Systèmes Embarqués 1 : Travail écrit no 1.

Nom : Zerbon **Prénom :** Yannick **Classe : T-2/I-2**
Date : 30.11.2017

5.8

Problème n° 1 (Programmation en assembleur)

Implémentez en langage assembleur ARM le code ci-dessous. Le module assembleur contiendra toutes les directives permettant un assemblage et un linkage correct du module et de ses fonctions.

```
unsigned checksum (char s[]) {
    int l = strlen(s);
    unsigned a = 0;
    unsigned b = 0;
    for (int i=l-1; i>=0; i--) {
          a += s[i];
          b += a;
    }
    return (a%256) + ((b%256)<<8);
}
```

Indications :

- La méthode `compute_cks` reçoit en paramètre la référence sur un tableau de caractères (adresse du 1er caractère), `char s[]`, lequel est passé par le registre `R0`. Elle retourne la valeur de la somme de contrôle (`checksum`) par le registre `R0`.
- La méthode `strlen` est une méthode externe à ce module et ne doit pas être implémentée. Elle reçoit en paramètre la référence sur un tableau de caractères, lequel est passé par le registre `R0` et retourne le nombre de caractères contenus dans le tableau par le registre `R0`. Il faut noter que les registres `R0` à `R3` seront modifiés par l'appel à cette fonction.

--- en assembleur --------

```
.global checksum
.data   .align 8
.text   .align 8

checksum:
    mov r4, r0 // on place s dans r4
    bl strlen // r0 => l
    mov r1, #0 // a=0
    mov r2, #0 // b=0
    sub r0, r0, #1 // i=l-1
loop: cmp r0, #0 // i>=0
    blt 1f
    add r1, r1, [r4, r0] // a+= s[i]
    add r2, r2, r1 // b+=a
    sub r0, #1 // i--
    b loop
1:  and r1, r1, #8 // a = a%256
    and r2, r2, #8 // b = b%256
    lsl r2, #8 // b<<8
    add r0, r1, r2 // a+b
    str r0, [r0];
    bx lr
```

Systèmes Embarqués 1 : Travail écrit no 1.

10

Problème n° 2 (Mode d'adressage)

a) Implémentez les 2 instructions assembleur permettant de restaurer le contenu des registres R0 à R14 depuis le tableau « `regs` » contenu en mémoire : `regs: .space 15*4`

2

```
ldr r0, regs;
ldmia r0, {r0-r14}^
```

b) Donnez l'instruction assembleur permettant de stocker sur la pile le contenu des registres r7 à r11 et lr (les instructions push et pop ne peuvent pas être utilisées).

2

```
stmfd sp!, {r7-r11, lr};
```

c) Pour le code assembleur, la représentation de la mémoire (**Little-Endian / 8-bits**) et l'état des registres du processeur ci-dessous, donnez le résultat des opérations (état des registres, état de la mémoire) :

Mémoire (little-endian / 8 bits)		(après)
0x80004310	0x34	
0x80004311	0xf5	
0x80004312	0x89	
0x80004313	0xc9	
0x80004314	0x25	0x01
0x80004315	0x94	0x51
0x80002116	0xa5	0x00
0x80004317	0xc2	0x80

	Registres (avant)	Registres (après)
R0	0x0000'0008	0x00 82'0000
R1	0x0000'8200	0x0000'00c989
R2	0x0000'000c	
R3	0x8000'4312	0x 8000'4 314
R4	0x0000'0002	
R5	0x8000'4310	
R6	0xffff'fff2	
R7	0x8000'5101	
R8	0x8000'5110	

1. `lsl r0, r1, r0`

2

`r0 = r1 << r0` → 8200 << 8 → 0001

on décale de "2 hexa" sur la gauche : 00 82'0000

2. `ldrh r1, [r3], #2`

1. On charge 16 bits sur r1 depuis l'adresse r3 → 0000 '00 C989

2. on décale r3 de 2

2

3. `str r7, [r5, #4]`

2

on store 32 bits de r7 à l'adresse r5 décalée de 4

Systèmes Embarqués 1 : Travail écrit no 1.

