MSP430 TP1 : prise en main et programmation assembleur

Introduction

Dans cette série de TP «msp430», on va étudier le fonctionnement d'un (petit) ordinateur pour mieux comprendre l'interface entre le logiciel et le matériel. Vous devrez donc faire les diverses manipulations demandées, et par moment écrire des bouts de programme.
Nous ne ramasserons pas de compte-rendu ; par contre, vous avez intérêt à prendre des notes tout au long du déroulement du TP pour pouvoir les relire par la suite : dans les TP d'après, mais aussi avant les QCM, et aussi avant l'examen ! Pour chaque exercice, mettez donc par écrit (sur papier ou sur ordinateur) les manips que vous faites, les questions que vous vous posez, et les nouvelles notions que vous comprenez.

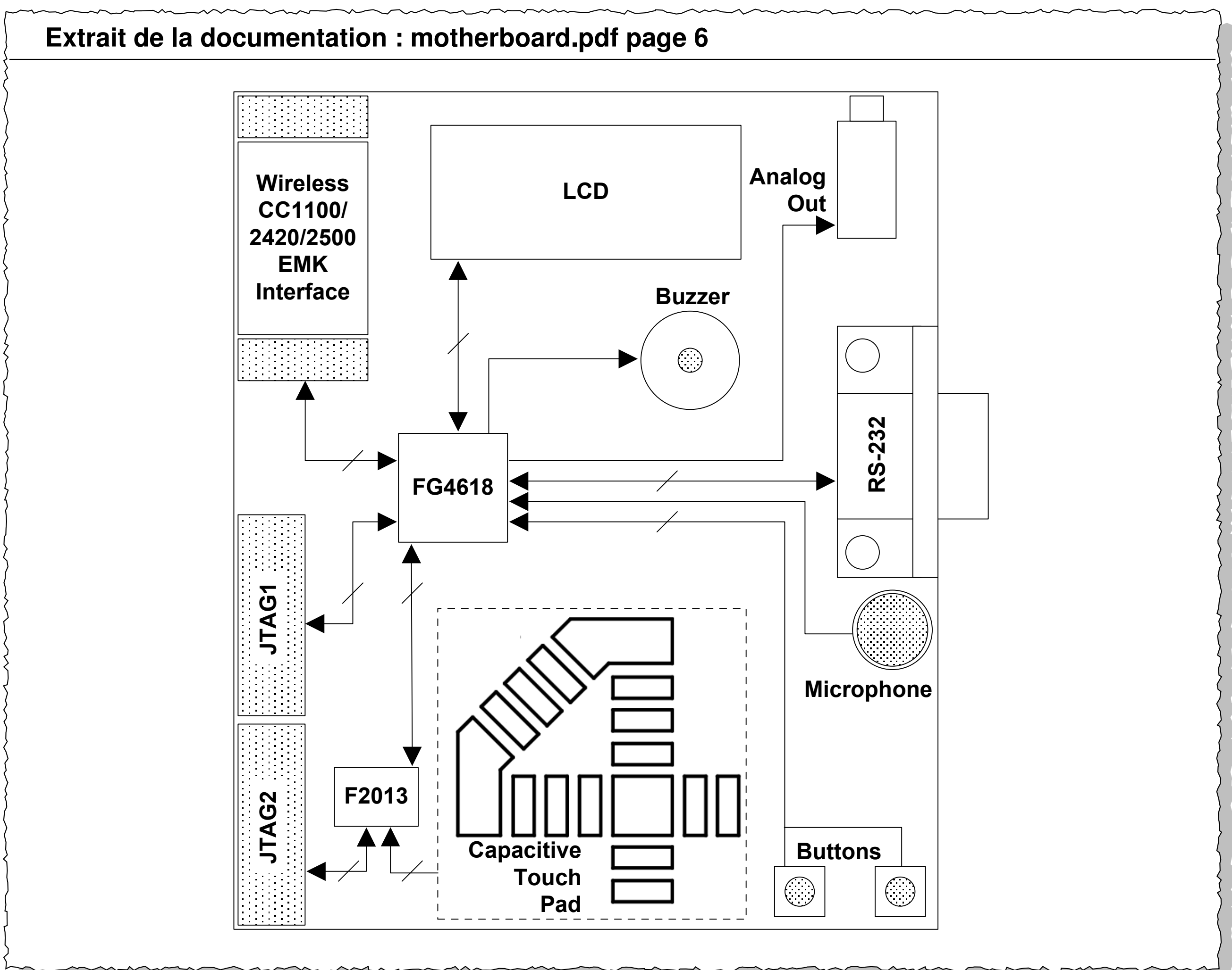
1 Découverte de la carte

Exercice 1 Pour chaque binôme, allez prendre le matériel nécessaire au TP : une carte d'expérimentation, une sonde JTAG (le boîtier gris avec une nappe d'un côté), et un câble USB.

1.1 La carte mère

La carte que nous allons utiliser en TP est illustrée dans le schéma ci-dessous. La puce principale est celle repérée FG4618 : c'est un microcontrôleur, on y reviendra par la suite. Mais d'abord, faisons le tour de la carte.

