ARC TD5

Prise en main d'une architecture simple : le MSP430
(4h séance sur machine), 9 avril 2024

Construit à partir du poly de TD-TP des cours IF-AC et IF-AO

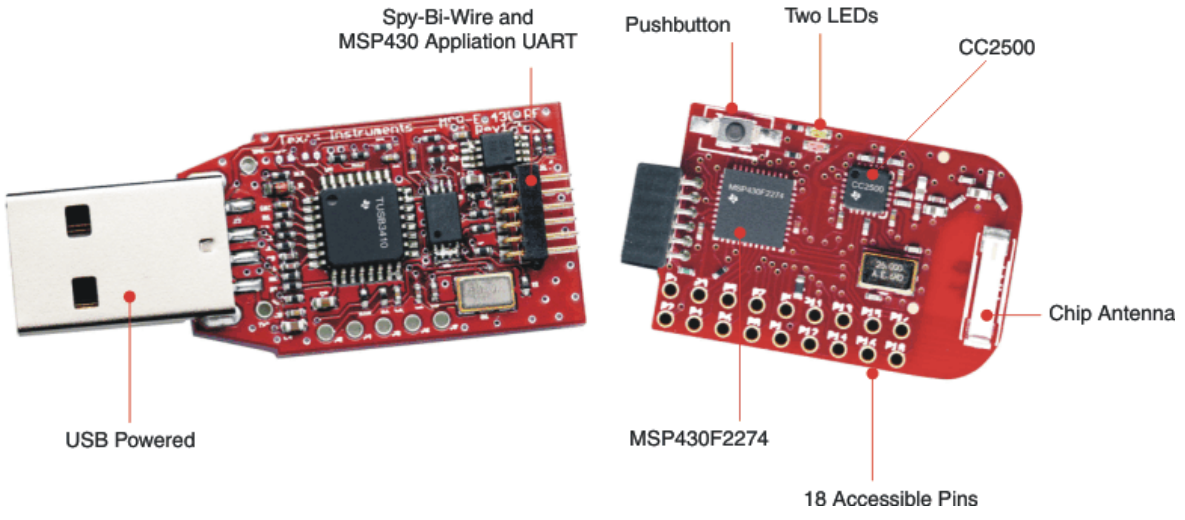
Dans ce TP «msp430», on va étudier – du point de vue matériel, contrairement au TP CRO MSP430 – le fonctionnement d'un système embarqué qui n'est rien d'autre qu'un *petit* ordinateur. Vous devrez donc faire les diverses manipulations demandées, et par moment écrire des bouts de programme.
Nous ne ramasserons 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.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.
2020 : Attention, si `mspdebug` ne marche pas c'est que vous le lancez depuis un terminal gnome (a fond blanc), il faut le lancer depuis un terminal `xterm` (à fond noir).
2023 : Attention. A partir de 2023, nous utilisons `gcc` version 8, en cas de problème incompréhensible lié au printf ou au `#include <stdio.h>`, vérifier que le compilateur est bien `msp430-elf-gcc` et non pas `msp430-gcc`.
2023 : Attention. A partir de 2023, vous avez à votre disposition une machine virtuelle TC sur laquelle mspgcc devrait fonctionner (vous l'avez initié en CRO normalement), la chaîne reste disponible sur les machines du département.

1 Découverte de la carte

Pour chaque binôme, allez prendre le matériel nécessaire au TP : dans chaque boîte, vous trouverez un genre de clé USB qui ressemble au schéma ci-dessous. Ne le branchez pas tout de suite.
Ce matériel est composé de deux parties. La bonne nouvelle est que nous n'étudierons pas de près la moitié compliquée à gauche, qui est une interface USB vers JTAG.

Exercice 1 Que signifie l'acronyme USB, au fait ? Expliquez en une phrase la signification du S. Faites valider cette phrase par un enseignant, mais n'attendez pas qu'il arrive pour passer à la suite.

