Architecture des Ordinateurs 2023/2024 - TD

Dans cette série de TP «msp430», 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.
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.

msp430x4xx.pdf est le manuel générique de la famille MSP430. Le processeur est documenté au chapitre 3 de ce document.

datasheet-msp430g4618.pdf donne les détails techniques de notre modèle précis de msp430.

Partie I: MSP 430 - Prise en main

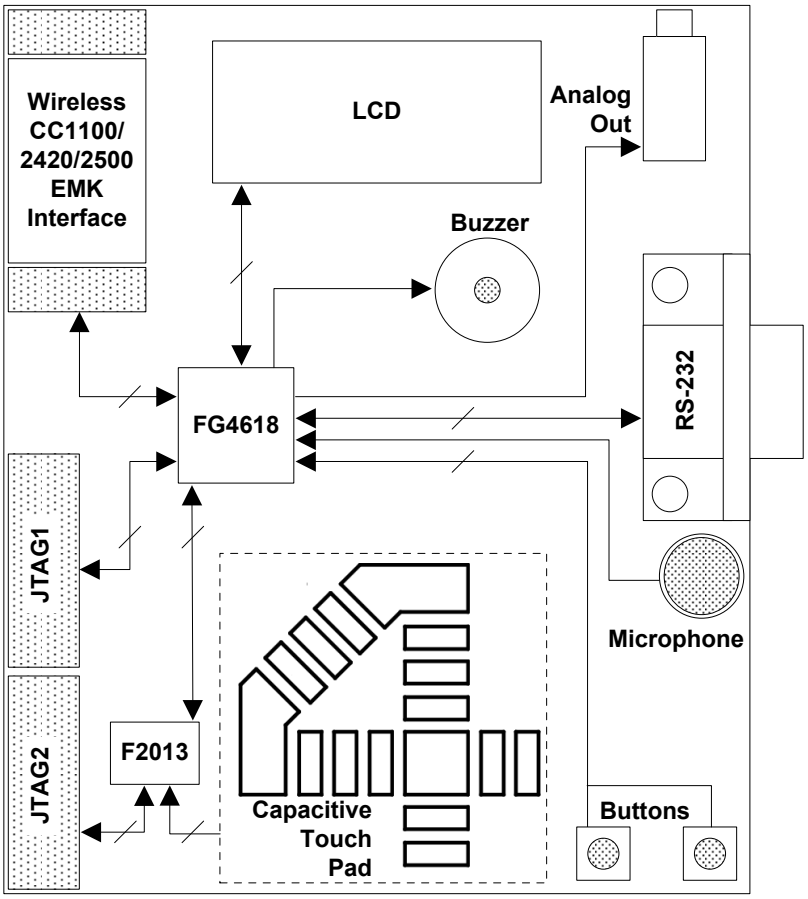
1 Découverte de la carte

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.

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.

Exercice 2 Que signifie l'acronyme JTAG ? L'explication détaillée est donnée dans l'encadré page 7, que nous lirons en temps utile.

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)