Extrait de la documentation : motherboard.pdf page 6



Sur cette carte mère, en plus du msp430, il y a tout un tas de périphériques :

1. un écran à cristaux liquides (pour afficher des chiffres et des icônes)
2. un microphone
3. un buzzer (pour jouer du son)
4. une prise casque (pour jouer du son aussi, mais plus joli)
5. un quartz (pour générer le signal d'horloge)
6. deux boutons poussoirs
7. des voyants lumineux (LED)
8. une roue tactile capacitive (*touchpad*) en forme de chiffre 4
9. un port série (RS-232)

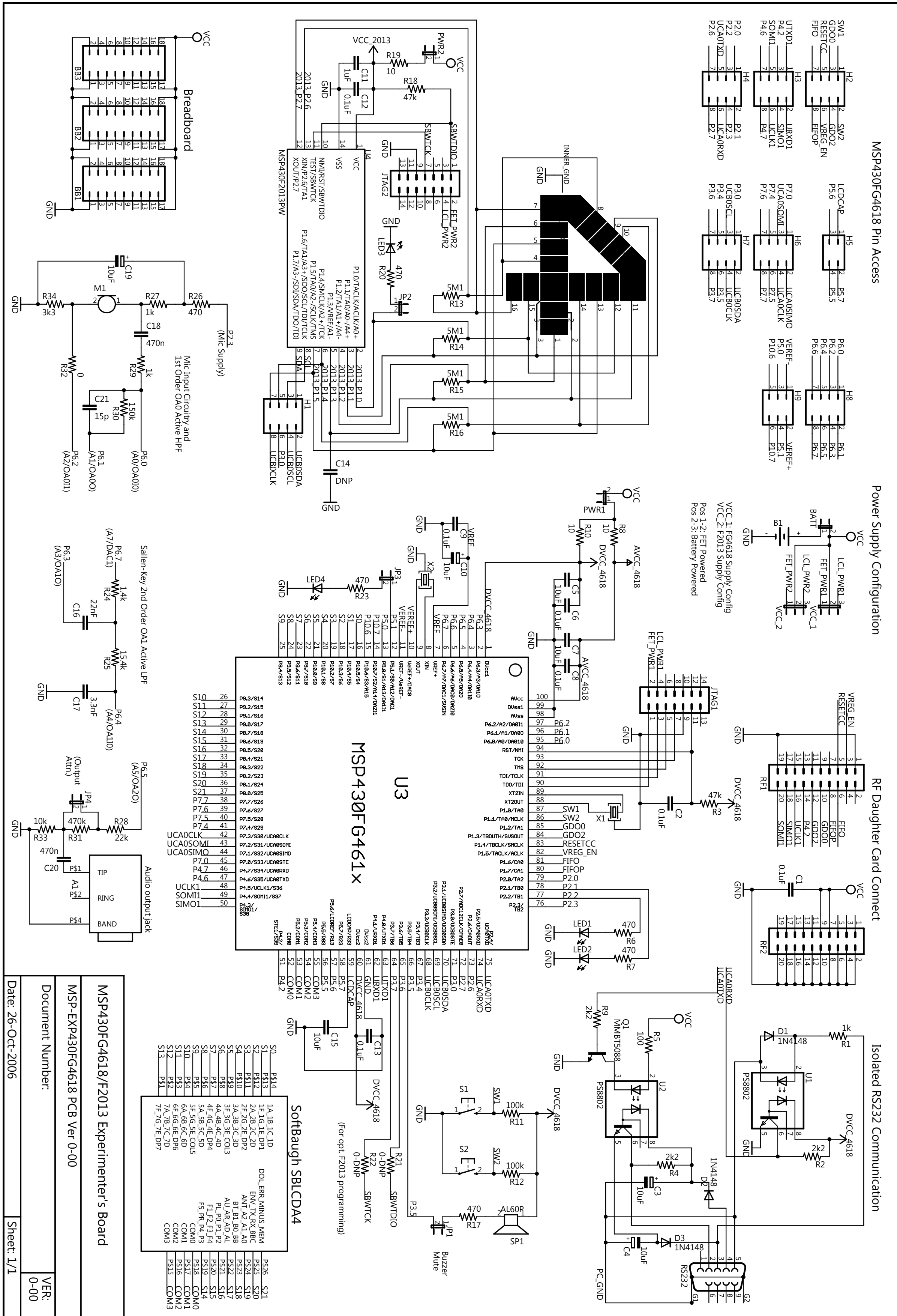
etc. ...

Exercice 2 Pour chacun de ces éléments, indiquez sur le schéma page précédente son emplacement approximatif. Certains éléments (LEDs, quartz) ne sont pas sur le schéma, vous devrez les chercher directement sur la carte. Le quartz est repéré `X2`, et les diodes sont repérées `LED1`, `LED2`, `LED3` et `LED4`.

Commentaire Vous aurez peut-être remarqué que la carte comporte *deux* microcontrôleurs. L'un est un MSP430F2013 (c'est le petit), et l'autre un MSP430FG4618 (c'est le gros). C'est avec ce second msp430 qu'on va travailler dans ces TP. Les diodes LED1, LED2, et LED4 y sont connectées par des pistes de la carte mère. La diode LED3, par contre, est connectée au F2013, qu'on ne va pas utiliser du tout. Vous pouvez dès maintenant oublier son existence, ainsi que celle de la LED3.

