# ARC TD6

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

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

Objectifs pédagogiques de ce TD/TP :

1. Prendre en main la carte, mspdebug, la programmation en assembleur
2. Comprendre les memory-mapped IO

En terme de compétences (à discuter) :

— suivre une documentation technique contenant des schémas d'architecture
— lire et écrire de l'assembleur
— débugger au niveau assembleur au moyen d'un débugger
— utiliser des entrées/sorties en bare metal

C'est un peu ras les paquerettes écrit comme cela.

Dans ce TP «msp430» (qui se prolongera dans le cours CRO), on va étudier le fonctionnement d'un (petit) ordinateur réel, 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

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.

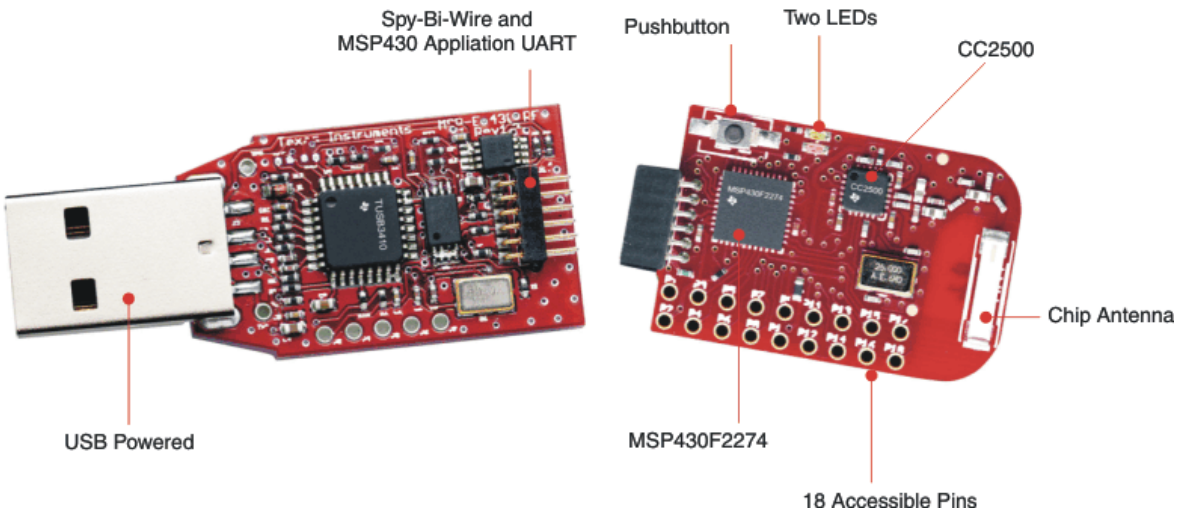
**Exercice 1** Que signifie l'acronyme USB, au fait ? Expliquez en une phrase la signification du S. Faites valider cette phrase par un enseignant, mais n'attendez pas qu'il arrive pour passer à la suite.

universal serial bus. Le S veut dire que les différents bits de chaque donnée transitent sur le bus USB les uns après les autres, sur le même fil. C'est par opposition, par exemple, au bus de données qu'on a entre le processeur et la mémoire qu'on a dans notre machine de von Neumann.

**Exercice 2** Que signifie l'acronyme JTAG ? L'explication détaillée est donnée dans l'encadré page **??**, que nous lirons en temps utile.

joint test action group. En voila de la culture qui ne sert à rien.

