MSP430 TP3 : Pile et conventions d'appel

Introduction

Dans ce TP, on va s'intéresser aux conventions d'appel, c'est à dire à la façon dont les différentes fonctions d'un programme communiquent entre elles. Pour cela, on va écrire en assembleur et en C plusieurs programmes de plus en plus sophistiqués. Créez un un nouveau répertoire TP3 et mettez tous vos nouveaux fichiers là-dedans.

À savoir : le rôle de la convention d'appel

Lorsqu'on se situe au niveau assembleur, le langage reflète directement l'architecture de la machine : registres, instructions, etc. On ne retrouve donc pas les notions classiques des langages de programmation : variables, boucles, conditionnelles, qu'il faut implémenter explicitement en utilisant le jeu d'instructions disponibles.
Pour les appels de fonction (aka «procédure», «méthode», «sous-programme») la plupart des architectures offrent des instructions dédiées. Ces instructions s'appellent par exemple `CALL` et `RET` sur msp430 (et sur x86), ou `BL` et `BX` sur ARM. Ainsi, `CALL #func` saute vers la fonction `func`, et `RET` retourne vers la fonction appelante. Mais ces instructions ne s'occupent pas particulièrement des arguments ni des résultats. Afin que nos différentes fonctions puissent communiquer entre elles, il faut donc se mettre d'accord, entre appelant et appelé, sur la façon de se passer les paramètres et de rendre les résultats : c'est la *convention d'appel*.

1 Appels de fonctions en assembleur

Exercice 1 Recopiez le programme ci-dessous dans un fichier `tp3.s`. Récupérez votre *driver* du TP précédent, compilez tout ça et chargez le programme sur la carte. Identifiez dans le code machine, les zones qui correspondent aux différentes fonctions.

```
.section .init9

main:
        /* initialisation de l'ecran */
        call #lcd_init

        /* emballage des arguments */
        MOV #6, R15
        MOV #7, R14

        call #mult

        /* deballage de la valeur de retour de mult
           qu'on re-emballe comme argument pour l'appel suivant */
        MOV R13, R15

        call #lcd_display_number

/* infinite loop */
done:
        jmp done

mult:
        MOV #0, R13

        ret
```

Rappel : la chaîne de compilation

Vous avez déjà vu toutes les commandes utiles dans les TP précédents :

— Pour *assembler* un programme ASM vers du langage machine :
```
msp430-as -mmcu=msp430fg4618 -o truc.o truc.s
```
— pour *compiler* un programme C et l'assembler dans la foulée (on obtient aussi du code machine) :
```
msp430-gcc -mmcu=msp430fg4618 -Wall -Werror -O1 -c -o truc.o truc.c
```
— Pour faire l'*édition de liens* entre un ou plusieurs modules et obtenir un exécutable :
```
msp430-gcc -mmcu=msp430fg4618 -mdisable-watchdog -o bidule.elf truc1.o truc2.o
```
— Pour *désassembler* un exécutable et obtenir un fichier texte lisible à l'oeil nu :
```
msp430-objdump -d machin.elf > machin.lst
```

Exercice 2 On veut maintenant que la fonction `mult` calcule la multiplication $a \times b$. Écrivez le code nécessaire pour additionner a sur lui-même b fois. Notre *convention d'appel* sera la suivante :

— à l'entrée dans la fonction, R14 et R15 contiennent les arguments, qui sont supposés strictement positifs et pas trop grands [1]

— au retour de la fonction, R13 contient la valeur de retour.

Faites valider par un enseignant.

Commentaire Vous aurez remarqué que notre programme invoque successivement `mult` puis une fonction C. Comme cette dernière aura été compilée par GCC, elle ne suivra pas particulièrement notre convention d'appel maison, mais plutôt celle du compilateur [2]. Les deux conventions d'appel sont différentes, mais similaires. En particulier, `lcd_display_number()` attend son paramètre *number* dans R15. C'est pourquoi notre programme copie $a \times b$ de R13 vers R15 avant d'invoquer `lcd_display_number()`.