Extrait de la documentation : ez430.pdf page 5

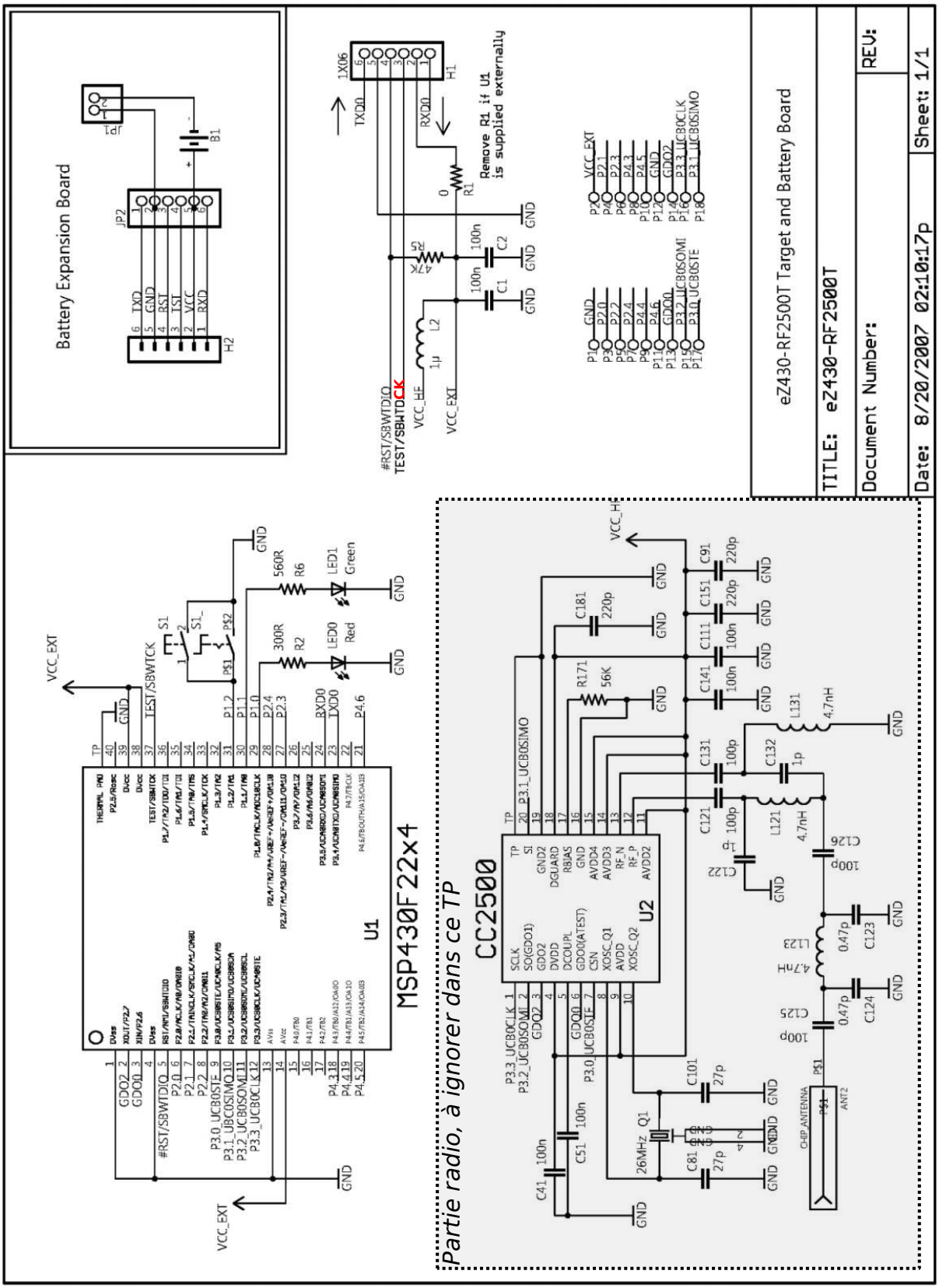


Sur la carte mère de droite (la plus petite), en plus du msp430, il y a quelques périphériques :
1. un bouton poussoir
2. deux voyants lumineux (LED)

3. une puce radio (CC2500), son quartz et son antenne
4. un port série
5. 18 broches libres pour ajouter vos propres interfaces vers ce que vous voudrez

Exercice 2 Retrouvez chacun de ces éléments sur le schéma page suivante.
L'encadré montre le schéma électrique de la carte mère de droite. Retrouvez-y les différents composants et indiquez leur emplacement sur le schéma.

Extrait de la documentation : ez430.pdf page 18



(On constate que les documents techniques constructeurs soufrent parfois d'une résolution insuffisante et de petits bugs. Que cela vous encourage à faire mieux).

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 :

ez430.pdf (27 pages) décrit notre carte d'expérimentation et les différents composants présents sur la carte.

MSP430.pdf (695 pages) est le manuel générique de la famille MSP430. Le processeur est documenté au chapitre 3 de ce document.

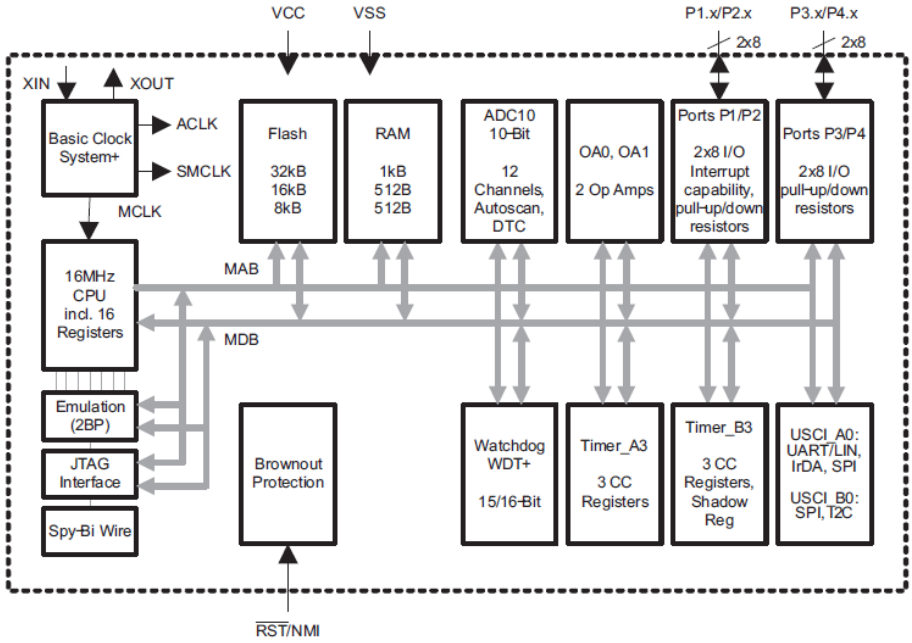
msp430F2274.pdf (87 pages) donne les détails techniques de notre modèle précis de msp430 : le `MSP430F2274`.

