# ESIR SYS 2012-2013
# Contrôle en Classe n.1

**Nom:** _____

**Prénom:** _____

**Durée: 1h**

**Notation: sur 20 points**

## Partie A (MCQ / QCM, 10 points) :

**Instructions:**

Entourez la lettre de la réponse que vous pensez être juste.

**Barème:**

+0.5 pour chaque réponse correcte

-0.5/m pour chaque réponse fausse (ou m+1 est le nombre de réponses possibles)

**Q1 : Code machine et code assembleur sont**

A : une et même chose.

B : Le code machine est utilisé pour générer le code assembleur.

C : Le code assembleur est utilisé pour générer le code machine.

**Q2 : Le code suivant est généralement rencontré au début des programmes TASM que nous avons vus:**

**MOV AX,@data**

**MOV DS,AX**

**Ce code sert à :**

A : À sauvegarder la valeur du registre AX and la variable @data, avant d'initialiser AX avec la valeur du registre DS.

B : À initialiser la position du segment de données.

C : À sauvegarder la valeur de @data dans la variable DS.

**Q3 : L'instruction MOV [20],[30] existe-t-elle ?**

A : Oui

B : Non

C : Cela dépend de la version du BIOS.

**Q4 : Considérez les 3 extraits de code suivants :**

***Code 1 :***

```
MOV AX,12345678H
```

***Code 2 :***

```
MOV AH,1234H
MOV AL,5678H
```

***Code 3 :***

```
MOV AL,1234H
MOV AH,5678H
```

A : Ces 3 extraits ont tous des effets différents.

B : Code 1 est équivalent à Code 2, mais pas à Code 3.

C : Code 1 est équivalent à Code 3, mais pas à Code 2.

D : Code 1 est à la fois équivalent à Code 2 et à Code 3.

**Q5 : Un programme assembleur 8086**

A : Peut utiliser une centaine de registres.

B : Ne peut utiliser qu'un petit nombre (moins de vingt) de registres.

C : Peut créer des registres de façon dynamique, selon ses besoins propres.

**Q6 : Considérez le code suivant :**

```
   .MODEL  SMALL
   .STACK
   .DATA
X DW  100
Y DW  200
Z DW  1 DUP(?)
   .CODE
Debut:
   MOV AX,@data ; allways there
   MOV DS,AX
   MOV AX,X
   CMP AX,Y
   JLE label1
   MOV Z,AX
   JMP fin
label1:
   MOV AX,Y
   MOV Z,AX
fin:
   END Debut
```

A : Ce code implémente : si x<=y alors z := y finsi

B : Ce code implémente : si x>y alors z := y finsi

C : Ce code implémente : si x<=y alors z :=x sinon z :=y finsi

D : Ce code implémente : si x>y alors z := x sinon z := y finsi

**Q7 : Soit le code suivant :**

```
.CODE
ecrire:
  MOV DX,OFFSET MSG ; affichage de MSG
  MOV AH,9
  INT 21H
  JMP retour
debut:
  MOV AX,@data ; allways there
  MOV DS,AX;
  JMP ecrire
  JMP ecrire
retour:
  END debut
```

A : Ce code permet d'appeler la procédure 'ecrire' deux fois.

B : Ce code ne permet pas d'appeler la procédure 'ecrire' deux fois.

**Q8 : Dans le 8086, la pile :**

A : Grandit des adresses les plus petites vers les adresses les plus grandes.

B : Grandit des adresses les plus grandes vers les adresses les plus petites.

C : Soit l'un, soit l'autre, selon les instructions assembleur utilisées.

**Q9 : Pour appeler la procédure P avec les paramètres X et Y il est possible d'écrire en assembleur :**

**CALL P(X,Y)**

A : Oui, toujours

B : Non, jamais

C : Oui dans certains cas

**Q10 : Combien de segments le 8086 possède-t-il ?**

A : 2

B : 3

C : 4

**Q11 : L'instruction MOV AX,[BP] est équivalente à**

A : MOV AX,ES:[BP]

B : MOV AX,DS:[BP]

C : MOV AX,SS:[BP]

**Q12 : Considérez le code suivant**

**MOV AX,1**

**MOV BX,2**

**PUSH AX**

**PUSH BX**

**MOV BX,3**

**MOV AX,4**

**POP AX**

**POP BX**

**À la suite de ce code**

A : AX et BX contiennent 1 et 2 respectivement

B : AX et BX contiennent 3 et 4 respectivement

C : AX et BX contiennent 2 et 1 respectivement

D : AX et BX contiennent 4 et 3 respectivement

**Q13 : L'instruction assembleur**

**MOV AX, [BX+SI]**

A : assigne au registre AX la somme des valeurs contenues dans les registres BX et SI.