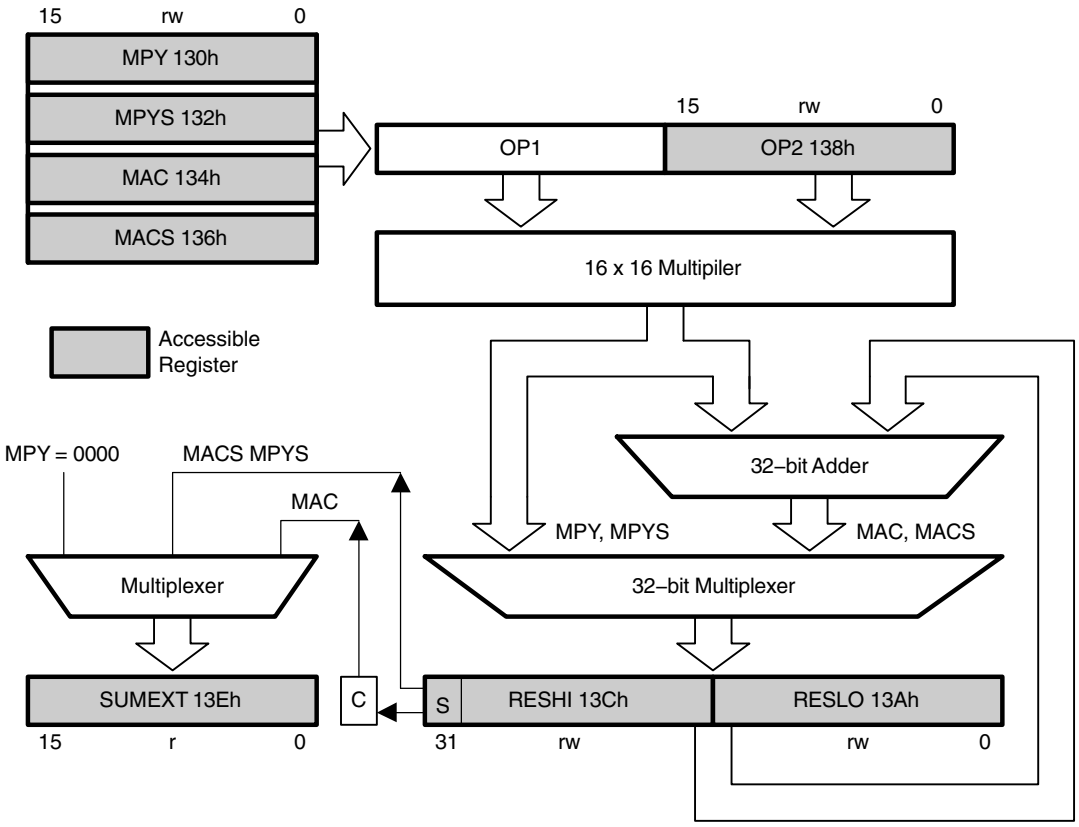
Le multiplieur matériel

Dans l'exercice suivant, on va implémenter une seconde version de mult, cette fois-ci en utilisant un accélérateur matériel présent dans notre MSP430 et dédié aux calculs de multiplication. On vous a copié-collé ci-dessous la documentation de ce multiplieur.

Extrait de la documentation : msp430x4xx.pdf page 346

Introduction The hardware multiplier is a peripheral and is not part of the MSP430 CPU. This means that its activities do not interfere with the CPU activities. The multiplier registers are peripheral registers that are loaded and read with CPU instructions.
The module performs 16 × 16, 16 × 8, 8 × 16, and 8×8 bit operations. The module is capable of supporting signed and unsigned multiplication, as well as signed and unsigned multiply-and-accumulate operations. The result of an operation can be accessed immediately after the operands have been loaded into the peripheral registers.
The hardware multiplier block diagram is shown below.

1. l'idée est de simplifier l'exercice : on suppose que a et b sont assez petits pour que $a \times b$ tienne sur 16 bits
2. pour les plus curieux, la convention d'appel de notre compilateur : `http://mspgcc.sf.net/manual/x1248.html`



Hardware Multiplier Operation The hardware multiplier supports unsigned multiply, signed multiply, unsigned multiply accumulate, and signed multiply accumulate operations. The type of operation is selected by the address the first operand is written to.

The hardware multiplier has two 16-bit operand registers, OP1 and OP2, and three result registers, RESLO, RESHI, and SUMEXT. RESLO stores the low word of the result, RESHI stores the high word of the result, and SUMEXT stores information about the result. The result is ready in three MCLK cycles and can be read with the next instruction after writing to OP2, except when using an indirect addressing mode to access the result. When using indirect addressing for the result, a NOP is required before the result is ready.

