Sujet informatique embarquée – examen M1

J.-M Friedt, 28 mars 2018

Tous les documents sont autorisés, toutes les connexions à internet sont autorisées mais les communications par téléphone mobile sont **proscrites**. Chaque question sera notée sur 1 point. Il est probablement plus judicieux de persévérer sur un thème que se disperser sur les 3 axes d'investigations si le temps manque.

1 SPI 16-bits sur STM32

Nous avons vu que divers protocoles de communication enrichissent les perspectives d'utilisation de petits cœurs de microcontrôleurs par l'ajout de périphériques riches en fonctionnalités. La carte mémoire compatible avec le protocole Multi-MediaCard (MMC), désormais couramment disponible sous forme Secure Digital (SD), fournit un exemple parfait, avec une capacité de stockage quasi-illimitée accessible par bus SPI. Le bus SPI est relativement rapide, avec quelques Mb/s de débit de transfert, et un petit cœur de microcontrôleur limitera le débit de communication en proposant souvent des transactions par paquets de 8 bits (1 octet) entre lesquels son temps de calcul introduit une latence avant que la prochaine donnée ne puisse être transmise. Dans le cas de données acquises sur 16 bits (par exemples issues d'un convertisseur analogique-numérique), il peut être avantageux de transmettre les données par paquets de 16 bits et ainsi diviser par 2 les latences dues aux calculs sur le microcontrôleur : le STM32 propose une telle fonctionnalité qui peut nous faciliter la vie et augmenter le débit des transactions. Nous allons nous efforcer de le mettre en œuvre ci-dessous.

1. Rappeler la caractéristique d'un bus de communication synchrone, et en quoi il s'oppose à un bus de communication asynchrone. Quel gain le protocole synchrone amène-t-il ?
2. Supposons que le résultat d'un calcul sur un microcontrôleur est un mot `w` codé sur 32 bits. Nous supposons avoir une fonction `spi_send` qui prend en entrée un argument sur 16 bits : proposer le prototype de cette fonction, puis son appel, dans la fonction principale `main()` afin de transmettre `w` sur le bus SPI, en prenant soin de sélectionner les sous-ensembles appropriés de `w` à chaque appel. Combien d'appels à `spi_send` faut-il pour transmettre `w` ?
3. Au cours de nos explorations du STM32, nous avons utilisé la bibliothèque `libopencm3` qui permet d'initialiser le bus SPI au travers de la fonction `spi_init_master()`. En consultant le prototype de la fonction définie dans les fichiers d'entête (`/home/jmfriedt/sat/` contient le compilateur et fichiers d'entêtes pour développer sur ARM Cortex) dans lesquels on cherchera l'occurence de cette fonction, et en recherchant les constantes définies dans ces mêmes fichiers d'entête, proposer les arguments de la fonction `spi_init_master` afin de configurer les transactions par paquets de 16 bits.
4. Si au lieu de passer par la fonction `spi_init_master` de `libopencm3`, nous allons considérer une approche "manuelle" consistant à affecter le bit approprié sur le registre approprié "à la main" (i.e. sans passer par une bibliothèque). Quel registre contient quel bit à cet effet ? On pourra rechercher la solution dans la *datasheet* disponible à `www.st.com/resource/en/reference_manual/dm00180366.pdf` : où se trouve la solution dans ce document ? Donner la séquence de recherche dans ce document qui permet d'identifier exactement l'adresse du registre en question (et non sa position relativement à une autre valeur).
5. Quelle opération logique permet de passer à 1 un bit d'un registre sans toucher aux autres bits ? Proposer la ligne de code en C, qui devra inclure l'adresse du registre adéquat et sa manipulation pour permettre des transactions sur 16 bits. La recherche pourra se faire dans l'arborescence de `libopencm3` qui se trouve quelquepart sous `/home/jmfriedt/sat`. Quelles sont les constantes de `libopencm3` qui permettent de parcourir le même cheminement logique que celui proposé à la lecture de la datasheet dans la question précédente ?

2 Cross-compilation MIPS

Notre objectif est de développer sur une architecture MIPS sans avoir accès au matériel. Nous allons présenter les outils de mise en œuvre d'un environnement d'émulation et de cross-compilation pour MIPS [1]. Ces étapes du travail ne font pas partie de l'examen et ont déjà été effectuées sur vos environnements de travail : le sujet porte exclusivement sur *l'utilisation* de ces outils. Un émulateur permet de pallier la déficience de disponibilité de matériel en exécutant un code compilé pour une certaine architecture cible, ici MIPS, sur l'architecture hôte, ici Intel x86. Qemu fournit de telles fonctionnalités : il s'obtient par `git clone git://github.com/prplfoundation/qemu.git` et se compile sans subtilité dans un répertoire propre (par exemple `build`) par `../configure --target-list=mips64el-softmmu`. Une fois l'émulateur compilé, il nous faut un ensemble cohérent d'outils pour développer sur l'architecture cible, tel que fourni par exemple par `buildroot` : nous en récupérons les sources (`git clone git://git.buildroot.net/buildroot`) et compilons pour la "bonne" architecture (`make qemu_mips64el_malta_defconfig`). Après quelques heures de compilation, nous voici prêts à explorer une nouvelle architecture matérielle.