1.1 Vous avez dit microcontrôleur ?

Le MSP430F2274 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 : msp430F2274.pdf page 6

MSP430F22x4 Functional Block Diagram



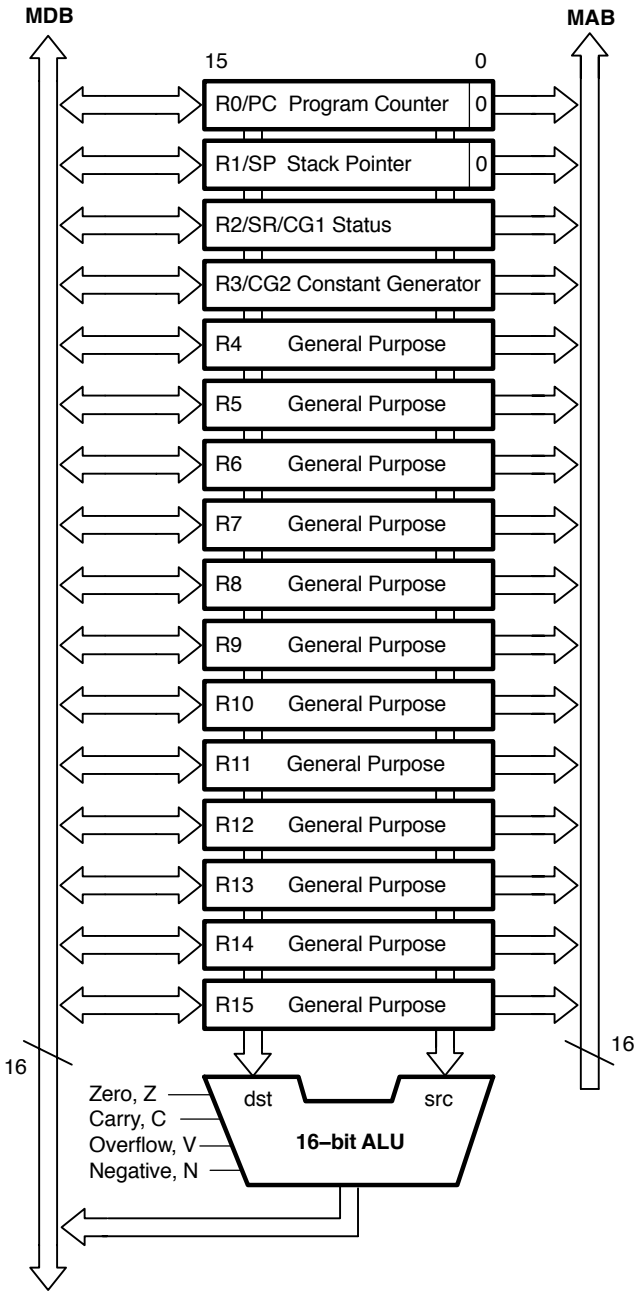
Les flèches repérées MAB et MDB sont respectivement le *Memory Address Bus* et le *Memory Data Bus* (les mêmes que dans la micro-machine). Ce sont eux qui relient le processeur au reste-du-monde, comme dans toute machine de von Neumann qui se respecte.

Exercice 3 Repérez sur ce diagramme le processeur, la RAM, la mémoire flash. Vérifiez que vous connaissez le sens des acronymes CPU, RAM, ADC (sinon cherchez sur internet ou demandez à un enseignant). Ignorez les autres acronymes pour le moment.

1.2 Zoom sur le processeur

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

Extrait de la documentation : MSP430.pdf page 43



Commentaire Attention, ce schéma ne montre que la vue ISA (instruction-set architecture), c'est à dire du point de vue de l'utilisateur du processeur. Elle cache les détails de microarchitecture que le programmeur n'a pas besoin de connaître : automate de contrôle, registre d'instruction, etc

Les seuls éléments représentés sur le schéma sont donc ceux qui sont accessibles au programmeur : les 16 registres architecturaux, les drapeaux (Z,C,V,N), ainsi que l'unité arithmétique et logique. Remarquez au passage que les 4 premiers registres sont *spécialisés* pour un usage particulier (R0 est le *program counter*, etc). À l'inverse les 12 autres registres sont *généraux*, on peut y mettre ce qu'on veut.

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

Exercice 5 Explicitez l'acronyme ALU.

Exercice 6 Tiens, il manque les flèches sur les fils entre l'ALU et les drapeaux. Ajoutez-les.

Exercice 7 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*.

2 Installation des outils : msp430-elf-gcc et mspdebug

2.1 Activation de la chaîne de compilation sur les machines du département (déjà fait en CRO normalement)

La programmation du eZ430-R2500 se fait en C, nous utiliserons le compilateur `msp430-elf-gcc` qui est installé sur les machines du département. Si vous avez déjà utilisé cette chaîne de compilation, vous pouvez directement taper la commande `go_mspgcc` et passer à la section suivante.
Pour vérifier que vous avez installé correctement la chaîne de compilation `mspgcc`, vous tapez la commande :

```
msp430-elf-gcc -v
```

Attention : nouvelle chaîne de compilation `gcc` v8

Depuis 2023 nous utilisons une nouvelle chaîne de compilation (`gcc v8`) fournie par Texas Instrument. La manipulation du `printf` est un peu délicate, donc à chaque fois que vous utilisez un `printf` bien mettre `\r\n` (dans cet ordre) à la fin de la chaîne que vous affichez, sinon il restera dans le buffer de sortie jusqu'au prochain `printf`.

