TP4, TC-BED, ez430-RF2500
Pile et convention d'appel

October 1, 2019

But du TP

Nous revenons un peu "en arrière", c'est à dire vers une programmation plus bas niveau pour comprendre un peu mieux le mécanisme de la pile utilisée à l'exécution ainsi que ce que l'on appelle les *convention d'appel* (ABI pour Applicaction Binay Interface).

1 Retour sur les outils proposés par `mspdebug`

L'outil `mspdebug` permet de contrôler très précisément ce qu'il se passe dans la mémoire du `MSP430`, mémoire incluant les registres de contrôle des périphériques et les registres du CPU.

Exercice 1 Démarrez mspdebug en tapant la ligne commande suivante:

`mspdebug rf2500`

Vous devez obtenir une série d'informations techniques compliquées, puis une liste des commandes disponibles, et enfin un *prompt* de la forme `(mspdebug)` en début de ligne. Commencez par effacer complètement la puce en tapant dans mspdebug la commande `erase`.

On va maintenant se servir de `mspdebug` pour allumer et éteindre la diode LED0 (rouge) en écrivant directement dans la mémoire. Pour initialiser la diode, on rappelle qu'il faut écrire la valeur 0 à l'adresse 0x26 (P1SEL) puis 3 à l'adresse Ox22 (P1DIR) et enfin 1 à l'adresse 0x21 (P1OUT) .

Exercice 2 Toujours dans mspdebug, tapez `help mw` et lisez l'aide de la commande *memory write*. Remarquez au passage que vous pouvez aussi taper `help` tout court pour obtenir la liste des commandes disponibles, et `help` *bidule* pour obtenir de l'aide sur la commande *bidule*.

Exercice 3 Faites s'allumer et s'éteindre la diode quelques fois, faites allumer et éteindre les deux diodes en même temps. Constatez le changement dans la mémoire en utilisant la commande `md`, faites valider par l'enseignant

2 Appels de fonctions en assembleur

Exercice 4 Recopiez le programme ci-dessous dans un fichier `tp4.s`. Compilez tout ça et chargez le programme sur la carte. exécutez le et avec la commande `regs` de `mspdebug` vérifiez que les registres ont bien les valeurs que vous attendez.

```
.section .init9

main:

        /* mise en place des arguments */
        MOV #6, R15
        MOV #7, R14
        call #mult
```

```
        /* infinite loop */
done:
        jmp done

mult:
        MOV #41, R13

        ret
```

Évidement ce programme n'est pas correct: la multiplication renvoit systématiquement 41 dans le registre 13. Nous allons écrire une multiplication correcte.

Exercice 5 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. L'algorithme sera simplement:

$res = 0$
$cpt = b$
tant que $(cpt! = 0)$
    $res = res + a$
    $cpt = cpt - 1$
fin tant que

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.

Remplacer le code `MOV #41, R13` par votre code et vérifiez que votre multiplication fonctionne avec la commande `regs`. Faites valider par un enseignant.

3 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 6 Mettez un point d'arret sur votre programme au début (`setbreak 0x08XXX` ou `0x08XXX` est l'adresse de l'étiquette main). executez pas a pas et surveillez jusqu'à avant l'execution du `CALL mult`.

- étudiez l'état de la pile avant l'appel CALL
- étudiez l'état de la pile après l'appel CALL, quelle est l'utilité du 38 80 présent en somment de pile? Combien d'octet l'appel de CALL as-t'il ajouté (ou retranché) à la pile?

Exercice 7 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 8 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 ?

[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, on a rappelé en fin de sujet (p 8) les base de l'assembleur MSP430 et les instructions assembleur disponible.

Exercice 9 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.

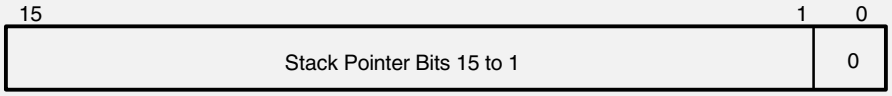
Extrait de la documentation: msp430x4xx.pdf page 45

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.