**Extrait de la documentation : ez430.pdf page 5**

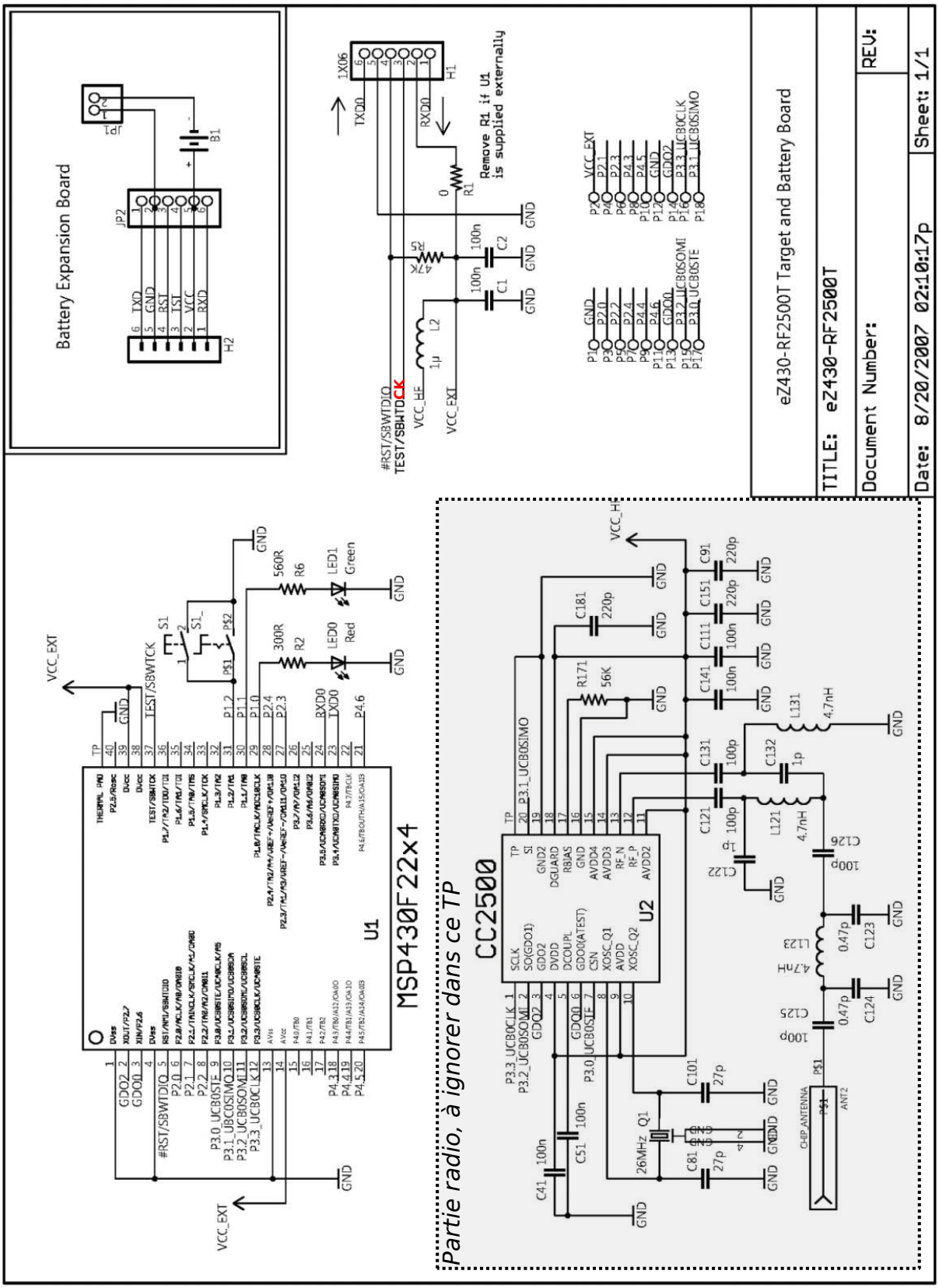



Sur la carte mère de droite (la plus petite), en plus du msp430, il y a quelques périphériques :

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 3** Retrouvez chacun de ces éléments sur le schéma ci-dessus.
L'encadré ci-dessous 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** décrit notre carte d'expérimentation et les différents composants présents sur la carte.

