E.N.S.E.I.R.B. Année 2004-2005
Filière Informatique $1^{ère}$ année
Filière Télécom $1^{ère}$ année

STRUCTURE DES ORDINATEURS

EXAMEN

CORRIGÉ

N.B. : - Ceci doit être considéré comme un corrigé-type : les réponses qu'il contient sont justes, mais leur rédaction n'était pas la seule possible.
- Le barème est donné à titre définitif. Outre l'exactitude des réponses aux questions posées, il a été tenu compte de leur concision et, dans une moindre mesure, de la présentation.

Question 1 (3 points)

La traduction et l'interprétation sont deux mécanismes permettant l'exécution sur un ordinateur utilisant un certain langage L_0 d'un programme écrit dans un langage L_1 de plus haut niveau, c'est-à-dire tel que toute instruction de L_1 soit sémantiquement équivalente à une ou plusieurs instructions de L_0.

La traduction consiste à utiliser une seule fois un programme spécifique en L_0, appelé compilateur, pour traduire le programme en L_1 en un programme complet en L_0, qui pourra alors être exécuté autant de fois que nécessaire de façon autonome, le programme en L_1 et le compilateur n'étant plus nécessaires.

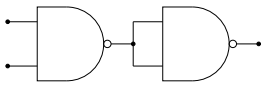
L'interprétation consiste à utiliser, chaque fois que l'on veut exécuter le programme en L_1, un programme spécifique en L_0 appelé interpréteur, qui associera à l'instruction courante du programme en L_1 un ensemble sémantiquement équivalent d'instructions en L_0, qui seront immédiatement exécutées.

La différence entre traduction et interprétation repose donc sur la production, avant toute exécution, d'un programme en L_0 issu du programme en L_1.

Question 2 (6 points)

(2.1) (1 point)

Une porte NON-ET fournit l'inverse de ce que doit fournir la porte ET à construire. Comme les deux broches d'entrée d'une porte NON-ET sont symétriques, que la fonction à câbler est symétrique, et qu'il ne faut plus utiliser qu'une autre porte, ce n'est donc pas au niveau des broches d'entrée qu'il faut intervenir. Il nous faut donc câbler un inverseur avec une porte NON-ET. Puisque $x = x.x$, alors $\overline{x} = \overline{x.x}$, et il suffit donc simplement de connecter ensemble les deux broches d'entrée de la porte NON-ET pour réaliser l'inverseur.



(2.2) (3 points)

Si on étiquette les quatre entrées de E0 à E3, et les deux sorties de S0 à S1 (où S0 peprésente le bit de poids faible et S1 le bit de poids fort), la table de vérité de l'encodeur est la suivante :

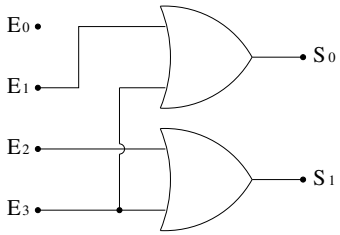
Entrée	*Sorties*	
	$S1$	$S0$
$E0$	0	0
$E1$	0	1
$E2$	1	0
$E3$	1	1

Le fait que seule une des entrées soit au niveau logique 1, alors que toutes les autres sont au niveau 0, est une contrainte très forte, qui permet de simplifier grandement les équations logiques. On a alors

les fonctions logiques suivantes :

$$\begin{aligned} S0 &= E1 + E3 \ , \\ S1 &= E2 + E3 \ . \end{aligned}$$

Le schéma de l'encodeur est donc le suivant :

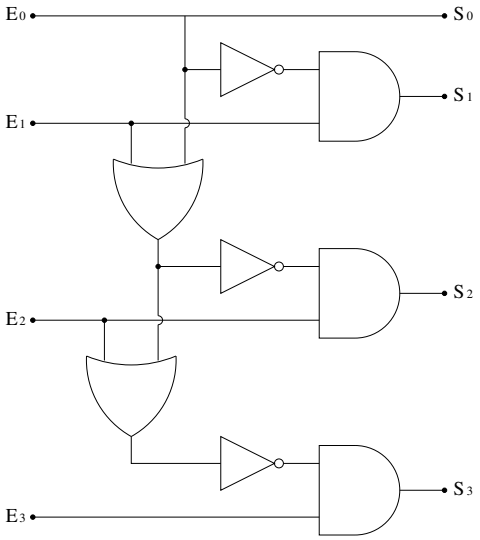


(2.3) (2 points)