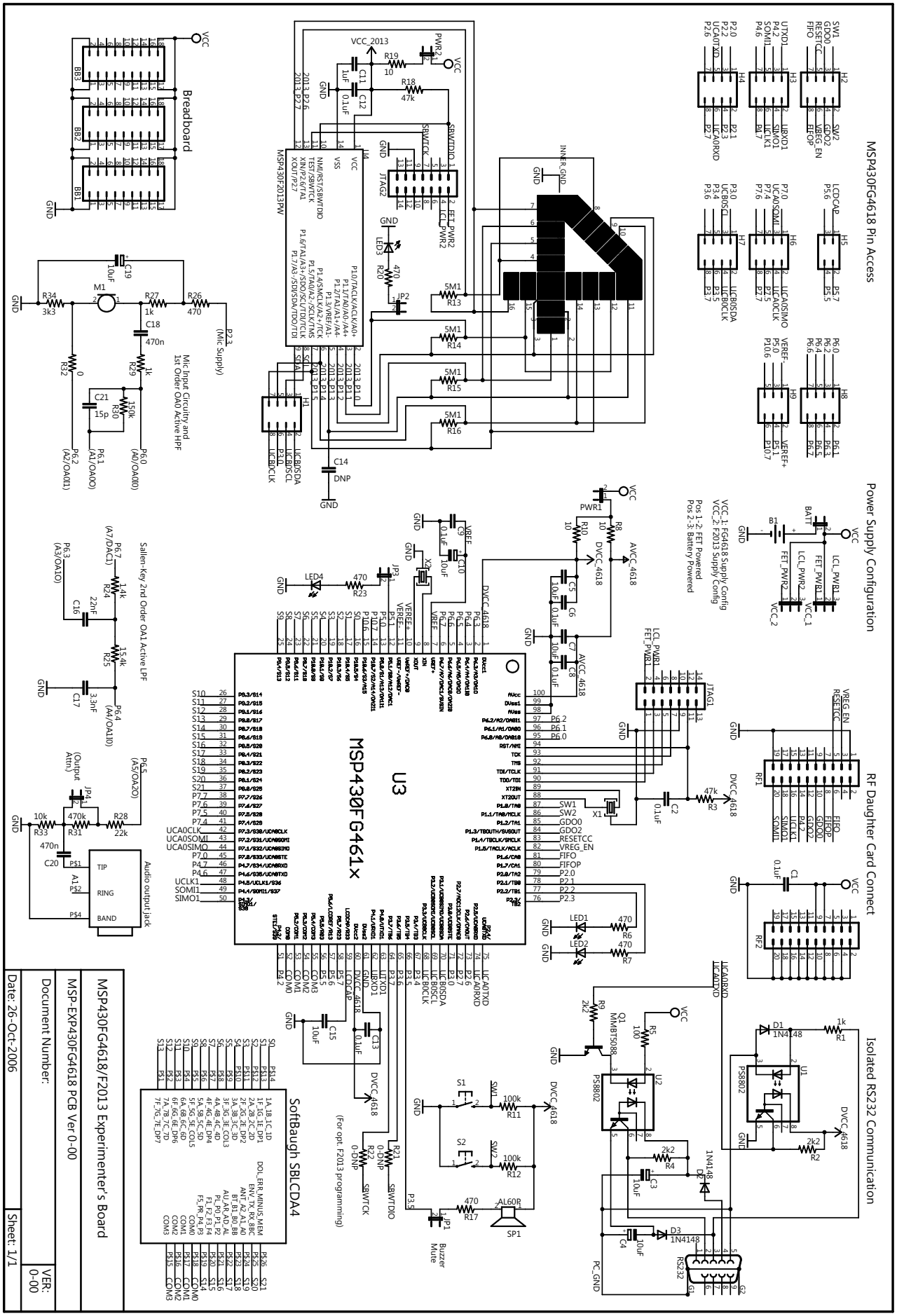
etc. ...

Exercice 3 Pour chacun de ces éléments, indiquez sur le schéma ci-dessus 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 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 4 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.

