mémoire

Les instructions mémoire permettent de lire ou d'écrire des valeurs en mémoire à une adresse donnée, c'est-à-dire de lire ou d'écrire une valeur dans les variables déclarées dans la section `.data`.

Question 1

Faites le tour des instructions d'accès mémoire en lecture et en écriture de votre mémento. Combien d'opérandes sources et destination ont ces instructions ? De quelle nature sont-ils ? À quoi correspondent les opérandes ?

Question 2
Quel est le format de codage de ces instructions ? Donnez le codage de l'instruction `sw $9, 4($8)`.

Question 3
Quelle est la différence entre les instructions `lw`, `lh`, `lb`, `lhu` et `lbu` ?

Question 4

L'adresse du mot à lire/écrire est la somme d'une valeur contenue dans un registre (adresse de base) et d'un déplacement (valeur immédiate). Par exemple $lw\ r_{dest},\ imm(r_{addr})$. Pour pouvoir exécuter cette instruction, il faut que le registre r_{addr} contienne une valeur telle que sa somme avec l'immédiat corresponde à une adresse mémoire valide. Avant tout accès à la mémoire, il faut donc charger l'adresse de base dans le registre r_{addr}.

1. Quelle est la taille d'une adresse mémoire ? Donnez la suite d'instructions permettant de mettre dans le registre $8 l'adresse d'un mot mémoire implanté à l'adresse 0x1001 0004 correspondant à la variable `var1`.
2. Écrivez un programme assembleur avec la déclaration d'un mot mémoire d'adresse `var1` initialisé à 0xFF et d'un mot d'adresse `var2` non initialisé. Le programme principal charge dans $9 la valeur contenue en mémoire à l'adresse `var1`, ajoute 5 à cette valeur et stocke le résultat en mémoire à l'adresse `var2`.
3. Donnez le code C correspondant au programme que vous avez écrit.

Exercice 3 – Contenu mémoire et rangement mémoire
Soit le programme suivant :

```
.data
   var1: .word 0xCCDDEEFF
   var2: .byte 0x11
   var3: .byte 0x22
   var4: .byte 0x33
   var5: .byte 0x44
   var6: .asciiz "123"

.text
   lui   $8, 0x1001
   lw    $9, 0($8)
   lw    $10, 4($8)
   lb    $11, 0($8)
   lbu   $12, 0($8)
   addiu $12, $12, 1
   addiu $11, $11, 1
   sw    $11, 0($8)
   sb    $9, 7($8)

   ori   $2, $0, 10
   syscall
```

Question 1
Donnez le contenu de la mémoire, octet par octet et mot par mot, après le chargement du programme et avant son exécution :

Adresse	octet n	octet (n+1)	octet (n+2)	octet (n+3)	mot
0x10010000					
0x10010004					
0x10010008					
0x1001000c					

Question 2

Déroulez à la main l'exécution du programme et donnez le contenu des registres à la fin du programme ainsi que le contenu de la mémoire (faites évoluer le contenu de la mémoire instruction par instruction).

registre	$8	$9	$10	$11	$12
contenu					

Adresse	octet n	octet (n+1)	octet (n+2)	octet (n+3)	mot
0x10010000					
0x10010004					
0x10010008					
0x1001000c					

Exercice 4 – Accès à un tableau, codage de caractère

Remarque :

— On fait l'hypothèse que le type int en C est codé sur 4 octets (c'est généralement le cas sur les architectures 32 bits)

— Un tableau `tab` est un ensemble de valeurs rangées consécutivement en mémoire et `tab[0]` désigne le premier élément du tableau.

— Une chaîne de caractères `ch` est un tableau de caractères, déclarée comme un tableau, par exemple `char ch[] = "exemple";`.

Question 1

Écrivez un programme assembleur correspondant au code C suivant :

```
int tab[] = { 1, 2, 34, 256, -1 }; /* tableau d'entiers */
char chaine[] = "toto";            /* chaîne de caractères */

void main() {
   printf("%d", tab[3]);
   printf("%d", chaine[2]);
   exit();
}
```

Question 2

Quelles sont les valeurs affichées lors de l'exécution ?

Exercice 5 – Appels système autour des chaines de caractères

Question 1

1. Écrivez un programme avec une variable globale de type chaine de caractères contenant la valeur "test d'affichage". Écrivez le programme principal qui affiche la chaîne de caractères à l'aide de l'appel système correspondant (cherchez-le dans le mémento).
2. Modifiez votre programme en déclarant une zone mémoire non allouée de 32 octets. Le programme lit au clavier une chaîne de caractères qu'il stocke dans cette zone mémoire (cherchez l'appel système correspondant dans votre mémento). Ensuite, il affiche le résultat de la lecture (la chaine de caractères lue) à l'aide de l'appel système correspondant (voir le mémento).

TME4 : Variables globales et instructions mémoire

IMPORTANT : Dans le menu Settings :

1. Décochez l'utilisation des pseudo-instructions (*"Permit extended (pseudo) instructions and formats"*) : vous devez toujours utiliser uniquement les instructions de base dans votre programme.
2. Cochez l'affichage des valeurs et des adresses en hexadécimal (*"Values (resp. Addresses) displayed in hexadecimal"*).

Objectif(s)

- ✯ Familiarisation avec les déclarations de variables globales et la réservation d'espace mémoire pour les données
- ✯ Écriture de programmes assembleur comportant des instructions de lecture et d'écriture en mémoire
- ✯ Familiarisation avec le codage ASCII et manipulation de chaînes de caractères.

Exercice(s)

Exercice 1

Question 1

Déclarez 4 octets $o1, o2, o3$ et $o4$ valant respectivement 1, 2, 3 et 4 puis un mot $m1$ valant `0xAABBCCDD`. Assemblez ce programme et regardez comment ces variables sont implantées en mémoire.

Question 2

Donnez les adresses correspondant aux étiquettes du programme $o1, o2, o3$, $o4$ et $m1$.

Question 3

Dans l'onglet *Settings*, activez **Show Labels Window**. Qu'observez-vous ? Vérifiez ce que vous avez répondu à la question précédente.