Operand Registers The operand one register OP1 has four addresses, shown in the table below, used to select the multiply mode. Writing the first operand to the desired address selects the type of multiply operation but does not start any operation. Writing the second operand to the operand two register OP2 initiates the multiply operation. Writing OP2 starts the selected operation with the values stored in OP1 and OP2. The result is written into the three result registers RESLO, RESHI, and SUMEXT.

Repeated multiply operations may be performed without reloading OP1 if the OP1 value is used for successive operations. It is not necessary to re-write the OP1 value to perform the operations.

Register	Short Form	Register Type	Address	Initial State
Operand one - multiply	MPY	Read/write	0130h	Unchanged
Operand one - signed multiply	MPYS	Read/write	0132h	Unchanged
Operand one - multiply accumulate	MAC	Read/write	0134h	Unchanged
Operand one - signed multiply accumulate	MACS	Read/write	0136h	Unchanged
Operand two	OP2	Read/write	0138h	Unchanged
Result low word	RESLO	Read/write	013Ah	Undefined
Result high word	RESHI	Read/write	013Ch	Undefined
Sum Extension register	SUMEXT	Read	013Eh	Undefined

Exercice 3 Mettez de côté votre version précédente de `mult`, et réécrivez-la en utilisant le multiplieur matériel. On garde les mêmes hypothèses : *a* et *b* positifs et suffisamment petits pour que leur produit tienne sur 16bits.

2 Étude du fonctionnement de la pile

Dans cette partie, on va s'intéresser aux mécanismes de la pile d'exécution. Répondez aux questions ci-dessous, en vous aidant si nécessaire des morceaux de documentation reproduits en pages suivantes.

Exercice 4 Écrivez un programme qui fait des `PUSH` et des `POP`, et exécutez-le en mode pas-à-pas. Dans quelle direction croît la pile : vers les petites adresses, ou vers les grandes adresses ?

Exercice 5 Le registre SP pointe-t-il toujours vers la première case vide au-dessus de la pile, ou sur la dernière case pleine en sommet de pile ?

Exercice 6 Le registre SP nous indique où est le sommet de la pile. Mais quelle est l'adresse de l'autre extrémité, la *base* de la pile ? Trouvez la, puis dessinez la région occupée par la pile sur la memory map du TP précédent.

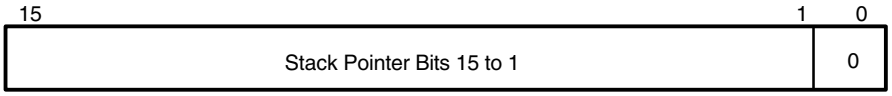
Extrait de la documentation : msp430x4xx.pdf page 45

3.2.2 Stack Pointer (SP)

The stack pointer (SP/R1) is used by the CPU to store the return addresses of subroutine calls and interrupts. It uses a predecrement, postincrement scheme. In addition, the SP can be used by software with all instructions and addressing modes. Figure 3–3 shows the SP. The SP is initialized into RAM by the user, and is aligned to even addresses.

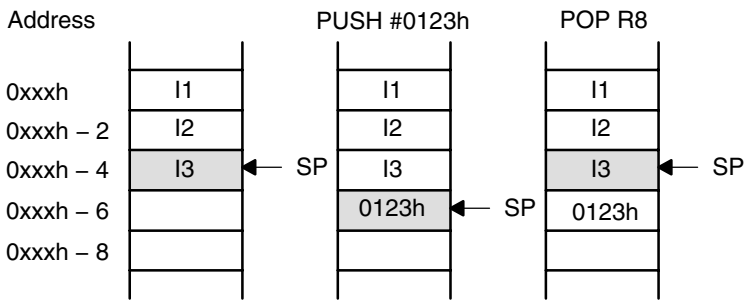
Figure 3–4 shows stack usage.

Figure 3–3. Stack Pointer




```
MOV   2(SP),R6  ; Item I2 -> R6
MOV   R7,0(SP)  ; Overwrite TOS with R7
PUSH  #0123h    ; Put 0123h onto TOS
POP   R8        ; R8 = 0123h
```

Figure 3–4. Stack Usage



PUSH[.W] Push word onto stack
PUSH.B Push byte onto stack

Syntax

```
PUSH    src    or    PUSH.W    src
PUSH.B  src
```

Operation

SP – 2 → SP
src → @SP

Description The stack pointer is decremented by two, then the source operand is moved to the RAM word addressed by the stack pointer (TOS).