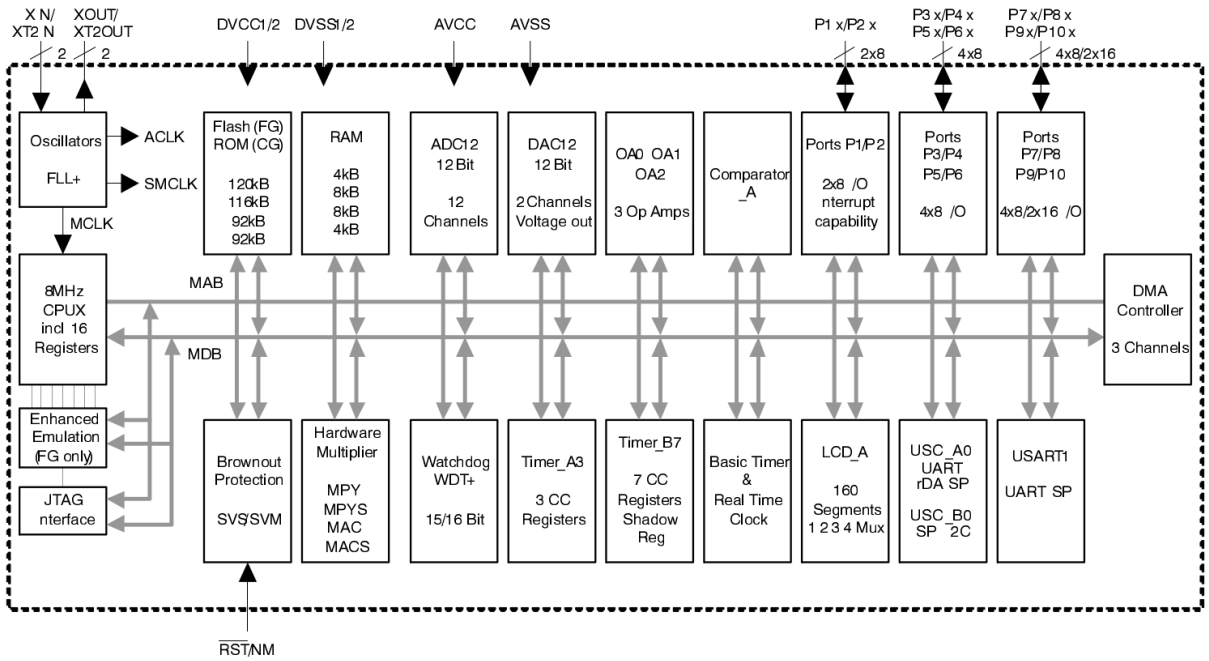
Extrait de la documentation : motherboard.pdf page 19



1.1 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 contrôleurs de périphériques. Si on zoome sur l'intérieur de la puce, on a donc affaire à l'architecture illustrée ci-dessous.

Extrait de la documentation : datasheet-msp430g4618.pdf page 5



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.
Vous pouvez constater qu'ici, le reste du monde ne se limite pas à la mémoire comme dans notre micro-machine... Nous allons détailler tout cela.

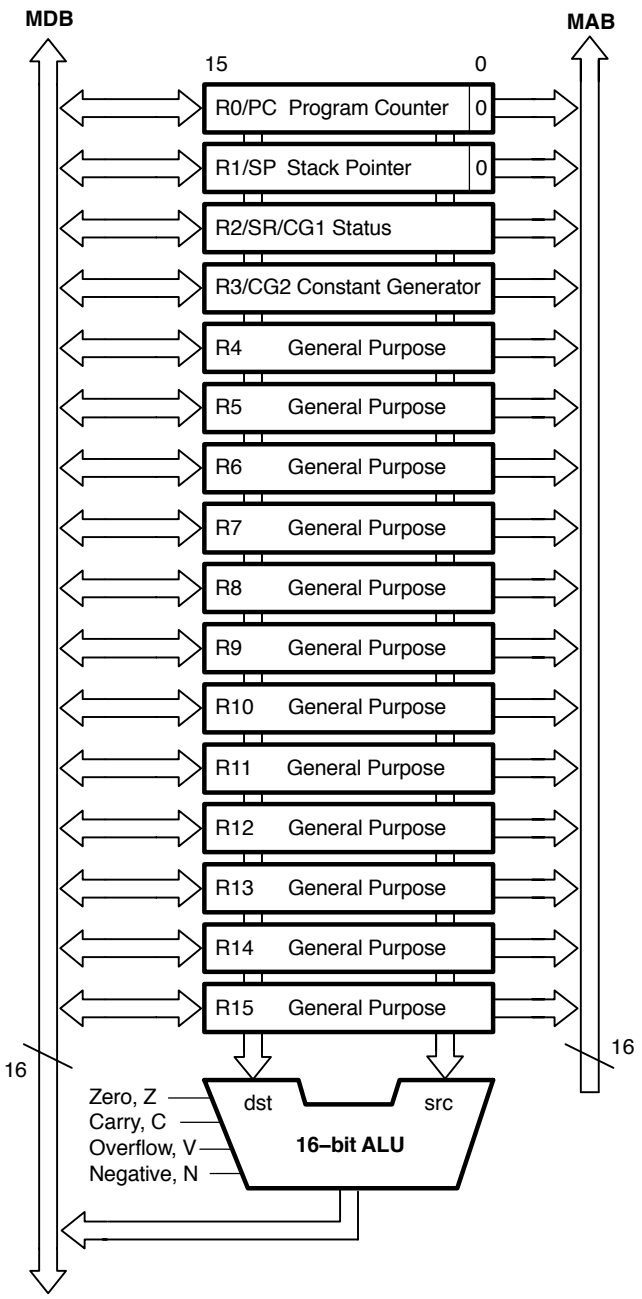
Exercice 5 Repérez sur ce diagramme le processeur, la RAM, la mémoire flash. Vérifiez que vous connaissez le sens des acronymes RISC, CPU, RAM, ADC, DAC. Sinon, ouvrez le glossaire qui est tout au début de msp430x4xx.pdf (p. 4). Demandez des explications à un enseignant si nécessaire. Ignorez les autres acronymes pour le moment.