**MSP430.pdf** est le manuel générique de la famille MSP430. Le processeur est documenté au chapitre 3 de ce document.

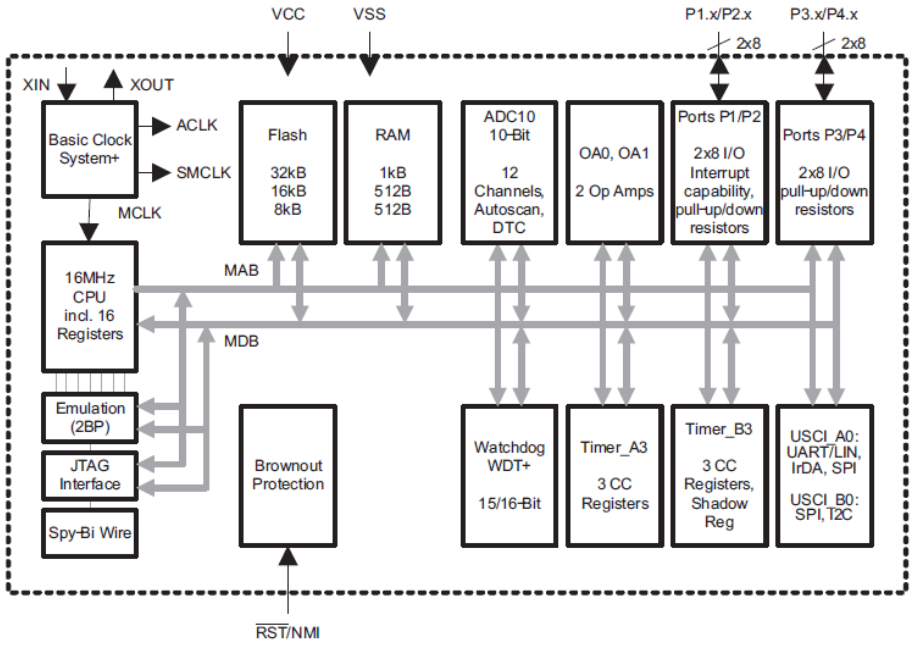
**msp430F22x4.pdf** donne les détails techniques de notre modèle précis de msp430.

## 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 : msp430F22x4.pdf page 7**

**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.

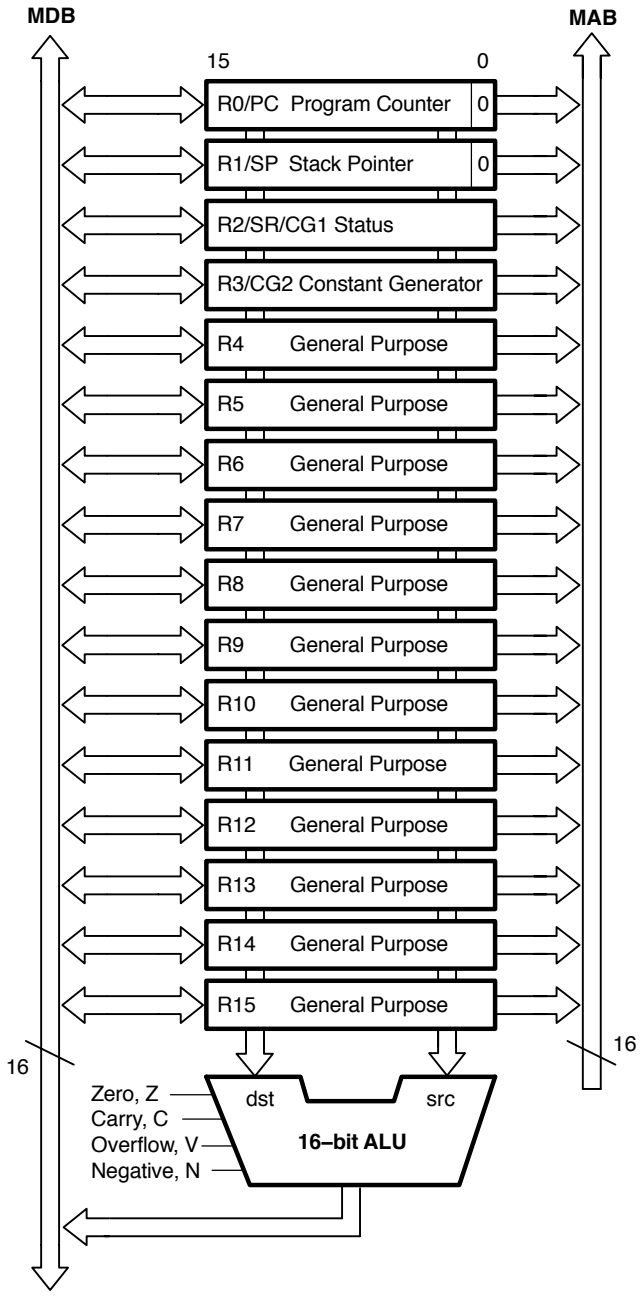
**Exercice 4** Repérez sur ce diagramme le processeur, la RAM, la mémoire flash. Vérifiez que vous connaissez le sens des acronymes CPU, RAM, ADC (sinon cherchez sur internet ou demandez à un enseignant). Ignorez les autres acronymes pour le moment.