Exercice 2 – Programme avec accès mémoire

Question 1

Écrivez un programme assembleur comportant l'allocation aux adresses v1 et v2 de deux mots mémoire initialisés respectivement à -1 et 0xFF. Le programme doit charger les valeurs contenues aux adresses v1 et v2 dans les registres $8 et $9, puis afficher les deux valeurs. Assemblez et testez votre programme.

Question 2

Modifiez le programme pour qu'il ajoute 1 à v1 et à v2 puis range les nouvelles valeurs en mémoire. Assemblez et exécutez le programme. Vérifiez le contenu de la mémoire à la fin de l'exécution (l'exécution a-t-elle bien modifié les valeurs implantées en mémoire aux adresses v1 et v2 ?).

Question 3
Écrivez un programme qui déclare un octet de valeur `0xFF`, qui charge cet octet dans le registre $9 en considérant sa valeur comme signée et qui charge cet octet dans $10 en considérant sa valeur comme non signée. Le programme affiche ensuite le contenu des deux registres.

Exécutez le programme et regardez le contenu des registres $9 et $10 après chargement. Quelles sont les valeurs affichées ? Expliquez.

Exercice 3 – Chaine de caractères

Question 1
Écrivez un programme qui déclare en variable globale une chaine de caractères `ch` initialisée à "coucou", puis qui affiche cette chaine de caractères à l'écran. Exécutez votre programme.

Question 2
Complétez votre programme pour qu'il échange les deux premiers caractères de la chaine `ch` et affiche la chaine modifiée.

Exercice 4 – Chaîne de caractères et introduction au codage ASCII

Question 1

1. À l'aide de votre mémento, donnez le codage de la chaine `"123456"` en pensant à mettre le caractère de fin de chaine à la fin (valeur `0x00`).
2. Déclarez un tableau d'octets initialisé avec les valeurs trouvées à la question précédente.
3. Pour vérifier votre codage, écrivez un programme qui utilise l'appel système d'affichage d'une chaine de caractères pour afficher le contenu du tableau.
4. Chargez le 3ème caractère de la chaine dans le registre $16. Affichez sa valeur en décimal. Quelle est-elle ?
5. Déterminez une méthode pour retrouver un chiffre à partir du code ASCII du caractère représentant ce chiffre. Regardez pour cela le codage ASCII des caractères qui représentent un chiffre.
6. Programmez votre solution et appliquez-la à la valeur qui a été stockée dans $16 : vous devez obtenir la valeur 3, que vous stockerez dans le registre $17. Affichez la valeur obtenue.
7. Vérifiez votre méthode en l'appliquant à d'autres caractères représentant un chiffre, par exemple `4`.

Exercice 5 – Tableau
Écrivez un programme assembleur qui implante le programme C suivant :

```
int tab[] = {4, 23, 12, 3, 8, 1};

int s;
int p;

void main() {
   s = tab[3];
   p = tab[4];

   tab[0] = s + 1;
   tab[1] = s + p;
```

```
    tab[2] = tab[5];

    exit();
}
```

Exécutez votre programme pas à pas en regardant évoluer le contenu de la mémoire. Vérifiez que les valeurs contenues dans le tableau sont correctes à la fin de l'exécution.

TD5 : Instructions de sauts et structures de contrôle

Objectif(s)

- ✯ Familiarisation avec les instructions de rupture de séquence : les branchements (saut conditionnels ou inconditionnels).
- ✯ Écriture de programmes contenant des conditionnelles et/ou des boucles.
- ✯ Parcours de tableau.

Exercice(s)

Exercice 1 – Instructions de saut conditionnel et inconditionnel

A l'aide de votre mémento, dites ce que font les instructions suivantes et quel est leur format de codage :

```
beq  $8, $9, etiquette
bne  $8, $9, etiquette
blez $8, etiquette
bltz $8, etiquette
bgez $8, etiquette
bgtz $8, etiquette
j etiquette
jr r1
```

Quelles sont les instructions de saut conditionnel et de saut inconditionnel parmi les instructions ci-dessus ? Quelles sont les conditions que l'on peut exprimer dans les instructions de saut conditionnel ?

Dans les programmes écrits dans des langages de haut niveau, on utilise des structures de contrôle qui cassent l'exécution séquentielle des instructions d'un programme. Les deux types de structures de contrôle les plus employés sont l'alternative (if cond then ... [else ...]) et les boucles (boucles for ou boucles while).

Exercice 2 – L'alternative

Question 1

Donnez le code assembleur correspondant au programme C ci-dessous. Simulez l'exécution du programme et vérifiez que la valeur de `a` en mémoire à l'issue du programme est correcte. Faites de même avec une valeur initiale de `a` permettant de tester le deuxième chemin d'exécution possible.

```
int a = -5;
int b = 3;
void main() {
  if (a == 0) {
     a = a + b;
  }
```

```
    else {
        a = a - b;
    }
    exit();
}
```

Question 2

Donnez le calcul de la condition et le saut conditionnel dans les cas où la condition `a==0` est remplacée dans le code source par :

1. `a != 0`
2. `a > 0`
3. `a >= 2`
4. `a < b`

Exercice 3 – Boucles for

Question 1

Donnez un programme assembleur qui contient deux variables globales `p` et `q` initialisées (à 1 et 10 par exemple). Le programme principal calcule la somme des entiers compris entre `p` et `q` inclus et affiche cette somme.

Attention : si `q` est strictement inférieur à `p`, la somme doit valoir `0` ; si les deux entiers sont égaux, alors somme doit valoir `p`.

Exercice 4 – Boucle while avec conditionnelle et manipulation de tableau

Question 1

Donnez un programme qui calcule le maximum d'un tableau d'entiers relatifs (codés en compléments à 2) strictement positifs et affiche ce maximum ; le mot à 0 marque la fin du tableau.

Rappel : un tableau de `N` entiers correspond à `N` entiers rangés consécutivement en mémoire. Ainsi, il vous faut allouer un ensemble d'entiers non nuls en mémoire et indiquer la fin du tableau en allouant un mot initialisé à 0 après la déclaration des entiers du tableau.