Le schéma du circuit logique est très régulier : il s'agit, dès qu'une entrée est positionnée à 1, d'inhiber la traversée du signal sur les lignes de numéro supérieur. Si l'on étiquette les quatre entrées de E0 à E3, et les quatre sorties de S0 à S3, on a les fonctions logiques suivantes :

$$\begin{aligned} S0 &= E0 \ , \\ S1 &= E1.\overline{E0} \ , \\ S2 &= E2.\overline{E0 + E1} \ , \\ S3 &= E3.\overline{E0 + E1 + E2} \ . \end{aligned}$$

Le schéma du filtre est donc le suivant :



Il est à noter que l'on pourrait très bien se passer de l'entrée E3 et de la porte AND associée, en posant simplement $S3 = \overline{E0 + E1 + E2}$. Ceci permet en plus de filtrer le cas invalide $\{0, 0, 0, 0\}$, qui est alors transformé en $\{1, 0, 0, 0\}$.

Question 3 (6 points)

Dans tous les programmes suivants, on prendra l'hypothèse conservatrice qu'il ne faut pas altérer les données initiales.

– Architecture à zéro adresses : comme il n'y a pas, dans l'énoncé, d'informations précises sur la sémantique des opérations `SUB` et `DIV` (est-ce le total ou la quantité à soustraire qui doit être placé en sommet de pile, et est-ce le diviseur ou le dividende qui doit être placé au sommet ?), il est possible d'utiliser ces opérateurs comme on l'entend. Cela n'a en fait aucune incidence sur le nombre d'instructions, puisqu'on peut toujours placer les arguments dans la pile de façon à les retrouver à la bonne place au moment du `SUB` puis du `DIV`, sans avoir besoin de faire aucun `POP` de sauvegarde

temporaire (la pile sert justement à cela) :

```
PUSH   E          // 2 + 4 demi-octets
PUSH   F          // 2 + 4
MUL               // 2
PUSH   D          // 2 + 4
SUB               // 2
PUSH   B          // 2 + 4
PUSH   C          // 2 + 4
MUL               // 2
PUSH   A          // 2 + 4
ADD               // 2
DIV               // 2
POP    X          // 2 + 4
```

On a donc 12 instructions et 52 demi-octets.

– Architecture à une adresse : ici, la sémantique des instructions `SUB` et `DIV` est parfaitement définie, et impose des sauvegardes temporaires, réalisées dans la variable `X` :

```
LOAD   E          // 2 + 4 demi-octets
MUL    F          // 2 + 4
STORE  X          // 2 + 4
LOAD   D          // 2 + 4
SUB    X          // 2 + 4
STORE  X          // 2 + 4
LOAD   B          // 2 + 4
MUL    C          // 2 + 4
ADD    A          // 2 + 4
DIV    X          // 2 + 4
STORE  X          // 2 + 4
```

On a donc 11 instructions et 66 demi-octets.

– Architecture à deux adresses : on a plusieurs écritures possibles, selon qu'on accumule les résultats dans R0 ou directement dans la case mémoire X. Dans le premier cas, on a une instruction MOV en plus pour sauver le résultat final de R0 dans X, mais on utilise moins de demi-octets que pour la deuxième version. C'est cette version que l'on présente ici :

```
MOV    R0,E       // 2 + 1 + 4 demi-octets
MUL    R0,F       // 2 + 1 + 4
MOV    R1,D       // 2 + 1 + 4
SUB    R1,R0      // 2 + 1 + 1
MOV    R0,B       // 2 + 1 + 4
MUL    R0,C       // 2 + 1 + 4
ADD    R0,A       // 2 + 1 + 4
DIV    R0,R1      // 2 + 1 + 1
MOV    X,R0       // 2 + 1 + 4
```

On a donc 9 instructions et 57 demi-octets.

– Architecture à trois adresses :

```
LOAD   R0,E       // 2 + 1 + 4 demi-octets
LOAD   R1,F       // 2 + 1 + 4
MUL    R0,R0,R1   // 2 + 1 + 1 + 1
LOAD   R1,D       // 2 + 1 + 4
SUB    R2,R1,R0   // 2 + 1 + 1 + 1
LOAD   R0,B       // 2 + 1 + 4
LOAD   R1,C       // 2 + 1 + 4
MUL    R0,R0,R1   // 2 + 1 + 1 + 1
LOAD   R1,A       // 2 + 1 + 4
ADD    R0,R0,R1   // 2 + 1 + 1 + 1
DIV    R0,R0,R2   // 2 + 1 + 1 + 1
STORE  R0,X       // 2 + 1 + 4
```

On a donc 12 instructions et 74 demi-octets.