Exercice 3 L'encadré page suivante montre le schéma électrique de la carte mère. Retrouvez les différents composants vus jusqu'ici, et indiquez leur emplacement sur le schéma.

Extrait de la documentation : motherboard.pdf page 19



1.2 Documentations techniques

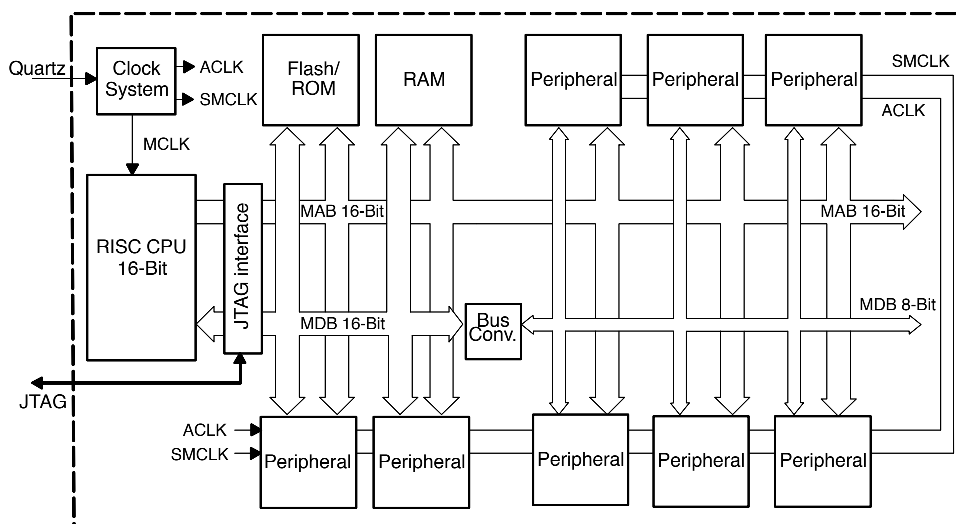
Comme tout objet technologique, notre plate-forme de TP s'accompagne d'une documentation technique abondante. Pour ne pas vous noyer sous la doc, nous vous en avons copié les extraits essentiels directement dans le sujet, sous forme d'encadrés. Pour les plus curieux, nous vous avons aussi mis à disposition les documents sur Moodle :

Motherboard.pdf décrit notre carte d'expérimentation et les différents composants présents sur la carte.
MSP430.pdf est le manuel générique de la puce MSP430. Le processeur est documenté au chapitre 3.
datasheet.pdf donne les détails techniques de notre modèle précis de msp430.
LCD.pdf décrit l'écran à crystaux liquides (qu'on utilisera à partir du TP2).

1.3 Vous avez dit microcontrôleur ?

Le MSP430FG4618 est un microcontrôleur, c'est à dire un *System-on-Chip* : une même puce qui contient à la fois un processeur, de la mémoire, et des blocs périphériques. Si on zoome sur l'intérieur de la puce, on a donc affaire à l'architecture illustrée ci-dessous.

Extrait de la documentation : msp430x4xx.pdf page 21



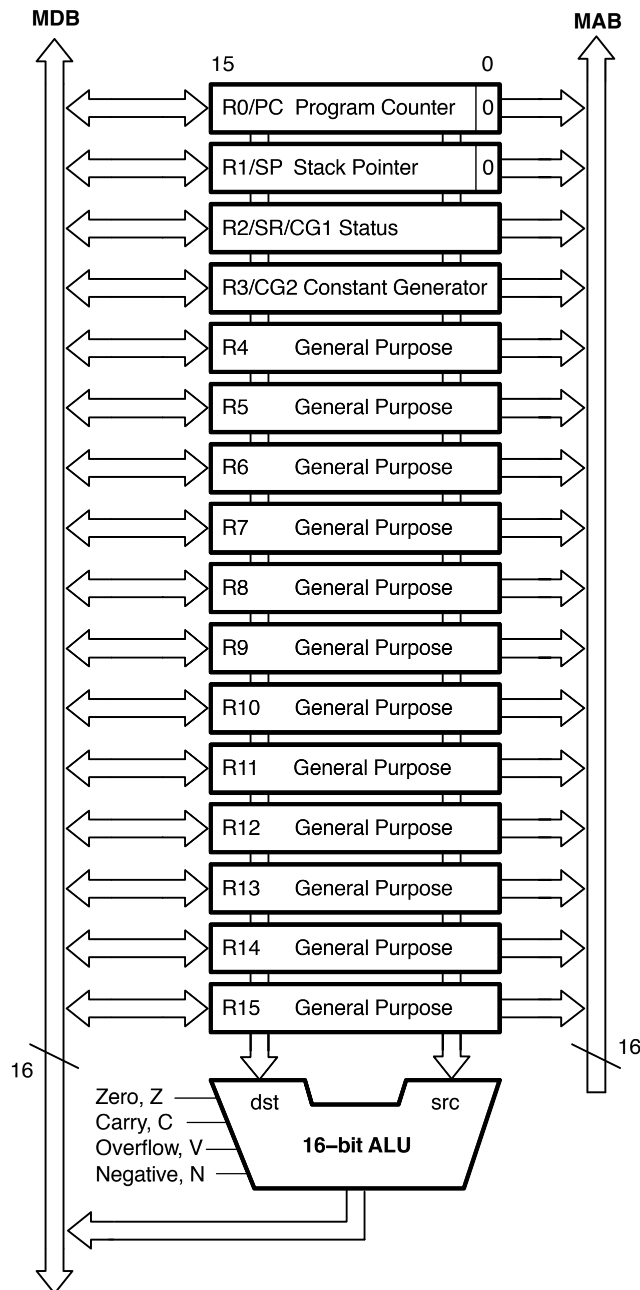
Commentaires Les flèches repérées MAB et MDB sont respectivement le *Memory Address Bus* et le *Memory Data Bus* (les mêmes que dans la micro-machine). Ce sont eux qui relient le processeur au reste-du-monde, comme dans toute machine de von Neumann qui se respecte. Le bloc repéré *JTAG interface* permet justement de venir intercepter tout ce qui passe sur le bus, et donc de contrôler finement le comportement de la machine. C'est ce qu'on va faire dans ces TP.