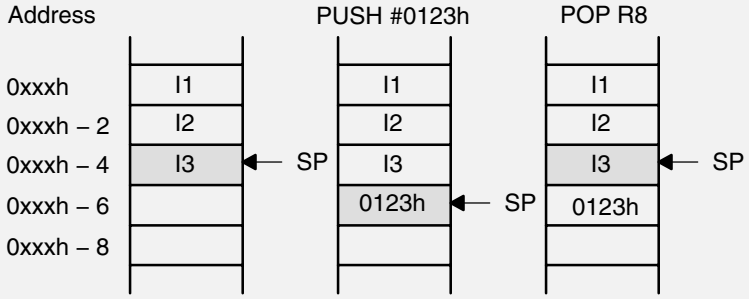
Figure 3–4 shows stack usage.

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



Extrait de la documentation: msp430x4xx.pdf page 95

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

Instruction Set

*** 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.

A Annexe: rappel des instruction du MSP

Utile pour le TP: La syntaxe ASM du msp430

Vous avez déjà rencontré ce jeu d'instructions en TD, mais avec une syntaxe un peu simplifiée. On vous présente ici la syntaxe que vous allez devoir utiliser en TP.

Opérations La plupart des instructions est de la forme `OPCODE SRC, DST`. OPCODE est l'opération souhaitée, par exemple ADD, XOR, MOV, etc. La liste complète est donnée page 8. SRC et DST indiquent les opérandes sur lesquels travailler. Chaque opérande est de l'une des formes suivantes:

- un nom de registre: `R7`, `R15`... (utilisez les numéros, pas de «`SP`» ni «`PC`» etc.)
- une constante immédiate: `#42`, `#0xB600`...
- une case mémoire désignée par son addresse: `&1234`, `&0x3100`...

Par exemple, l'instruction `ADD &1000, R5` calcule la somme de R5 et de la valeur contenue dans la case d'adresse 1000, et range le résultat dans R5. Attention, certaines combinaisons n'ont pas de sens, et seront rejetées par l'assembleur avec un message d'erreur. Par exemple l'instruction `MOV R8, #36` ne veut rien dire.
Certaines instructions travaillent sur un seul opérande, et ont donc une syntaxe légèrement différente. Par exemple `INV DST` inverse chacun des bits de DST, ou `CLR DST` met DST à zéro. Reportez-vous à la liste page 8 pour plus de détails, et/ou à la doc: msp430x4xx.pdf page 61 et suivantes.

Drapeaux Certaines instructions, notamment les opérations arithmétiques et logiques, modifient le registre d'état (`R2`, cf TP1), en particulier les drapeaux Z, N, C, V:

- Z est le *Zero bit*. Il passe à 1 lorsque le résultat d'une opération est nul, et il passe à 0 lorsqu'un résultat est non-nul.
- N est le *Negative bit*. Il passe à 1 lorsque le résultat d'une opération est négatif (en complément à deux) et il passe à 0 lorsqu'un résultat est non-négatif.
- C est le *Carry bit*. Il passe à 1 lorsqu'un calcul produit une retenue sortante, et il passe à 0 lorsqu'un calcul ne produit pas de retenue sortante.
- V est le *Overflow bit*. Il est mis à 1 lorsque le résultat d'une opération arithmétique déborde de la fourchette des valeurs signées (en complément à deux), et à 0 sinon.

La liste page 8 détaille l'effet de chaque instruction sur les quatre drapeaux: un tiret lorsque le drapeau n'est pas affecté, un 1 ou un 0 lorsque le drapeau passe toujours à une certaine valeur, et une étoile lorsque l'effet sur le drapeau dépend du résultat.

Sauts conditionnels Les instructions de branchement sont de la forme `JUMP label`. Le saut peut être soit inconditionnel (instruction `JMP`), soit soumis à une condition sur les drapeaux. Par exemple, l'instruction `JNZ label` est un *Jump if Non-Zero*: elle sautera vers *label* si et seulement si le bit `Z` est faux.

Opérandes «word» ou «byte» Chaque instruction peut travailler sur des mots de 16 bits, ou sur des octets. Il faut pour cela on préciser `OPCODE.B`. Par exemple, l'instruction `MOV.B R10, &42` copie les 8 bits de poids faible de R10 vers l'octet situé à l'adresse 42, alors que l'instruction `MOV R10, &42` copie tout le contenu de R10 vers les deux octets situés aux adresses 42 et 43[a].

[a] Précision: les 8 bits de poids faible vont en 42, et les 8 bits de poids fort vont en 43. On dit que le msp430 est de type *little-endian*. Allez lire `https://fr.wikipedia.org/wiki/Endianness` si c'est la première fois que vous voyez ce mot.