Status Bits Status bits are not affected.

Mode Bits OSCOFF, CPUOFF, and GIE are not affected.

Example The contents of the status register and R8 are saved on the stack.

```
PUSH    SR      ; save status register
PUSH    R8      ; save R8
```

Example The contents of the peripheral TCDAT is saved on the stack.

```
PUSH.B  &TCDAT  ; save data from 8-bit peripheral module,
                ; address TCDAT, onto stack
```

Note: The System Stack Pointer

The system stack pointer (SP) is always decremented by two, independent of the byte suffix.

Extrait de la documentation : msp430x4xx.pdf page 94

*** POP[.W]** Pop word from stack to destination
*** POP.B** Pop byte from stack to destination

Syntax
```
POP     dst
POP.B   dst
```

Operation
```
@SP  –> temp
SP + 2  –> SP
temp –> dst
```

Emulation `MOV @SP+,dst` or `MOV.W @SP+,dst`
Emulation `MOV.B @SP+,dst`

Description The stack location pointed to by the stack pointer (TOS) is moved to the destination. The stack pointer is incremented by two afterwards.

Status Bits Status bits are not affected.

Example The contents of R7 and the status register are restored from the stack.

```
POP     R7      ; Restore R7
POP     SR      ; Restore status register
```

Example The contents of RAM byte LEO is restored from the stack.

```
POP.B   LEO     ; The low byte of the stack is moved to LEO.
```

Example The contents of R7 is restored from the stack.

```
POP.B   R7      ; The low byte of the stack is moved to R7,
                ; the high byte of R7 is 00h
```

Example The contents of the memory pointed to by R7 and the status register are restored from the stack.

```
POP.B   0(R7)   ; The low byte of the stack is moved to the
                ; the byte which is pointed to by R7
                : Example:  R7 = 203h
                ;           Mem(R7) = low byte of system stack
                : Example:  R7 = 20Ah
                ;           Mem(R7) = low byte of system stack
POP     SR      ; Last word on stack moved to the SR
```

Note: The System Stack Pointer

The system stack pointer (SP) is always incremented by two, independent of the byte suffix.

3 Implémentation d'une fonction récursive

La raison d'être d'une pile d'exécution est de sauvegarder des valeurs (avec PUSH) que l'on veut pouvoir restaurer plus tard (avec POP). Ainsi, on peut empiler temporairement une variable, et réutiliser le registre qu'elle occupait. C'est particulièrement utile dans le contexte de la récursivité, où une même variable va exister simultanément en plusieurs exemplaires.

Exercice 7 Écrivez (en assembleur) une fonction récursive qui calcule la factorielle : `fact(0)`=1, et `fact(n) = n`×`fact(n-1)` pour `n`>0. Pour la multiplication, vous pouvez réutiliser l'une ou l'autre de vos implémentations de `mult`, ou accéder directement au multiplieur matériel. Pour la mise au point, exécutez

votre programme en mode pas-à-pas avec mspdebug. Pensez à écrire explicitement les invariants vérifiés à divers endroits de votre programme.
Faites valider par un enseignant.

4 Compilation séparée et conventions d'appel

À savoir : *scratch registers* VS *preserved registers*

Dans l'exercice précédent, vous avez sauvegardé la valeur de n sur la pile, le temps de calculer `fact(`$n-1$`)`, et vous l'avez restaurée ensuite. En effet, le registre R15 qui contient n au début de `fact(`n`)` doit être libéré pour y stocker $n-1$. Ce raisonnement peut se généraliser aux autres registres : avant d'appeler une fonction quelconque, je dois empiler tous les registres qui contiennent mes variables, car ils risquent d'être écrasés (en anglais *clobbered registers*) par cette fonction.
Cette nécessité est dûe au fait que les registres sont pour l'instant considérés *scratch* : c'est à dire, chaque fonction peut s'en servir comme bon lui semble. Si un appelant veut conserver une certaine valeur, c'est à lui de la sauvegarder avant l'appel, et de la restaurer après l'appel. On dit ainsi que les registres sont *caller-saved*. Cette approche a l'inconvénient de causer des sauvegardes excédentaires : dans le doute, je suis obligé de sauvegarder tous mes registres précieux avant d'appeler une fonction, alors que celle-ci ne les touche peut-être pas du tout.
L'approche inverse consisterait à obliger chaque fonction à rendre tous les registres dans le même état qu'elle les a trouvés en arrivant (on parle alors de registres *callee-saved*). Ainsi, l'appelant n'a pas besoin de se préoccuper de sauvegarde : tous ses registres sont automatiquement *préservés* (en anglais *preserved registers*) malgré les appels. Mais au fond le problème reste le même : si ça se trouve, une fonction appelée va consciencieusement sauvegarder et restaurer un registre que la fonction appelante n'utilisait pas.