Exercice 5 – Manipulation de bits : décalage et masque (à faire en TME si non traité en TD)

Question 1

Déclarez un entier `n` en variable globale et initialisez-le à la valeur 123. On souhaite écrire un programme qui calcule puis affiche le nombre de bits à 1 dans le mot binaire représentant l'entier.

Quelle valeur doit afficher ce programme quand `n` vaut 123 ? quand `n` vaut -1 ou 0xFEDCBA98 ?

Pour écrire ce programme, il est nécessaire de tester la valeur des 32 bits du mot binaire correspondant à `n`. Il faut donc écrire l'équivalent d'une boucle `for` en assembleur qui traite le ième bit du mot à la ième itération. Si ce bit vaut 1, il faut le compter dans le résultat.

Pour récupérer la valeur du bit de poids faible d'un mot binaire, il suffit de réaliser une opération de masquage (opération booléenne ET) avec la valeur 1 :

$$a_3a_2a_1a_0 \ \& \ 0b0001 = 000a_0$$

Pour traiter tous les bits d'un mot, il suffit de décaler le mot vers la droite de manière logique ce qui permet de positionner sur le bit de poids faible du mot résultant le bit voulu :

$$a_3a_2a_1a_0 \gg_{unsigned} 1 = 0a_3a_2a_1$$

$$a_3a_2a_1a_0 \gg_{unsigned} 2 = 00a_3a_2$$

TME5 : Instructions de sauts et structures de contrôle

Objectif(s)

- ✯ Écriture de programmes assembleur comportant des boucles et des conditionnelles
- ✯ Comparaison à une valeur pour les sauts conditionnels
- ✯ Parcours de tableau et manipulation de chaines de caractères

Exercice(s)

Exercice 1 – Boucle for avec comparaison à une valeur non nulle

Question 1

Donnez un programme assembleur qui comporte deux variables globales `p` et `q` non initialisées et qui lit leur valeur au clavier. Ensuite, le programme principal calcule la somme des entiers de `p` à `q` non inclus et l'affiche. Notez que quand $\texttt{p} \geq \texttt{q}$ la valeur affichée doit être 0.

Testez votre programme avec différentes valeurs pour vous assurer qu'il fonctionne correctement quelles que soient les valeurs de `p` et `q` entrées.

Exercice 2 – Calcul du PGCD

Question 1

On rappelle l'algorithme permettant de calculer le PGCD de deux nombres `a` et `b` :

```
tmpa = a
tmpb = b
TANT QUE tmpa != tmpb FAIRE
   SI tmpa > tmpb ALORS
      tmpa = tmpa - tmpb
   SINON
      tmpb = tmpb - tmpa
   FIN SI
FIN TANT QUE
```

Écrivez un code assembleur qui calcule le PGCD de deux entiers `a` et `b` entrés au clavier et affiche le résultat.

Exercice 3 – Manipulation de chaines

Écrivez un programme qui, étant donnée `ch` une chaine de caractères déclarée comme variable globale, calcule la taille de cette chaine de caractères et l'affiche.

Exercice 4 – Parcours de tableau

Écrivez un programme qui, étant donné `tab` un tableau d'entiers strictement positifs se terminant par l'élément -1 et `val` une valeur, tous deux déclarés comme variables globales (`val` en premier), compte le nombre d'éléments du tableau strictement inférieurs à `val` et l'affiche.

Exercice 5 – Exercices de TD

Terminez les exercices du TD5 non traités.

TD6 : Variable locales et pile d'exécution, tableau et codage ASCII

Objectif(s)

- ✯ Notion de pile et implantation des variables locales
- ✯ Traduction littérale de code C
- ✯ Optimisation des variables locales et des accès mémoire
- ✯ Manipulations de tableau, en variable globale puis locale
- ✯ Codage ASCII et chaîne de caractères
- ✯ Correspondance entre un caractère ASCII représentant un chiffre et la valeur numérique du chiffre ; utilisation de masque
- ✯ Recopie mémoire

Exercice(s)

Exercice 1 – Pile d'exécution, variables locales et traduction littérale de code C

Question 1

Qu'est ce que la pile d'exécution ? À quoi sert-elle ? Comment s'en sert-on ?
Quelles sont les premières instructions d'un programme (ou d'une fonction) ? Quel est leur but ? Que doivent faire les dernières instructions d'un programme ?

Question 2

Soit un programme principal qui a trois variables locales `a`, `b` et `c` de type `int`, `char`, `short` déclarées dans cet ordre et initialisées avec les valeurs 12, 3 et 5 respectivement. Quelles seront les premières instructions ainsi que les dernières instructions du programme principal ? Pendant l'exécution du programme, quelles seront les adresses d'implantation des variables locales ?

Question 3

Rappelez les règles de traduction littérale en assembleur d'une instruction C de type assignation, par exemple `var1 = var2 + var3`.

Exercice 2 – Chaine de caractères et codage ASCII

Question 1

- Donnez le codage ASCII de la chaine `"AaBb"`
- Quel est le codage ASCII correspondant aux chiffres 0, 1, ..., 9 ?
- Donnez le codage ASCII de la chaine de caractères `"1024"`
- Quelle est l'adresse mémoire du ième caractère d'une chaine ?

— Comment retrouver à partir du codage ASCII d'un chiffre (par exemple le codage du caractère '3') la valeur de l'entier correspondant (3 dans l'exemple) ?

Exercice 3 – Calcul du nombre associé à une chaîne de caractères représentant un nombre et affichage de la valeur correspondante

Question 1

On considère le programme C ci-dessous qui comporte une variable globale `ch` de type chaine de caractères représentant un nombre entier positif. Le programme principal calcule la valeur numérique correspondante puis affiche cette valeur. Écrivez un programme assembleur qui correspond à la traduction littérale du code ci-dessous.

```
char ch[] = "1234";

void main() {
   int i = 0;
   int val = 0;
   char c;

   while (ch[i] != 0) {
      c = ch[i];
      c = c & 0x0F; /* récupération de la valeur du chiffre */
      val = val * 10 + c;
      i += 1;
   }
   printf("%d", val);
   exit();
}
```

Remarque:
Ce programme utilise le fait que :

— 3456 = (((3 * 10 + 4) * 10 + 5) * 10 + 6)

— une chaine de caractères déclarée par la directive `.asciiz` se termine toujours par le caractère nul (qui vaut 0) qui correspond au caractère de fin de chaine.

