Nom : Groupe :
Correction

Note : **60** /60

Contrôle Long n°1

Calculatrice et documents interdits - Durée 3 heures - Répondre sur la feuille
Bien lire entièrement les questions qui ne sont pas toujours indépendantes ; de plus de nombreuses indications vous aident.

/5

I. Quantité d'information

Un candidat a oublié de joindre à son dossier d'inscription un document très important. Au téléphone, la secrétaire lui suggère de l'envoyer de toute urgence par fax ou par mail. Ne disposant pas de fax, le candidat se décide à scanner son document et à l'envoyer par mail. Après un premier essai, sons fournisseur d'accès à Internet lui signale (par mail) qu'il est limité dans la taille des documents qu'il envoie, à 600 ko. Le document, un diplôme, est une feuille A4 (200 mm x 300 mm utiles). Il n'est pas nécessaire de le scanner en couleur (il n'y en a pas), mais comme il comporte un logo, 256 niveaux de gris sont préférables. On supposera toujours égales les résolutions optique verticale et horizontale. On donnera les résolutions en points par millimètre.

1. Calculer la résolution optique maximum que l'on peut adopter pour envoyer l'image scannée.

/3

Nombre de points pour une résolution R : (200×R) × (300×R) = 60000×R²
Nombre de bits pour coder un des 256 niveaux de gris pour un point : 8 bits = 1 octet
Total = 60000×R² octets.
Comme on est limité à 600 ko, on a au maximum R² ≈ 10 soit R ≈ 3 pts / mm

2. Qu'en pensez vous ? Proposez une solution pour l'envoyer par mail.

/2

C'est un peu grossier comme définition ; les petits caractères ne passeront pas bien.
On peut faire de la compression ou utiliser moins de niveaux de gris (voire N&B).

/6

II. Architecture d'un processeur

1. Représentez la structure interne (registres et bus) d'un processeur.

Le processeur est un processeur 8 bits d'espace adressable 64ko, évidement basé sur l'architecture Von Neumann. Ce processeur est très simple : il n'a qu'un seul accumulateur. Malgré l'utilisation d'adressages implicite, immédiat et direct, son jeu d'instruction est réduit ce qui permet de limiter le code de ses opérations à un seul octet (et au plus une opérande).

/4

Il faut :
un bus de données (data-bus) de 8 bits entre MP et CPU,
et un bus d'adresses (@-bus) de 8 bits entre MP et CPU,

un registre RI connecté au data-bus,
et un registre IP connecté au @-bus,

un registre ACC connecté au data-bus,
et registre RTUAL connecté au data-bus,
et une UAL connecté à l'ACC et à un RTUAL,

un registre RTA connecté au @-bus,

2. Quel(s) registre(s) doit-on ajouter à notre processeur pour lui faire gérer une pile ?

/2

un (ou plusieurs) registre(s) repérant l'@ du haut de la pile (SS:SP dans les x86),
un (ou plusieurs) registre(s) repérant l'@ du bas de la pile (SS:0000 dans les x86),

/16

III. Correction d'un exécutable

On voudrait utiliser un programme, mais lorsqu'on lance son exécution, l'ordinateur "se fige" (le processeur semble faire quelque chose, mais ne s'arrête pas). On voudrait donc corriger ce programme qui doit être erroné. Malheureusement, on ne dispose pas du code source. On se sert alors de DEBUG pour désassembler le programme (pour décoder le langage machine binaire en langage symbolique). Voici ce que donne DEBUG (extraits) :

```
-u 0 30
1AC9:0000 B80000          MOV     AX,0000
1AC9:0003 B90000          MOV     CX,0000
1AC9:0006 B8C81A          MOV     AX,1AC8
1AC9:0009 8ED8            MOV     DS,AX
1AC9:000B 803E030008      CMP     Byte Ptr [0003],08
1AC9:0010 7D17            JGE     0029
1AC9:0012 A00000          MOV     AL,[0000]
1AC9:0015 22060100        AND     AL,[0001]
1AC9:0019 7404            JZ      001F
1AC9:001B FE060200        INC     Byte Ptr [0002]
1AC9:001F D0260100        SHL     Byte Ptr [0001],1
1AC9:0023 FE060300        INC     Byte Ptr [0004]
1AC9:0027 EBE2            JMP     000B
1AC9:0029 B44C            MOV     AH,4C
1AC9:002B CD21            INT     21
1AC9:002D B80000          MOV     AX,0000
1AC9:0030 B90000          MOV     CX,0000
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=1AC7  ES=1ACA  SS=1AC8  CS=1AC9  IP=0006   NV UP EI PL NZ NA PO NC
1AC9:0006 B8C81A          MOV     AX,1AC8
-
```