En général, on coupe donc la poire en deux : une partie des registre sera considérée *préservée* i.e. *callee-saved* et une autre partie sera considérée *scratch*, i.e. *caller-saved*. Typiquement, les registres scratch serviront également à communiquer entre fonction appelante et fonction appelée : passage des paramètres et des valeurs de retour. La convention d'appel et les conventions d'usage des registres sont des éléments importants de l'ABI (*Application Binary Interface*) d'un système. Au niveau langage machine, deux morceaux de code ne pourront fonctionner ensemble que s'ils respectent la même ABI.

Dans cette partie, on va observer de plus près l'ABI de notre compilateur.[3] Celle-ci consiste à passer les arguments via registres si on en a quatre ou moins, et à utiliser la pile à partir du cinquième. Pour mettre en évidence ce fonctionnement, on va donc invoquer une fonction avec plein de paramètres.

Exercice 8 Recopiez le programme ci-dessous dans un nouveau fichier `sum.c`. Compilez-le, désassemblez l'exécutable, et ouvrez le listing dans un éditeur de texte.

```
#include "lcd.h"

int sum_eight(int a, int b, int c, int d, int e, int f, int g, int h)
{
    int x=a+b+c+d+e+f+g+h;

    return x;
}

int main(void)
{
    volatile int x=0;

    lcd_init();

    x=sum_eight(100,200,300,400,500,600,700,800);
```

3. Pour les plus curieux, l'ABI de mspgcc est documentée là : `http://mspgcc.sourceforge.net/manual/c1225.html`

```
    lcd_display_number(x);

    for(;;);
}
```

Exercice 9 Mettez un point d'arrêt (`setbreak <ADDR>`) sur l'entrée dans la fonction `sum_eight()`, et exécutez le programme jusque là. Recopiez le contenu du processeur dans le tableau ci-dessous.

R0	R4	R8	R12
R1	R5	R9	R13
R2	R6	R10	R14
R3	R7	R11	R15

Exercice 10 Examinez le contenu de la mémoire (`md <ADDR> <LEN>`) et recopiez ci-dessous le contenu de la pile. Chaque ligne représente un mot de 16 bits, donc deux octets. Attention, le MSP430 est une architecture *little-endian*[4] : en mémoire, l'octet de poids faible vient *avant* l'octet de poids fort du même mot. Dans la colonne «divers», vous indiquerez la provenance de chaque valeur : quelle est l'instruction qui a écrit cette valeur ici.

addresse	hexa	décimal	divers

5 Facultatif : fonction variadique

Certaines fonctions, par exemple *printf()*, ont un nombre variable d'arguments. En général, la convention d'appel fait une exception pour ces fonctions-là. La solution classique est de passer tous les arguments via la pile d'exécution.

4. D'ailleurs, allez lire `https://fr.wikipedia.org/wiki/Endianness` si ce n'est pas déjà fait.

Exercice 11 En assembleur, écrivez une fonction `sum_many` qui attend un premier argument entier dans R15, et tous les autres sur la pile. R15 indique le nombre d'arguments empilés. Cette fonction calcule la somme de tous les arguments empilés, et renvoie cette somme dans R15. On pourra donc brancher cette fonction dans le programme suivant :

```
.section .init9
main:
    call #lcd_init

    push #100
    push #200
    push #300
    push #400
    push #500
    push #600
    push #700
    push #800
    push #900
    push #1000
    mov #10, r15
    call #sum_many
    add #20, r1 /* stack cleanup */
    /* return value R15 becomes first argument of next call */
    call #lcd_display_number

loop:
    jmp loop
```