Si la commande n'est pas connue, activez la *toolchain* comme indiqué ci-dessous. A priori le TP est à exécuter sur les machines du département ou la chaîne de compilation est installée. En annexe **??**, vous trouverez les instruction pour installer la chaîne sur votre machine Ubuntu si vous le souhaitez. Une fois la *toolchain* activée, il faut encore télécharger les sources des programmes que l'on va utiliser (section **??**).

2.2 Activation de la chaîne de compilation sur les machines du département

La programmation du eZ430-R2500 se fait en C, nous utiliserons le compilateur `msp430-elf-gcc` qui est installé sur les machines du département. la DSI a déployé la chaîne de compilation msp430 dans le répertoire `/opt/msp430-2023`. Pour pouvoir l'exécuter il faut copier le contenu du fichier `/opt/msp430-2023/env.sh` dans votre `.bashrc`, cela va mettre à jour les variables `$PATH`, `$MSPINCLUDES`, etc. Voici le contenu du fichier `env.sh` que vous trouverez aussi sur Moodle, copiez le dans votre `.bashrc`, vous comprenez ce code ?
Enfin, lancer une nouvelle fenêtre shell et tapez la commande :
`go_mspgcc`
Cela modifie votre invite (en rajoutant `[MSPGCC8]` devant) et vous avez maintenant accès à la commande `msp430-elf-gcc`

```
# -*-sh-*-

function go_mspgcc()
{
    # use your own mspgcc
    export MSPHOME=${HOME}/tools/msp430/
    export MSP430DIR=${MSPHOME}/msp430-gcc-8.3.1.25_linux64
    export MSPINCLUDES=${MSPHOME}/msp430-gcc-support-files/include
    export MSPLDLIBS=${MSPINCLUDES}
    export MSP430_GCC_INCLUDE_DIR=${MSPINCLUDES}

    export MSPGCCDIR=${MSP430DIR}
    export MSP430PREFIX=msp430-elf
    export CC=${MSP430PREFIX}-gcc
    export LD=${MSP430PREFIX}-ld
    export MSPFLAGS=-mhwmult=none

    export PATH=${PATH}:${MSPGCCDIR}/bin
    export PS1='[MSPGCC8]'${PS1}
}
```

Code à copier dans le fichier `$HOME/.bashrc`

2.3 Activation de `mspdebug`

L'outil `mspdebug` sert lui à télécharger le code binaire sur la carte elle-même (en passant par le port USB grâce à un protocole JTAG). La figure 1 explique comment est chargé sur la clé le binaire depuis l'ordinateur (protocol JTAG) et comment ensuite la clé peut communiquer avec l'ordinateur par un port série (protocole UART).

- Lancer un terminal de type `xterm` (le fond doit être noir).
- Branchez sur un port USB la carte ez430.
- Lancez la command `mspdebug rf2500`
- Si tout se passe bien, vous vous retrouvez avec une invite :

  `(msp-debug)`
- **2020 : Attention, si `mspdebug` ne marche pas c'est que vous le lancez depuis un terminal gnome, il faut le lancer depuis un terminal `xterm`**

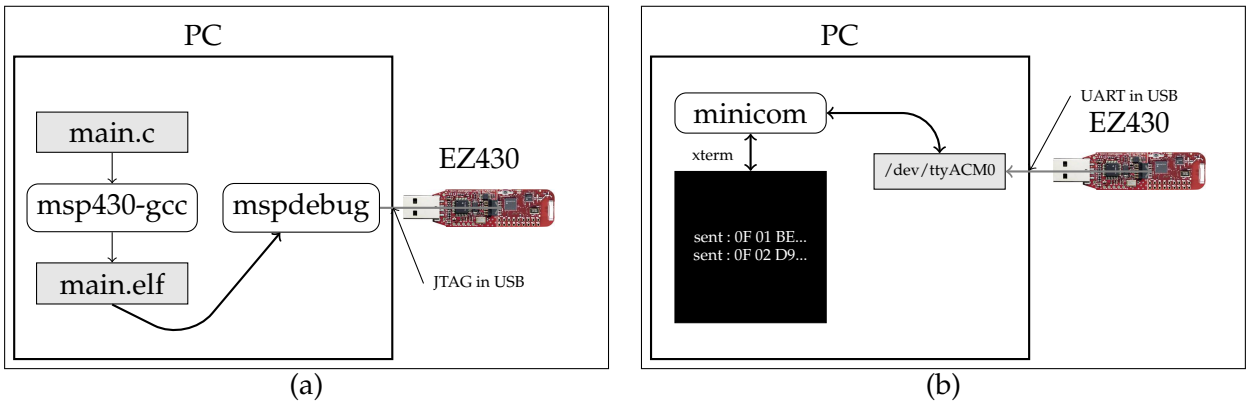


FIGURE 1 – (a) Cross-compilation et chargement du binaire sur le MSP430 via le protocol JTAG dans USB, et (b) communication série (ou UART) entre le PC et le MSP430 à travers le device linux `/dev/ttyACM0`.