1. Dites à quelle adresse (`segment:offset`) commence le programme (justifiez).

/2 **(1AC9:)0006**
C'est là qu'est initialisé IP par le chargeur du SE.
(en plus ça commence toujours par l'initialisation de DS)

- Dites à quelle adresse se termine le programme (justifiez).

/1 **(1AC9:)002B (ou 2C ou 2D selon comment on interprète la question)**
La dernière instruction INT 21 (sur 2C-2D) avec AH à AC est une instruction de retour au DOS.

2. Donnez la valeur du segment de données (justifiez).

/1 **1AC8 parce que c'est à cette valeur que l'on l'initialise avec les premières instructions.**

- Donnez les adresses du premier octet et du dernier octet du segment de données,
sous la forme `segment:offset`, et sous la forme réelle en hexadécimal (20 bits).

adresse premier octet :

/1 **>1AC8:0000** **>1AC8 0 + 0 0000 = 1AC80**

adresse du dernier octet :

/1 **>1AC8:FFFF** **>1AC8 0 + 0 FFFF = 2AC7F**
(= 1AC80 + 10000 - 00001 = 2AC80 - 1)

- Quelle est la taille du segment de données ?

/1 **adresses de 0000 à FFFF, soit 2^{16} octets = $2^6 \times 2^{10}$ donc 64 ko**

/7 3. Complétez le tableau suivant au fil de l'exécution du programme.

On a représenté ici uniquement les registres utilisés et la partie de la mémoire utilisée pour les données.
Notez uniquement les valeurs pour les registres qui changent.

AX : **0000** CX : **0000** DS : **1AC7** Mémoire : **0A 01 00 00 00 00 00 00**
CS : **1AC9** IP : **0006**
Instruction : **MOV AX,1AC8**

AX : **1AC8** CX : DS : Mémoire :
CS : IP : **0009**
Instruction : **MOV DS,AX**

AX : CX : DS : **1AC8** Mémoire :
CS : IP : **000B**
Instruction : **CMP Byte Ptr [0003],08**

AX : CX : DS : Mémoire :
CS : IP : **0010**
Instruction : **JGE 0029**

AX : CX : DS : Mémoire :
CS : IP : **0012**
Instruction : **MOV AL,[0000]**

AX : **1A0A** CX : DS : Mémoire :
CS : IP : **0015**
Instruction : **AND AL,[0001]**

AX : **1A00** CX : DS : Mémoire :
CS : IP : **0019**
Instruction : **JZ 001F**

AX : CX : DS : Mémoire :
CS : IP : **001F**
Instruction : **SHL Byte Ptr [0001],1**

AX : CX : DS : Mémoire : **2**
CS : IP : **0023**
Instruction : **INC Byte Ptr [0004]**

AX : CX : DS : Mémoire : **1**
CS : IP : **0027**
Instruction : **JMP 001B**

AX : CX : DS : Mémoire :
CS : IP : **001B**
Instruction : **CMP Byte Ptr [0003],08**

AX : CX : DS : Mémoire :
CS : IP : **0010**
Instruction : **JGE 0029**

4. A votre avis, où est située l'erreur ? Proposez une correction.

/2 **On fait un test de fin de boucle sur [0003] sans jamais y toucher !**
L'erreur doit être sur INC Byte Ptr [0004]
Il devait s'agir de INC Byte Ptr [0003]

IV. UAL

/7

A. Additionneur

1. Complétez les tables de vérité des opérateurs et, ou, ou exclusif.

/3

a	b	AND a,b	OR a,b	XOR a,b
0	0	**0**	**0**	**0**
0	1	**0**	**1**	**1**
1	0	**0**	**1**	**1**
1	1	**1**	**1**	**0**

2. Complétez la table de vérité d'un demi-additionneur.

Il y a deux sorties : somme S et retenue C.