10

Problème n° 3 (traitement numérique des nombres)

a) Prévoyez l'état des flags Z, C, N et V ainsi que le résultat contenu dans le registre R2 (en décimal) suite à l'exécution des instructions assembleur suivantes :

Remarque : toutes les opérations sont faites avec des registres de 8 bits au lieu de 32 bits

1. `ldr r2, =129`
   `subs r2, #127`

127 : 0111 1111
-127 : 1000 0001
=> 1000 0001 + 1000 0001 = 1 0000 0010

Z= 0 C= 1 N= 0 V= 1 R2(non signé) = 2 R2 (signé) = 2

2. `ldr r2, #-5`
   `adds r2, #254`

+5 : 0000 0101
-5 : 1111 1011
1111 1011 + 1111 1110 = 1 1111 1001
128 64 32 16 8 1
1111 1001 → 0000 0111

Z= 0 C= 1 N= 1 V= 0 R2(non signé) = 249 R2(signé) = -7

3. `ldr r2, #-8`
   `cmp r2, #248`

1111 1000 + 0000 1000 = 1 0000 0000
1111 1000 → 0000 1000

Z= 1 C= 1 N= 0 V= 0 R2(non signé) = 248 R2(signé) = -8

b) Représentez en hexadécimal sur 32 bits (simple précision) la valeur réelle ci-dessous et donnez le développement (pour rappel : exposant est codé sur 8 bits avec un biais de 127)

-19,75 / 4 :

S = 1

19,75 : 10011,11 · 2^{-2} => 1,00111 11 · $2^4 \cdot 2^{-2}$ = 1,00111 1 · 2^2

127 + 2 = 129 = 1000'0001

=> 1 1000'0001 0011 1100 0000 0000 0000 0000

=> 0x C0 9E 00 00

c) Citez les fanions (flags) utilisés pour tester les conditions des nombres signés et citez les 6 suffixes correspondants aux 6 opérations conditionnelles possibles sur ces nombres signés (==, !=, >, >=, <, <=), p.ex. eq pour ==

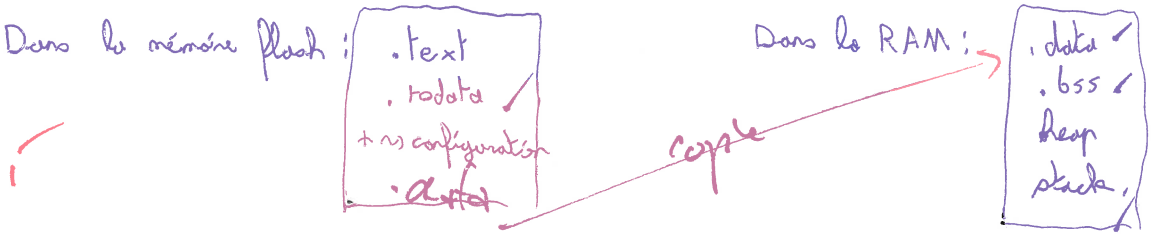
signés : N, V et Z

== eq
!= ne
> GT
>= GE
< : LE
<= : LT

Systèmes Embarqués 1 : Travail écrit no 1.

Problème n° 4 (architecture générale)

a) Citez les différents segments d'une application et leur organisation dans la mémoire pour un système sur puce (System on Chip – SoC)



b) Selon la classification de Flynn, citez l'architecture (l'abréviation) du µP TI-AM3358 (un ARM Cortex-A8) que nous utilisons au laboratoire et indiquez laquelle des 2 architectures Von Neumann et Harvard est mise en œuvre sur le processeur et pour quelle raison

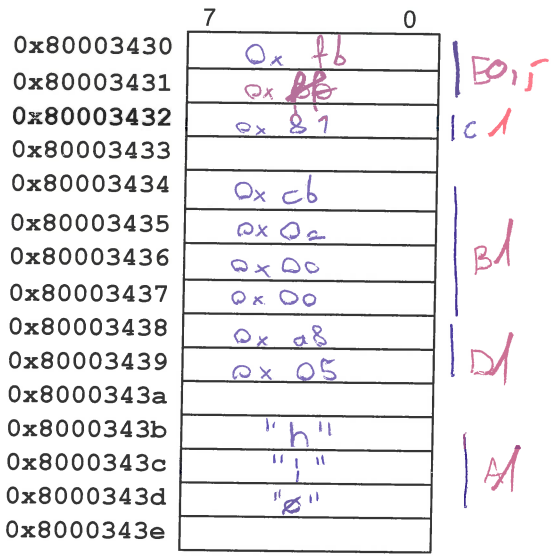
On utilise un SISD avec l'architecture von Neumann.