3 Prise en main des outils : mspdebug

Pour communiquer avec notre MSP430 au travers de l'interface USB/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 8 Branchez la carte, et lancez mspdebug en tapant la ligne commande suivante :

```
mspdebug rf2500
```

L'argument est le nom du driver à utiliser, ici celui de notre carte qui s'appelle 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 une diode LED. Un concept universel aujourd'hui en informatique est le concept de *périphérique mappé en mémoire*. Tous les périphériques sont *mappés* sur des adresses mémoire, c'est à dire accessible en écrivant (ou lisant) à certaines addresses mémoires. En écrivant les bonnes valeurs aux bonnes adresses, on peut contrôler ces périphériques.
Par exemple, pour allumer la diode rouge, il faut d'abord écrire la valeur 1 à l'adresse 34. Cela fait, on allumera la diode en écrivant la valeur 1 à l'adresse 33, et on l'éteindra en écrivant 0 à l'adresse 33. Admettons ces valeurs pour l'instant, nous les expliquerons dans un moment.

Exercice 9 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 10 Faites s'allumer et s'éteindre la diode quelques fois.

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

Exercice 11 Créez un nouveau répertoire TPMSP430, et retapez[1] dans un fichier `ex1.s` le programme suivant :

```
        .global main

main:
        /* disable watchdog */
        mov.w   #23168, &WDTCTL
        /* initialisation de la diode rouge */
        mov.b #1, &34
        /* eteindre */
        mov.b #0, &33
        /* allumer */
        mov.b #1, &33
loop:
        jmp loop
```

Dans ce programme,

- `.global main` est une commande à destination de msp430-elf-gcc pour lui indiquer où placer ce code – voir l'encadré ci-dessous.
- `mov.b` est l'instruction assembleur qui réalise une copie (*move*) d'un octet (b pour *byte*).

1. 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).

— en assembleur msp430, `#17` désigne la valeur 17, alors que `&17` désigne le contenu de la case mémoire d'adresse 17.

— donc `mov.b #1, &34` est une instruction assembleur qui réalise une copie de la valeur constante 1 vers le contenu de la case mémoire d'adresse 34. Attention, les arguments sont dans l'ordre inverse de la commande `mw` de mspdebug... Moyen mnémotechnique : en assembleur MSP430, la virgule se lit "to".

— `jmp` est une instruction MSP430 de saut (pour *jump*)

— `main:` et `loop:` sont des définitions d'étiquettes *(label)*. Une étiquette indique une adresse au programme assembleur. On peut les utiliser ensuite en place de la vraie adresse comme destination de sauts (ou autres). En cas de saut relatif, le programme assembleur calculera le déplacement par soustraction de l'adresse du saut à l'adresse de l'étiquette.

— Ici, remarquez qu'on finit notre programme par une boucle infinie dont il ne sortira pas : cela assure que notre pointeur de programme ne part pas se balader au hasard dans la mémoire...

— La première ligne de la fonction `main` (`mov.w #23168, &WDTCTL`) sert à débrancher le *watchdog* qui fait rebooter le système lorsqu'il est inactif trop longtemps. Allez lire le premier paragraphe de la page wikipedia "watchdog timer" et vous comprendrez par quel mécanisme votre téléphone reboote lorsqu'il ralentit trop.

Attention, si on fait une faute de frappe, en général il n'y a pas de message d'erreur mais le programme fera n'importe quoi. Nottament si on oublie le "#" devant les constantes.

Exercice 12 Traduisez ce programme en un exécutable en langage machine avec la commande suivante :

```
msp430-elf-gcc -mmcu=msp430f2274 -o ex1.elf ex1.s
```

L'option `-mmcu=msp430f2274`, indique la puce exacte ciblée.

À savoir : assemblage et 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 dans notre cas msp430-elf-as), et permet de passer d'un fichier `bidule.s` à un fichier `bidule.o`.
   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 donnée.

Exercice 13 Désassemblez le programme obtenu par

```
msp430-elf-objdump -d ex1.elf
```

Cherchez, dans la sortie de cette commande, votre `main`, et répondez aux questions suivantes :

— Quel est le code binaire de l'instruction `jmp loop` ?

— A quelle adresse est-elle assemblée ?

— Est-ce un saut relatif ou un saut absolu ?

— Qui s'est permis de rajouter toutes ces instructions autour de votre programme ?

— Qu'elle est l'adresse de la fonction `main`

Exercice 14 Depuis mspdebug, transférez votre programme sur la carte (on dit *flasher*) en utilisant la commande
`prog ex1.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`.

5 Exécution d'un programme pas à pas

Exercice 15 Dans `ex1.s`, déplacez les instructions d'allumage et d'extinction à l'intérieur de la boucle infinie : le but est de faire clignoter la diode. Chargez par `prog` puis exécutez de nouveau votre programme par `run` dans mspdebug (n'oubliez pas de le ré-assembler avant, i.e. de re-générer le `ex1.elf`).
Si tout va bien, on dirait que la diode reste encore toujours allumée. C'est peut-être que vous vous êtes trompés. C'est peut-être aussi qu'elle clignote bien, mais trop rapidement pour notre œil. En effet, 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. Décidez ainsi si la diode clignote ou si vous vous êtes plantés. Auquel cas, corrigez.

