Microprocesseurs 1 & 2: Travail écrit no 3. 5 7

Nom : [redacted] **Prénom :** [redacted] **Classe : I/2** **Date : 19.04.2011**

Problème n° 1 (Interfaçage assembleur – C)

1. Implémentez en assembleur la fonction C « subtract » ci-dessous :

```
extern int subtract (int a, int b, int c, int d, int e, int f, int g);
int subtract (int a, int b, int c, int d, int e, int f, int g)
{ return a-b-c-d-e-f-g; }
```

Code assembleur:

```
subtract :  mov   r1, sp        @ copy sp
            ldr   r3, #0        @ r3 = offset

sub :       ldr   r0, [r1, r3]  @ load A in r1 = res, post indexed
            add   r3, #4
            ldr   r2, [r1, r3]  @ load next param
            sub   r0, r0, r2    @ r0 = r0 - r2

            cmp   r3, #24
            bne   sub

            bx    lr            @ return (result in r0)
```

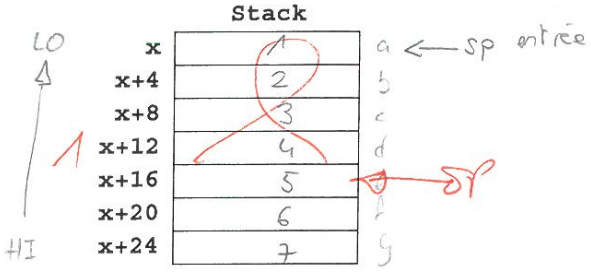
2. Représentez l'état des registres (R0 à R5) et de la pile (position du stack pointer, contenu) à l'entrée et à la sortie de la fonction, pour l'appel ci-dessous :

```
int main() { return subtract (1,2,3,4,5,6,7); }
```

Entrée :

R0	?
R1	?
R2	?
R3	?
R4	?
R5	?

Stack



Sortie :

R0	-26
R1	x (sp)
R2	7
R3	24
R4	?
R5	?

Microprocesseurs 1 & 2: Travail écrit no 3.

8

Problème n° 2 (Interruptions - concept et logiciel)

Sur un système d'exploitation, un client logiciel (mode utilisateur) désire accéder à 3 fonctions (voir prototype ci-dessous) situées dans le noyau (mode superviseur). Ces fonctions ne sont donc accessible qu'en utilisant l'instruction assembleur « **swi** #numéro » ; la fonction 0 est adressée avec le numéro 0, la fonction 1 avec le numéro 1 et la fonction 2 avec le numéro 2.

```
long function0 (long arg1, long arg2);
long function1 (long arg1);
long function2 (long arg1, long arg2, long arg3);
```

Implémentez les segments de code (avec déclaration) permettant d'accéder à chacune de ces fonctions, du côté client et du côté serveur. Du côté client, ces méthodes seront appelées à partir de code écrit en C. Du côté serveur seul le « swi_handler » doit être réalisé.

Pour rappel, « swi » ne cause pas décalage et permet de spécifier jusqu'à 2^{24} identificateurs différents.

client.h

```
#include "system.h"
extern long function0 (long arg1, long arg2);
extern long function1 (long);
extern long function2 (long, long, long);
```

2

client.c

```
long function0 (long arg1,) {
  call_function (0, arg1, arg2, 0)
}
long function1 (long arg1) {
  call_function (1, arg1, 0, 0)
}
long function2 (long arg1, a2, a3) {
  call_function (2, arg1, arg2, arg3)
}
```

pas nickel

system.h

```
extern void init();
extern long call_function (int key, long arg1, long arg2, long arg3);
```

system.s

```
init    :  . . .   @ attach   swi_handler

swi_handler : nop
              stmfd  sp!, {r1-r12, lr}   @ save context

              ldr    r4, [lr-#4]         @ get call instruction
              and    r4, #0x00FFFFFF     @ mask sur SWI #?
              cmp    r4, #0
              beq    function0_sys       @ function 0
              cmp    r4, #1
              beq    function1_sys       @ function 1
              cmp    r4, #2
              beq    function2_sys       @ function 2

              ldmfd  sp!, {r1-r12, pc}^  @ return, r0 = function return value

call_function : @ on génère ici l'interruption SWI #key   ok vous comm' ?
```

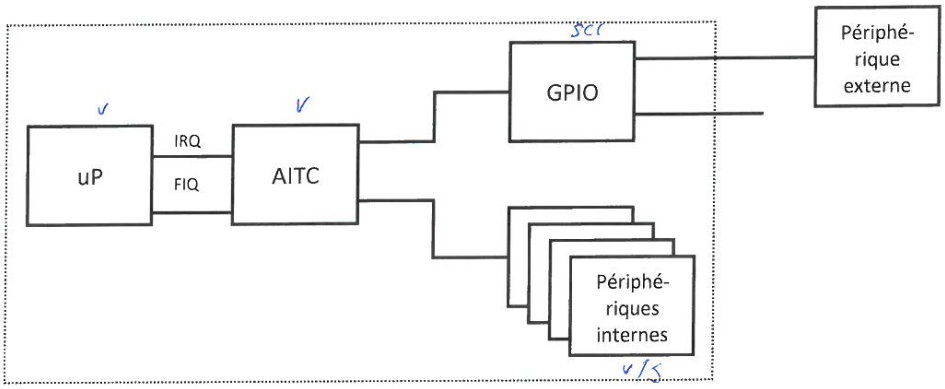
4.0

Microprocesseurs 1 & 2: Travail écrit no 3.