6. Sachant que la version de `buildroot` pour MIPS est installée dans `/home/jmfriedt/buildroot_mips`, quel est le compilateur qui génère un exécutable contenant des opcodes pour MIPS ?
7. Exporter le chemin des exécutables selon la question ci-dessus, et cross-compiler un programme affichant "Hello World" en ligne de commande sur l'émulateur Qemu (login=root, pas de mot de passe). Pour ce faire, nous lançons GNU/Linux sur l'émulateur au moyen de

1. D. Sweetman, *See MIPS Run, 2nd Edition*, The Morgan Kaufmann Series in Computer Architecture and Design (2006)

```
qemu-system-mips64el -M malta -cpu I6400 -kernel /home/etudiant/vmlinux \
-serial stdio -drive file=/home/etudiant/rootfs.ext2,format=raw -append "root=/dev/hda" \
-virtfs local,path=/home/etudiant/nfs_arma,mount_tag=host0,security_model=passthrough,id=host0
```

qui donne accès au répertoire du PC `/home/etudiant/nfs_arma` comme un disque virtuel accessible depuis `/mnt` de l'émulateur (cf `/etc/fstab` de l'image exécutée sur l'émulateur). Démontrer à l'enseignant (présentation du code et de son exécution) le bon fonctionnement de votre programme sur l'émulateur MIPS. ATTENTION : ne jamais tuer un programme par CTRL-C, qui induit la mort de qemu et non de l'application qui s'exécute dans l'émulateur.

8. Après avoir identifié l'emplacement des sources du noyau fourni par buildroot (fournir sa version en réponse à cette question), proposer et compiler un module qui se contente d'afficher "Hello" au chargement et "Bye" au déchargement. Démontrer son bon fonctionnement sur l'émulateur.
9. Un programme trivial de la forme `int main() {volatile long i;i=365000;while (1) {}}` est compilé pour trois architectures : x86 (`gcc`), ARM (`arm-buildroot-linux-uclibcgnueabihf-gcc` tel que nous l'avons utilisé pour A13), et MIPS. Les diverses déclinaisons de `objdump` nous informent du code assembleur correspondant au désassemblage de l'exécutable. Nous obtenons respectivement pour x86, ARM et MIPS les résultats suivants :

```
0000844c <main>:
    844c:       e52db004        push    {fp}            ; (str fp, [sp, #-4]!)
    8450:       e28db000        add     fp, sp, #0
    8454:       e24dd00c        sub     sp, sp, #12
    8458:       e30931c8        movw    r3, #37320      ; 0x91c8
    845c:       e3403005        movt    r3, #5
    8460:       e50b3008        str     r3, [fp, #-8]
    8464:       eafffffe        b       8464 <main+0x18>
```

pour ARM,

```
0000000000000660 <main>:
 660:   55                      push   %rbp
 661:   48 89 e5                mov    %rsp,%rbp
 664:   48 c7 45 f8 c8 91 05    movq   $0x591c8,-0x8(%rbp)
 66b:   00
 66c:   eb fe                   jmp    66c <main+0xc>
 66e:   66 90                   xchg   %ax,%ax
```

pour intel x86, et

```
0000000120000aa0 <main>:
   120000aa0:   67bdffe0        daddiu  sp,sp,-32
   120000aa4:   ffbe0018        sd      s8,24(sp)
   120000aa8:   03a0f025        move    s8,sp
   120000aac:   3c020005        lui     v0,0x5
   120000ab0:   344291c8        ori     v0,v0,0x91c8
   120000ab4:   ffc20000        sd      v0,0(s8)
```

pour MIPS.
Quelle différence "évidente" observez vous entre les opcodes associées aux instructions assembleurs sollicitées dans chaque cas. Analysez les divers codes et expliquez l'affectation de la variable selon le code C d'origine (sans connaître le sens de chaque mnémonique, leur abréviation et le résultat de la question suivante doit permettre de suivre la séquence de traitement). On pourra s'aider dans cette analyse en identifiant la taille de la variable affectée et la taille des opcodes selon les architectures ? Que vous inspire cette différence entre les 3 codes ?

10. Quelle conséquence cette différence peut-elle avoir sur l'unité arithmétique et logique (ALU) ?
11. Comment classer, au sein des familles CISC et RISC, ces trois processeurs ?

