4.8 bonus

Systèmes Embarqués 1 & 2: Travail écrit no 3. 36pt 4.6

Nom : [redacted] **Prénom :** [redacted]

Classe : T-2/I-2

Date : 28.04.2017

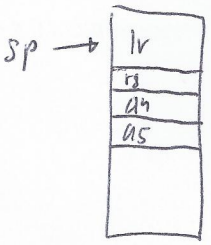
Problème n° 1 (interfaçage C - assembleur)

a) Codez en assembleur la fonction « f » ci-dessous. Note : les « **long** » ont 32 bits.

```
typedef long (*fnct_t) (long p1, long p2);
unsigned f (fnct_t fnct, long a1, long a2, long a3, long a4, long a5)
{return a1 + (a2 << 2) + fnct(a4, a3) + a5;}
```

r0 r1 r2 r3 stack stack

```
push {lr, r8}
add r8, r8, r0      // r8 = a1
mov r8, r1
ldr r1, [sp, #12]   // r1 = a5
add r8, r8, r1      // r8 = a1 + a5   l'ordre n'est pas respecté
lsl r2, #2
add r8, r8, r2      // r8 = a1 + a5 + (a2<<2)
mov r2, r0          // r2 = fnct_t address
ldr r0, [sp, #8]    // r0 = a4
mov r1, r3          // r1 = a3
bl r2
add r0, r0, r8
pop {pc, r8}
```



5 1/2

b) Pour la fonction « foo » avec la signature suivante

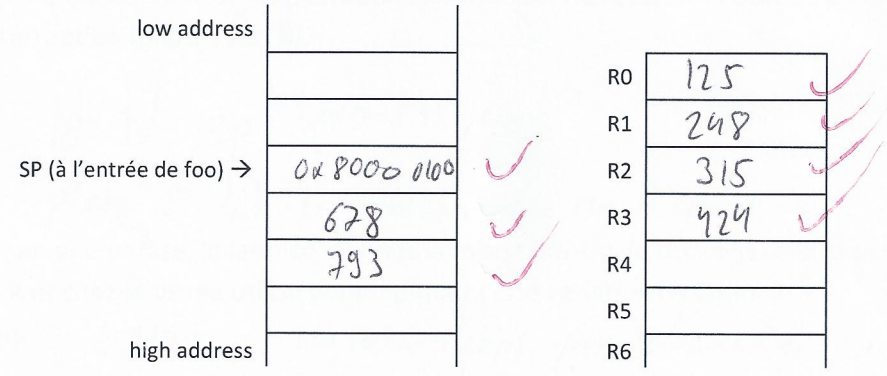
```
extern int foo (int a1, int a2, int a3, int a4, int* a5, int a6, int a7);
```

représentez l'état du processeur - sa pile et ses registres - à l'entrée de la fonction, alors qu'aucune instruction de « foo » n'a encore été exécutée, pour les valeurs des arguments suivants :

r0 r1 r2 r3 stack

```
int res = foo (125, 248, 315, 424, 0x8000'0100, 678, 793);
```

Note : la pile, les registres et tous les arguments sont sur 32 bits 32 bits



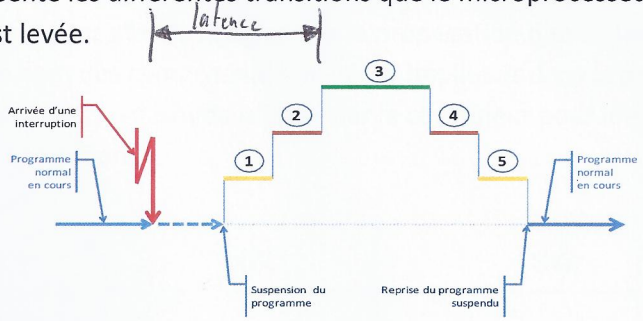
4

10

Systèmes Embarqués 1 & 2: Travail écrit no 3.

Problème n° 2 (Interruptions)

La figure ci-dessous représente les différentes transitions que le microprocesseur et son logiciel effectuent lorsqu'une interruption est levée.



a) Implémentez en assembleur toutes les opérations que l'on doit effectuer dans la phase 2 pour appeler en toute sécurité la routine « `isr (int vector)` » qui sera exécutée de la phase 3.

```
sub lr, #4      oh, mais pas toujours
stmfd sp!, {r0-r12, lr}
bl ...
```

1 1/2

b) Décrivez l'utilité de la table de vecteur, indiquez ce que contient cette table de vecteur et indiquez où celle-ci se situe sur le µP ARM am3358 de TI (µP ARM Cortex-A8).

Elle peut se trouver n'importe où (adresse) mais généralement à l'adresse 0 de la stack ou le SRAM
Elle contient ~~L'adresse de la~~ routine d'interruption à appeler pour un vecteur d'interruption.
utilité ?

1 1/2

c) Implémentez en assembleur les opérations permettant de restaurer l'état du µP après le traitement d'une interruption (phases ~~3~~ 4 et ~~4~~ 5)

```
ldmfd sp!, {r0-r12, pc}^
```

phase 5 faite en hardware donc rien à coder

1

d) Décrivez, en une phrase, la latence, donnez la raison principale pouvant celle-ci peut varier très fortement et citez le terme utilisé pour indiquer cette variation de latence.

variation = gigue raisons = - temps pour désactiver les interruptions (principale)
- temps d'exécution de la plus longue instruction machine

