M2P CCI : corrigé Langage Machine, Novembre 2015

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

Table des matières

Conventions communes à toutes les questions

Toutes les variables locales des fonctions et procédures sont stockées dans des registres (le numéro de registre à utiliser est indiqué en commentaire). Seules les variables déclarées à l'extérieur des fonctions et sans l'attribut register sont stockées en mémoire.

La convention d'appel des fonctions et procédures est la suivante :
— Tous les paramètres sont passés par les registres et non en mémoire
— Le premier paramètre est passé dans le registre r0
— Le deuxième paramètre est passé dans le registre r1
— Le résultat des fonctions est retourné dans le registre r0 à la place du premier paramètre

La taille d'un char est de 8 bits, celle d'un int ou unsigned int est de 32 bits.

Remarques générales sur le contenu des copies

Dans beaucoup trop de copies, il n'a été tenu aucun compte des conventions de stockage des variables locales et d'appel de procédure, ni de la présence ou de l'absence d'atribut register.

Une variable déclarée avec l'attribut register est une variable stockée directement dans un registre (de préférence celui donnée en commentaire).

Les déclarations suivantes :

```
register unsigned int somme;    /* a stocker dans r5 */
register int i;                 /* a stocker dans r6 */

register unsigned int decimal;  /* a stocker dans r4 */

register unsigned int *ptri;    /* a stocker dans r4 */
register unsigned int longueur; /* a stocker dans r6 */
```

signifient que, dans le corps des fonctions ou procédures auxquelles elles appartiennent, les variables ont les propriétés suivantes :

1. somme est le registre r5 (et n'a donc ni adresse mémoire ni étiquette somme associée, et utiliser l'instruction ldr r8,=somme n'a aucun sens) et son contenu est de type entier naturel. L'instruction mov r5,#32 correspond dans ce cas à somme=32 en C.
2. même chose pour i, décimal et longueur, dans les registres respectifs r6,r4 et r6, avec des contenus de type entier naturel excepté pour i (entier relatif)

3. ptri est le registre r4 (et n'a donc ni adresse mémoire ni étiquette somme associée, et ni ldr r8,=ptri ni ldr r8,=*ptri n'ont de sens), elle est de type unsigned int *, ce qui signifie que son contenu est de type adresse d'un emplacement mémoire (une variable) contenant un entier naturel.

La déclaration de ptri a été souvent interprétée à tort ainsi :
— la variable ptri est stockée en mémoire dans la section data ou bss
— r4 contient l'adresse à laquelle se trouve la variable ptri

Cette interprétation correspondrait à ce genre de déclaration :

```
unsigned int ptri;                      /* sans register, ptri sera stockée dans bss */
register unsigned int *r4 = &ptri;      /* r4 pointe sur ptri                        */
```

et donnerait lieu à une réservation de place dans bss et à une initialisation de r4 en début de corps de la fonction :

```
                    .bss
                    ... /* reservations de memoire precedentes */
ptri:               .skip  4


                    .text
                   @ dans le début du corps de la fonction
                   ldr  r4,=ptri
```

A contrario, dans main.c, la variable val1 n'est pas une variable locale de fonction, et elle ne pourrait pas avoir l'attribut register puisqu'on en passe l'adresse lors de l'appel de la fonction string_to_int. Elle sera donc obligatoirement stockée dans bss, et en supposant que v et tmp soient 2 variables stockées dans des registre rv et rtmp :

1. v = val1 équivaut à v = *&val1, et suppose une opération d'initialisation d'un registre avec l'adresse de val1 (ldr regadr,= val1) suivi d'une opération de lecture à cette adresse (ldr rv,[regadr]).
2. val1 = v équivaut à *&val1=v et suppose une opération d'initialisation d'un registre avec l'adresse de val1 (ldr regadr,= val1) suivi d'une opération d'écriture à cette adresse (str rv,[regadr]).
3. pour v = 35, il faut d'abord placer la constante dans un registre temporaire, on se retrouve ensuite avec un cas analogue au précédent (mov rtmp,#35 ; ldr regadr,=val1 ; str rtmp
4. une incrémentation val1 = val1 + 2 nécessite 4 instructions (initialisation de regadr, lecture par ldr, addition, écriture par str).

Si vous avez décider d'ignorer l'attribut register des variables locales somme, i, décimal, longueur, restez au minimum cohérent avec votre choix. Si vous ajoutez quelquechose à somme, la séquence doit ressembler à ceci :
ldr rx,= somme ; ldr r5,[rx] ; add r5,r5,... ; str r5,[rx] **sans oublier** la mise à jour du contenu en mémoire avec str.