central processing unit ; random access memory ; analog to digital converter.
Remarques : random, dans RAM, ne veut pas dire "au hasard" mais "ou on veut". C'est par opposition à la bande magnétique de l'époque (ou au CDROM tant qu'il y en a) dont l'accès est séquentiel.

## 1.2 Zoom sur le processeur

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

**Extrait de la documentation : MSP430.pdf page 44**




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

réponse : tout ça est dans le CPU.

**Exercice 6** Explicitez l'acronyme ALU.

arithmetic and logic unit

**Exercice 7** Tiens, il manque les flèches sur les fils entre l'ALU et les drapeaux. Ajoutez-les.

Tanguy : bi-directionnelles( ?), c'est pour vérifier qu'ils ont bien compris le principe des drapeaux

**Exercice 8** 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*.

spécialisé :
- ne peut contenir que certains types de donnée (e.g. des adresses, des drapeaux)
- a un rôle prédéfini en hw, et peut changer de valeur sans intervention explicite de la part du soft

général :
- pas de rôle prédéfini en hw, c'est le sw qui donne un rôle
- les registres généraux sont interchangeables

## 2 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 9** 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. Comme le montre la figure p. **??**, tous les périphériques sont mappés sur des adresses mémoire. 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 10** 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*.

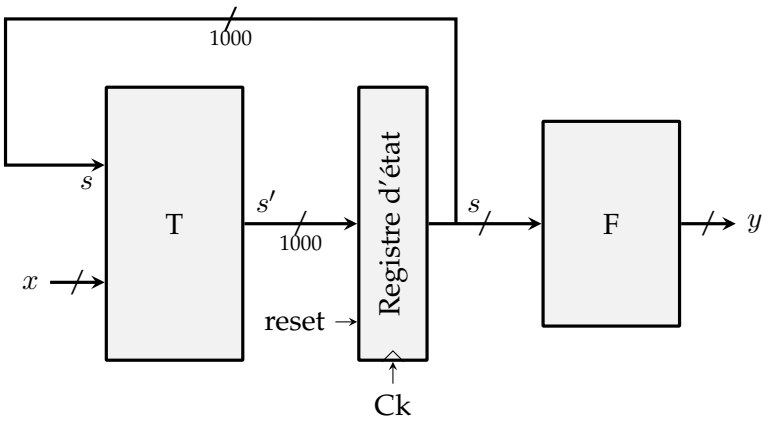
**Exercice 11** Faites s'allumer et s'éteindre la diode quelques fois.

`mw 34 1`, puis `mw 33 0` / `mw 33 1`

La mort dans l'âme, j'ai passé le cours sur JTAG dans les commentaire pour les profs, je le laisse parce que c'est bien expliqué...

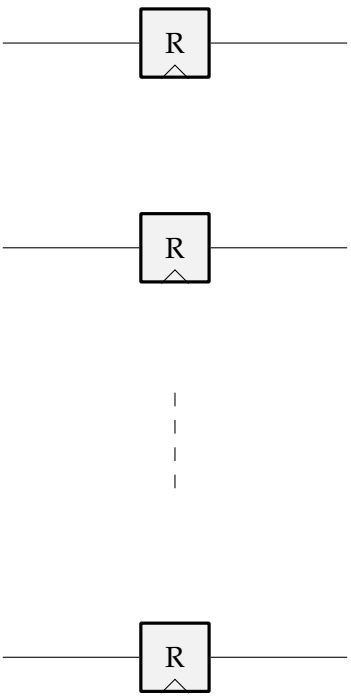
**À savoir : le JTAG**

La norme JTAG permet de lire ou d'écrire n'importe quel bit de mémoire d'un circuit combinatoire qui l'implémente.
Pour cela, on considère virtuellement l'ensemble du circuit (ici le MSP 430) comme un seul gros automate selon la figure suivante (que vous connaissez maintenant bien).

