Master CCI LM Examen final 2011

Durée Deux heures, documents autorisés, calculatrice et ordinateur interdits

Avant de commencer ce sujet, prennez bien connaissance de l'annexe D qui vous donne des consignes sur l'ABI à respecter au cours de cet examen. Vous pourrez ensuite vous laisser guider par les diverses consignes de ces trois exercices indépendants.

1 Variables et pointeurs

Q1) Traduire en langage d'assemblage ARM les déclarations et affectatins suivantes :

```
short  a;
char   car;

char  *pcar = &car;
char   car2 = 'b';
short  b    = 3;

[...]
   a = b;
   car = *pcar;
[...]
```

Attention : Ici, toutes les variables sont placées en mémoire et sont globales.

2 Sérialisation (Tableaux, structures, ...)

2.1 Définition de l'enregistrement

On a la structure suivante :

```
struct record {
   int   id;
   char  initial;
   short age;
   short poids;
```

```
};

struct record a;
```

Q2.1) Traduire en langage d'assemblage ARM, la déclaration de `a`. Vous donnerez par la même la taille de la structure `struct toto`.

2.2 Sérialisation des enregistrement

On possède une grande collection d'enregistrement du type `struct record` placée sous la forme d'un tableau dans la mémoire centrale. On souhaite maintenant le transmettre par le réseau à une autre machine. Nous avons pour celà deux contraintes :

- rien ne sert de transmettre des octets inutiles, le débit de transfert est une ressource rare, on va donc éviter de la gaspiller. On ne transmettra donc pas les octets de bourrage.
- toutes les machines n'ont pas la même endianness (*c.f.* annexe E). L'ARM est naturellement little-endian. Malheureusement, il y a fort longtemps les constructeurs se sont mis d'accord pour que les valeurs transmises sur les réseaux soit big endian. Il nous faut donc écrire les valeurs dans la bonne endianness.

Pour réaliser celà, on utilise une fonction `serialize` (présentée en annexe A qui prend pour argument l'adresse d'un tableau d'enregistrements placés en mémoire, l'adresse d'un buffer d'octet assez grand pour contenir la serialisation de nos enregistrements, ainsi qu'un dernier argument qui nous donne le nombre d'enregistrement à serialiser.

Q2.2) Traduire en langage d'assemblage ARM la fonction `serialize` en respectant les conventions qui se doivent.

Attention : l'operation `LE_to_net` n'est pas une fonction qu'il vous faut appeler, mais bien une série d'instructions ARM à définir dans votre programme en langage d'assemblage.

3 Recursions

3.1 Recursion simple : calcul de puissance

Pour élever un nombre quelconque à une puissance quelconque, on peut avoir recours à une fonction récursive présentée en annexe B.

Q3.1.1) Traduire en langage d'assemblage ARM la fonction `pow` en respectant les conventions qui se doivent.

On produit un programme qui appelle directement la fonction `pow` dans le `main`. En observant le simulateur, dans le `main`, on trouve l'appel à la fonction `pow` :

```
[PC]            insn LM       insn LA
[0x00400038]    0xeb00005C    bl 0x0040009C  [pow] ; 24: bl pow
```

Juste avant l'exécution de cette instruction, le contenu des registres est le suivant :

```
$r0 = 00000008
$r1 = 00000006
$sp = 7FFFEBC0
```

Q3.1.2) En fonction de votre programme, quel paramètre représente $r0 et $r1 ? En observant le simulateur, à quelle adresse commence votre fonction `pow` ?

Q3.1.3) Dessiner la pile en précisant bien les adresses et le contenu correspondant à l'exécution de votre programme pour 3 appels à votre fonction récursive.

3.2 BONUS : Récursion double : Calcul des combinaisons

Pour calculer les coefficients binomiaux de deux entiers positifs, on peut avoir recours à une fonction récursive `cnp` dite double présentée en annexe C.

Q bonus) Traduire en langage d'assemblage ARM la fonction `cnp` en respectant les conventions qui se doivent.