Question 2

Donnez une version dans laquelle toutes les variables locales sont optimisées en registre.

Question 3

Donnez une version dans laquelle les calculs ou chargements mémoire redondants sont éliminés. Quels autres types d'optimisation peut-on faire ?

Question 4

On considère désormais la variante ci-dessous dans laquelle la chaine de caractère est locale au programme principal. Que faut-il changer au programme écrit à la question précédente pour obtenir un code optimisé correspondant à cette variante ?

```
void main() {
   int i = 0;
   int val = 0;
   char c;
   char ch[] = "1234";

   while (ch[i] != 0) {
```

```
      c = ch[i];
      c = c & 0x0F; /* récupération de la valeur du chiffre */
      val = val * 10 + c;
      i += 1;
   }
   printf("%d", val);
   exit();
}
```

Exercice 4 – Recopie mémoire d'une chaine de caractère

Question 1

On considère le programme C qui réalise la recopie de `N` caractères de la chaine de caractères `ch1`, globale et initialisée, dans une autre chaine de caractères `ch2`, globale mais non initialisée. Le caractère de fin de chaine est ajouté après les `N` caractères. Le programme affiche la chaine recopiée à la fin du programme.

On supposera que le nombre de caractères à copier sera toujours inférieur à la taille de la chaine `ch1`, qui sera elle toujours strictement inférieure à 20.

```
char ch2[20];
int  N = 2;           /* N <= strlen(ch1) < 20, faire varier pour vos tests */
char ch1[] = "Hello";

void main() {
   int i = 0;
   for (; i < N; i += 1) {
      ch2[i] = ch1[i];
   }
   ch2[i] = '\0';
   printf("%s", ch2);
   exit();
}
```

Donnez 2 versions du code assembleur de ce programme : une version sans optimisation et une version optimisée.

Question 2

On considère désormais que les deux chaines sont des variables locales du programme principal :

```
int  N = 2;       /* N <= strlen(ch1) < 20, faire varier pour les tests */

void main() {
   int i = 0;
   char ch2[20];
   char ch1[] = "Hello";

   for (; i < N; i += 1) {
      ch2[i] = ch1[i];
   }
   printf("%s", ch2);
   exit();
}
```

Que faut-il changer dans le programme précédent pour produire le code assembleur correspondant à ce programme, sans optimisation et avec optimisation ?

TME6 : Variables locales et pile d'exécution, tableau et codage ASCII

Objectif(s)

- ✯ Maitrise de la notion d'écriture mémoire de données, locales ou globales
- ✯ Maitrise de la notion de codage ASCII
- ✯ Maitrise des tableaux et chaines en données locales ou globales

Exercice(s)

Exercice 1 – Passage d'un nombre à sa représentation en mémoire sous forme de chaîne de caractères

Le but de cet exercice est de construire la chaine de caractères correspondant à un entier positif lu au clavier. Par exemple, si l'entier lu au clavier est 5678 alors on veut construire la chaine "5678". L'entier peut valoir 0 et la chaine sera alors "0".

Il s'agit donc de faire le travail inverse de l'exercice 2 du TD6 qui vise à calculer la valeur correspondant à un entier représenté par une chaine de caractères.

L'exercice se compose de 3 questions permettant de construire la chaine voulue, pas nécessairement de manière optimale mais permettant de travailler des notions importantes : écriture mémoire, calcul de l'adresse d'un élément de tableau, implantation et manipulation de variables locales.

Question 1

On considère dans un premier temps le programme C ci-dessous. Ce programme construit une chaine de caractères à partir d'un entier positif lu au clavier. Par divisions successives par 10, il est possible de récupérer tous les chiffres qui composent le nombre en commençant par le chiffre le plus à droite, le chiffre des unités. Ce chiffre des unités correspond au dernier caractère de la chaine que l'on cherche à construire. Comme on ne connait pas le nombre de chiffres du nombre, on ne sait pas quelle est la place du chiffre des unités dans la chaine que l'on veut construire. Par contre on peut construire une chaine de taille fixe en ajoutant des "0".

Sachant que le nombre lu est codé sur 32 bits, il est toujours strictement inférieur à 2^{32} soit 4294967296. Ainsi, la chaine de caractères (caractère de fin de chaine inclus) ne comportera jamais plus de 11 caractères avec 10 chiffres (quitte à n'avoir que des 0). Dans cette question, quel que soit le nombre lu, on construit une chaine de 10 chiffres avec autant de 0 que nécessaire afin de remplir les 10 caractères de la chaîne.

```
char chaine[11]; /* au plus 10 chiffres + 1 fin de chaine */

void main() {
  int i, nb, r, nbzero;
  scanf("%d", &nb); /* lecture au clavier d'un nombre */

  chaine[10] = 0;  /* caractère fin de chaine */

  /* remplissage de la chaine représentant le nombre,
     avec autant de 0 devant que nécessaire */
  for (i = 9; i >= 0; i -= 1) {
    r = nb % 10;     /* chiffre des unités */
```

```
    nb = nb / 10;   /* nombre de dizaines */
    chaine[i] = r + 0x30;
  }
  printf("%s", chaine);
  exit();
}
```

Implantez le programme assembleur correspondant sans optimisation. Puis dans un nouveau fichier, à partir de la version sans optimisation, donnez une version avec optimisation.

Question 2

Le programme augmenté ci-dessous comporte, à la suite de ce qu'il y avait déjà, une boucle pour compter le nombre de '0' qui se trouvent en début de la chaine construite (pour les éliminer ensuite dans la question 3).