Exercice 4 Branchez maintenant la sonde JTAG sur la carte. Vous devriez pouvoir choisir sans difficulté entre les deux connecteurs JTAG.

1.4 Zoom sur le processeur

Si on se rapproche encore, on tombe sur l'architecture suivante :

Extrait de la documentation : msp430x4xx.pdf page 43



Commentaire Attention, ce schéma ne montre que la vue «externe», c'est à dire du point de vue de l'utilisateur. Une «vue interne» [a] comporterait d'autres éléments nécessaires à l'implémentation (automate de contrôle, registre d'instruction, etc).
Les seuls éléments représentés sur le schéma sont ceux qui sont accessibles au programmeur : les 16 registres architecturaux, ainsi que l'unité arithmétique et logique. Remarquez au passage que les 4 premiers registres sont *spécialisés* pour un usage particulier (R0 est le compteur ordinal, etc). À l'inverse les 12 autres registres sont *généraux*, on peut y mettre ce qu'on veut.

Nous vous avons reproduit en annexe (page 13 et suivantes) les passages correspondants de la documentation. Vous n'en avez pas besoin pour l'instant, mais pensez à y revenir lorsque vous voudrez des détails sur l'usage des registres.

a. voir par exemple les figures 9.2 et 9.3 du poly de cours (page 80)

Exercice 5 Allez lire la page `https://fr.wikipedia.org/wiki/Registre_de_processeur` et résumez, en une phrase, la différence entre un registre *spécialisé* et un registre *général*.

Exercice 6 Sur le schéma de la page 4, indiquez où se trouvent nos 16 registres, ainsi que l'automate de contrôle.

2 Prise en main des outils : mspdebug

Pour communiquer avec la carte au travers de la sonde JTAG, on va utiliser un programme appelé `mspdebug`. Cet outil va nous permettre de charger des programmes dans la mémoire, d'observer et de contrôler l'exécution du programme, d'inspecter le contenu du CPU et de la mémoire, etc.

Exercice 7 Démarrez mspdebug en tapant la ligne commande suivante :

```
mspdebug -j -d /dev/ttyUSB0 uif
```

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 LED4 sans passer par le CPU. Pour initialiser la diode, il faut écrire la valeur 2 à l'adresse 49. Ensuite, on l'allumera en écrivant la valeur 2 à l'adresse 50, et on l'éteindra en écrivant 0 à l'adresse 50. [1]

Exercice 8 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 9 Faites s'allumer et s'éteindre la diode quelques fois.

3 Exécution d'un programme en mode pas-à-pas

Exercice 10 Créez un nouveau répertoire TP1, et retapez [2] dans un fichier `tp1.s` le programme suivant :

```
.section .init9

main:
        /* initialisation de la diode rouge */
        mov.b #2, &49

        /* eteindre */
        mov.b #0, &50

        /* allumer */
        mov.b #2, &50

loop:
        jmp loop
```

Exercice 11 Traduisez ce programme en langage machine avec la commande suivante :

```
msp430-as -mmcu=msp430fg4618 -o tp1.o tp1.s
```

---

1. En effet, dans la vie, pour communiquer avec les périphériques depuis le processeur, on s'y prend de la même façon que pour communiquer avec la RAM. Ce sera d'ailleurs le sujet du TP2.
2. Vous pouvez aussi essayer de copier-coller depuis le PDF, mais il faudra pas venir vous plaindre que ça marche pas (ce qui sera le cas). Et puis c'est réellement formateur de retaper les exemples (si, si).