Exercice 16 Quittez `mspdebug` (commande `exit`) et flasher directement l'ez430 en une ligne (si vous n'avez pas à faire de pas-à-pas, c'est beacoup plus rapide de flasher de cette manière là) :

```
mspdebug rf2500 "prog ex1.elf"
```

6 Programmation en assembleur : variables et boucles

Note à partir d'ici, le temps peut s'allonger, essayer de répondre à chaque questions, si vous ne finissez pas ce n'est pas grave
Vous allez maintenant devoir modifier votre programme un peu plus sérieusement. Pour la syntaxe ASM, aidez-vous des explications qui sont données dans les deux encadrés on the next page et on page 12.

Débuggage : points d'arrêts

Pour la mise au point, utilisez `mspdebug`. `mspdebug` vous permet de débugger, un peu comme `gdb` (en moins puissant).
En plus des commandes qu'on a vues jusqu'ici, vous aurez peut-être besoin de la commande `md` *(memory display)* pour lire la mémoire, et de `setbreak` pour mettre des points d'arrêt.
En particulier : `setbreak 0x800e` pour mettre un breakpoint à la fonction `main`
Pour plus de détails, `help md` et `help setbreak`.

Exercice 17 Introduisons d'abord les registres et les opérations logiques. Modifiez le programme comme suit :

```
        .global main
main:
        /* disable watchdog */
        mov.w   #23168, &WDTCTL
        /* initialisation de la diode rouge */
        mov.b #1, &34
```

```
        mov.b #1, r15
loop:
        /* transferer r15 vers la diode */
        mov.b r15, &33
        /* que fait cette ligne? */
        xor #1, r15
        jmp loop
```

La nouveauté est l'utilisation l'un des registres introduits dans le dessin de la page 5. L'instruction `xor #1, r15` met dans r15 le ou-exclusif (xor), bit à bit, de r15 et de la valeur 1. Essayez de prédire ce que fait ce programme. Executez ce programme pas-à-pas, et observez dans la fenêtre `mspdebug` la valeur du registre r15 au cours de l'exécution.

Exercice 18 Ajoutez au programme précédent ce qu'il faut pour que, tout en faisant clignoter la diode, il compte dans le registre r14 (il ajoute à chaque tour de boucle 1 au registre r14). Vérifiez que r14 augmente bien dans `mspdebug`.

Exercice 19 Modifiez votre programme afin de ralentir suffisamment la boucle infinie pour pouvoir observer le clignotement à l'oeil nu. Pour cela, vous allez rajouter, à l'intérieur de la boucle existante, une seconde boucle qui ne fait rien sauf perdre du temps. Ce sera l'équivalent d'une boucle `for` : elle incrémente un registre jusqu'à atteindre une certaine valeur, par exemple 20000. Pour sortir de cette boucle, vous pourrez utiliser une instruction de comparaison `CMP` et un saut conditionnel, par exemple `JNE`.
C'est une question difficile (la première fois) : si la page 11 ne vous suffit pas, ne restez pas bloqué, faites appel à un enseignant.

Survol de la syntaxe assembleur du msp430

On vous présente ici la syntaxe que vous allez devoir utiliser en TP. Elle est en général insensible à la casse (majuscules ou minuscules, c'est pareil).

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 on the next page. `SRC` et `DST` indiquent les opérandes (source et destination) sur lesquels travailler. La destination est aussi le premier opérande de l'opération, ainsi la virgule peut souvent se lire "to". Par exemple `ADD #1, R5` peut se lire "ADD 1 to R5" et, en C++, s'écrirait `R5=R5+1;`.
Une instruction spéciale est l'instruction `MOV`, par exemple `MOV R7, R5`, qui peut se lire "MOV R7 to R5" et s'écrirait en C++ `R5=R7;` [a]
En détail, 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, à préfixer par `#` : `#42`, `#0xB600`...
— le contenu d'une case mémoire désignée par son addresse, à préfixer [b] par `&` : `&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 on the following page pour plus de détails, et/ou à la doc : MSP430.pdf pages 56 et suivantes.

Drapeaux Certaines instructions, notamment les opérations arithmétiques et logiques, modifient le registre d'état (`R2`, cf encadré on page 18), 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 on the following page 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 on page 8. 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 (par défaut), ou sur des octets (il faut pour cela remplacer `OPCODE` par `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 [c]. Reportez-vous à l'encadré on page 20 pour plus de détails.

a. Et donc en termes Unix c'est cp, pas mv.

b. Si par mégarde on écrit `mov 42, R5` au lieu d'écrire `mov #42, R5` alors non seulement ça ne cause aucun message d'erreur, mais surtout le programme fera n'importe quoi. Vous voila prévenu. Et si vous voulez savoir ce qui se passe dans ce cas, assemblez puis désassemblez, puis cherchez dans la doc ce qu'on vous a caché.

c. 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.

Liste compacte des instructions MSP430

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 (MSP430.pdf, section 3.4). Il faut s'y reporter si vous avez besoin de précisions.