Exercice 6 Branchez maintenant la sonde JTAG sur la carte. Vous devriez pouvoir choisir sans difficulté entre les deux connecteurs JTAG en regardant la figure de la page 2.

1.2 Zoom sur le processeur

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

Extrait de la documentation : msp430x4xx.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 : l'automate de contrôle, le 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 compteur ordinal, etc). À l'inverse les 12 autres registres sont *généraux*, on (le programmeur) peut y mettre ce qu'on veut.

Exercice 7 Sur le schéma de la page 4, indiquez où se trouvent nos 16 registres, ainsi que l'automate de contrôle.

Exercice 8 Explicitez l'acronyme ALU.

Exercice 9 Tiens, il manque les flèches sur les fils entre ALU et les drapeaux. Ajoutez-les.

Exercice 10 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 Prise en main des outils : mspdebug

Pour que la suite marche il faut avoir tapé une fois dans votre terminal la ligne suivante (attention, par défaut il faudra répéter cette commande à chaque ouverture d'un nouveau terminal) :

```
source /opt/msp430-toolchain/env.sh
```

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 11 Branchez la carte, et lancez mspdebug en tapant la ligne commande suivante :

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

L'argument `uif` est le nom du driver à utiliser, ici celui de notre boîtier JTAG.
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 les mémoires de la puce en tapant dans mspdebug la commande `erase`.

On va maintenant se servir de mspdebug pour allumer et éteindre la diode LED4. Comme illustré par la figure p. 4, tous nos périphériques sont «mappés» sur des adresses mémoire : en écrivant les bonnes valeurs aux bonnes adresses, on peut contrôler ces périphériques.
Par exemple, pour activer cette diode, il faut tout d'abord écrire la valeur 2 à l'adresse 50. Ensuite, on allumera la diode en écrivant la valeur 2 à l'adresse 49, et on l'éteindra en écrivant 0 à l'adresse 49. Admettons ces valeurs pour l'instant, nous les expliquerons dans un moment.