/2

a	b	C	S
0	0	**0**	**0**
0	1	**0**	**1**
1	0	**0**	**1**
1	1	**1**	**0**

- En déduire le schéma du demi additionneur.

On rappelle les symboles des portes et [&], ou [≥1], ou exclusif [=1]

/2

Si l'on compare les tables du 2°/ et du 1°/, on voit aisément que :
C = AND a,b
S = XOR a,b

/14

B. Multiplieur

On essaie de faire un programme pour faire une multiplication d'entiers. Nous ne cherchons pas à faire un programme en langage assembleur mais seulement en langage symbolique (pas de directives de déclaration de segment…). Vous n'avez pas à concevoir l'algorithme, il est donné ci-après dans une version simplifiée (donc limitée) que vous n'êtes *pas* obligé de comprendre pour faire l'exercice. Les questions vous guideront.

En binaire, un nombre X s'écrit :

$X = x_7x_6\ldots x_1x_0 = \Sigma_{i\in[0,7]}\, x_i.2^i = x_7.2^7 + x_6.2^6 + \ldots + x_1.2^1 + x_0.2^0$

Le produit de deux entiers binaires de n bits X et Y s'écrit alors :

$X.Y = X.\{y_7.2^7 + y_6.2^6 + \ldots + y_1.2^1 + y_0.2^0\} = y_7.X.2^7 + y_6.X.2^6 + \ldots + y_1.X.2^1 + y_0.X.2^0 = \Sigma_{i\in[0,7]}\, y_i.\{X.2^i\}$

En soulignant qu'un bit y_i vaut soit 0 soit 1, on peut écrire l'algorithme suivant :

initialiser le résultat à 0,
pour tous les bit y_i de Y :
    si le bit y_i de Y vaut 1 alors
        ajouter $(X \times 2^i)$ au résultat.

1. On code maintenant la boucle "pour tous les bits y_i de Y".

Ne faites *que* la boucle laissez un espace pour le reste. Pour simplifier vous pouvez utiliser ici des étiquettes.

Donnez les instructions nécessaires à la réalisation de la boucle "pour i allant de 0 à 7".

Vous utiliserez CX pour stocker votre variable de compteur.

/2

```
        MOV CX,0
boucle:CMP CX,8
        JGE sortie



        INC CX
        JMP boucle
sortie:
```

2. On code maintenant le test "si". On suppose que Y est stocké dans l'accumulateur AL.

Donnez deux méthodes permettant de tester la valeur du bit de poids 0 dans AL.

Les deux méthodes sont chacune composées d'une instruction logique **et** d'un test (différents).

- la première masque tous les bits de AL sauf celui de poids 0,

/2
```
AND AL,0000 0001b
JZ plus loin si rien
```

- la seconde met le bit de poids 0 dans CF.

/2
```
SHL AL,1
JNC plus loin si rien
```

- A l'itération suivante, on voudra tester le bit suivant de Y.

Laquelle de ces deux méthodes est préférable ? Justifiez.

/1 **La seconde, car pour tester le bit suivant il suffit de la refaire sans rien toucher.**

3. On mémorise le terme $T_i = (X \times 2^i)$ à la $i^{ème}$ itération.

- Quelle est la relation entre T_i et T_{i+1} ?

/1 $T_{i+1} = 2 \times T_i$

- T_i est stocké dans l'accumulateur AH.

Donnez une instruction permettant de mettre la valeur de T_{i+1} dans AH.

/1
```
SHL AH,1
```

4. Donnez les instructions d'initialisation nécessaires à notre programme.

Les valeurs X et Y à multiplier sont rangées aux adresses @0000H et @0001H.

/2
```
MOV AL,[0001]
MOV AH,[0000]
```

- Donnez l'instruction qu'il manque pour faire l'addition des termes T_i.

On ne demande *qu'une seule* instruction. Le résultat P est rangé en @0002H.

/1
```
ADD [0002],AH
```

5. On revient sur le codage de la boucle. On remarque que les bits à 0 de Y n'apportent rien au résultat de la multiplication. La boucle serait plus efficace si on l'on se contentait de ne traiter que les bits à 1 de Y. Une manière de l'écrire est "tant que l'on a pas traité tous les bits y_i à 1 de Y".