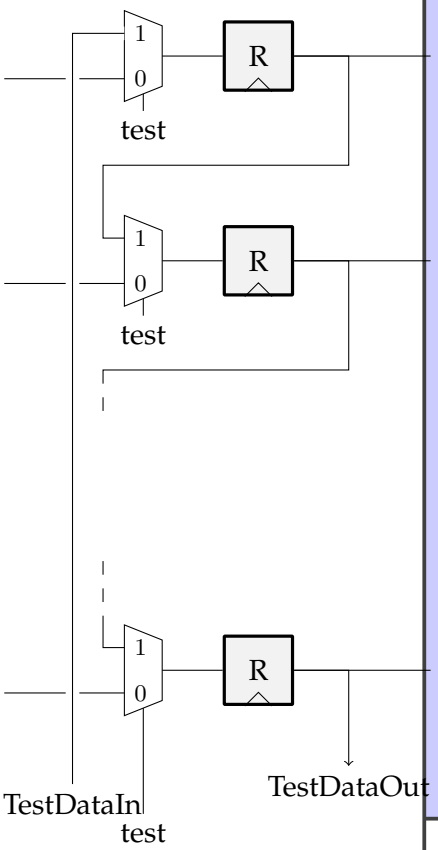



Dans cette figure, le registre d'état est un énorme registre (1000 bits sur notre exemple) qui contient le registre de l'automate de contrôle, mais aussi tous les registres de la partie opérative : les registres de la boîte à registres, tous les registres de pipeline, chaque bit de la mémoire RAM embarquée sur la puce, etc. Tout l'état du processeur, quoi. Si vous n'avez pas compris ce paragraphe, faites-le vous expliquer par un enseignant.
On ajoute (de manière automatique) à chaque flip-flop de cet immense registre un tout petit peu de circuiterie pour faire de l'ensemble des 1000 registres binaires un unique immense registre à décalage. C'est une transformation automatique qui est décrite par la figure ci-dessous :

Le registre d'état de la figure ci-dessus, avant...




... et après sa transformation en JTAG



## Le JTAG, suite

Avec tout cela, le circuit fonctionne normallement lorsque test est à 0. Et on peut, en 1000 cycles, mettre le circuit dans un état quelconque. Il suffit de mettre test à 1, et de pousser l'état qu'on veut dans le grand registre à décalage ainsi obtenu. Dans le même temps, l'état précédent du circuit sort sur testOut : on peut également, toujours en 1000 cycles d'horloge, lire l'état complet du processeur. C'est ce qu'on va faire dans ce TP pour copier notre programme dans la mémoire du MSP430, mais aussi pour observer quand on le désire les valeurs des registres.
Mais pourquoi cela s'appele JTAG ? Parce que cela sert surtout à tester chaque puce, y compris les plus complexes comme votre Pentium, avant de le mettre en boîte. En effet, lors du processus de fabrication, il arrive souvent qu'une poussière malencontreuse rende un transistor inopérant. Comment détecter cette situation pour jeter les puces défectueuses au plus tôt ?
Bien sûr, on pourrait lui faire booter Linux puis Windows et jouer un peu à Quake dessus, et on se dirait qu'on a tout testé. Mais cela prendrait de longues minutes par puce, et le temps c'est de l'argent. Voici une technique qui permet de tester toute la puce en quelque centaines de milliers de cycles seulement (comptez combien de cycles à 4GHz il faut pour booter Linux en 20s).
Une fois ceci en place, on teste le circuit comme ceci :

— On met *test* à 1, puis on pousse un état connu, pas forcément utile, dans le processeur.

