

Organisation des ordinateurs
Examen de juin 2022
Énoncés et solutions

Énoncés

1. Un dispositif de télécommunication génère des signaux radio en émettant des ondes sinusoïdales de six fréquences différentes, s'échelonnant de 433,075 MHz à 434,775 MHz par pas de 0,34 MHz.
Parmi les six fréquences possibles, les trois premières présentent la probabilité p d'être émises, et les trois dernières la probabilité $2p$.
 - (a) Quelle est la quantité d'information contenue dans un signal dont la fréquence figure parmi les trois premières ? parmi les trois dernières ?
 - (b) On mémorise les signaux reçus dans une mémoire Flash d'une capacité de 256 MB. Si 10^6 signaux radio sont reçus et décodés correctement chaque seconde, après combien de temps en moyenne cette mémoire sera-t-elle entièrement remplie ?
2.
 - (a) Quelle est la plus petite représentation (en terme de nombre de bits) du nombre -1 par la méthode du complément à deux ?
 - (b) Effectuer l'addition $-12,875 + 6,375$ en représentant les nombres en virgule fixe par complément à deux, avec 5 chiffres avant et 3 chiffres après la virgule.
 - (c) Si l'on applique l'algorithme d'addition de nombres entiers non signés à des nombres représentés par complément à un, qu'obtient-on comme résultat dans le cas où les deux opérands et le résultat de l'opération possèdent tous les trois un bit de poids fort égal à 1 ? (Justifier votre réponse à l'aide d'un développement mathématique.)
 - (d) Construire la représentation IEEE754 en simple précision du plus grand nombre négatif distinct de 0, et en donner une écriture hexadécimale.
3.
 - (a) Pourquoi les processeurs implémentent-ils des mécanismes de gestion d'une pile ? À quoi cela sert-il ?
 - (b) L'architecture x86-84 est petit-boutiste. Qu'est-ce que cela signifie ? (Illustrer votre réponse à l'aide d'un exemple concret.)
 - (c) En quoi le langage d'assemblage diffère-t-il du code machine ?

- (d) Expliquer comment se déroulera l'exécution du fragment de code assembleur x86-64 suivant. Combien d'itérations de la boucle seront-elles effectuées ?

```
XOR RCX, RCX
boucle: LOOP boucle
```

4. On souhaite créer un programme capable de générer une table des cubes de tous les nombres entiers entre 1 et 100. Cette table doit être affichée sur la sortie standard, dans le format suivant :

```
1: 1
2: 8
3: 27
4: 64
5: 125
:
99: 970299
100: 1000000
```

- (a) Écrire, en pseudocode ou en langage C (au choix), un algorithme permettant de résoudre ce problème.
- (b) Traduire cet algorithme en un programme assembleur x86-64 complet, en veillant à respecter la convention d'appel de fonctions des systèmes *Unix*.

Note : L'affichage des données doit être réalisé grâce à la fonction `printf` de la bibliothèque standard : si `s` est une chaîne de caractères et `x`, `y` deux entiers, alors `printf(s, x, y)` affiche la chaîne `s` dans laquelle la première occurrence de “%d” est remplacée par la valeur de `x`, et la seconde par celle de `y`.

Exemples de solutions

1. (a) Il y a 6 signaux différents, dont 3 (ceux dont la fréquence est située entre 434,095 et 434,775 MHz) présentent une probabilité deux fois plus grande que les 3 autres (ceux dont la fréquence est située entre 433,075 et 433,755 MHz) d'être reçus.

On en déduit que la probabilité de recevoir un signal donné est égale à $\frac{1}{9}$ pour ceux des trois fréquences les plus basses, et à $\frac{2}{9}$ pour ceux de

fréquence plus élevée. Les quantités d'information correspondantes sont donc respectivement égales à

$$\log_2 9 \approx 3,170 \text{ bits}$$

et

$$\log_2 \frac{9}{2} \approx 2,170 \text{ bits.}$$

- (b) En moyenne, un tiers des signaux reçus apportent 3,170 bits d'information, et deux tiers d'entre eux 2,170 bits. Mémoriser un signal nécessite donc en moyenne

$$\frac{1}{3}3,170 + \frac{2}{3}2,170 \approx 2,503 \text{ bits.}$$

Une mémoire de 256 MB est donc capable de retenir

$$\frac{256 \cdot 2^{20} \cdot 8}{2,503} \approx 858 \cdot 10^6$$

signaux, ce qui correspond à 858 secondes, ou encore 14 minutes et 18 secondes.

2. (a) Les nombres représentables sur n bits par complément à deux sont ceux qui appartiennent à l'intervalle $[-2^{n-1}, 2^{n-1} - 1]$. Pour le nombre -1 , la plus petite valeur de n qui convient est donc $n = 1$. La représentation demandée est donc 1.