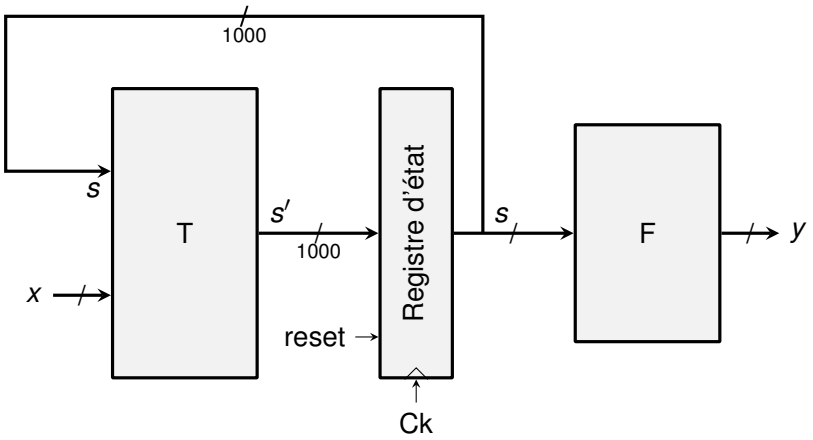
Exercice 12 Toujours dans mspdebug, tapez `help mw` et lisez l'aide de la commande *memory write*. Remarquez au passage que vous pouvez aussi taper `help` tout court pour obtenir la liste des commandes disponibles, et `help` *bidule* pour obtenir de l'aide sur la commande *bidule*.

Exercice 13 Faites s'allumer et s'éteindre la diode quelques fois.

À savoir : le JTAG

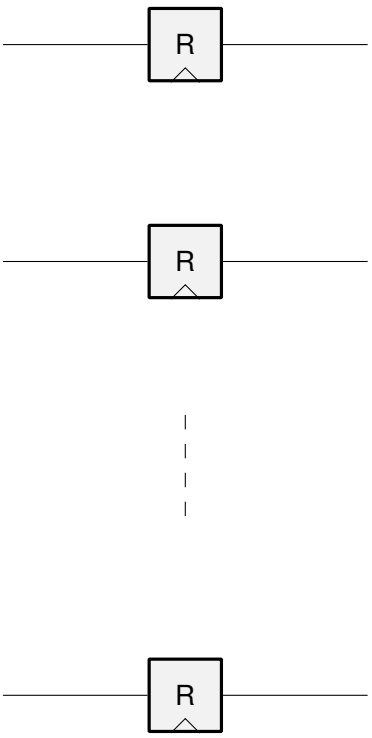
L'acronyme JTAG désigne une méthode permettant de lire ou d'écrire n'importe quel bit de mémoire d'un circuit séquentiel. Cette méthode nécessite «seulement» des modifications mineures à l'intérieur du circuit, ainsi qu'une poignée de signaux connectés au monde extérieur (de deux à cinq fils, suivant les variantes du protocole).

Le principe de base est simple : il s'agit de considérer virtuellement l'ensemble du circuit (ici le MSP 430) comme un seul gros automate, selon la figure suivante que vous connaissez maintenant bien.

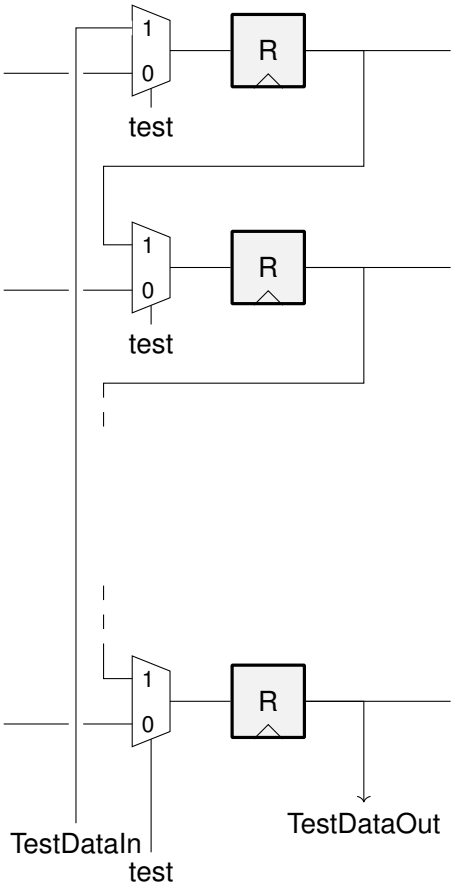


Dans cette figure, le registre d'état est un énorme registre (1000 bits sur notre exemple) qui contient le registre de l'automate de contrôle, mais aussi tous les registres de la partie «datapath» : les registres de la boîte à registres, tous les registres de pipeline, la valeur des flags etc. Tout l'état 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 contrôler l'exécution de notre programme en pas-à-pas, 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).

