Université Internationale de Casablanca

LAUREATE INTERNATIONAL UNIVERSITIES

Filières GI & GE Première Année Tronc Commun
Examen microprocesseurs
Année scolaire 2017/2018
Durée : 2h

Exercice 1

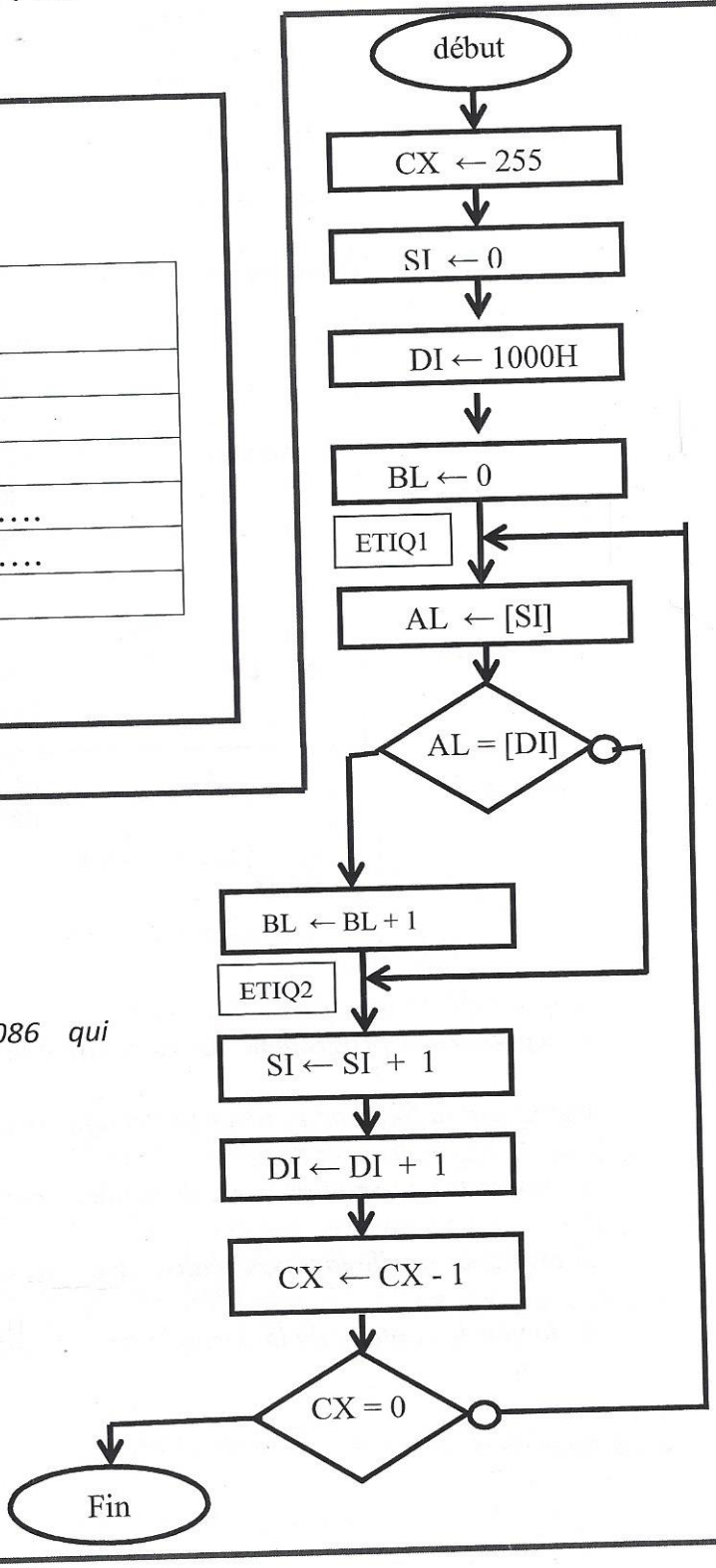
Compléter le tableau suivant :

Instruction	Résultat
mov ax,2050h	ax = ….
inc ax	ax = …..
shr ah,1	ah = ……
mov [0100h],30h	[0100h] = ….
mov [0101h],70h	[0101h]= ….
mov bx,[0100h]	bx= …

Exercice 2

On propose l'organigramme ci-contre :

Donner le code en assembleur 8086 qui correspond à cet organigramme.