1 Introduction sans question (5 mn)

En utilisant le format %d, les fonctions de type scanf permet de convertir une chaîne de caractères [1] représentant l'écriture d'un entier en décimal en la valeur entière correspondante.

L'objectif est de traduire en langage d'assemblage un programme C qui effectue un travail analogue et génère par exemple l'entier 1234 à partir de la chaîne de caractères "1234".

Outre main, le programme comprend 3 routines affectuant chacune une tâche :

1. **char10_to_val** prend un (code ASCII de) caractère chiffre décimal en paramètre et calcule sa valeur entière (calcule 5 à partir de '5').
2. **chaine_to_chiffres** applique char10_to_val à chaque caractère de la chaîne et remplit un tableau d'entiers avec les valeurs des chiffres (de "1234" à {1,2,3,4}).
3. **string_to_int** utilise chaine_to_chiffres pour convertir la chaîne en valeur entière suivant la formule $x = \sum_{i=0}^{n-1} x_i 10^i$ avec $x_i \in [0..9]$.

2 String_to_int : boucle for et tableau (1h)

Le programme principal main.c qui convertit deux chaînes, est donné en annexe :

```
mandelbrot> armrun string_to_int
Chaine : 43  -> 43
Chaine : 26543  -> 26543
```

Et voici le fichier string_to_int.c :

```
#include "chaine_to_chiffres.h"
#include <stdio.h>

#define MAX 11
unsigned int valeurs[MAX];

/* Conversion d'une chaine en valeur entiere (s -> valentier) */
void string_to_int (char *s, unsigned int *valentier)
{
   register unsigned int nb_val;   /* a stocker dans r4 */
   register unsigned int somme;    /* a stocker dans r5 */
   register int i;                 /* a stocker dans r6 */

   somme = 0;
```

1. sscanf pour une chaîne stockée dans un tableau, scanf et fscanf pour une chaîne lue au clavier ou dans un fichier

```
   nb_val = chaine_to_chiffres (s,valeurs);
   for (i=0;i<nb_val;i++) {
     somme = 10*somme + valeurs[i];
   }
   *valentier = somme;
}
```

et un squelette de traduction à compléter :

```
                  .global   string_to_int
MAX=11
                  /* ajouter sections bss et/ou data */

                  .text
string_to_int:
prologue:         stmfd sp!,{r4-r9,lr}     @ sauvegarde des registres en pile
                  mov   r9,r1              @ copie de parametre reçu valentier
                                           @ (L'appel de chaine_to_chiffres
                                           @ detruira le contenu de r1)
                  ldr   r7,= valeurs
corps:            ...

epilogue:         ldmfd sp!,{r4-r9,lr}     @ restauration registres depuis pile
                  mov   pc,lr
                  .ltorg
```

2.1 Sections data /bss (15mn)

Traduire en langage d'assemblage la déclaration du tableau valeurs.

Traduire en langage d'assemblage la déclaration des variables ch1, val1, ch2 et val2 du fichier main.c.

```
                  .global   string_to_int
#define MAX 11

                  .bss
valeurs:          .skip     MAX*4


                  .data
ch1:              .asciz  "43"   @ ou bien .ascii "43"; .byte 0
                                 @ ou bien .byte '4'; .byte '3'; .byte '0'

ch2:              .asciz "26543" @ meme principe : six fois  .byte possible
```

```
format:             .asciz "Chaine : %s -> %u \012"    @ '\012' = \n

@ La taille cumulée des chaines precedentes a peu de chances d'etre multiple de 4
@ donc directive d'alignement sur la taille (plsu elevee) de la variable qui suit

                    .balign 4
val2:               .word   7



                     .bss
val1:               .skip   4
```

Traduire en langage d'assemblage (section text) l'affectation **v = val1** de main.c. Vous supposerez que v est stockée dans le registre r4.

Attention : dans le fichier main.c, la valeur val1 est déclarée à l'extérieur de la fonction main. Elle n'a pas d'attribut register et ne pourrait en avoir puisqu'on en passe l'adresse lors de cet appel : string_to_int (ch1,&val1). La variable val1 est donc stockée en mémoire dans la section bss (pusique déclarée sans valeur initiale).