— 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.
Il y a même une boîte nommée *JTAG interface* qui permet, à travers le JTAG, d'observer aussi tout le contenu de la RAM et des périphériques.
Bien sûr, le nombre des registres et l'ordre dans lequel ils sont chaînés dépend du microcontrôleur utilisé, c'est pourquoi on doit passer les bons argumentsà mspdebug.

3 Assemblage et exécution d'un programme

Exercice 14 Créez un nouveau répertoire TPMSP430, et retapez dans un fichier `ex14.s` le programme suivant :

```
.section .init9

main:
        /* initialisation de la diode rouge */
        mov.b #2, &50

        /* eteindre */
        mov.b #0, &49

        /* allumer */
        mov.b #2, &49

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 la case mémoire d'adresse 17.
- donc `mov.b #2, &49` est une instruction assembleur qui réalise une copie de la valeur constante 2 vers la case mémoire d'adresse 49. 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 désigne un emplacement dans notre programme, qu'on peut utiliser par exemple comme destination dans les instruction de saut (ou autres). Pour un saut absolu comme notre `jmp loop`, GCC utilisera dans le langage machine l'adresse réelle de l'étiquette. Pour un saut relatif, il calculera la distance de saut (aka déplacement ou offset) en faisant une soustraction entre l'adresse de départ et l'adresse de destination.
- 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 15 Traduisez ce programme en un exécutable en langage machine avec la commande suivante :

```
msp430-gcc -mmcu=msp430fg4618 -mdisable-watchdog -o prog.elf ex14.s
```

Les deux options sont importantes. La première, `-mmcu=msp430fg4618`, indique la puce exacte ciblée. La seconde, `-mdisable-watchdog`, débranche le *watchdog*, un composant matériel 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.

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.

Exercice 16 Désassemblez le programme obtenu par

```
msp430-objdump -d prog.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 cette instruction est-elle assemblée ?
- Est-ce un saut relatif ou un saut absolu ?
- D'où viennent toutes ces instructions supplémentaires, autour de votre programme ?

Exercice 17 Depuis mspdebug, transférez votre programme sur la carte en utilisant la commande `prog prog.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

À partir d'ici, il est productif d'avoir deux terminaux ouverts : l'un dans lequel `mspdebug` reste ouvert, l'autre dans lequel vous exécutez vos `msp430-gcc`. Cela permet de conserver l'historique des commandes passées dans `mspdebug`.

Exercice 18 Copiez `ex14.s` en un nouveau fichier `ex18.s`, déplacez les instructions d'allumage et d'extinction à l'intérieur de la boucle infinie : le but est de faire clignoter la diode. Assemblez par `msp430-gcc`, puis dans mspdebug chargez votre programme par `prog` et exécutez-le de nouveau par `run`.
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.

5 Programmation en assembleur : variables et boucles

Vous allez maintenant devoir modifier votre programme un peu plus sérieusement. Pour la syntaxe de l'assembleur MSP430, aidez-vous des explications qui sont données dans les deux encadrés page 12 et page 13.

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

```
.section .init9
main:
  mov.b #2, &50   /* initialisation de la diode */
  mov   #2, r15   /* valeur initiale de la valeur de la diode */
loop:
  mov.b r15, &49  /* transferer r15 vers la diode */
  xor #2, r15     /* que fait cette ligne? */
  jmp loop
```

La nouveauté est l'utilisation de l'un des registres visibles sur le dessin de la page 5. L'instruction `xor #2, r15` met dans r15 le ou-exclusif (xor), bit à bit, de r15 et de la valeur 2. Remarquez que nous travaillons sur r15 avec des instruction sans le suffixe `.b` : ces instructions travaillent sur 16 bits, pas juste 8. 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 20 Ajoutez au programme précédent ce qu'il faut pour que, tout en faisant clignoter la diode, il compte les tours de boucle 1 dans le registre r14). Vérifiez que r14 augmente bien dans `mspdebug`.