Exercice 3

On propose le programme assembleur suivant :

```
data            SEGMENT
     table           DB 20,30,0,15,0,0,8,24,26,0
     result          DB 0
data                 ENDS

code            SEGMENT

debut:               MOV        AX,data
                     MOV        DS,AX

                     MOV        CX,10
                MOV         SI,OFFSET table

encore:         MOV         AL,[SI]
                CMP         AL,00
                JNE         suite
                INC         result

suite:          INC         SI
                LOOP        encore

                MOV         AH,4CH
                INT         21H

code                 ENDS
                     END   debut
```

1/ reprendre ce programme sur votre copie ligne par ligne en lui rajoutant des commentaires qui expliquent la fonction réalisée par chaque instruction

2/ proposer un organigramme de la partie encadrée

3/ expliquer quelle fonction réalise ce programme

4/ donner le contenu de la variable 'result' après exécution de ce programme

Exercice 4

Donner un organigramme et expliquer que fait ce programme

Calculer la durée d'exécution de ce code sachant que le microprocesseur fonctionne à la fréquence de 10 Mhz

```
                ASSUME      CS : Code, DS : Data
Data            SEGMENT
Tempo           DW          0100H
Data            ENDS

Code            SEGMENT                          Nombre de µc
Debut :         MOV         AX , Data              10
                MOV         DS , AX                 2

                MOV         AX , Tempo             10
Boucle :        ADD         AX , -1                 4
                JNE         Boucle                 16

                MOV         AH , 4CH                4
                INT         21H                    52

Code            ENDS
                END         Debut
```

Exercice 5

On dispose d'un fichier de chaîne de caractères stocké en mémoire que nous représenterons dans notre programme comme étant une variable tableau identifiée par le nom FICHIER. La fin de ce fichier qui ne dépasse pas 65536 octets est indiquée par le caractère '$'.

1/ Ecrire le code en assembleur 8086 qui permet de fournir dans le registre DX la longueur de ce fichier

2/ Dans le but de réaliser la compression de ce fichier par l'algorithme de Huffman, on doit procéder à une étude statistique qui fournit le nombre d'occurrence de chaque caractère présent dans ce fichier. Pour simplifier le traitement, on ne considère que les 10 premières lettres majuscules (code ASCII compris entre 65 et 74).

Ecrire un programme en assembleur qui stocke dans un tableau de 10 cases le nombre de chacun des 10 caractères présents dans le fichier : la case 0 contiendra le nombre de lettre 'A', la case 1 contiendra le nombre de lettre 'B' et ainsi de suite.

Liste des instructions du CPU Intel 8086

Nom (mnémonique)		Action
AAA	ASCII Adjust for Addition	opération complexe, voir littérature
AAD	ASCII Adjust for Division	opération complexe, voir littérature
AAM	ASCII Adjust for Multiplic.	opération complexe, voir littérature
AAS	ASCII Adjust for Subtr..	opération complexe, voir littérature
ADC	Add with Carry	*op1* + *op2* (+1) → *op1* (+1 si *CF* = 1)
ADD	Add	*op1* + *op2* → *op1*
AND		*op1 AND op2* → *op1*
CALL		appel de procédure
CBW	Convert Byte to Word	AL → AX (extension bit de signe)
CLC	Clear Carry flag	0 → *CF* ('Carry' bit, 'flag-register')
CLD	Clear Direction flag	0 → *DF* ('Direction' bit, 'flag-reg.')
CLI	Clear Interrupt flag	0 → *IF* ('Interrupt' bit, 'flag-register')
CMC	Complement Carry flag	*NOT CF* → *CF* ('Carry' bit, 'flag-reg.')
CMP	Compare	*op1* – *op2*
CMPB	Compare Byte	*zone-mémoire* – *zone-mémoire* (byte)
CMPW	Compare Word	*zone-mémoire* – *zone-mémoire* (mot)
CWD	Convert Word to Double	*AX* → *DX:AX* (extension bit de signe)
DAA	Decimal Adjust for Addition	opération complexe, voir littérature
DAS	Decimal Adjust for Subtr.	opération complexe, voir littérature
DEC	Decrement	*op1* – 1 → *op1*
DIV	Divide	*DX:AX* / *op1* → *AX*, reste → *DX* *AX* / *op1* → *AL*, reste → *AH*
ESC	Escape	*op1* → bus (pas d'autre action CPU)
HLT	Halt	arrêter le CPU
IDIV	Integer Divide	*DX:AX* / *op1* → *AX*, reste → *DX* *AX* / *op1* → *AL*, reste → *AH* (op. abs.)
IMUL	Integer Multiply	*op1* * *AX* → *DX:AX* *op1* * *AL* → *AX* (op. abs.)
IN	Input	entrée/sortie → *AL*
INC	Increment	*op1* + 1 → *op1*
INT	Interrupt	interruption par vecteur numéro *op1*
INTO	Interrupt Overflow	interr. par vecteur numéro 4 (si *OF* = 1)