À savoir : assemblage VS éditions de liens

Pour passer d'un programme en langage assembleur à un programme exécutable, il faut réaliser deux opérations :

1) l'*assemblage* consiste à convertir un fichier texte contenant des instructions vers un fichier binaire contenant les même instructions, mais en langage machine. L'outil qui fait ça, l'assembleur, est typiquement nommé `as`, et permet de passer d'un fichier `bidule.s` à un fichier `bidule.o`. C'est ce qu'on vient de faire dans l'exercice précédent. Mais ce n'est pas fini : le programme consiste peut-être en plusieurs morceaux, qu'il faut maintenant coller ensemble.
2) l'*édition de liens* consiste à coller ensemble plusieurs fichiers `machin.o`, et à placer chacun d'entre eux aux bonnes adresses, par exemple pour s'assurer qu'ils ne se marchent pas les uns sur les autres. L'outil qui fait ça, l'éditeur de liens, est typiquement nommé `ld`, et produit un fichier `truc.elf`

Invoquer ces différents outils comme il faut avec les bonnes options est compliqué et souvent source d'erreur. Heureusement, il existe aussi une commande générique `gcc` qui est beaucoup plus simple d'usage, et qui se charge d'appeler `as` et `ld` dans le bon ordre et avec les bons arguments. Ainsi, vous pouvez obtenir directement un exécutable avec la commande suivante :

```
msp430-gcc -mmcu=msp430fg4618 -mdisable-watchdog -o truc.elf truc.s
```

Exercice 12 Depuis mspdebug, transférez votre programme sur la carte en utilisant la commande `prog tp1.elf`, puis lancez-le avec la commande `run`. Constatez que la diode reste toujours allumée (c'est normal, on ne l'éteint jamais). Interrompez l'exécution en appuyant sur `Ctrl+C`.

Exercice 13 Dans `tp1.s`, déplacez les instructions d'allumage et d'extinction à l'intérieur de la boucle infinie, et exécutez de nouveau votre programme. Constatez que la diode reste encore toujours allumée. En réalité, elle clignote bien, mais trop rapidement pour notre œil. C'est parce que la boucle tourne trop vite ! La fréquence du CPU est de 1MHz, et chaque instruction prend une poignée de cycles d'horloge, donc notre boucle tout entière tourne à plus de 100kHz. Interrompez de nouveau l'exécution, et au lieu de la relancer avec `run`, utilisez cette fois la commande `step` qui exécute une seule instruction machine. (faites donc `help run` et `help step` au passage). Constatez qu'en exécutant ainsi le programme en *mode pas-à-pas*, on arrive maintenant à voir ce qui se passe.

4 Programmation en assembleur

Vous allez maintenant devoir modifier votre programme. Pour la syntaxe ASM, aidez-vous des explications qui sont données dans les deux encadrés page suivante et page 9. Pour la mise au point, utilisez mspdebug. En plus des commande qu'on a vues jusqu'ici, vous aurez peut-être besoin de la commande `md` pour lire la mémoire, et de `setbreak` pour mettre des points d'arrêt. Commencez donc par taper `help md` et `help setbreak`.

Exercice 14 Modifiez votre programme afin de ralentir suffisamment la boucle infinie pour pouvoir observer le clignotement à l'oeil nu. Pour cela, vous allez rajouter une boucle (ou plusieurs) qui incrémente une variable jusqu'à atteindre une certaine valeur, par exemple 20000. Faites valider par un enseignant.

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 suivante. 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 suivante 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 encadré page 15), 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 suivante 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`. Regardez par exemple le programme page 6. 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]. Reportez-vous à l'encadré page 17 pour plus de détails.

---

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 avex besoin de précisions.

5 Programmation en C

Contrairement au langage d'assemblage qui est différent pour chaque jeu d'instructions, le langage C permet de programmer en s'abstrayant largement des détails de l'architecture cible. Les opérations de calcul et les structures de contrôle (boucles, fonctions, etc) ont une syntaxe unique, qui sera traduite par le *compilateur* vers un programme ASM avec les bonnes instructions. Cependant, même s'il est découplé de l'architecture concrète de la machine, le langage C offre au programmeur une vue abstraite calquée sur l'architecture de von Neumann : un processeur d'un côté, et une mémoire de l'autre côté. La syntaxe des *pointeurs* fait le lien entre les deux, permettant de désigner une case mémoire par son adresse, d'obtenir l'adresse d'une variable, etc.

Dans cette partie du TP, on va s'intéresser à la manière dont un programme C est traduit en assembleur.