Question 4 (5 points)

(4.1) (1 point)

La règle des « *des 20-80* » est due principalement aux boucles et aux appels de sous-programmes. Puisqu'un programme est en général destiné à effectuer un même traitement sur un ensemble de

données, on implémente les algorithmes sous forme de corps de boucle ou de sous-programmes qui traitent, les unes après les autres, les données du problème. Un autre facteur, de moindre impact, est la présence de code de traitement d'erreurs, pratiquement jamais utilisé, qui renforce la prédominance des principaux corps de boucles.

(4.2) (2 points)

Afin de tirer avantage de ce constat, il faut faire en sorte que l'ordinateur puisse exécuter plus rapidement les programmes lorsque cette exécution reste localisée sur une petite portion du code.

Parmi les mécanismes spécifiquement conçus dans ce but, on trouve les caches. Le rôle d'un cache est de stocker, au plus près du processeur, et donc avec une vitesse de lecture plus importante, les informations les plus récemment accédées. Lorsqu'on entre dans une boucle, ou un sous-programme, pour la premiere fois, les instructions correspondantes sont conservées dans le cache, et pourront être présentées plus rapidement au processeur lors des itérations suivantes.

La mémoire virtuelle paginée et le swap participent également de ce principe : on ne conserve en mémoire centrale que les pages de code utiles à l'avancement immédiat du programme, toutes les autres pages pouvant servir à stocker des données utiles.

En fait, c'est l'ensemble de la hiérarchie mémoire, des registres au swap, qui est conçu afin de tirer parti des principes de localité.

(4.3) (2 points)

Analyse des trois possibilités :

i. Si l'on écrit entièrement le programme en langage C, le codage prendra 10 mois-homme. Si l'on prend l'efficacité de ce programme comme référence, nous dirons que ce programme s'exécute avec une vitesse 1. C'est la solution la plus rapide, mais la moins efficace.

ii. Si l'on écrit entièrement le programme en assembleur, le codage prendra 100 mois-homme, soit plus de 8 années-homme. C'est énorme, et représente des coûts de développement bien trop élevés pour la majorité des projets. Le programme cependant sera quatre fois plus rapide que la version de référence.

iii. Si l'on écrit entièrement le programme en langage C, puis qu'on recode ensuite les 1% du programme en assembleur, le développement du programme prendra $10+10\times\frac{1}{100}=11$ mois-homme. L'efficacité globale du programme sera alors de $\frac{1}{2}\times 1+\frac{1}{2}\times 4=\frac{5}{2}$, soit deux fois et demie plus rapide que la version de référence.

La question à se poser, face à ces trois choix de développement, est de savoir quel est le facteur limitant (temps de développement ou performance du logiciel), et si d'ailleurs il y en a même un.

La première solution convient aux programmes interactifs, pour lesquels la portabilité et la maintenabilité dominent.

La troisième solution est clairement la meilleure si l'efficacité du programme est un facteur limitant, au détriment d'une légère perte de réutilisabilité, puisque le coût de codage de la partie en langage machine devra être repayé pour chaque nouveau portage. Cependant, on n'a pas de perte de portabilité, car il reste toujours la version de référence codée intégralement en langage C, qui peut servir de démonstrateur lors d'un nouveau portage.

La deuxième option est peu réaliste pour un logiciel généraliste, sauf à vouloir mobiliser assez de personnel pour réaliser le projet en moins de trois ans. En effet, comme la puissance des machines double tous les 16 mois, il suffit d'attendre moins de trois ans pour trouver sur le marché des machines capables d'exécuter le programme en langage C quatre fois plus vite que la version de référence d'il y a trois ans. Qui plus est, un tel développement n'est absolument pas pérenne, et donc affreusement coûteux à long terme lorsque les matériels évoluent... Cependant, pour certains logiciels critiques embarqués, pour lesquels, une fois le matériel validé, il ne sera pas possible d'en changer, cela pourra être la seule solution possible.