Tout accès à la variable val1 nécessite donc un accès mémoire, que l'on peut mettre en évidence en remplaçant dans le code C val par *&val1. L'affectation
ldr r4,= val1 correspond en C à r = &val1 (avec un problème de typage puisque &vl1 est de type unsigned int * et r de type unsigned int) et non à r=val1.

```
                    .text
@ v   : r4
@ tmp : r5,r6
main:               stmfd   sp!,{r4,r5,r6,lr}

                    ldr     r0,= ch1            @ string_to_int(ch1,&val1)
                    ldr     r1,=val1
                    bl      string_to_int

@ Début de traduction de v = val1
                    ldr     r4,=val1            @ v = *&val1
                    ldr     r4,[r4]
@ Fin de traduction de v=val1

                    ldr     r0,=format          @ printf ("...",ch1,v)
                    ldr     r1,=ch1
                    mov     r2,r4
                    bl      printf
```

```
ldr    r0,= ch2          @ string_to_int(ch2,&val2)
ldr    r1,=val2
bl     string_to_int

ldr    r4,=val2          @ v = *&val2
ldr    r4,[r4]

ldr    r0,=format        @ printf ("...", ch2,val2)
ldr    r1,=ch2
ldr    r2,= val2
ldr    r2,[r2]
bl     printf

ldmfd  sp!,{r4,r5,r6,lr}
mov    pc,lr

.ltorg
```

Imaginons que l'on ajoute dans le corps de main l'affection suivante : **val1 = 35**. **Comment** la traduirait-on en langage d'assemblage (en veillant à ne pas faire d'accès mémoire inutile).

```
@ Noter qu'il n'y a aucune raison de lire l'ancienne valeur de val1 avant
@ de lui en affecter une nouvelle : str et non une séquence ldr + str.
@ La constante 35 est codable sur 8 bits : mov # possible

ldr   r5,=val1    @ *&val1 = 35
mov   r6,#35
str   r6,[r5]
```

2.2 Boucle for (35mn)

Traduire en langage d'assemblage l'affectation **somme = 10*somme + valeurs[i]**

```
corpswhile:     add   r5,r5,r5, LSL #2      @ somme = 5*somme
                mov   r5,r5,LSL #1          @ somme = 2*somme
                mov   r8,r6,LSL #2          @ i*sizeof(unsigned int)
                ldr   r8,[r7,r8]
                add   r5,r5,r8              @ somme = somme + valeurs[i]
```

Remplacer la boucle for par une séquence d'instructions C équivalente utilisant une boucle while.

```
i=0;
while (i < nb_val) {
   somme = somme * 10 + valeurs[i];
   i++;
}
```

Traduire en langage d'assemblage la boucle for (indiquer par un commentaire l'endroit où placer la traduction de l'affectation précédente).

L'endroit où se trouvent les paramètres s et valentier est défini par la convention d'appel rappelée en début de sujet.

```
                                    @ for (i=0;i<nb_val;i++)
                mov   r6,#0         @ i=0
                b     condwhile     @ goto  condwhile

corpswhile:     ...                 @ somme = 10*somme + valeurs[i]
                add   r6,r6,#1      @ i++

condwhile:      cmp   r6,r4         @ if (i<nb_val) goto corpswhile
                blt   corpswhile    @ rappel : i n'est pas unsigned
```

Si vous préférez le test avant le corps :

```
condwhile:      cmp   r6,r4         @ if (i >= nb_val) goto finwhile
                bge   finwhile
corps:          ...                 @ somme = ...
                add   r6,r6,#1      @ i++
                b     condwhile     @ goto condwhile
finwhile:       ...                 @ suite après for
```

Remarque : la traduction (pas demandée) de *valentier=somme est str r5,[r1] puisque valentier, en tant que deuxième paramètre, doit être passé dans r1 par l'appelante, et que le registre stockant somme est r5.

2.3 Appel de fonction (10mn)

Traduire en langage d'assemblage l'appel **string_to_int (ch1,&val1)** de main.c.

```
ldr    r0,= ch1          @ string_to_int(ch1,&val1)
ldr    r1,=val1
bl     string_to_int
```