5.1 Compilation et exécution

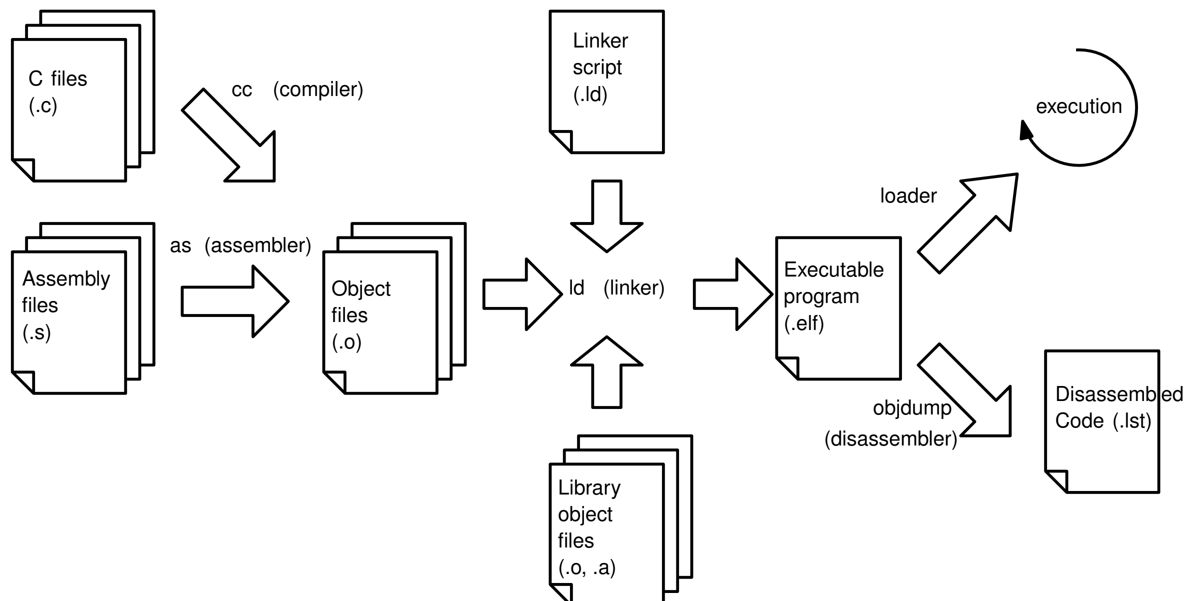
Exercice 15 Retapez dans un fichier `blink.c` le programme suivant :

```
unsigned char* p5dir;
unsigned char* p5out;
unsigned int i;

int main(void)
{
    // initialisation de la diode
    p5dir = (unsigned char*) 50;
    *p5dir = 2;

    p5out = (unsigned char*) 49;

    while(1)
    {
        // software delay
        for(i=0;i<20000;i++)
            {}//do nothing

        // allumer
        *p5out=2;

        // software delay
        for(i=0;i<20000;i++)
            {}//do nothing

        // eteindre
        *p5out=0;
    }
}
```

Exercice 16 En vous aidant des explications de l'encadré page suivante, compilez ce programme et transférez-le sur le msp430. Constatez que la diode rouge clignote.

À savoir : la chaîne de compilation

La chaîne de compilation complète est illustrée ci-dessous :



Vous connaissez déjà certains outils : `as`, `ld`. Le *chargement* du programme, sous Linux, serait effectué par le noyau. Dans ce TP, on utilise mspdebug pour transférer notre exécutable sur le msp430.
Les nouveaux éléments sont `cc` le compilateur C, et `objdump` le désassembleur.

Compilation Plutôt que d'invoquer `cc` individuellement, on va encore une fois passer par la commande `gcc` qui est plus confortable :

— pour faire seulement la compilation C vers assembleur, utilisez cette commande :
```
msp430-gcc -mmcu=msp430fg4618 -Wall -Werror -O1 -S -o truc.s truc.c
```
— pour faire la compilation et l'assemblage (C vers code machine), utilisez cette commande :
```
msp430-gcc -mmcu=msp430fg4618 -Wall -Werror -O1 -c -o truc.o truc.c
```
— pour faire aussi l'édition de liens dans la foulée, utilisez cette commande :
```
msp430-gcc -mmcu=msp430fg4618 -mdisable-watchdog -o truc.elf truc.c
```

Désassemblage Pour comprendre précisément ce qui se passe lors de l'exécution, il est souvent utile d'examiner précisément le code machine. L'outil qui permet de faire ça s'appelle un *désassembleur*. Il prend en entrée un programme en langage machine (`.o` ou `.elf`) et produit en sortie un fichier texte lisible à l'oeil nu (c'est presque de l'ASM). La ligne de commande à utiliser est la suivante :
```
msp430-objdump -d truc.elf > truc.lst
```

Exercice 17 Désassemblez l'exécutable, et ouvrez le listing. Repérez les différents morceaux : en plus de votre fonction `main()`, le *linker* a aussi inclus du code avant et après le programme. On ne va pas chercher à comprendre ce code ajouté, mais par contre dans la suite on va étudier d'un peu plus près le code assembleur de la fonction `main()`. Repérez respectivement l'initialisation des diodes, les deux boucles de temporisation, la boucle infinie, et les instructions qui allument et éteignent la diode.

5.2 Compilation optimisante

Jusqu'ici, la traduction C vers assembleur restait assez directe. Mais dans le cas général, le compilateur n'a aucune obligation de traduire ligne-par-ligne, ni de respecter l'ordre des instructions, etc. Sa seule garantie est de respecter la *sémantique* de notre programme, c'est à dire intuitivement de ne pas changer «ce que calcule» le programme. [3] Cette distinction est importante, car il existe évidemment plusieurs programmes

3. Pour avoir des idées plus précises sur les «règles du jeu» quand on utilise un compilateur, suivez un cours de compilation !