Extrait de la documentation: msp430x4xx.pdf page 115

Mnemonic		Description	Operation	V	N	Z	C
ADC(.B)	dst	Add C to destination	dst + C → dst	*	*	*	*
ADD(.B)	src,dst	Add source to destination	src + dst → dst	*	*	*	*
ADDC(.B)	src,dst	Add source and C to destination	src + dst + C → dst	*	*	*	*
AND(.B)	src,dst	AND source and destination	src .and. dst → dst	0	*	*	*
BIC(.B)	src,dst	Clear bits in destination	.not.src .and. dst → dst	–	–	–	–
BIS(.B)	src,dst	Set bits in destination	src .or. dst → dst	–	–	–	–
BIT(.B)	src,dst	Test bits in destination	src .and. dst	0	*	*	*
BR	dst	Branch to destination	dst → PC	–	–	–	–
CALL	dst	Call destination	PC+2 → stack, dst → PC	–	–	–	–
CLR(.B)	dst	Clear destination	0 → dst	–	–	–	–
CLRC		Clear C	0 → C	–	–	–	0
CLRN		Clear N	0 → N	–	0	–	–
CLRZ		Clear Z	0 → Z	–	–	0	–
CMP(.B)	src,dst	Compare source and destination	dst – src	*	*	*	*
DADC(.B)	dst	Add C decimally to destination	dst + C → dst (decimally)	*	*	*	*
DADD(.B)	src,dst	Add source and C decimally to dst.	src + dst + C → dst (decimally)	*	*	*	*
DEC(.B)	dst	Decrement destination	dst – 1 → dst	*	*	*	*
DECD(.B)	dst	Double-decrement destination	dst – 2 → dst	*	*	*	*
DINT		Disable interrupts	0 → GIE	–	–	–	–
EINT		Enable interrupts	1 → GIE	–	–	–	–
INC(.B)	dst	Increment destination	dst +1 → dst	*	*	*	*
INCD(.B)	dst	Double-increment destination	dst+2 → dst	*	*	*	*
INV(.B)	dst	Invert destination	.not.dst → dst	*	*	*	*
JC/JHS	label	Jump if C set/Jump if higher or same		–	–	–	–
JEQ/JZ	label	Jump if equal/Jump if Z set		–	–	–	–
JGE	label	Jump if greater or equal		–	–	–	–
JL	label	Jump if less		–	–	–	–
JMP	label	Jump	PC + 2 x offset → PC	–	–	–	–
JN	label	Jump if N set		–	–	–	–
JNC/JLO	label	Jump if C not set/Jump if lower		–	–	–	–
JNE/JNZ	label	Jump if not equal/Jump if Z not set		–	–	–	–
MOV(.B)	src,dst	Move source to destination	src → dst	–	–	–	–
NOP		No operation		–	–	–	–
POP(.B)	dst	Pop item from stack to destination	@SP → dst, SP+2 → SP	–	–	–	–
PUSH(.B)	src	Push source onto stack	SP – 2 → SP, src → @SP	–	–	–	–
RET		Return from subroutine	@SP → PC, SP + 2 → SP	–	–	–	–
RETI		Return from interrupt		*	*	*	*
RLA(.B)	dst	Rotate left arithmetically		*	*	*	*
RLC(.B)	dst	Rotate left through C		*	*	*	*
RRA(.B)	dst	Rotate right arithmetically		0	*	*	*
RRC(.B)	dst	Rotate right through C		*	*	*	*
SBC(.B)	dst	Subtract not(C) from destination	dst + 0FFFFh + C → dst	*	*	*	*
SETC		Set C	1 → C	–	–	–	1
SETN		Set N	1 → N	–	1	–	–
SETZ		Set Z	1 → C	–	–	1	–
SUB(.B)	src,dst	Subtract source from destination	dst + .not.src + 1 → dst	*	*	*	*
SUBC(.B)	src,dst	Subtract source and not(C) from dst.	dst + .not.src + C → dst	*	*	*	*
SWPB	dst	Swap bytes		–	–	–	–
SXT	dst	Extend sign		0	*	*	*
TST(.B)	dst	Test destination	dst + 0FFFFh + 1	0	*	*	1
XOR(.B)	src,dst	Exclusive OR source and destination	src .xor. dst → dst	*	*	*	*

Remarque chacune de ces instructions est documentée en détail dans la doc (msp430x4xx.pdf page 61 et suivantes). N'hésitez pas à vous y reporter si vous avez besoin de précisions.