3 Char10_to_val : if, paramètre adresse (30 mn)

Voici le code C de la procédure char10_to_val et un squelette de traduction à compléter :

```
#include "char10_to_val.h"
#include <stdio.h>

/* Conversion d'un caractere chiffre -> valeur du chiffre ('4' -> 4) */
/* Convention d'appel : chiffre dans r0, pvalchiffe dans r1,         */
/*                      resultat dans r0 à la place de chval         */
/* Retourne 0 en cas d'erreur (pas un caractere dans ['0' ... '9']   */
/* Retourne 1 si la conversion est reussie                           */

int char10_to_val (char chiffre, unsigned int *pvalchiffre)
{
  register unsigned int decimal;        /* a stocker dans le registre r4 */
  register int r;                       /* a stocker dans le registre r5 */

  decimal = chiffre;
  if (decimal >= '0' && decimal <= '9') {
    decimal   = decimal - '0';
    *pvalchiffre = decimal;
    r = 1;
  } else {
    r = 0;
  }
  return r;
}
```

```
                    .global      char10_to_val
                    .text
char10_to_val:
prologue:           stmfd     sp!, {r4,r5}     @ sauvegarde des registres en pile

corps:              ...

si:                 ...

alors:              ...

sinon:              ...

finsi:
epilogue:           ...                        @ return r
```

```
                    ldmfd       sp!,{r4,r5}
                    mov  pc,lr
```

Avec une condition composée de type ET, il suffit que l'un ou l'autre des termes de la condition soit faux pour que l'on saute dans la branche sinon du if. La sémantique du C indique que l'opérateur && est de type "et puis" : la deuxième partie de la condtion n'est évaluée que si la première partie de la condition est vérifiée.

La traduction d'un if avec un condition de type ET suit le même principe que celle d'un if avec une condition simple, mais avec une séquence d'autant couples test+branchement si faux que de termes dans la condition composée.

Traduire en langage d'assemblage les instructions du corps de char10_to_val en décompsant au préalable la construction if en équivalent C avec if et goto. Vous pouvez écrire deux blocs de code séparés ou bien placer la forme if ... goto en commentaire des instructions en langage d'assemblage.

Remarque : il faut se souvenir que 'c' désigne l'entier code ASCII du caractère c minuscule, soit l'entier 0x63 et que le code ASCII des caractères chiffres décimaux sont différents des entiers qu'ils représentent. Les caractères chiffres 0 et 9 ont pour codes ASCII 0x30 et 0x39 et non 0 et 9 : '0' $\neq$ 0 et '9' $\neq$ 9.

Une déclaration C char valcar = '9' se traduirait dans la section data par
valcar : .byte 0x39 @ ou bien .byte 99 ou bien (plus,lisible) à .byte '9'.

```
                    .text
char10_to_val:
prologue:           stmfd     sp!, {r4,r5}  @ sauvegarde des registre en pile

corps:              mov  r4,r0              @ decimal = chiffre

condif:             cmp  r4, #'0'           @ if (decimal < '0') goto sinon
                    blo  sinon              @ blt si deicmal n'etait pas unsigned
                    cmp  r4, #'9'           @ if (decimal > '9') goto sinon
                    bhi  sinon              @ bgt si decimal n'etait pas unsigned

alors:              sub  r4,r4,#'0'         @ decimal = decimal - '0'
                    str  r4,[r1]            @ *pvalchiffre = decimal
                    mov  r5,#1              @ r = 1
                    b    finsi              @ goto finsi

sinon:              mov  r5,#0              @ r = 0

finsi:
epilogue:           mov  r0,r5              @ return r
```

```
                    ldmfd      sp!,{r4,r5}
                    mov  pc,lr
```

A noter : si la première condition est vraie, il ne faut pas poursuivre avec le bloc alors, mais avec l'évaluation de la deuxième partie de la condition.

```
/* Cette structure de code correspond a une condition de type OU : */
/* if ((decimal >= '0') || (decimal <= '9')) {                     */
/*     ...                                                         */
/*     r=1;                                                        */
/* } else {                                                        */
/*     r=0;                                                        */
/* }                                                               */

condif:    cmp  r4,#'0"       @ if (decimal >= '0') goto alors
           bhs  alors
           cmp  r4,#'9'       @ if (decimal >  '9') goto sinon
           bhi  sinon

alors:     ...                @ decimal = decimal - '0' etc
           b    finsi

sinon:     mov  r5,#0

finsi:
```