Nom (mnémonique)		Action	
INW	Input Word	entrée/sortie → *AX*	
IRET	Interrupt Return	retour du traitement d'interruption	
JA	Jump if Above = JNBE	branchement cond. (*op2* > *op1*)	opérandes absolues
JAE	Jump if Above or Eq. = JNB	branchement cond. (*op2* ≥ *op1*)	
JB	Jump if Below = JNAE	branchement cond. (*op2* < *op1*)	
JBE	Jump if Below or Eq. = JNA	branchement cond. (*op2* ≤ *op1*)	
JCXZ	Jump if CX = 0	branchement si *CX* = 0	
JE	Jump if Equal = JZ	branchement cond. (*op2* = *op1*)	
JG	Jump if Greater = JNLE	branchement cond. (*op2* > *op1*)	
JGE	Jump if Greater or Eq. = JNL	branchement cond. (*op2* ≥ *op1*)	
JL	Jump if Less = JNGE	branchement cond. (*op2* < *op1*)	
JLE	Jump if Less or Equal = JNG	branchement cond. (*op2* ≤ *op1*)	
JMP	Jump	branchement inconditionnel	
JNA	Jump if Not Above = JBE	branchement cond. (*op2* ≤ *op1*)	opérandes absolues
JNAE	Jump if Not Above or Eq. = JB	branchement cond. (*op2* < *op1*)	
JNB	Jump if Not Below = JAE	branchement cond. (*op2* ≥ *op1*)	
JNBE	Jump if Not Below or Eq. = JA	branchement cond. (*op2* > *op1*)	
JNE	Jump if Not Equal = JNZ	branchement cond. (*op2* ≠ *op1*)	
JNG	Jump if Not Greater = JLE	branchement cond. (*op2* ≤ *op1*)	
JNGE	Jump if Not Greater or Eq. = JL	branchement cond. (*op2* < *op1*)	
JNL	Jump if Not Less = JGE	branchement cond. (*op2* ≥ *op1*)	
JNLE	Jump if Not Less or Eq. = JG	branchement cond. (*op2* > *op1*)	
JNO	Jump if Not Overflow	branchement cond. (si pas 'overflow')	
JNP	Jump if Not Parity = JPO	branchement cond. (si parité impaire)	
JNS	Jump if Not Sign	branchement cond. (si valeur positive)	
JNZ	Jump if Not Zero = JNE	branchement cond. (si résultat¼ ¼≠ 0)	
JO	Jump if Overflow	branchement cond. (si 'overflow')	
JP	Jump if Parity = JPE	branchement cond. (si parité paire)	