En partant de la version optimisée de la question précédente, dans un nouveau fichier, implantez le code ci-dessous. Seule une version optimisée est demandée.

```
char chaine[11]; /* au plus 10 chiffres + 1 fin de chaine */

void main() {
  int i, nb, r, nbzero;
  scanf("%d", &nb); /* lecture clavier d'un nombre */

  chaine[10] = 0;  /* caractère fin de chaine */

  /* remplissage de la chaine représentant le nombre,
     avec autant de 0 devant que nécessaire */
  for (i = 9; i >= 0; i -= 1) {
    r = nb % 10;  /* chiffre des unités */
    nb = nb / 10;   /* nombre de dizaines */
    chaine[i] = r + 0x30;
  }
  printf("%s", chaine);
  /* détermination du nombre de caractères '0' dans chaine[0..8] */
  nbzero = 0;
  i = 0;
  while (i < 9 && chaine[i] == 0x30){
    nbzero += 1;
    i++;
  }
  printf("%d", nbzero);  /* affichage du nb de zeros calculé */
  exit();
}
```

Question 3

Le programme encore augmenté ci-dessous contient (à la suite de ce qu'il y avait déjà) une recopie mémoire en place de la chaine permettant d'éliminer les '0' non significatifs en début de chaine. Le caractère de fin de chaine est recopié aussi. La chaine finale est affichée.

Implantez dans un nouveau fichier en partant du code réponse de la question précédente. Là encore, seule la version optimisée est demandée.

```
char chaine[11];            /* au plus 10 chiffres + 1 fin de chaine */

void main() {
  int i, nb, r, nbzero;
  scanf("%d", &nb);         /* lecture clavier d'un nombre */

  chaine[10] = 0;           /* caractère fin de chaine */

  /* remplissage de la chaine représentant le nombre,
```

```
  avec autant de 0 devant que nécessaire */
  for (i = 9; i >= 0; i -= 1) {
    r = nb % 10;           /* chiffre des unités */
    nb = nb / 10;          /* nombre de dizaines */
    chaine[i] = r + 0x30;
  }
  printf("%s", chaine);   /* affichage de la chaine avec des '0' devant*/
  /* détermination du nombre de caractères '0' dans chaine[0..8] */
  nbzero = 0;
  for (i = 0; i < 9; i += 1) {
    if (chaine[i] == 0x30) {
      nbzero += 1;
    }
  }
   printf("%d", nbzero);
  /* recopie en place de la chaine pour éliminer les premiers caractères '0'
     (caractère de fin de chaine inclus) */
  for (i = 0; i < 10 - nbzero + 1; i += 1) {
    chaine[i] = chaine[i + nbzero];
  }

  printf("%s", chaine);  /* affichage de la chaine représentant le nombre*/
  exit();
}
```

Question 4
Changer le programme de la question précédente de sorte que la chaine de caractères soit maintenant une variable locale du programme principal.

Exercice 2 – Manipulation d'enregistrements et de pointeurs

On considère des enregistrements comme définis dans le code ci-dessous et un programme principal qui les manipule.

Question 1
Quelle est la taille en octets d'un donnée de type `struct point` ? Si une telle donnée est implantée à l'adressse `A`, à quelle adresse se trouve chacun des 3 champs en fonction de `A` ?

```
struct point {
   char[2] nom; /* nom composé d'un caractère + fin de chaine */
   int abs;
   int ord;
};

struct point p1 = {"X"; 2; 6;};
struct point p2 = {"Y"; 4; 4;};
struct point p3;
struct point * ptr;

void main() {
   ptr = &p3;
   ptr->abs = (p1.abs + p2.abs) / 2;
   ptr->ord = (p1.ord + p2.ord) / 2;
   ptr->nom[0] = 'Z';
   ptr->nom[1] = '\0'; /* caractère fin de chaine */

   /* affichage des champs de p3 */
```

```
   printf("%s", p3.nom);
   printf("%d", p3.abs);
   printf("%d", p3.ord);

   exit();
}
```

Question 2
Implanter en assembleur le programme et vérifier le contenu de la mémoire de données au cours de son exécution.

TD7 : Appels de fonctions

Objectif(s)

- ✯ Maitrise des conventions d'appel en MIPS
- ✯ Écriture de fonctions et d'appels de fonction en assembleur

Exercice(s)

Exercice 1 – Questions préliminaires

Question 1

1. Rappelez les informations à faire passer entre une fonction appelante et une fonction appelée lors d'un appel de fonction et le sens de ces échanges.
2. Quelles sont les informations à sauvegarder lors d'appel de fonction et pourquoi ?
3. Que définissent les conventions d'appel ? A quoi servent-elles ?
4. Rappelez les conventions d'appel en MIPS ?

La figure 1 contient la procédure pour l'implantation de fonction, vous suivrez cette procédure pour TOUTES les fonctions que vous aurez à écrire. Complétez les "trous".

Exercice 2 – Moyenne de 3 nombres

On souhaite écrire le code du programme C suivant :

```
/* trois variables globales */
int n = 15;
int m = -1;
int l = 124;

/* fonction moyenne3 */
int moyenne3(int p, int q, int r) {
   int sum = p + q + r;
   return sum / 3;
}

/* programme principal */
void main() {
  int tmp;
  tmp = moyenne3(n, m, 5);
  printf("%d", tmp);
  tmp = moyenne3(m, l, m + 5);
  printf("%d", tmp);
  exit();
}
```

Procédure pour l'implantation d'une fonction

0. Se représenter la pile à l'entrée de la fonction
1. Écrire le code du corps de la fonction en :
   (a) choisissant les registres qu'on veut associer avec les variables locales (si optimisées en registre)
   (b) écrivant les lectures et écriture en pile (de paramètres ou variables locales) en laissant un `??` pour l'immédiat de ces instructions de transfert mémoire (adresse relative au pointeur de pile)
   (c) mettant dans le registre `$_` le résultat de la fonction
2. Déterminer la taille du contexte de la fonction à partir de
   (a) `nr` le nombre de registres persistants utilisés dans le corps de la fonction
   (b) `na` le nombre max de mots nécessairee au stockage des arguments des fonctions appelées par la fonction. Pour determiner ce nombre max de mots, on calcul le nombre de mots pour les arguments de chaque fonction appelée comme suit : on compte 1 mot pour les 4 premiers arguments auquel on ajoute le nombre de mots permettant de stocker les arguments suivants en respectant leur contrainte d'alignement.
   (c) du nombre de mots `nv` nécessaire aux stockages variables locales, déterminable à partir du code source de la fonction
3. Écrire le prologue comportant :
   (a) l'allocation des emplacements sur la pile,
   (b) la sauvegarde des registres persistants et du registre `$__` (= écriture sur la pile) qui doivent être rangés par ordre croissant de leur numéro des adresses les plus petites aux plus grandes.