- (b)

$$\begin{array}{r} -12,875 : \quad \boxed{1 \mid 0 \mid 0 \mid 1 \mid 1 \mid 0 \mid 0 \mid 1} \\ \phantom{\boxed{1 \mid 0 \mid 0 \mid 1 \mid 1 \mid 0 \mid 0 \mid 1}} \\ 6,375 : \quad + \quad \boxed{0 \mid 0 \mid 1 \mid 1 \mid 0 \mid 0 \mid 1 \mid 1} \\ \hline -6,5 : \quad \boxed{1 \mid 1 \mid 0 \mid 0 \mid 1 \mid 1 \mid 0 \mid 0} \end{array}$$

- (c) Notons w et w' les deux suites de n bits qui sont additionnées, et w'' le résultat de l'opération. Étant donné que w , w' et w'' possèdent toutes les trois un bit de poids fort égal à 1, on a $[w]_{ns} = [w]_{c_1} + 2^n - 1$, $[w']_{ns} = [w']_{c_1} + 2^n - 1$ et $[w'']_{ns} = [w'']_{c_1} + 2^n - 1$.

L'algorithme d'addition a nécessairement produit un report à la position n . On a donc

$$\begin{aligned}[w'']_{c_1} &= [w'']_{ns} - 2^n + 1 \\ &= [w]_{ns} + [w']_{ns} - 2^{n+1} + 1 \\ &= [w]_{c_1} + [w']_{c_1} - 1.\end{aligned}$$

En résumé, le résultat obtenu est égal à la somme des deux nombres moins 1.

- (d) Premièrement, le nombre est négatif, donc son bit de signe est égal à 1. Il reste ensuite à construire une mantisse de valeur la plus petite possible, tout en restant différente de 0. Une telle mantisse est nécessairement dénormalisée, ce qui conduit à un exposant égal à -127 . La représentation de cet exposant est 00000000.

La plus petite mantisse dénormalisée différente de 0 s'obtient en fixant tous les bits de sa représentation à 0, sauf le dernier qui est égal à 1. On obtient donc 000000000000000000000001.

En résumé, la représentation recherchée est composée de 32 bits égaux à 0, à l'exception du premier et du dernier qui sont égaux à 1. En notation hexadécimale, cette représentation s'écrit 0x80000001.

3. (a) La pile permet de gérer les appels de fonctions, en retenant (sous la forme de *stack frames*) l'adresse de retour, les arguments et les variables locales de chaque appel. La pile peut également être utilisée pour sauvegarder et restaurer des données temporaires.
- (b) Cela signifie que lorsqu'une donnée doit être mémorisée dans plusieurs cellules de la mémoire, ses 8 bits de poids faible sont placés dans la cellule d'adresse la plus faible, puis les 8 bits suivants dans la cellule d'adresse suivante, et ainsi de suite jusqu'aux 8 bits de poids forts qui sont placés dans la cellule d'adresse la plus élevée.
Par exemple, mémoriser l'entier de 16 bits 0x1234 à l'adresse 0x100 s'effectue en écrivant 0x34 à l'adresse 0x100 et 0x12 à l'adresse 0x101.
- (c) Le langage d'assemblage est un formalisme textuel lisible, au contraire du code machine qui est une représentation numérique d'un programme directement compréhensible par le processeur.
- (d) Ce fragment de code commence par calculer le "ou exclusif" de chaque bit du registre **RCX** avec lui-même, ce qui a pour effet de mettre la valeur de ce registre à 0.

Ensuite, l'instruction `LOOP` décrémente d'abord la valeur de `RCX`, et ensuite effectue à nouveau cette opération tant que le résultat obtenu est différent de 0. Étant donné que la valeur initiale de `RCX` est nulle, 2^{64} itérations de la boucle seront donc nécessaires pour en sortir (ce qui correspond en pratique à un nombre d'opérations prohibitivement élevé).

4. (a) `#include <stdio.h>`

```
int main()
{
    int i;

    for (i = 1; i <= 100; i++)
        printf("%d: %d\n", i, i * i * i);

    return 0;
}
```

(b)

```
.intel_syntax noprefix
.data
str: .asciz "%d: %d\n"
.text
.global main
.type main, @function
main:
    push RBP
    mov  RBP, RSP
    mov  RSI, 1                # i = 1
boucle:
    mov  RAX, RSI
    imul RSI
    imul RSI
    mov  RDI, offset flat:str
    mov  RDX, RAX              # i * i * i
    push RSI                  # sauvegarde RSI
    sub  RSP, 8                # (RSP multiple de 16)
    call printf
    add  RSP, 8                # (rétablissement RSP)
    pop  RSI                   # récupération RSI
    inc  RSI                    # i++
    cmp  RSI, 100              # i <= 100?
```

```
jle boucle  
mov EAX, 0  
pop RBP  
ret
```