Nom (mnémonique)		Action	
JPE	Jump if Parity Even = JP	branchement cond. (si parité paire)	
JPO	Jump if Parity Odd = JNP	branchement cond. (si parité impaire)	
JS	Jump if Sign	branchement cond. (si valeur négative)	
JZ	Jump if Zero = JE	branchement cond. (si résultat¼ = 0)	
LAHF	Load AH with Flags	bits arithmétiques du 'flag-reg.' → AH	
LDS	Load pointer to DS	adresse de *op2* → *DS:op1*	
LEA	Load Effective Addr.	adresse de *op2* → *op1*	
LES	Load pointer to ES	adresse de *op2* → *ES:op1*	
LOCK		réservation du bus pour > 1 cycle	
LODB	Load Byte	*zone-mémoire* → *AL*	
LODW	Load Word	*zone-mémoire* → *AX*	
LOOP		branchement si *CX* = 0	1ère action: *CX* – 1 → *CX*
LOOPE	Loop while Equal = LOOPZ	branchement si *CX* = 0 et *ZF* = 1	
LOOPNE	Loop while Not Eq. = LOOPNZ	branchement si *CX* = 0 et *ZF* = 0	
LOOPNZ	Loop while Not Zero = LOOPNE	branchement si *CX* = 0 et *ZF* = 0	
LOOPZ	Loop while Zero = LOOPE	branchement si *CX* = 0 et *ZF* = 1	
MOV	Move	*op2* → *op1*	
MOVB	Move Byte	*zone-mémoire* → *zone-mémoire*	
MOVW	Move Word	*zone-mémoire* → *zone-mémoire*	
MUL	Multiply	*op1* * *AX* → *DX:AX* *op1* * *AL* → *AX*	
NEG	Negate	0 – *op1* → *op1*	
NOT		*NOT op1* → *op1*	
OR		*op1 OR op2* → *op1*	
OUT	Output	*AL* → entrée/sortie	
OUTW	Output Word	*AX* → entrée/sortie	
POP		des-empiler → *op1*	
POPF	Pop Flags	des-empiler → 'flag-register'	
PUSH		*op1* → empiler	
PUSHF	Push Flags	'flag-register' → empiler	
RCL	Rotate Left with Carry	décalage circulaire gauche par *CF*	
RCR	Rotate Right with Carry	décalage circulaire droite par *CF*	
REP	Repeat	pré-fixe: répétition sur zone-mémoire	
RET	Return	retour d'une procédure	

Nom (mnémonique)		Action
ROL	Rotate Left	décalage circulaire gauche
ROR	Rotate Right	décalage circulaire droite
SAHF	Store AH to Flags	*AH* → bits arithm. du 'flag-register'
SAL	Shift Arithm. Left = SHL	décalage à gauche (0-remplissage)
SAR	Shift Arithmetic Right	décalage à droite, extension du signe
SBB	Subtract with Borrow	*op1* – *op2* (–1) → *op1* (–1 si *CF* = 1)
SCAB	Scan Byte	*AL* – *zone-mémoire*
SCAW	Scan Word	*AX* – *zone-mémoire*
SHL	Shift Logical Left = SAL	décalage à gauche (0-remplissage)
SHR	Shift Logical Right	décalage à droite (0-remplissage)
STC	Set Carry flag	1 → *CF* ('Carry' bit, 'flag-register')
STD	Set Direction flag	1 → *DF* ('Direction' bit 'flag-reg.')
STI	Set Interrupt flag	1 → *IF* ('Interrupt' bit, 'flag-register')
STOB	Store Byte	*AL* → *zone-mémoire*
STOW	Store Word	*AX* → *zone-mémoire*
SUB	Subtract	*op1* – *op2* → *op1*
TEST		*op1 AND op2*
WAIT		le CPU entre dans une boucle d'attente
XCHG	Exchange	*op1* ⇔ *op2*

Nom (mnémonique)		Action
XLAT	Translate	conversion de code par table de corresp.
XOR	Exclusive Or	*op1 XOR op2*

ZF	SF	CF	OF	PF	opération	condition de branchement (à la suite de l'exécution de l'instruction "CMP A,B")	
1					JE, JZ	A=B	
0					JNE, JNZ	A≠B	
	1				JS		
	0				JNS		
		1			JB, JNAE	A<B	comparaison de valeurs sans signe
		0			JNB, JAE	A≥B	
	CF∨ZF = 1				JBE, JNA	A≤B	
	CF∨ZF = 0				JNBE, JA	A>B	
	SF∀OF = 1				JL, JNGE	A<B	comparaison de valeurs avec signe
	SF∀OF = 0				JNL, JGE	A≥B	
	(SF∀OF)∨ZF = 1				JLE, JNG	A≤B	
	(SF∀OF)∨ZF = 0				JNLE, JG	A>B	
			1		JO	dépassement arithmétique	
			0		JNO	pas de dépassement arithmétique	
				1	JP, JPE	parité paire ("even")	
				0	JNP, JPO	parité impaire ("odd")	

∨ = "ou" inclusif
∀ = "ou" exclusif