4 Chaine_to_chiffres : paramètres, pointeurs (25mn)

Voici le contenu du fichier chaine_to_chiffres.c et un squelette de traduction :

```
#include "char10_to_val.h"

/* 1er parametre  : une chaine de caracteres parmi ['0'... '9']            */
/* 2eme parametre : tableau chiffre a remplir avec les valeurs des chiffres */
/* La taille du tableau doit etre au moins egale a la longueur de la chaine s   */
/* Valeur retournee  : nombre de valeurs stockees dans chiffres                 */

/* Convention d'appel :                                                        */
/*   s dans r0, chiffres dans r1, valeur de retour dans r0 a la place de s     */

unsigned int chaine_to_chiffres (char *s, unsigned int *chiffres)
{
  register unsigned *ptri;             /* a stocker dans r4 */
  register char *ptrc;                 /* a stocker dans r5 */
  register unsigned int longueur;      /* a stocker dans r6 */
```

```
  register int ok;                    /* a stocker dans r7 */

  ptri = chiffres;
  ptrc = s;
  longueur = 0;

  /* do...while : comme while mais le corps est executé au moins 1 fois */

  do {
     ok = char10_to_val(*ptrc, ptri);
     if (ok != 0) {
        longueur = longueur + 1;
        ptrc = ptrc + 1;
        ptri = ptri + 1;
        }
  } while (ok != 0);
  return longueur;
}
```

```
                        .global chaine_to_chiffres
                        .text
chaine_to_chiffres:
prologue:               stmfd   sp!, {...}   @ sauvegarde des registres dans la pile
corps:                  ...

epilogue:               ...                  @ return longueur
                        ldmfd  sp!,{...}     @ restauration des registres
                        mov    pc,lr
```

Traduire en langage d'assemblage l'affectation **ptri = chiffres**.

Traduire en langage d'assemblage l'affectation **ptri = ptri + 1**.

Traduire en langage d'assemblage l'affectation
ok = char10_to_val(*ptrc, ptri)

```
        .text
        mov   r4,r1            @ ptri = chiffres
        add   r4,r4,#4         @ ptri = ptri + 1 *sizeof(unsigned int)

        mov     r1,r4          @ pvalchiffre de char10_to_val = ptri
        ldrb    r0,[r5]        @ chiffre de char10_to_val = *ptrc
        bl      char10_to_val
        mov     r7,r0          @ ok=char10_to_val(...)
```

Remarque : ne pas oublier la sémantique de l'arithmétique sur un pointeur. En supposant qu'un pointeur p contienne l'adresse de l'élément d'indice i d'un tableau

(p=&(t[i]), l'expression p+j doit correspondre à l'adresse de l'élément d'indice j du même tableau (p=&(t[i+j])). Sachant que la formule de calcul de l'adresse d'un élément d'indice k d'un tableau t est t + k * sizeof(type_du_tableau), il faut donc implicitement multiplier j par la taille d'un élément avant de l'ajouter à l'adresse contenue dans p.

Donner la liste des registres à sauvegarder et restaurer (dans stmfd et ldmfd). Un de ces rgistres doit impérativement être sauvegardé et restauré, faute de quoi l'exécution du programme ne sortira jamais de la fonction chaine_to_chiffres. **Quel est ce registre ?** (expliquer pourquoi).

Il ne faut pas sauver/restaurer r0 qui stocke la valeur de retour de la fonction. On peut se poser la question d sauvegarder le paramètre s avant d'appeler char10_to_val puisque *ptrc va écraser le contenu de r0 lors de cet appel. En fait, s est copiée dans ptrc avant ct appel, et n'est plus utilisée après, donc la réutilisation de r0 en tant que paramètre passé ne nécessite pas de sauvegarde spéficique de s.

On peut ne pas sauver/restaurer r1 qui est inclus dans la convention d'appel.

Il faut sauvegarder tous les autres registres affectés dans le corps de la fonction. Ce sont les registres r4 à r7 qui stockent les variables locales de la fonction, ainsi que le paramètre implicite adresse de retour : le contenu du registre lr sera détruit lors de l'appel de char10_to_val.

Le registre a ne pas oublier de sauvegarder et restaurer (même s'il n'apparait pas explicitement dans l'instruction bl qui le modifie) est lr.

Chaque nouvel appel d'une routine écrase dans lr l'adresse de retour reçue de l'appelante (main) et la remplace par l'adresse de retour (instruction suivant bl) dans la fonction (chaine_to_chiffres). En l'absence de sauvegarde/restauration de lr, l'instruction mov pc,lr à la fin de chaine_to_chiffres ne va donc pas retourner à l'appelante main, mais effectuer un branchement incontionnel dans le corps de chaine_to_chiffres, d'où une boucle infinie.

5 Annexe 1 : main.c

```
#include <stdio.h>
#include "string_to_int.h"

char ch1 [ ] = "43";  /* La taille du tableau est deduite de celle de la chaine */
unsigned int val1;
char ch2 [ ] = "26543";
unsigned int val2=7;
```

```
int main (void)
{
  register unsigned int v;

  string_to_int (ch1,&val1);
  v = val1;
  printf ("Chaine : %s  -> %u \n",ch1,v);
  string_to_int (ch2,&val2);
  printf ("Chaine : %s  -> %u \n",ch2,val2);

  return 0;
}
```