7 Memory-mapped IO

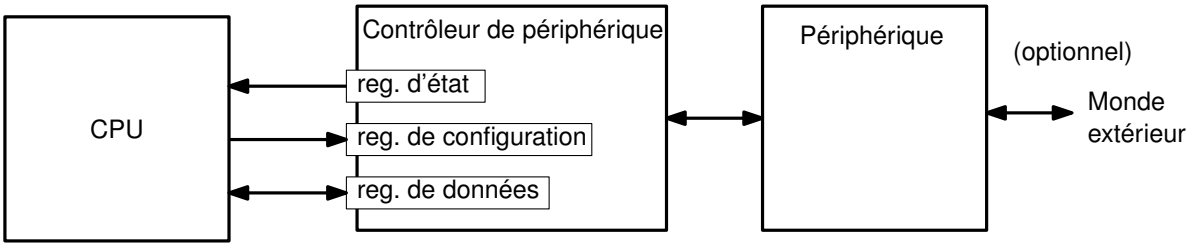
À savoir : Les entrées-sorties

Du point de vue du processeur, un périphérique se présente comme un ensemble de registres (au sens du cours d'AC), qui permettent d'échanger de l'information entre le CPU et le périphérique.
On peut distinguer informellement trois sortes de registres dans un périphérique :

— les *registres d'état* du périphérique fournissent de l'information sur l'état du périphérique : est-il actif, est-il prêt, a-t-il quelquechose à dire, etc. Ils sont typiquement accessibles en lecture seulement : le processeur peut lire leur contenu, mais pas le modifier.

— les *registres de contrôle* ou *de configuration* du périphérique sont utilisés par le CPU pour configurer et contrôler le périphérique. Ils seront typiquement accessibles en lecture-écriture, ou parfois en écriture seulement.

— les *registres de données* du périphérique permettent de lui envoyer des données (en écrivant dedans depuis le CPU) ou de recevoir des données de la part d'un périphérique (en lisant dedans).



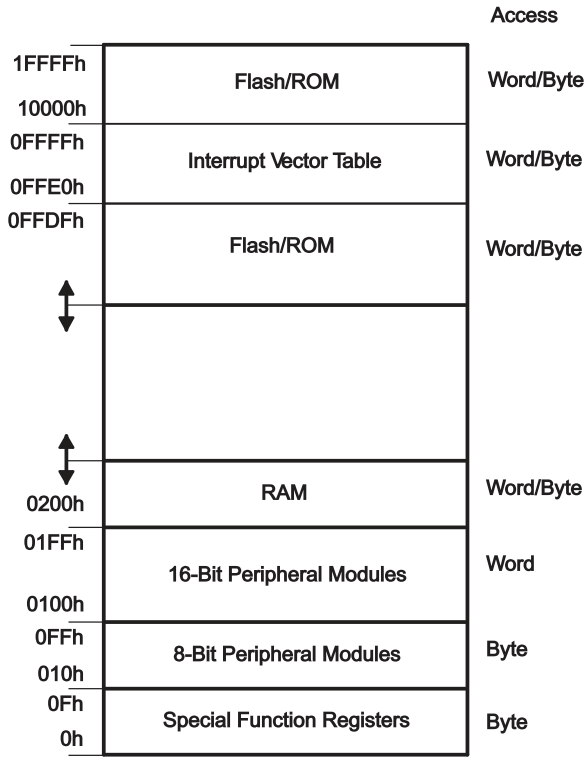
Tout cela est assez informel. Dans certains cas, un même registre peut appartenir à plusieurs de ces catégories, par exemple s'il contient à la fois des informations d'état (en lecture seule) et des information de configuration (en lecture/écriture).

La circuiterie contenant ces registres est appelée le *contrôleur* du périphérique. La plupart des boites sur la figure de la page 4 sont des contrôleurs de périphériques. Physiquement parlant, le contrôleur est parfois situé sur le périphérique lui-même, par exemple un contrôleur de disque dur. Parfois au contraire il est placée plus près du processeur (ceux de la page 4 sont tous intégrés sur la même puce). et reliée ensuite au périphérique proprement dit par un moyen quelconque. Par exemple, votre carte vidéo est reliée à votre écran par un câble VGA ou HDMI. L'architecture générique est illustrée ci-dessous :

Les registres matériels doivent pouvoir être accédés individuellement par le CPU. Comme pour les cases mémoire, on leur donne donc chacun une adresse distincte. Certains processeurs distinguent les adresses de mémoire et les adresses de registres matériels ; ils offrent alors des instructions distinctes pour accéder aux uns et aux autres. À l'inverse, la majorité des processeurs, dont notre MSP430, utilisent un unique *espace d'adressage* : certaines adresses correspondent à de la mémoire, et d'autres à des registres matériels. Les entrées-sorties se font alors avec les mêmes instructions que les accès mémoire classiques. De plus, les contrôleurs de périphériques et la mémoire se partagent les mêmes bus d'adresse et de donnée : à nouveau, voir la figure de la page 4.
On parle alors d'entrées/sorties «projetées en mémoire», ou *Memory-Mapped Input/Output*.

Utile pour le TP : le plan mémoire du msp430

Du point de vue du CPU, la mémoire principale et les périphériques se présentent tous comme des cases mémoire. Certains registres matériels font 16 bits, et occupent donc deux adresses consécutives (à gauche sur le schéma ci-dessous). Certains autres registres ne font que 8 bits, et occupent une seule adresse (à droite sur le schéma ci-dessous). Vous aurez aussi remarqué que la «mémoire» est elle-même composée d'une région de RAM (en lecture-écriture) et d'une région de mémoire flash (en lecture seule).
Pour s'y retrouver, la documentation technique nous indique le «plan d'adressage» (en VO, la *memory map*) c'est à dire une cartographie des différentes régions de l'espace d'adressage de la machine :



Ce schéma est coupé de MSP430.pdf page 25.

Exercice 20 Pour allumer notre diode on écrivait aux adresses 34 et 33. Traduisez-les en hexa et placez-les sur le plan mémoire.