   NB : si besoin utiliser un dessin de la pile pour déterminer les emplacement des registres, les adresses des variables locales et celles des paramètres.
   (d) Si besoin, sauvegarder les 4 premiers paramètres dans leur emplacement
   (e) Si besoin, initialiser les variables locales
4. Dans le corps de la fonction, adapter les déplacements relatifs aux pointeurs de pile quand nécessaire (accès à aux variables locales, ou aux paramètres de la fonction).
5. Écrire l'épilogue soit dans l'ordre :
   (a) la restauration des registres (= lecture des valeurs sauvegardées sur la pile),
   (b) la désallocation des emplacements sur la pile,
   (c) le retour à l'appelant avec l'instruction _____

FIGURE 1 – Étapes préconisées pour l'implantation d'une fonction

Question 1 – Corps de la fonction
Donnez le code assembleur correspondant au corps de la fonction `moyenne3` en optimisant la variable `sum` dans le registre `$16`. Vous utiliserez les registres `$8`, `$9`, `$10` pour contenir des résultats de calculs ou des valeurs intermédiaires.

Question 2 – Prologue
Déterminez combien il faut allouer d'octets sur la pile dans le prologue et écrivez le prologue de la fonction `moyenne3`.

Question 3 – Epilogue
Écrivez l'épilogue de la fonction `moyenne3`.

Question 4 – Appel de fonction avec des paramètres
Quelles sont les étapes à suivre pour écrire un appel à une fonction qui a des paramètres ?

Question 5 – Code du programme principal
Quelle est la taille du contexte du programme principal ? Justifiez votre réponse.
Donnez les directives de déclaration des variables globales et le code correspondant au programme principal (`main`).

Question 6 – Représentation graphique de la pile
Représentez graphiquement l'évolution de la pile au cours de l'exécution du programme. On fera grandir la pile "vers le bas". On rappelle que, par convention, le pointeur de pile pointe sur le sommet de la pile, c'est-à-dire sur la dernière case occupée de la pile au moment où l'on entre dans une fonction.

Exercice 3 – Moyenne de 3 et 5 nombres
On souhaite avoir en plus une fonction qui calcule la moyenne de 5 entiers. Voici le code du programme C que l'on considère désormais :

```
/* trois variables globales */
int n = 15;
int m = -1;
int l = 124;

/* fonction moyenne3 */
int moyenne3(int p, int q, int r) {
   int sum = p + q + r;
   return sum / 3;
}

/* fonction moyenne5 */
int moyenne5(int p, int q, int r, int s, int t) {
   int sum = p + q + r + s + t;
   return sum / 5;
}

/* programme principal */
void main() {
  int tmp;
  tmp = moyenne3(n, m, 5);
  printf("%d", tmp);
  tmp = moyenne5(m, l, m + 5, 12, 35);
  printf("%d", tmp);
  exit();
}
```

Question 1
Quelle est conséquence de la différence du nombre de paramètres entre les fonctions `moyenne5` et `moyenne3` lors de l'écriture du corps de la fonction `moyenne5` ?
En suivant les étapes préconisées, donnez le code du corps la fonction `moyenne5` en utilisant le registre $10 pour contenir la valeur du paramètre `t`. Puis, donnez le prologue et l'épiloque de la fonction `moyenne5`.

Question 2
Quelle est la taille du contexte de ce nouveau programme principal ?
Donnez le code correspondant au programme principal.

Question 3
Représentez la pile avant à l'entrée de la fonction `moyenne5` et après son prologue.

Question 4
Que faudrait-il changer au code précédemment écrit si les paramètres de la fonction `moyenne5`, étaient déclarés comme des **`char`** (**donc des entiers signés codés sur un octet**) ?

Exercice 4 – Tableau en paramètre
On souhaite écrire en assembleur le programme suivant, qui calcule le nombre d'éléments d'un tableau d'entiers positifs et se terminant par -1.

```
int tab1[] = {23, 4, 5, -1};
int tab2[] = {2, 345, 56, 23, 45, -1};

int nb_elem(int tab[]) {
  int nb_elem = 0;
  int i = 0;
  while (tab[i] != -1) {
    nb_elem += 1;
    i += 1;
  }
  return nb_elem;
}

/* programme principal */
void main() {
  printf("%d", nb_elem(tab1));
  printf("\n"); /* affichage du caractère retour à la ligne */
  printf("%d", nb_elem(tab2));
  exit();
}
```

Question 1
Donner le code correspondant à la fonction `nb_elem`.

Question 2
Donner le code correspondant au `main` et aux déclarations des tableaux.

TME7 : Appels de fonctions

Exercice 1 – Variation autour des appels de fonctions

Il est conseillé de préparer la première question (niveau TD6) en amont de ce TME afin d'avoir le temps de traiter toutes les autres questions.

Question 1 – Programme principal sans fonction

Écrivez une programme principal qui transforme tous les caractères minuscules d'une chaîne de caractères `ch`, variable globale du programme, en majuscule. Le programme affiche la chaîne de caractères avant la transformation et après. Rappel : une chaîne de caractères en C se termine toujours par le caractère nul dont la valeur est 0x00.

```
char ch[] = "1 exemple d'exemple\n";

int main(){
  int i = 0;
  printf("%s", ch);
  while (ch[i] != '\0'){
     if (ch[i] >= 'a' && ch[i] <= 'z')
          ch[i] = ch[i] - 0x20;   /* transformation en majuscule si minuscule*/
     i++;
  }
  printf("%s", ch);
  exit();
}
```