Extrait de la documentation: msp430x4xx.pdf page 115

Mnemonic		Description	Operation	V	N	Z	C
ADC(.B)	dst	Add C to destination	dst + C → dst	*	*	*	*
ADD(.B)	src,dst	Add source to destination	src + dst → dst	*	*	*	*
ADDC(.B)	src,dst	Add source and C to destination	src + dst + C → dst	*	*	*	*
AND(.B)	src,dst	AND source and destination	src .and. dst → dst	0	*	*	*
BIC(.B)	src,dst	Clear bits in destination	.not.src .and. dst → dst	–	–	–	–
BIS(.B)	src,dst	Set bits in destination	src .or. dst → dst	–	–	–	–
BIT(.B)	src,dst	Test bits in destination	src .and. dst	0	*	*	*
BR	dst	Branch to destination	dst → PC	–	–	–	–
CALL	dst	Call destination	PC+2 → stack, dst → PC	–	–	–	–
CLR(.B)	dst	Clear destination	0 → dst	–	–	–	–
CLRC		Clear C	0 → C	–	–	–	0
CLRN		Clear N	0 → N	–	0	–	–
CLRZ		Clear Z	0 → Z	–	–	0	–
CMP(.B)	src,dst	Compare source and destination	dst – src	*	*	*	*
DADC(.B)	dst	Add C decimally to destination	dst + C → dst (decimally)	*	*	*	*
DADD(.B)	src,dst	Add source and C decimally to dst.	src + dst + C → dst (decimally)	*	*	*	*
DEC(.B)	dst	Decrement destination	dst – 1 → dst	*	*	*	*
DECD(.B)	dst	Double-decrement destination	dst – 2 → dst	*	*	*	*
DINT		Disable interrupts	0 → GIE	–	–	–	–
EINT		Enable interrupts	1 → GIE	–	–	–	–
INC(.B)	dst	Increment destination	dst +1 → dst	*	*	*	*
INCD(.B)	dst	Double-increment destination	dst+2 → dst	*	*	*	*
INV(.B)	dst	Invert destination	.not.dst → dst	*	*	*	*
JC/JHS	label	Jump if C set/Jump if higher or same		–	–	–	–
JEQ/JZ	label	Jump if equal/Jump if Z set		–	–	–	–
JGE	label	Jump if greater or equal		–	–	–	–
JL	label	Jump if less		–	–	–	–
JMP	label	Jump	PC + 2 x offset → PC	–	–	–	–
JN	label	Jump if N set		–	–	–	–
JNC/JLO	label	Jump if C not set/Jump if lower		–	–	–	–
JNE/JNZ	label	Jump if not equal/Jump if Z not set		–	–	–	–
MOV(.B)	src,dst	Move source to destination	src → dst	–	–	–	–
NOP		No operation		–	–	–	–
POP(.B)	dst	Pop item from stack to destination	@SP → dst, SP+2 → SP	–	–	–	–
PUSH(.B)	src	Push source onto stack	SP – 2 → SP, src → @SP	–	–	–	–
RET		Return from subroutine	@SP → PC, SP + 2 → SP	–	–	–	–
RETI		Return from interrupt		*	*	*	*
RLA(.B)	dst	Rotate left arithmetically		*	*	*	*
RLC(.B)	dst	Rotate left through C		*	*	*	*
RRA(.B)	dst	Rotate right arithmetically		0	*	*	*
RRC(.B)	dst	Rotate right through C		*	*	*	*
SBC(.B)	dst	Subtract not(C) from destination	dst + 0FFFFh + C → dst	*	*	*	*
SETC		Set C	1 → C	–	–	–	1
SETN		Set N	1 → N	–	1	–	–
SETZ		Set Z	1 → C	–	–	1	–
SUB(.B)	src,dst	Subtract source from destination	dst + .not.src + 1 → dst	*	*	*	*
SUBC(.B)	src,dst	Subtract source and not(C) from dst.	dst + .not.src + C → dst	*	*	*	*
SWPB	dst	Swap bytes		–	–	–	–
SXT	dst	Extend sign		0	*	*	*
TST(.B)	dst	Test destination	dst + 0FFFFh + 1	0	*	*	1
XOR(.B)	src,dst	Exclusive OR source and destination	src .xor. dst → dst	*	*	*	*