3

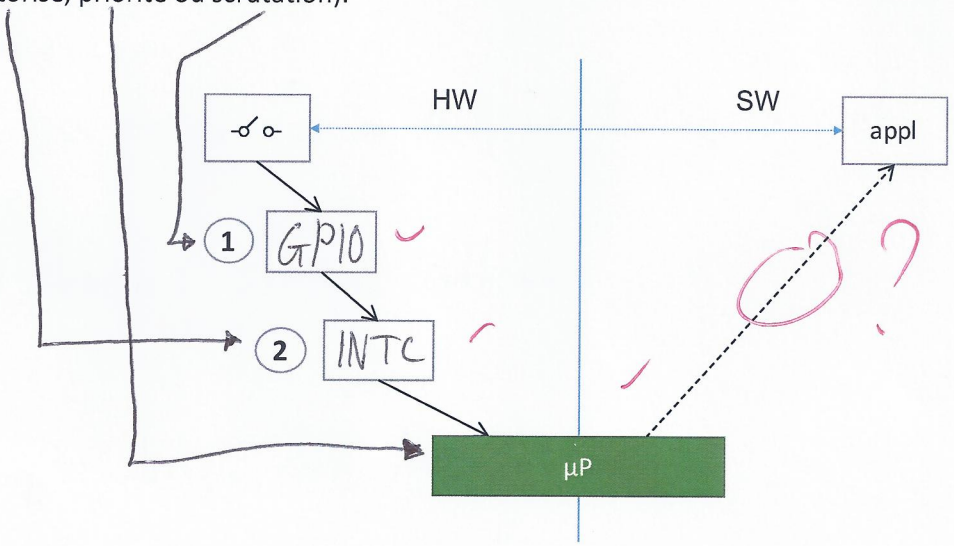
latence = temps entre l'arrivée d'une interruption et le début de la routine qui traite cette interruption

7

Systèmes Embarqués 1 & 2: Travail écrit no 3.

Problème n° 3 (Interruptions)

a) Complétez la figure ci-dessous afin de représenter la propagation d'une alarme HW et son traitement par logiciel en indiquant le nom des composants HW et SW impliqués dans la propagation et le traitement de l'alarme. Indiquez également la méthode utilisée par le contrôleur pour identifier la source d'interruption (vectorisé, priorité ou scrutation).



3

b) Pour les composants HW n° 1 et n° 2 de la figure ci-dessus, décrivez succinctement le mécanisme mis en place par ces composants afin de permettre au logiciel d'identifier la source d'interruption, ceci sachant que chacun de ces 2 composants est capable de traiter plusieurs sources d'interruptions simultanément.

- Composant n° 1 Scrutation (priorité fixe ou tourniquet)

Il va regarder qui a fait l'interruption parmis les I/O et enregistrer l'info ~~dans un de ses registres~~.

2 1/2

- Composant n° 2 interruption vectorisé

Grâce au gpio, il va regarder dans ~~la~~ sa table qu'elle vecteur d'interruption correspond quelle table?

l'INTC est indépendant de GPIO

6

Systèmes Embarqués 1 & 2: Travail écrit no 3.

Problème n° 4 (Interruptions)

a) Sachant que les variables a, b, c, et d stockées en mémoire sont de type « uint32_t», indiquez et argumentez si l'opération ci-dessous est atomique ou n'est pas atomique

1. a += 10 ;
2. b = 10 ;
3. c = a + b
4. d = d * 10 ;

1) pas atomique car on doit load & store a en plus de faire +10

2) atomique : str est suffisant

6

3) ~~pas~~ atomique, comme pour (1)

4) " " " " "

1, 3, 4 ne sont pas atomique car on doit load la valeur des variables avant de pouvoir travailler dessus. Après les avoir load, il pourrait avoir une interruption qui nous bloque et va modifier a, b ou c et quand le programme va reprendre, on continuera de travailler avec les anciennes valeurs.

2) est atomique car on load une adresse donc une interruption, même si elle modifie b, ne fera pas que de mettre +10 après utilisera l'ancienne valeur.

b) Décrivez le mécanisme à mettre en œuvre pour rendre atomique des opérations.

Utiliser des Semaphore / mutex pour protéger ses sections critiques.

2

(8)

Systèmes Embarqués 1 & 2: Travail écrit no 3.

Problème n° 5 (Système d'exploitation - noyau)

a) Implémentez un type en C, pour représenter les états d'un thread pour un mini-noyau coopératif.

1/2 enum state = {"RUNNING", "READY", "TERMINATED", "BLOCKED"};

b) Décrivez en C la structure (struct tcb) permettant de stocker le contexte et l'état d'un thread pour un mini-noyau coopératif. Seuls les attributs indispensables à la gestion du thread doivent être décrits.

1 1/2

```
struct tcb {
    char* state;
    uint32_t register[15];
    uint32_t stack[size];
    tcb* next;
}
```

OK, si pour par

c) Sachant que la variable « struct tcb* running_thread ; » pointe sur le thread en cours d'exécution dans un mini-noyau coopératif, implémentez en C la méthode « void yield() » permettant de passer la main à un autre thread prêt à s'exécuter.

1

```
void yield(){
    running_thread.state = "BLOCKED";
    running_thread.next.state = "RUNNING";
    running_thread = running_thread.next;
    running_thread.state = "RUNNING";
}
```

d) Le sémaphore est un composant indispensable pour tout système d'exploitation.

a. Décrivez succinctement le fonctionnement d'un sémaphore

1

Il permet de synchroniser des sections threads pour qu'ils ne puissent pas faire leur sections critiques si x autre threads sont en section critique.

Par exemple on peut rendre tout un code atomique on le protégeant avec un Sémaphore, ce qui fera que si un thread veut effectuer ce code, aucun autre ne pourra le faire tant qu'il n'aura pas fini (dans ce cas le semaphore est un mutex)

Citez 2 usages possibles des sémaphores sur des systèmes d'exploitation préemptifs

1/2

→ Rendre des code atomique

→ Bloquer les thread quand ils sont pas running

(5)