B : assigne au registre AX l'adresse de la variable BX après y avoir ajouté le contenu de SI.

C : assigne au registre AX le contenu de l'adresse obtenue en ajoutant le contenu du registre BX et celui du registre SI

D : assigne à BX et SI le contenue de AX

**Q14 : Considérez l'extrait assembleur suivant.**

**MOV AX,5h**

**PUSH AX**

**CALL P1**

**L'exécution de ce code sur le 8086 déplace le somment de la pile de :**

A : 1 octet

B : 2 octets

C : 3 octets

D : 4 octets

**Q15 : L'extrait assembleur suivant représente le corps d'une procédure.**

```
PUSH BP
MOV BP,SP
SUB SP,4

MOV AX,[BP+4]
MOV [BP-2],AX

...

ADD SP,4
POP BP
RET 2
```

**En supposant les entiers codés sur 2 octets, et les caractères sur 1. Cette procédure peut avoir comme signature :**

A : procédure P fixe(ent x1)

B : procédure P fixe(car b)

C : procédure P fixe(ent x1) mod(end x2)

D : procédure P fixe(ent x1, car x2)

**Q16 : Après l'exécution de l'instruction CALL P, le sommet de la pile contient**

A : l'adresse du début de la procédure P

B : l'adresse de la fin de la procédure P

C : l'adresse de l'instruction CALL P

D : l'adresse de l'instruction qui suit CALL P

**Q17 : Dans le code assembleur suivant, en supposant que 1234h et 1236h sont les positions en mémoire des variables X et Y, on considère le code suivant d'appel de la procédure P.**

```
MOV AX,[1234h]

PUSH AX

MOV AX,[1236h]

PUSH AX

CALL P
```

Dans cet appel de procédure, les paramètres d'appel sont passés à la procédure P :

A : par valeur

B : par adresse

C : par valeur et par adresse

**Q18 : Pourquoi le registre BP est-il généralement empilé en début de procédure**

A : pour initialiser les variables locales

B : pour transmettre la valeur de BP à d'éventuels appels emboîtés contenu dans la procédure

C : pour permettre des appels récursifs

**Q19 : Le 8086 permet d'adresser une mémoire de :**

A : $2^{16}$ = 65536 octets (64 KB)

B : $2^{20}$ = 1048576 octets (1 MB)

C : $2^{32}$ = 4294967296 octets (4 GB)

**Q20 : Les paramètres passés à une procédure sont empilés :**

A : avant l'adresse de retour

B : après l'adresse de retour

C : parfois avant, parfois après selon leur type (par valeur ou par adresse)

## Partie B : Problèmes

### Question 1 [5 points]

Rappel : un entier est représenté en mémoire sur deux octets et un caractère sur un octet. Soit la définition suivante en pseudo code:

```
type étudiant =
   struct // déclaration du type étudiant
    car[10] nom;
    ent age;
    ent note;
   fin_struct;

étudiant ET; // déclaration de la variable ET, de type étudiant
```

### Question 1.a

Combien d'octets sont nécessaires au stockage d'une variable de type étudiant ? Quels sont les déplacements relatifs ( ) nécessaires pour accéder à chacun des attributs d'une variable de type étudiant ?

### Question 1.b

Soit l'entête de la procédure suivante (avec la notation pseudo-code du cours) :

```
procédure inscription = mod (étudiant f1) fixe (étudiant f2)
début
  (* personne ne sait ce que doit faire cette procédure *)
fin
```

Donner le schéma de pile de cette procédure ainsi que sa traduction en langage d'assemblage (bien sûr, la partie code ne pourra contenir que le prologue et l'épilogue de cette procédure).

## Question 2 [5 points]

Soit la procédure suivante (écrit en C ou C++) :

```
int * fonc () {
    int  i, T[10];
    for (i=0; i <10 ;i++) T[i] = i;
    return (T);
}
```

### Question 2.a

Donner le schéma de pile et le code assembleur de cette procédure.

### Question 2.b

Cette procédure contient une erreur de programmation : laquelle et pourquoi ?