ASM qui «calculent la même chose» qu'un unique programme C. Le compilateur va donc essayer de construire la meilleure traduction possible, c'est à dire celle qui calcule la même chose mais le plus vite possible. On appelle cela *l'optimisation*.
Par exemple, dans notre cas, la boucle sur `i` n'a pas d'intérêt apparent pour la logique du programme, et donc le compilateur peut être tenté de la supprimer complètement.

Exercice 18 Refaites l'exercice précédent en essayant les différents niveaux d'optimisation offerts par gcc : `-O0`, `-O1`, `-O2`, et `-O3`. (Attention : la syntaxe correcte est «tiret», suivi de la lettre O majuscule, puis d'un chiffre). Quelles sont les principales différences entre les programmes assembleur que vous obtenez ?

Exercice 19 Ajoutez à la déclaration de la variable `i` le mot-clé `volatile`, dont l'effet est d'interdire au compilateur d'optimiser les accès à `i`. Une fois de plus, compilez votre programme avec les différents niveaux d'optimisation et observez le code assembleur obtenu.

5.3 Représentation en machine des variables du programme

Pour faire cette partie, gardez le mot-clé `volatile` sur `i` et revenez au niveau d'optimisation `-O1`.

Exercice 20 Dans notre programme, la variable `i` est déclarée avec le type `unsigned int`. En observant le code assembleur, indiquez quelle est la taille (en octets) de cette variable.

Exercice 21 Modifiez votre programme en donnant à `i` les types `int`, `unsigned long` et également `unsigned long long`. Quelle est la taille de `i` dans chacun de ces cas ?

Exercice 22 Exécutez les quatre versions différentes du programme et comparez les fréquences de clignotement dans chacun des cas. Comment expliquez-vous ces différences ?

Exercice 23 En observant de près le code des différentes versions, et/ou en les exécutant pas-à-pas, expliquez comment est incrémentée la variable `i` dans chaque version.

Annexe : documentation des registres du CPU

Extrait de la documentation : msp430x4xx.pdf page 44

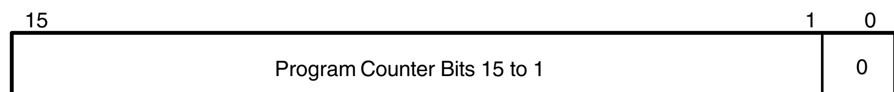
3.2 CPU Registers

The CPU incorporates sixteen 16-bit registers. R0, R1, R2 and R3 have dedicated functions. R4 to R15 are working registers for general use.

3.2.1 Program Counter (PC)

The 16-bit program counter (PC/R0) points to the next instruction to be executed. Each instruction uses an even number of bytes (two, four, or six), and the PC is incremented accordingly. Instruction accesses in the 64-KB address space are performed on word boundaries, and the PC is aligned to even addresses. Figure 3–2 shows the program counter.

Figure 3–2. Program Counter



The PC can be addressed with all instructions and addressing modes. A few examples:

```
MOV   #LABEL,PC ; Branch to address LABEL
MOV   LABEL,PC  ; Branch to address contained in LABEL
MOV   @R14,PC   ; Branch indirect to address in R14
```

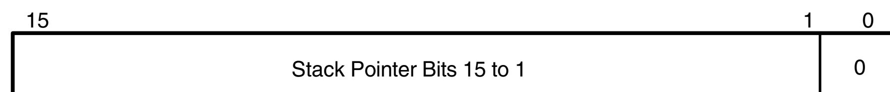
Extrait de la documentation : msp430x4xx.pdf page 45

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.

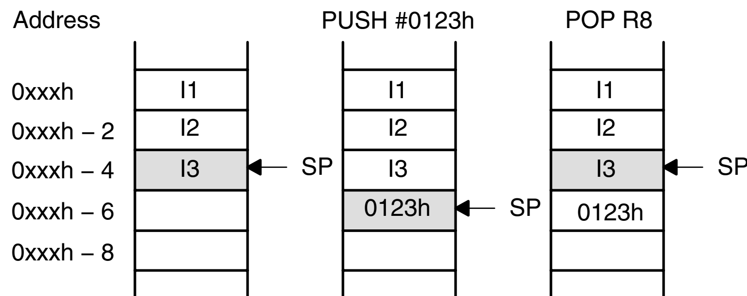
Figure 3–4 shows stack usage.

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



The special cases of using the SP as an argument to the PUSH and POP instructions are described and shown in Figure 3–5.

Figure 3–5. PUSH SP - POP SP Sequence

The stack pointer is changed after a PUSH SP instruction.

The stack pointer is not changed after a POP SP instruction. The POP SP instruction places SP1 into the stack pointer SP (SP2=SP1)

Extrait de la documentation : msp430x4xx.pdf page 46

3.2.3 Status Register (SR)

The status register (SR/R2), used as a source or destination register, can be used in the register mode only addressed with word instructions. The remaining combinations of addressing modes are used to support the constant generator. Figure 3−6 shows the SR bits.

Figure 3−6. Status Register Bits

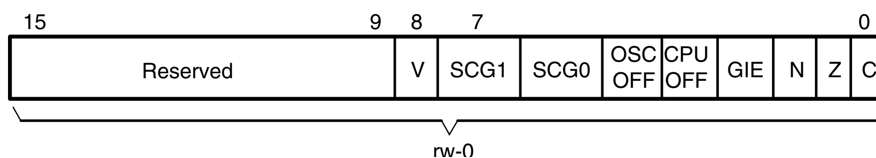


Table 3−1 describes the status register bits.