A Code C réalisant la serialisation

```
1
struct record {
3
    int    id;
5   char   initial;
    short  age;
7
};
9
void
11 serialize( struct record *src, char *dst, int n){

13    int i;

15    for(i = 0; i < n; i++){

17        *(int *)dst) = LE_to_net(src[i].id);
          dst += 4;
19
          *dst = LE_to_net(src[i].initial);
21        dst += 1;

23        *(short *)dst = LE_to_net(src[i].age);
          dst += 2;
25
          *(short *)dst = LE_to_net(src[i].poids);
27        dst += 2;

29    }

31 }
```

serialize.c

B Code C réalisant le calcul de puissance

```
1
int pow(int a, int n)
3 {
    if (n == 0) return 1;
5   return a * pow(a, n - 1);
}
```

recursion_1.c

C Code C réalisant le calcul des combinaisons

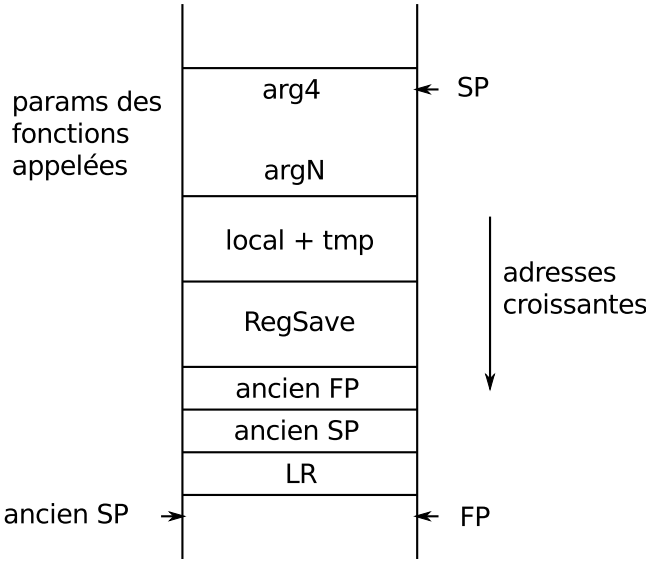
```
2 int cnp(int n, int p)
  {
4   if (p > n)
      return 0;
6   if (p == 0)
      return 1;
8
    return cnp(n - 1, p) + cnp(n - 1, p - 1);
10 }
```

recursion_2.c

D ABI simplifiée

Pour la traduction de fonction/procédure dans cette examen, il vous est demandez de respecter une ABI simplifiée (par rapport à celle produite par exemple par les chaînes de compilation GNU).

Il vous est proposé d'adopter l'agencement de pile suivant :



Dans le cas de l'ARM, nous sommes bien en présence d'un pile FD (Full Descending). C'est à dire que la pile grandit vers les adresses décroissantes et que le sommet de pile (SP) pointe sur le dernier élément plein. On rappelera de même que les quatres premiers arguments des fonctions sont passés par registres (`r0 - r3`) et que la valeur de retour est passée par registre (`r0`). Aucun registre n'est sûr au travers des appels de fonction.

Pour la sauvegarde et la restauration de registres vous n'utiliserez que des `ldr` et `str` (pas de `ldm` et `stm`).

E Big et Little Endian

Soit un entier sur 32 bits stocké en mémoire à l'adresse 4X : chaque octet contient 8 bits de l'entier. Il existe deux méthodes de rangement de ces octets en mémoire :

- big-endian [1] : les octets d'adresses croissantes contiennent les paquets de 8 bits de poids croissant.
- little-endian [2] : les octets d'adresses croissantes contiennent les paquets de 8 bits de poids décroissant.

Exemple : stockage de l'entier 0xabcdef12 à l'adresse 0x1000

Adresse	Big Endian	Little Endian
0x1000	0xab	0x12
0x1000	0xcd	0xef
0x1000	0xef	0xcd
0x1000	0x12	0xab

1. dit grand boutiste
2. dit petit boutiste