Harvard est un peu dépassé! La séparation des mémoires rend la gestion des commandes plus compliquées et la performance globale moins efficace, et vous l'utilisez en interne

Von Neumann n'a pas ce problème (une seule mémoire et un bus unique) mais elle est en externe

c) Pour une organisation de la mémoire en « **Little-Endian** », représentez (en hexadécimal pour les entiers et en caractère ascii pour les strings) dans le tableau ci-dessous les variables suivantes

	Adresse :	variable :	taille/type :	valeur :
A	0x8000343b	text:	.asciz	"hi"
B	0x80003434	var1:	.long → 32	$1313_8 = 3 + 8 + 3 \cdot 8^2 + 8^3 = 523 + 192 = 715 = 0x02cb$
C	0x80003432	var2:	.byte → 8	$129_{10} = 1000\ 0001 \rightarrow 0x81$
D	0x80003438	var3:	.short → 16	$5a8_{16}$
E	0x80003430	var4:	.short → 16	$-5_{10} \rightarrow 0xfb$



Systèmes Embarqués 1 : Travail écrit no 1.

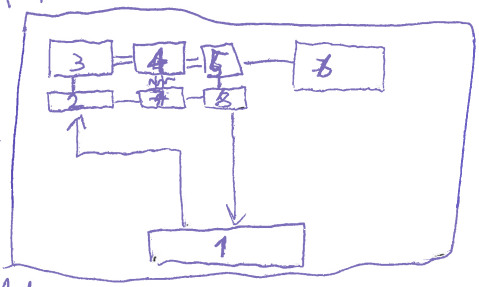
10

Problème n° 5 (architecture interne)

a) Citez dans l'ordre et décrivez succinctement les 8 composants que doit traverser l'instruction « `ldr r0,[r0]` » lors de son exécution depuis la mémoire principale

1. Mémoire Cache de niveau 2 : fait tampon avec la mémoire principale
2. Mémoire cache fetch de niveau 1 : fournit au fetch les opds mémoires requis
3. Fetch Instruction Unit : s'occupe de "fetch", collecter la commande
4. Decode Instruction Unit : transforme et décrypte la commande
5. Execute Instruction Unit : Performe l'instruction (avec l'ALU et la banque de registre)
6. Eventuellement le Neon processor pour certaines opérations avancées.
7. C15-MMU : Va s'occuper de ~~transformer~~ faire le lien entre les pages mémoires physiques et virtuelles
8. Mémoire cache exec de niveau 1 : fournit et transfert les opds mémoires vers

Passage à nouveau par la mémoire cache de niveau 2



4

11

6

b) Indiquez la fonction/l'usage des différents registres ci-dessous et indiquez le numéro du registre utilisé, s'il existe

1. SP : Stack Pointer, R13, pointe sur le "top", permet de faire des fonctions récursives par exemple
2. LR : Link Register, R14, permet de faire des retours de fonctions.
3. PC : Program Counter, R15, pointe sur l'instruction courante dans la mémoire.
4. CPSR : Registre de status, contient entre autres les différents flags.

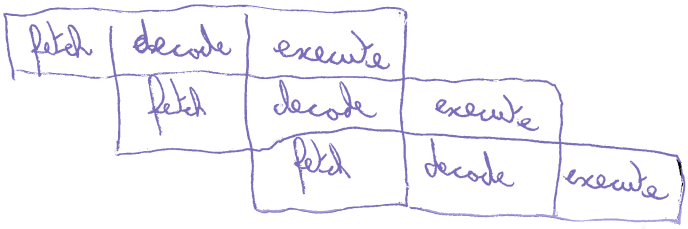
2

c) Indiquez les raisons ayant motivé les concepteurs des µP ARM v7 d'implémenter plusieurs modes de fonctionnement

Les "utilisateurs" (user, OS, ...) ont besoin de différentes ressources. Un utilisateur normal n'a pas besoin des modes privilégiés tandis qu'un OS si. Cela permet d'allouer les bonnes ressources à qui il faut tout en maintenant une certaine sécurité (système de permission).

2.5

d) Qu'apporte le pipelining du µP ARM Cortex-A8 et pour quelle raison



1

: Cela permet de gérer plusieurs instructions simultanément et donc de diminuer le temps/programme (performance ↑)