Le problème réside dans la manière de déterminer si on a traité tous les bits à 1. Voici la technique : vérifiez qu'avec la seconde méthode du 2, si l'on a bien choisi l'instruction logique, AL contient 0 lorsqu'on a mis le dernier bit à 1 de AL dans CF. En déduire une méthode plus rapide et économique (qui permet de se passer de l'utilisation de CX) pour faire la boucle.

Donnez les instructions pour le codage de la boucle améliorée.

Ne faites *que* la boucle laissez un espace pour le reste. Pour simplifier vous pouvez utiliser ici des étiquettes.

/2
```
boucle:CMP AL,0
       JZ sortie

       JMP boucle
sortie
```

/12

V. Assembleur

1. Donnez l'ensemble des directives et des instructions nécessaires à l'écriture d'un programme .

On ne demande donc ici que le squelette d'un programme : on ne détaillera pas les données du segment de données (marquez juste le commentaire " `;ici declaration des donnees` "), pas plus que les instructions du programme (marquez juste " `;ici instructions du programme` ").

/5

```
ASSUME DS:data, CS:code
data SEGMENT
;ici declaration des donnees
data ENDS
code SEGMENT
debut: MOV AX,data
       MOV DS,AX
       ;ici instructions du programme
       MOV AH,4C
       INT 21
data ENDS
       END debut
```

2. Donnez les directives pour la déclaration de `CHAIN` initialisée à "`Ho non !$`", d'un tableau `TAB` de 1234 caractères (non prédéfinis), de `N` initialisé à la taille de `TAB`, de `I` qui ira de 0 à `N`.

Les directives sont les lignes que vous placeriez à la place du commentaire précédent dans le segment de données.

/3

```
CHAIN DB "Ho non !$"
TAB   DB 1234 dup(?)
N     DW 1234
I     DW 0
```

3. Ecrire un programme qui détermine la taille de la chaîne `CHAIN`, et range cette taille dans `N`.

La chaîne `CHAIN` se termine par le caractère '`$`'. On suppose bien sûr ici que l'on a fait les déclarations des questions 1 et 2, et que l'on n'a donc à écrire que les lignes que vous placeriez à la place du commentaire précédent dans le segment de code.

/4

```
       MOV BX, offset CHAINE
       MOV CX,0
suite: MOV AL,[BX]
       INC BX
       CMP AL,'$'
       JNE suite
       MOV N,CX
```

Extrait du jeu d'instruction des 80x86

On rappelle ici les symboles couramment utilisés pour les instructions, et une brève description de la fonction. On précise aussi les types d'adressage autorisés pour chacune d'elle :

* signifie « un registre » (adressage implicite) (les registres de segment ne sont pas concernés désignés *S*).
# signifie « une valeur » (adressage immédiat),
@ signifie « une adresse mémoire » (adressage direct ou indirect).

Transferts
,[,#,@,*S*]
@,[*,#,*S*]
S,[*,@]

MOV copie une valeur d'un endroit à un autre

Opérations arithmétiques
,[,#,@]
@,[*,#]

ADD addition
SUB soustraction
CMP comparaison
AND et logique
OR ou logique
XOR ou exclusif

Opérations logiques
*
@

SHL décalage à gauche
SHR décalage à droite
ROL rotation à gauche
ROR rotation à droite
RCL rotation à gauche via CF
RCR rotation à droite via CF

Comparaison
Cf. opérations arithmétiques
Rappel :
ZF zéro
CF retenue
SF signe négatif (en code complément à 2)
OF débordement (en code complément à 2)
PF parité paire

Sauts *à l'adresse relative*
#

JMP

Branchements conditionnels
Test et saut à l'adresse relative
*

JE (JZ) test == (ou ==0)
JNE (JNZ) test !=
JL/JLE test </<= (nombres signés)
JG/JGE test >/>= (nombres signés)
JA/JAE test >/>= (nombres non signés)
JB/JBE test </<= (nombres non signés)
JC/JNC test si CF / non CF
JO/JNO test si OF / non OF
JS/JNS test si SF / non SF
JP/JNP test si PF / non PF