DESS CCI : examen Langage Machine, Décembre 2006

Deux heures, tous documents et calculatrices autorisés. Ordinateurs (PC) interdits.

On rappelle que le format **%s** dans printf sert à afficher une chaîne de caractères.

1 Appel de procédures (environ 6 points)

On considère l'extrait de programme C suivant. L'algorithme utilisé dans ce programme est incorrect : il n'y a pas de test pour arrêter la recherche lorsque l'on a parcouru tout le tableau sans trouver la valeur recherchée.

```
#include <stdio.h>
#define TAILLE_TAB 4

/*********************************************/
/* rechercher la valeur wanted dans le tableau */
/* a partir de l'indice debut                  */
/* retourne l'indice de wanted dans le tableau */
/*********************************************/

int chercher (char *f1, char *f2, char *f3, char *f4, int wanted, int debut)
{
int indice;
int tab [TAILLE_TAB] = { 63, -6, 0, 8};

printf ("%s%s%08x,%s%s%08x\n",
       f1, f2, (unsigned int) &wanted, f3, f4, (unsigned int) &indice);
if (wanted == tab[debut])
   /* trouve dans debut : on ne cherche pas plus loin */
   indice = debut;
else
   /* pas dans le dernier element : chercher a partir de tab[debut+1] */
   indice = chercher (f1,f2,f3,f4,wanted, debut+1);
return (indice);
}

void tester (int v)
{
printf ("valeur %d : indice = %d\n",v,
        chercher( "&wanted ="," 0x"," &indice ="," 0x", v,0));
}
```

```
void main ()
{
tester (63);
tester (8);
tester (-1);
tester (-6);
tester (5);
}
```

L'exécution de ce programme (compilé avec armgcc) génère l'affichage suivant :

```
&wanted = 0x007fffcc, &indice = 0x007fffa8
valeur 63 : indice = 0
&wanted = 0x007fffcc, &indice = 0x007fffa8
&wanted = 0x007fff8c, &indice = 0x007fff68
&wanted = 0x007fff4c, &indice = 0x007fff28
&wanted = 0x007fff0c, &indice = 0x007ffee8
valeur 8 : indice = 3
&wanted = 0x007fffcc, &indice = 0x007fffa8
&wanted = 0x007fff8c, &indice = 0x007fff68
&wanted = 0x007fff4c, &indice = 0x007fff28
&wanted = 0x007fff0c, &indice = 0x007ffee8
&wanted = 0x007ffecc, &indice = 0x007ffea8
&wanted = 0x007ffe8c, &indice = 0x007ffe68
&wanted = 0x007ffe4c, &indice = 0x007ffe28
&wanted = 0x007ffe0c, &indice = 0x007ffde8
&wanted = 0x007ffdcc, &indice = 0x007ffda8
&wanted = 0x007ffd8c, &indice = 0x007ffd68
&wanted = 0x007ffd4c, &indice = 0x007ffd28
&wanted = 0x007ffd0c, &indice = 0x007ffce8
&wanted = 0x007ffccc, &indice = 0x007ffca8
&wanted = 0x007ffc8c, &indice = 0x007ffc68
valeur -1 : indice = 13
&wanted = 0x007fffcc, &indice = 0x007fffa8
&wanted = 0x007fff8c, &indice = 0x007fff68
valeur -6 : indice = 1
&wanted = 0x007fffcc, &indice = 0x007fffa8
&wanted = 0x007fff8c, &indice = 0x007fff68
&wanted = 0x007fff4c, &indice = 0x007fff28
&wanted = 0x007fff0c, &indice = 0x007ffee8
&wanted = 0x007ffecc, &indice = 0x007ffea8
&wanted = 0x007ffe8c, &indice = 0x007ffe68
&wanted = 0x007ffe4c, &indice = 0x007ffe28
&wanted = 0x007ffe0c, &indice = 0x007ffde8
&wanted = 0x007ffdcc, &indice = 0x007ffda8
```

```
&wanted = 0x007ffd8c, &indice = 0x007ffd68
&wanted = 0x007ffd4c, &indice = 0x007ffd28
&wanted = 0x007ffd0c, &indice = 0x007ffce8
&wanted = 0x007ffccc, &indice = 0x007ffca8
&wanted = 0x007ffc8c, &indice = 0x007ffc68
valeur 5 : indice = 13
```

Question a : Expliquer pourquoi les adresses de wanted et indice varient.

Question b : A combien de mots de 32 bits correspond la différence entre les adresses de wanted et indice. A quoi servent ces mots ?

Question c : Expliquer pourquoi la recherche d'une valeur absente du tableau se termine et pourquoi avec un indice à 13.