Table 3−1. Description of Status Register Bits

Bit	Description	
V	Overflow bit. This bit is set when the result of an arithmetic operation overflows the signed-variable range.	
	`ADD(.B),ADDC(.B)`	Set when: Positive + Positive = Negative Negative + Negative = Positive, otherwise reset
	`SUB(.B),SUBC(.B),CMP(.B)`	Set when: Positive − Negative = Negative Negative − Positive = Positive, otherwise reset
SCG1	System clock generator 1. This bit, when set, turns off the DCO dc generator, if DCOCLK is not used for MCLK or SMCLK.	
SCG0	System clock generator 0. This bit, when set, turns off the FLL+ loop control	
OSCOFF	Oscillator Off. This bit, when set, turns off the LFXT1 crystal oscillator, when LFXT1CLK is not use for MCLK or SMCLK	
CPUOFF	CPU off. This bit, when set, turns off the CPU.	
GIE	General interrupt enable. This bit, when set, enables maskable interrupts. When reset, all maskable interrupts are disabled.	
N	Negative bit. This bit is set when the result of a byte or word operation is negative and cleared when the result is not negative.	
	Word operation:	N is set to the value of bit 15 of the result
	Byte operation:	N is set to the value of bit 7 of the result
Z	Zero bit. This bit is set when the result of a byte or word operation is 0 and cleared when the result is not 0.	
C	Carry bit. This bit is set when the result of a byte or word operation produced a carry and cleared when no carry occurred.	

3.2.4 Constant Generator Registers CG1 and CG2

Six commonly-used constants are generated with the constant generator registers R2 and R3, without requiring an additional 16-bit word of program code. The constants are selected with the source-register addressing modes (As), as described in Table 3–2.

Table 3–2. Values of Constant Generators CG1, CG2

Register	As	Constant	Remarks
R2	00	– – – – –	Register mode
R2	01	(0)	Absolute address mode
R2	10	00004h	+4, bit processing
R2	11	00008h	+8, bit processing
R3	00	00000h	0, word processing
R3	01	00001h	+1
R3	10	00002h	+2, bit processing
R3	11	0FFFFh	–1, word processing

The constant generator advantages are:

- ❑ No special instructions required
- ❑ No additional code word for the six constants
- ❑ No code memory access required to retrieve the constant

The assembler uses the constant generator automatically if one of the six constants is used as an immediate source operand. Registers R2 and R3, used in the constant mode, cannot be addressed explicitly; they act as source-only registers.

Constant Generator – Expanded Instruction Set

The RISC instruction set of the MSP430 has only 27 instructions. However, the constant generator allows the MSP430 assembler to support 24 additional, emulated instructions. For example, the single-operand instruction:

```
CLR        dst
```

is emulated by the double-operand instruction with the same length:

```
MOV        R3,dst
```

where the #0 is replaced by the assembler, and R3 is used with As = 00.

```
INC        dst
```

is replaced by:

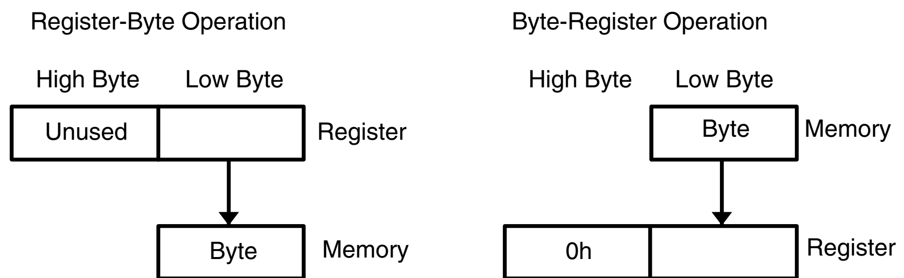
```
ADD        0(R3),dst
```

Extrait de la documentation : msp430x4xx.pdf page 48

3.2.5 General-Purpose Registers R4 to R15

Twelve registers, R4 to R15, are general-purpose registers. All of these registers can be used as data registers, address pointers, or index values, and they can be accessed with byte or word instructions as shown in Figure 3–7.

Figure 3–7. Register-Byte/Byte-Register Operations



Example Register-Byte Operation
R5 = 0A28Fh
R6 = 0203h
Mem(0203h) = 012h

```
ADD.B        R5,0(R6)
```

  08Fh
+ 012h
0A1h

Mem (0203h) = 0A1h
C = 0, Z = 0, N = 1

(Low byte of register)
+ (Addressed byte)
–>(Addressed byte)

Example Byte-Register Operation
R5 = 01202h
R6 = 0223h
Mem(0223h) = 05Fh

```
ADD.B      @R6,R5
```

  05Fh
+ 002h
00061h

R5 = 00061h
C = 0, Z = 0, N = 0

(Addressed byte)
+ (Low byte of register)
–>(Low byte of register, zero to High byte)