Exercice 21 **Question difficile, n'hésitez pas à appeler à l'aide.** 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 assembleur d'une boucle `for(i=2000; i>0; i--){}` Partant d'une certaine valeur, par exemple 20000, stockée

dans un registre, par exemple R13, elle décrémente ce registre à chaque tour. Pour sortir de la boucle il faut un saut conditionnel, par exemple `JNZ` (vous pouvez utiliser le fait quer le drapeau Z est mis à jour par l'instruction SUB qui décrémente votre registre). Attention, 20000 tient sur 16 bits mais pas sur 8 bits : utilisez des instructions sans l'extension `.b`.

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). Ne touchez pas aux registres R0 à R3, ils sont spéciaux, voir le dessin de la page 5.

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 (source et destination) sur lesquels travailler. La destination est aussi le second 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`...
- le contenu d'une case mémoire dont l'adresse est la valeur contenue dans un registre, alors ce registre est préfixé par `@`. Par exemple `MOV R7, @R5`, s'écrirait en C++ ainsi : `*R5=R7;`

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 pages 56 et suivantes.

Drapeaux Certaines instructions, notamment les opérations arithmétiques et logiques, modifient 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 `JMP label`. Regardez par exemple le programme 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].

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 (msp430x4xx.pdf, section 3.4). Il faut s'y reporter si vous avez besoin de précisions.

6 Memory-mapped IO

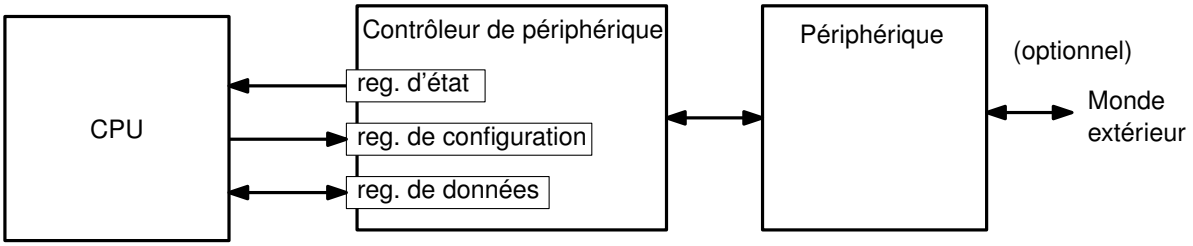
À savoir : Les entrées-sorties

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).



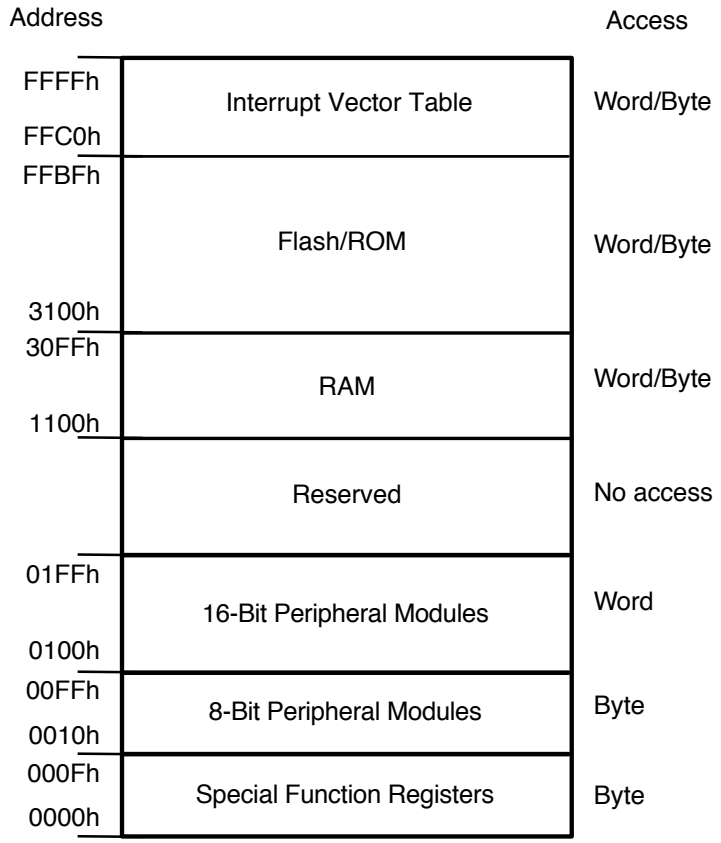
Tout cela est assez informel. Dans certains cas, un même registre peut appartenir à plusieurs de ces catégories, par exemple s'il contient à la fois des informations d'état (en lecture seule) et des information de configuration (en lecture/écriture).