Question 2 – Fonction avec un seul argument, de type adresse

Modifiez le programme principal de la question précédente pour que la transformation de la chaîne de caractères soit réalisé par une fonction `f` qui a un paramètre, l'adresse de la chaîne à modifier. Modifiez le programme principal pour que la fonction soit appliquée à 2 chaînes de caractères déclarées comme variables globales et nommées `ch1` et `ch2`.

```
char ch1[] = "1 exemple d'exemple\n";
char ch2[] = "Hello world!\n";

void min_to_maj_chaine(char * ch){
  int i = 0;
  while (ch[i] != 0){
    if (ch[i] >= 'a' && ch[i] <= 'z')
      ch[i] = ch[i] - 0x20;   /* transformation en majuscule si minuscule*/
    i++;
  }
  return;
}

int main(){
```

```
  printf("%s", ch1);
  min_to_maj_chaine(ch1);
  printf("%s", ch1);
  printf("%s", ch2);
  min_to_maj_chaine(ch2);
  printf("%s", ch2);
  exit();
}
```

Question 3 – Fonction avec argument de taille inférieure à 4 octets

On souhaite maintenant écrire et utiliser une fonction qui, étant donné un caractère, le transforme en majuscule si c'est une minuscule et le renvoie, ou le renvoie tel quel si ce n'est pas une minuscule. Cette fonction est appelée, dans le programme principal, sur tous les caractères d'une chaîne globale puis affiche la chaîne transformée.

Voici le code que vous devez implémenter.

```
char ch[] = "1 exemple d'exemple\n";
char min_to_maj_char(char c){
   if ( c >= 'a' && c <= 'z')
      return c - 0x20;
   else
      return c;
}

int main(){
   int i=0;
   printf("%s", ch);
   while (ch[i] != '\0'){
      ch[i] = min_to_maj_char(ch[i]);
      i++;
   }
   printf("%s", ch);
   exit();
}
```

Question 4 – Fonction avec argument de type adresse

On souhaite maintenant écrire et utiliser une fonction qui, étant donné un pointeur vers un caractère, transforme le caractère pointé en majuscule si c'est une minuscule. Le fonction ne renvoie rien. Cette fonction est appelée, dans le programme principal, avec comme paramètre tous les caractères d'une chaîne afin de transformer chacune de ses minuscules.

Voici le code que vous devez implémenter.

```
char ch[] = "1 exemple d'exemple\n";

void min_to_maj_ptr_char(char *c){
   if (*c >= 'a' && *c <= 'z')
      *c =  *c - 0x20;
   return;
}

int main(){
   int i=0;
```

```
   printf("%s", ch);
   while (ch[i] != '\0'){
      min_to_maj_ptr_char(&(ch[i]));
      i++;
   }
   printf("%s", ch);
   exit();
}
```

Exercice 2 – Variations suite et variables locales en paramètres effectifs

Les questions suivantes reprennent toutes les questions de l'exercice précédent mais considère des chaînes de caractères locales au programme principal et initialisées via une lecture au clavier. On entrera des chaînes comportant au plus 15 caractères.

Vous pourrez avantageusement partir de vos codes issus de l'exercice prédécent pour les adapter.

Question 1 – Fonction avec un seul argument, de type adresse

Modifiez le programme principal de la question précédente pour que la transformation de la chaîne de caractères soit réalisé par une fonction `f` qui a un paramètre, l'adresse de la chaîne à modifier. Modifiez le programme principal pour que la fonction soit appliquée à 2 chaînes de caractères déclarées comme variables globales et nommées `ch1` et `ch2`.

```
void min_to_maj_chaine(char * ch){
  int i = 0;
  while (ch[i] != 0){
    if (ch[i] >= 'a' && ch[i] <= 'z')
      ch[i] = ch[i] - 0x20;   /* transformation en majuscule si minuscule*/
    i++;
  }
  return;
}

int main(){
  int i=0;
  int ch1[16];     /* max 15 caractères hors celui de fin de chaine */
  int ch2[16];     /* max 15 caractères hors celui de fin de chaine */
  scanf("%s",ch1); /* lecture au clavier de la chaine */
  scanf("%s",ch2); /* lecture au clavier de la chaine */
  printf("%s", ch1);
  min_to_maj_chaine(ch1);
  printf("%s", ch1);
  printf("%s", ch2);
  min_to_maj_chaine(ch2);
  printf("%s", ch2);
  exit();
}
```

Question 2 – Fonction avec argument de type adresse appliquée à une variable locale de type tableau

Voici le code que vous devez implémenter.

```
char ch[] = "1 exemple d'exemple !";
char min_to_maj_char(char c){
```

```
   if ( c >= 'a' && c <= 'z')
      return c - 0x20;
   else
      return c;
}

int main(){
   int i=0;
   int ch[16];      /* max 15 caractères hors celui de fin de chaine */
   scanf("%s",ch);  /* lecture au clavier de la chaine */
   printf("%s", ch);
   while (ch[i] != '\0'){
      ch[i] = min_to_maj_char(ch[i]);
      i++;
   }
   printf("%s", ch);
   exit();
}
```

Question 3 – Fonction avec argument de type adresse appliquée aux élements d'une variable locale de type tableau

Voici le code que vous devez implémenter.

```
void min_to_maj_ptr_char(char *c){
   if (*c >= 'a' && *c <= 'z')
      *c = *c - 0x20;
   return;
}

int main(){
   int i=0;
   int ch[16];     /* max 15 caractères hors celui de fin de chaine */
   scanf("%s",ch); /* lecture au clavier de la chaine */
   printf("%s", ch);             */
   while (ch[i] != '\0'){
      min_to_maj_ptr_char(&(ch[i]));
      i++;
   }
   printf("%s", ch);
   exit();
}
```

TD8 : Fonctions récursives et appels imbriqués

Objectif(s)

✯ Ce TD a pour but de consolider les appels de fonctions et d'implémenter des fonctions récursives et imbriquées.

Une fonction imbriquée est une fonction qui est appelée par une autre fonction. Les conventions d'appel sont faites pour pouvoir appeler des fonctions écrite soi-même ou par d'autres, sans avoir en avoir se soucier d'autre chose que de les avoir bien suivies.

Une fonction récursive est une fonction qui s'appelle elle-même. C'est donc une fonction qui appelle une fonction, cette fonction a juste le même nom : cela ne change rien au niveau des conventions d'appel ! C'est au programmeur de s'assurer que la fonction récursive termine, c'est un problème d'algorithmique pas de programmation assembleur.