Exercice 21 Cherchez la diode rouge sur le schéma de la page 3. Comment s'appelle la broche du processeur auquel elle est reliée ? (c'est illisible mais ils ont donné le même nom au fil relié à cette broche, ouf).

Exercice 22 Même question pour la diode verte.

Exercice 23 Entourez, parmi les périphériques de la figure de la page 4, celui qui commande ces deux diodes.
Ces broches sont des *general purpose input/outputs* ou GPIO. Allez lire l'introduction au chapitre 8 de msp430x2xx.pdf. Elles peuvent être configurés en entrée (I) ou en sortie (O). Ce choix se fait par écriture dans un registre de contrôle, appelé ici P1DIR et mappé à l'adresse 34. Vous reconnaissez le 34 ?
Dans le cas de la carte EZ430, il faut configurer en sortie les deux broches reliées aux diodes avant de pouvoir écrire dedans.

Exercice 24 Retrouvez les adresses de P1DIR et P1OUT dans le tableau de la page 365 de MSP430.pdf. Ces registres font 8 bits car les GPIO vont par 8. Remarque : il y a tous les sordides détails électroniques dans msp430F2274.pdf, page 57. Cette page vaut une visite juste pour la specification de P1DIR.x comme I : 0 ; O : 1. C'est clair ?
Vous retrouvez aussi cette information dans le *peripheral file map* de la doc msp430F2274.pdf page 22

Exercice 25 Vous avez à présent tout ce qu'il faut pour savoir comment allumer la diode verte.

8 Sous routines

Les instructions CALL et RET

Les appels de fonction (aka «procédure», «méthode», «sous-programme», «routine») sont tellement courants en pratique que toutes les architectures offrent des instructions dédiées pour les implémenter. 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 située à l'adresse (ou à l'étiquette) `func`, et `RET` retourne vers la fonction appelante (en fait l'adresse appelante).
Attention, erreur fréquente ! si par mégarde vous écrivez `CALL func` au lieu de `CALL #func` alors votre programme sera assemblé sans message d'erreur mais il fera n'importe quoi à l'exécution.

Exercice 26 Écrivez, en utilisant CALL et RET une procédure qui fait une pause d'une seconde environ, testez votre fonction en faisant clignoter avec des appels à cette fonction. Essayez de comprendre, en regardant la documentation des instruction (page 12), comment le RET reviens à l'endroit d'ou on a effectué le CALL.

Exercice 27 Écrivez (et testez) une procédure qui lit un argument dans R15, et le sort en binaire sur nos deux diodes : la diode verte clignote comme une horloge, et la diode rouge s'allume pour les bits à 1 (en commençant par les bits de poids faible).

Exercice 28 Écrivez une procédure qui multiplie R14 par R15 et renvoie le résultat dans R15. Testez-la sur une série de multiplications en utilisant un point d'arrêt sur cette procédure.

Exercice 29 Écrivez un procédure qui "affiche" la table de multiplication par 3.

Annexe : Morceaux choisis de la documentation de l'assembleur MSP430

Extrait de la documentation : MSP430.pdf page 44

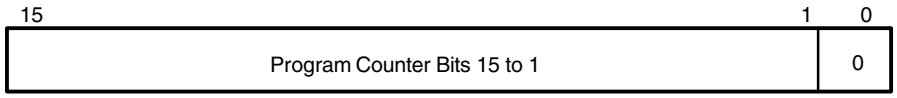
3.2 CPU Registers

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:

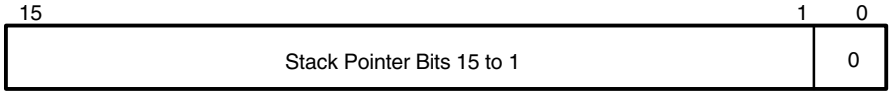
```
MOV   #LABEL,PC ; Branch to address LABEL
MOV   LABEL,PC  ; Branch to address contained in LABEL
MOV   @R14,PC   ; Branch indirect to address in R14
```

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.

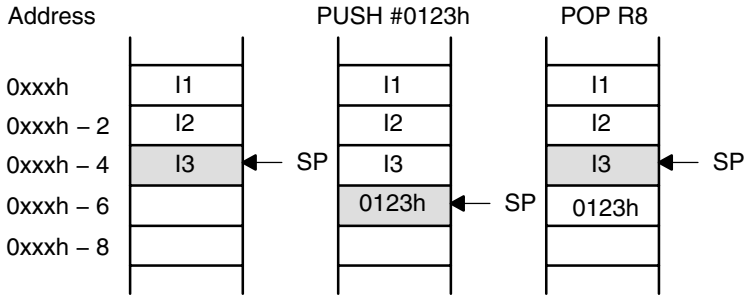
Figure 3–4 shows stack usage.

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)

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

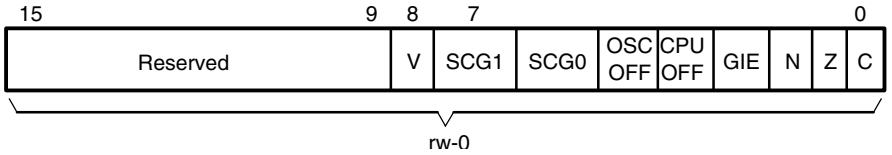


Table 3–1 describes the status register bits.

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
```

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