10

Problème n° 3 (Interruption – hardware)

La figure ci-dessous représente le système d'interruption de l'i.MX27 du point de vue HW.



1. Indiquez pour chaque composant du système d'interruption de l'i.MX27 la technique/méthode utilisée pour déterminer l'origine de l'interruption (vecteur)

3

µP : vecteur d'interruption

AITC : vecteur d'interruption

GPIO : mode par scrutation

Périphériques internes : dépend du périphérique/implémentation

2. Décrivez la séquence d'interruption (logicielle) pour l'appel de la routine d'interruption attachée à une entrée du module GPIO au travers du système d'interruption de l'i.MX27, du uP à l'application

5

Le GPIO reçoit l'interruption → envoie à l'AITC → qui la transmet au µProc sur la ligne FIQ ou IRQ, selon configuration

→ le µP va regarder dans sa table de vecteurs, par rapport au type d'interruption reçu quelle routine appelée et va le faire.

Cette routine doit regarder dans l'AITC quel périphérique a levé une interruption, et appeler la routine de traitement correspondant (GPIO ici)

Dans cette dernière, il faudra scruter les périphériques du GPIO pour trouver lequel a généré l'interruption et appeler une routine en conséquence.

3. Indiquez par quel moyen il est possible d'activer et de désactiver les interruptions au niveau du processeur ARM (le core)

2

Il faut modifier le CPSR du processeur.

On a accès à deux Disable Bit dans cpsr_c :

- I = 1 (IRQ désactivées)
- F = 1 (FIQ désactivées)

Microprocesseurs 1 & 2: Travail écrit no 3.

4

Problème n° 4 (Entrées / Sorties)

1. Décrivez le principe de base du mode de traitement des entrées/sorties par interruptions

3 c'est le périphérique qui va signaler au système :
- dès qu'il reçoit des informations
- dès qu'il est prêt à émettre.

Le système n'a plus qu'à recevoir et traiter ces interruptions. ✓

2. Citez les avantages, désavantages et les contraintes du mode de traitement par interruptions par rapport au mode par scrutation

3

⊕ pas de "busy waiting"
le processeur peut faire d'autre chose tant que le périphérique n'est pas prêt ✓

⊖ périphériques plus complexes à réaliser (plus chers)
codage du système de traitement plus crdu ✓

contraintes : utilisation d'un buffer de réception / émission ✓
↳ dimensionnement !
si le périphérique est rapide, il peut submerger le µP d'interruptions ✓

3. Implémentez la routine d'interruption « `void serial_isr()` »permettant la réception et l'émission de caractères pour le périphérique ci-dessous. La méthode « `void put_c(char c)` » permet de placer un caractère dans le tampon de réception et la fonction « int `get_c()` » retourne, si la valeur est positive, le prochain caractère à émettre.

```
#define STAT_TR 1<<1     /* transmitter ready: character could be sent */
#define STAT_RR 1<<0     /* receiver ready: character has been received */
#define CTRL_TE 1<<1     /* transmitter interrupt enable */
#define CTRL_RE 1<<0     /* receiver interrupt enable */
struct serial_ctrl_t {
   uint32_t rxbuf ;      /* registre contenant les données reçues  */
   uint32_t txbuf;       /* registre pour l'émission des données   */
   uint32_t stat;        /* registre de statut voir bits ci-dessus */
   uint32_t ctrl;        /* registre de contrôle du périphérique   */
}* serial = (struct serial_ctrl_t)0x10019000;
```

} activation dans une routine d'initialisation...

```
void serial_isr() {
  if ((serial -> stat && STAT_TR) != 0)
     serial -> txbuf = get_c();          // send
  if ((serial -> stat && STAT_RR) != 0)  // receive
     put_c (serial -> rxbuf)
  // quitter l'interruption ?
}
```

20 ✓

Microprocesseurs 1 & 2: Travail écrit no 3.

10

Problème n° 5 (systèmes d'exploitation)

1. Décrivez la commutation de contexte entre deux threads

Chaque thread verra son contexte sauvé dans un TCB (Thread control Block) contenant ~~ce~~ son ETAT général
- registres, cpsr, ~~...~~ stacks, état ✓

Lors d'une commutation de contexte, le système doit
- sauver l'état de thread actif dans son TCB
- restaurer l'état du thread suivant par rapport à son TCB dans le µP., incluant registres, pile, cpsr ...
- démarrer le thread (état = RUNNING) suivant qui était en état READY
- stopper le thread en cours (état = READY) ✓

2. Implémentez la fonction de transfert de contexte entre deux threads

```
// r0 : tcb of Thread to be stopped
// r1 : tcb of Thread to be started
// r2 : cpsr thread to be started

rtos_transfert :
    stmfd   r0!, {r0-r14}     @ save context from thread to be stopped
    mrs     r3, cpsr
    str     r3, [r0, #4]
    ldmfd   r1!, {r0-r14}     @ restore context from thread to be started
    ldr     r2, [r1, #17*4]
    msr     cpsr_cxfs, r2
    mov     pc, lr
```

```
struct tcb {
    long regs[15];
    long psr;
    enum state_t state;
};
```

3. Citez les 4 états d'un thread

2 TERMINATED, RUNNING, READY, WAITING