Pour l'écriture de fonctions, vous suivrez consciencieusement les étapes de programmation d'une fonction en assembleur à savoir pour rappel (cf. l'énoncé de la semaine dernière aussi). Vous porterez une attention particulière au contenu des registres non persistants (par exemple `$4`) avant les appels de fonctions dans des fonctions et au besoin de sauvegarder la valeur des arguments dans certains cas (déterminez lesquels).

Exercice 1 – Fonction puissance linéaire

Question 1

On considère le code suivant :

```
int puissance(int x, int n) {
    int tmp;
    if (n == 0) {
        return 1;
    }
    else {
        tmp = puissance(x, n - 1);
        return x * tmp;
    }
}

void main() {
   int x = 3;
   int p = 2;

   printf("%d", puissance(x, p));
   printf("%d", puissance(2, 6));

   exit();
}
```

Donnez le code de la fonction puissance puis le code du programme principal correspondant au code C ci-dessus. N'oubliez pas qu'il faut allouer de l'espace mémoire sur la pile pour les variables locales, dans les fonctions mais aussi dans le programme principal (le `main`).

Question 2 – Représentation graphique de l'arbre d'appels

Donnez les appels récursifs engendrés par l'appel `puissance(2,6)`

Question 3 – Représentation graphique de la pile

Numérotez les instructions de votre code assembleur.
Représentez graphiquement l'évolution de la pile au cours de l'exécution du premier appel à la fonction `puissance`.
Vous indiquerez par @i les adresses de retour que vous mettrez sur la pile, avec i correspondant au numéro de l'instruction associée à cette adresse de retour.

Exercice 2 – Fonctions imbriquées

On souhaite écrire en assembleur MIPS32 le programme C qui calcule la valeur moyenne de tous les entiers (non négatifs) stockés dans un tableau de dimension quelconque. Par convention, et pour pouvoir réutiliser la fonction `sumtab()`, on suppose que le dernier élément du tableau possède une valeur négative. Ce dernier élément n'est pas pris en compte dans le calcul de la moyenne.

Le programme principal appelle la fonction `arimean()` qui a pour seul argument un pointeur sur un tableau d'entiers. Elle renvoie la valeur de la moyenne arithmétique des éléments du tableau (tronquée à la valeur entière inférieure). La fonction `arimean()` appelle la fonction `sizetab()` qui prend pour seul argument le pointeur sur le tableau, et renvoie le nombre d'éléments non négatifs contenus dans le tableau. Elle appelle ensuite la fonction `sumtab()`, qui calcule la somme de tous les entiers non négatifs contenus dans le tableau. Elle effectue la division entière et renvoie le quotient au programme appelant.

```
#include <stdio.h>

/* variables globales initialisées */
int tab[] = {23, 7, 12, 513, -1} ;

/* programme principal */
void main(void) {
    int x = arimean(tab);
    printf("%d", x);
    exit();
}

/* cette fonction renvoie la moyenne arithmétique des éléments d'un tableau */
int arimean(int t[]) {
    int n = sizetab(t);
    int x = sumtab(t);
    return (x / n);
}

/* cette fonction renvoie le nombre d'éléments d'un tableau */
int sizetab(int t[]) {
    int index = 0;
    while (t[index] >= 0) {
        index += 1;
    }
    return index;
}

/* cette fonction renvoie la somme des éléments d'un tableau */
int sumtab(int t[]) {
```

```
    int accu = 0;
    int index = 0;
    while (t[index] >= 0) {
        accu = accu + t[index];
        index += 1;
    }
    return accu;
}
```

Question 1
Donnez le code de la fonction `arimean`.

Question 2
Donnez le code des fonction `sumtab` et `sizetab`.

Question 3
Donnez le code du programme principal ainsi que les déclarations des variables globales du programme.

Exercice 3 – Élément de la suite de Fibonnaci

Question 1
Écrivez le code assembleur correspondant au code C ci-dessous. Ce code contient une fonction calculant le nombre de Fibonnaci à l'ordre n et un programme principal affichant le résultat pour une valeur stockée dans une variable globale.

```
int n = 4;

void main() {
    printf("%d", fib(n));
    exit();
}

int fib(int n) {
    if (n == 0 || n == 1) {
        return 1;
    }
    else {
        return fib(n - 1) + fib(n - 2);
    }
}
```

Question 2 – Représentation graphique de l'arbre d'appels
Représentez graphiquement l'arbre des appels engendrés par l'appel `fib(n)` avec `n` qui vaut 4.

Question 3 – Représentation graphique de la pile
Numérotez les instructions de votre code assembleur.
Représentez graphiquement l'évolution de la pile au cours de l'exécution. Vous indiquerez par @i les adresses de retour que vous mettrez sur la pile, avec i correspondant au numéro de l'instruction associée à cette adresse de retour.

TME8 : Fonctions récursives

Objectif(s)

- ✯ Maîtrise totale des conventions d'appel de fonction.
- ✯ Ecriture de fonctions récursives.

Exercice(s)

Exercice 1 – Factorielle récursive

Voici le code que vous devez implémenter :

```
int p = 5;

int fact(int n){
   if (n==0)
      return 1;
   else
      return n * fact(n-1);
}

void main(){
   printf("%d", fact(p));
   exit();
}
```

Exercice 2 – Puissance récursive rapide

Voici le code que vous devez implémenter :

```
int p = 3;
int m = 5;

int puissance(int x, int n){
   int res;

   if (n==0)
      return 1;
   if (n==1)
      return x;
   else {
      res = puissance(x, n/2);    /* division par 2 */
      if (n & 0x01 == 1)          /* n est impair */
         return x * res * res;
      else
```

```
        return res * res;
    }
}

void main(){
    printf("%d", puissance(p,m));
    exit();
}
```

En C, `n & m` correspond au `AND` bit à bit entre les entiers `n` et `m` représentés en binaire.

Exercice 3 – Exercices de TD

Implantez le code correspondant à l'exercice sur les fonctions imbriquées du TD (avec la fonction `arimean`). Si le temps le permet, implantez aussi la fonction calculant un élément de la suite de Fibonnaci.