— Puis on met *test* à 0, et on fait tourner le processeur pendant quelques centaines de cycles.

— Il fait sans doute n'importe quoi, mais ce n'est pas grave.

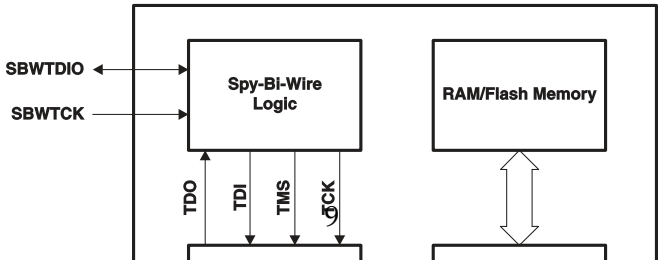
— On remet *test* à 1, et on sort l'état complet du processeur (tout en poussant un nouvel état).

— On compare l'état obtenu avec l'état (obtenu par simulation) dans lequel doit être le processeur si chacune de ses portes fonctionne correctement. S'il y a une différence, on le jette !

— Et on recommence plusieurs fois, avec des états construits pour faire fonctionner tous les transistors de l'énorme fonction $T$ – pas forcément des états dans lequel le processeur peut se trouver en fonctionnement normal.

Tout ceci est même normalisé par le Joint Test Action Group : JTAG.
Et le rapport avec notre interface JTAG ? Eh bien, une fois qu'on a ce mécanisme en place, on peut même s'en servir pour débugger : on peut aller observer ou changer la valeur de n'importe quel registre du processeur en quelque dizaines de milliers de cycle. Il suffit de lire l'état, changer les bits qu'on veut, et réécrire l'état modifié. C'est comme cela que vous pourrez, dans ce TP, observer dans mspdebug ce qui se passe à l'intérieur de votre MSP430.
Bien sûr, le nombre des registres et l'ordre dans lequel ils sont chaînés dépend du microcontrôleur utilisé, c'est pourquoi on doit passer l'argument rf2500 à mspdebug.

Parenthèse moins importante : Comme tout cela était trop simple, on a dans notre MSP430 une couche supplémentaire qui permet d'accéder au JTAG avec 2 fils seulement. Cela s'appelle Spy-By-Wire et c'est décrit sur le schéma ci-dessous. Vous y retrouvez d'un côté les 4 fils du JTAG : (TDI pour Test Data In, etc), et de l'autre côté les deux signaux SBWTDIO (spy by wire test data in/out) et SBWTCK (clock) qu'on avait sur le schéma de la page ??.



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

**Exercice 12** Créez un nouveau répertoire TPMSP430, et retapez[1] dans un fichier `ex12.s` le programme suivant :

```
.section .init9

main:
        /* 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,

- `.section .init9` est une commande à destination de msp430-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*).
- 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...

**Exercice 13** Traduisez ce programme en un exécutable en langage machine avec la commande suivante :

```
msp430-gcc -mmcu=msp430f2274 -mdisable-watchdog -o tp1.elf tp1.s
```

Les deux options sont importantes. La première, `-mmcu=msp430f2274`, indique la puce exacte ciblée. La seconde, `-mdisable-watchdog`, débranche 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 dans cette option, il n'y aura pas de message d'erreur mais le programme fera n'importe quoi, puisqu'il rebootera sans fin.

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).

**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-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 (et vous pouvez remplacer tp1 par ce que vous voulez.)

**Exercice 14** Désassemblez le programme obtenu par

```
msp430-objdump -d tp1.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 ?

Réponses attendues : ff3f, 803a, c'est un saut relatif puisqu'il est assemblé en jmp +0. Enfin, c'est le linker qui a, en particulier, inséré du code qui débranche le watchdog.

**Exercice 15** 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`.

## 4 Exécution d'un programme pas à pas

**Exercice 16** Dans `tp1.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 `load` puis exécutez de nouveau votre programme par `run` dans mspdebug.
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.

```
.section .init9

main:
        /* initialisation de la diode rouge */
        mov.b #1, &34
loop:
        /* eteindre */
        mov.b #0, &33

        /* allumer */
        mov.b #2, &1
        jmp loop
```

TODO : j'en suis là, elaguer la suite pour que ca fasse 4 heures environ

## 5 Programmation en assembleur : variables et boucles

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 ci-dessous et on this page.