3 Questions de cours

12. Que vaut 0xFFF9+1 en hexadécimal et en décimal ?
13. Que vaut 64 + 0x40 en hexadécimal et en décimal ?
14. Que vaut 64|0x40 en hexadécimal et en décimal ?
15. Quelle commande Unix permet de connaître les messages du système, et en particulier être informé du chargement d'un module noyau pour Linux.
16. Quelle est la fonction utilisée par le noyau Linux pour afficher un message dans le fichier de log du système ou la console, et ainsi communiquer une information à l'utilisation ?
17. Comment justifier l'utilisation d'un système d'exploitation sur un système embarqué ?
18. Quel(s) inconvénient(s) voyez vous à l'utilisation d'un système d'exploitation sur un système embarqué ?
19. Quelle étape de compilation traite la macro `#define x 1+2`
20. Suite à la question précédente, qu'affiche `printf("%d\n",x*3);` ?

Solutions

1. partage de l'horloge entre les interlocuteurs dans un bus synchrone, hypothèse de la même fréquence d'échantillonnage (*baudrate*) dans une communication asynchrone qui ne partage pas l'horloge. Gain de vitesse au détriment d'un signal (fil) additionnel.
2. `short spi_send(short s);` et `int main(){spi_send((w&0xffff0000)>>16;spi_send(w&0xffff);}` qui nécessite donc deux appels à la fonction `spi_send();` .
3. `#define SPI_CR1_DFF_16BIT (1 << 11)` dans `sat/arm-none-eabi/include/libopencm3/stm32/common/spi_common_l1f124.h` forme le cinquième argument de `spi_init_master()` tel que défini dans `sat/arm-none-eabi/include/libopencm3/stm32/common/spi_common_all`.
4. bit DFF, page 716 (tab. 25.7.1) de DM00180366.pdf qui est affecté par un masque sur un bit du SPI par `*(short*)(adresse)|=(1<<11);`. Il reste à trouver `adresse` : `SPI_CR1` est à l'offset 0 par rapport aux registres SPI en 0x4001 3000 - 0x4001 33FF d'après p.39 de DM00180366, cohérent avec les définitions de libopencm3 ...
5. ... `*(short*)(0x40013000)|=(1<<11);` passe le bit à 1 par un OU logique avec la valeur initiale du registre sans toucher aux valeurs des autres bits. L'adresse se trouve par

```
#define SPI1_BASE          (PERIPH_BASE_APB2 + 0x3000)
#define PERIPH_BASE_APB2   (PERIPH_BASE + 0x10000)
#define PERIPH_BASE        (0x40000000U)
```

4 Cross-compilation MIPS

6. `mips64el-buildroot-linux-uclibc-gcc`
7. `mips64el-buildroot-linux-uclibc-gcc -o hello hello.c` sur le PC pour générer le binaire qui s'exécute sur l'émulateur MIPS.
8. `output/build/linux-linux-4.9.6/`
9. x86 présente un nombre variable d'octets par instruction + arguments alors que MIPS et ARM présentent des instructions de taille fixes, toujours de 32 bits, incluant l'opcode et les arguments. Par conséquent, dans le second cas l'affectation d'une variable sur 32 bits doit se scinder en deux opérations, affectation du mot de poids fort puis du mot de poids faible, de 16 bits chacun. En effet, 365000=0x591C8 qui se divise en l'affectation de 5 dans le mot de poids fort et 0x91C8 dans le mot de poids faible sur ARM et MIPS, tandis que l'affectation se fait en une opération assembleur sur x86.
10. Dans une architecture qui manipule des instructions + arguments de taille fixe, l'ALU peut lire les instructions 4 octets par 4 octets, en sachant que les instructions sont nécessairement alignées sur des adresses multiples de 32 bits. Sur une architecture dont la taille des arguments varie avec la commande, l'ALU doit d'abord lire le premier octet correspondant à la commande, et en fonction de cet opcode, choisir le bon nombre d'octets contenant les arguments. L'architecture de l'ALU est donc plus complexe puisque le nombre d'arguments doit s'adapter à chaque commande.
11. MIPS et ARM sont des processeurs RISC, x86 est un processeur CISC. RISC permet d'effectuer les mêmes opérations en moins de cycle d'horloge (ALU plus simple permet d'être cadencée plus rapidement), au détriment de devoir découper des opérations simples en plusieurs étapes.
12. 0xFFF9+1=65529+1=65530=0xFFFA.
13. 64 + 0x40=64+64=128
14. 64|0x40=64
15. `dmesg`
16. `printk`
17. ordonnanceur, système de fichiers, pile TCP/IP
18. ressources additionnelles nécessaires (mémoire + latences variables de l'ordonnanceur)
19. préprocesseur
20. 7