2 Déclarations de variables (environ 6 points)

On considère l'extrait de programme C suivant (qui affiche bonjour !). Ce programme est exécuté sur une machine 32 bits [1].

```
unsigned short int us = 0x1234;
long  int l = 0x12345678;
char c1 = 0x61;
char c2 = 0x62;
char c3 = 0x63;
short int s = 0x6100;
char message [] = "bonjour!"
char *p = &c1;
void main ()
{
printf ("%s\n",message); /* Qu'afficherait printf ("%s\n",&c1); ? */
*p = c2;
}
```

Question d : Traduire en langage d'assemblage ARM les déclarations des variables. L'ordre de stockage en mémoire devra respecter celui des déclarations dans le programme. Combien d'octets occupe message ?.

Question e : Que fera le programme si l'on remplace **message** par **&c1** dans l'appel de printf, selon que le processeur est Little ou Big Endian[2] ?

[1] sizeof(char) = 1, sizeof(short) = 2, sizeof(int) = sizeof(long) = 4

[2] big endian : l'octet à l'adresse d'un entier 16 ou 32 bits contient les bits de poids fort (faibles si little endian)

Question f : Combien d'accès à la mémoire l'exécution de ***p=c2** génère-t-elle dans chacune des sections (text, data, et bss) ? .

3 Représentation des entiers en binaire (environ 8 points

On rappelle que l'affectation **x =x « n** effectue un décalage logique à gauche : elle élimine les n bits de poids forts de x et ajoute n chiffres à zéro en poids faible. De même, (appliquée à un entier naturel), l'opération C **x » n** est un décalage logique à droite : élimination des n bits de poids faible et ajout de n chiffres à zéro en poids forts.

3.1 Exemples sur quatre bits

Soit des processeurs fictifs travaillant respectivement sur 4, 8 ou 12 bits.

Question g : Donner les quatre paires de valeurs d'entier (naturel et relatif) dont les représentations en binaire sont les suivantes : **0110**, **0101**, **1101** et **1110** ?.

Question h : Si on suppose maintenant que ces entiers sont stockés dans une machine travaillant sur 12 bits. Quelle est la représentation en hexadécimal de ces quatre paires de valeurs ?

Question i : Quelle est la particularité commune des représentations en binaire de tous les entiers négatifs ?.

Question j : Quelle est la particularité commune des représentations en binaire de tous les entiers divisibles par 4 ?.

Question k : Soit x un entier naturel stocké dans une machine travaillant sur 8 bits. Que peut-on dire de x si l'expression C $(((x >> 3) << 3) - x)$ est égale à 0 ?

3.2 Manipulation de binaire en C

Voici une procédure C qui génère dans une chaîne de caractères la représentation en binaire (par paquets de 4 chiffres) d'un entier codé sur 32 bits.

```
void sprintbin (int x, char *ecriture)
{
register char *p;              /* stocker p dans r4 */
register unsigned int i;       /* stocker i dans r5 */

p = ecriture;
```

```
i = 0;

while (i < 32)
  {
  if (x < 0)
     *p++ = '1';   /* equivaut à *p = '1'; p++ */
  else
     *p++ = '0';
  x = x << 1;
  i++;
  if ((i%4) == 0) *p++ = ' ';   /* i%4 : module : reste de la division par 4 */
  }
*p = 0;
}
```

L'appelante de sprintbin passe deux paramètres : l'entier à convertir et l'adresse de stockage de la chaîne de caractères à générer.

```
#include <stdio.h>
#define MAX  40     /* 32 chiffres, 7 espaces */

char chaine [MAX];
char msgerr [] = "ce message ne devrait jamais etre affiche";

int main()
{
/* Ce programme affiche : 1010 1011 1100 1101 1110 1111 1001 0101   */
sprintbin (0xabcdef95,chaine);
printf ("%s\n", chaine);
}
```

Question l : Expliquer en quelques lignes le pincipe de fonctionnement de cet algorithme. Justifier en particulier le test de la condition $x < 0$.

Question m : L'affichage généré par ce programme est-il modifié si la dernière affectation ($*p = 0$) dans sprintbin est supprimée ? Si non, expliquer pourquoi. Si oui, comment faut-il modifier le fichier main.c de telle sorte que le message affiché ne soit correct que si l'affectation $*p = 0$ est présente dans sprintbin ?

Question n : En utilisant la convention d'appel standard (premiers paramètres dans $r0$ à $r3$), traduire le corps de la procédure printbin en langage d'assemblage, ainsi que l'instruction **sprintbin (x,chaine) ;** de la procédure main.

Question o : On décide de ne plus insérer d'espace entre les paquets de 4 chiffres et de supprimer purement et simplement la variable i. Donner maintenant la nouvelle condition de boucle du programme C et sa traduction en langage d'assemblage.