**Débuggage : points d'arrêts**

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` *(memory display)* pour lire la mémoire, et de `setbreak` pour mettre des points d'arrêt.
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 :

```
.section .init9
main:
  mov.b #1, &34   /* initialisation de la diode rouge */
        mov.b #1, r15   /* valeur initiale de la valeur de la diode */
loop:
  mov.b r15, &33  /* transferer r15 vers la diode */
        xor #1, r15     /* que fait cette ligne? */
  jmp loop
```

La nouveauté est l'utilisation l'un des registres introduits dans le dessin de la page **??**. 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 ?? 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 current 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++ `R7=R5;` [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 this 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 the current page), 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 this 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 the current page. 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 this page 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.

**Attention, piège !** 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. En effet, un coup de objdump révèle que `mov 42, R5` est assemblé en `mov -16(R0), R5` (pour une certaine valeur de 16) autrement dit on est en train de mettre n'importe quoi dans R5.
L'explication c'est que la syntaxe sans le dièse est interprétée comme le mode d'adressage «symbolique» (cf [MSP430.pdf section 3.3]) et donc on est en train de faire référence à une donnée située *42 octets après le début de la fonction courante* (on se rappelle que R0 est le PC).

Moralité : méfiance, ayez l'oeil sur les # qui pourraient manquer dans le code des étudiants.

GS : et d'ailleurs je ne vois vraiment pas à quoi peut servir ce mode d'adressage en pratique. À faire un «constant pool», avant le début de la fonction ? non, car l'offset -16 prend autant de place que la constante proprement dite. des idées ?
F2D : Je suppose que cela permet de faire des "variable pools" tout en gardant le code relogeable.

## 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.

# 6 Memory-mapped IO

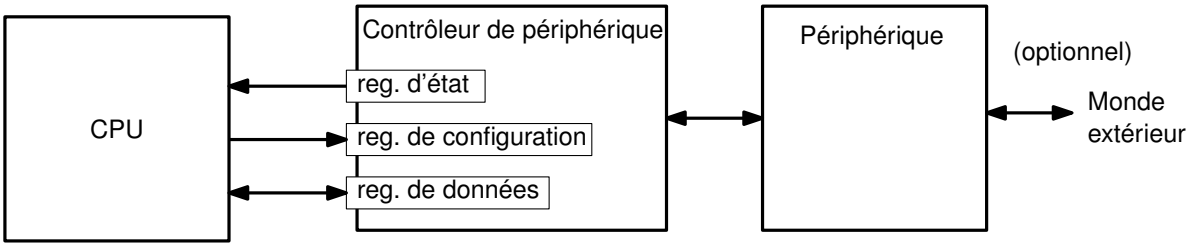
**À savoir : Les entrées-sorties**

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).



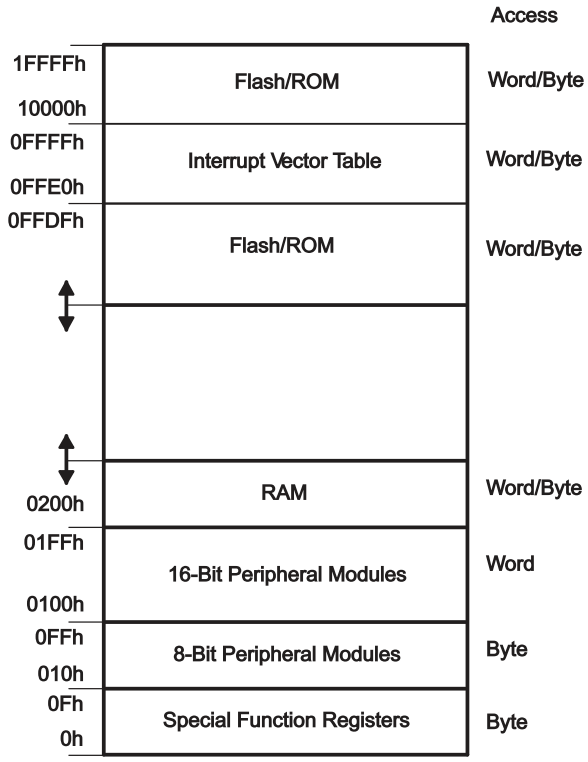

Tout cela est assez informel. Dans certains cas, un même registre peut appartenir à plusieurs de ces catégories, par exemple s'il contient à la fois des informations d'état (en lecture seule) et des information de configuration (en lecture/écriture).

La circuiterie contenant ces registres est appelée le *contrôleur* du périphérique. La plupart des boites sur la figure de la page ?? 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 ?? 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 ??.
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.

Je n'ai pas compris comment on accède aux 64Ko de flash d'adresses plus grandes que FFFF. La doc n'en parle pas.

**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 ??. 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 ??, celui qui commande ces deux diodes.

Les noms sont p1.0 et p1.1. Le périphérique est l'avant dernier en haut à gauche.

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 333 de MSP430.pdf. Ces registres font 8 bits car les GPIO vont par 8. Remarque : il y a tous les sordides détails électroniques dans msp43022x4.pdf, page 58. Cette page vaut une visite juste pour la specificationw de P1DIR.x comme I : 0 ; O : 1. C'est clair ?

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

# 7 Sous routines

**Les instructions CALL et RET**

**Exercice 26** 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 27** 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 28** 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**



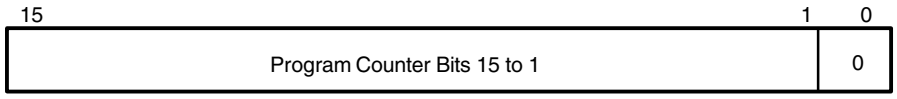
## 3.2 CPU Registers

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:

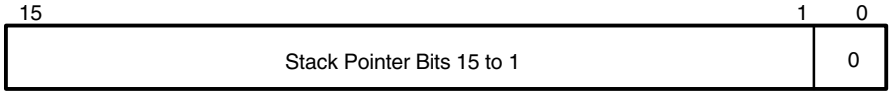
```
MOV   #LABEL,PC ; Branch to address LABEL
MOV   LABEL,PC  ; Branch to address contained in LABEL
MOV   @R14,PC   ; Branch indirect to address in R14
```

### 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.

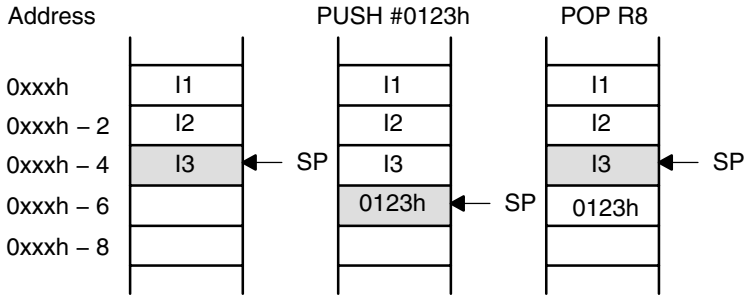
Figure 3–4 shows stack usage.

*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*

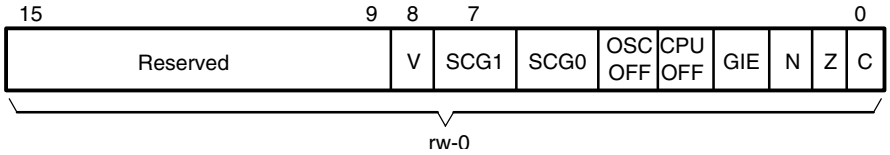



Table 3–1 describes the status register bits.

*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:<br>Positive + Positive = Negative<br>Negative + Negative = Positive,<br>otherwise reset |
| | `SUB(.B),SUBC(.B),CMP(.B)` | Set when:<br>Positive – Negative = Negative<br>Negative – Positive = Positive,<br>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*