La circuiterie contenant ces registres est appelée le *contrôleur* du périphérique. La plupart des boites sur la figure de la page 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. 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 l'espace d'adressage de la machine :



Exercice 22 Pour allumer notre diode on écrivait aux adresses 50 et 49. Traduisez-les en hexa (de tête !) et placez-les sur le plan mémoire.

Exercice 23 Cherchez la diode LED4 sur le schéma de la page 3. Comment s'appelle la broche du processeur auquel elle est reliée ? La partie intéressante commence par P (comme Port d'entrée-sortie).

Ces broches sont des *general purpose input/outputs*, ou GPIO. Comme écrit en introduction du chapitre 11 de msp430x4xx.pdf, *MSP430 devices have up to ten digital I/O ports implemented, P1 to P10. Each port has eight I/O pins. Every I/O pin is individually configurable for input or output direction, and each I/O line can be individually read from or written to.*
Ces registres font 8 bits, ainsi les GPIO sont groupés par paquets de 8, juste parce qu'il sont mappés sur des cases mémoire de 8 bits. La notation P5.1 se lit «le bit numéro 1 du port 5».
Chaque GPIO peut être configuré en entrée (I) ou en sortie (O). Ce choix se fait (bit à bit) par écriture dans un registre de contrôle. Pour le port 5 ce registre s'appelle P5DIR et est mappé à l'adresse 50. Vous reconnaissez le 50 ?
Les 8 bits de chaque port sont numérotés de 0 à 7. La valeur 2 qu'on écrivait à l'adresse 50 levait le bit numéro 1, dont la valeur en binaire est $2^1 = 2$. Si on voulait lever le bit 3 on écrirait la valeur $2^3 = 8$. Si on voulait lever à la fois le bit 1 et le bit 3, on écrirait la valeur binaire 00001010, soit 0x0A, ou 10 en décimal.

11.2 Digital I/O Operation

The digital I/O is configured with user software. The setup and operation of the digital I/O is described in the following sections. Each port register is an 8-bit register and is accessed with byte instructions. Registers for P7/P8 and P9/P10 are arranged such that the two ports can be addressed at once as a 16-bit port. The P7/P8 combination is referred to as PA and the P9/P10 combination is referred to as PB in the standard definitions file. For example, to write to P7SEL and P8SEL simultaneously, a word write to PASEL would be used. Some examples of accessing these ports follow:

```
BIS.B  #01h,&P7OUT     ; Set LSB of P7OUT.
                       ; P8OUT is unchanged
MOV.W  #05555h,&PAOUT  ; P7OUT and P8OUT written
                       ; simultaneously
CLR.B  &P9SEL          ; Clear P9SEL, P10SEL is unchanged
MOV.W  &PBIN,&0200h    ; P9IN and P10IN read simultaneously
                       ; as 16-bit port.
```

11.2.1 Input Register PxIN

Each bit in each PxIN register reflects the value of the input signal at the corresponding I/O pin when the pin is configured as I/O function.

Bit = 0: The input is low

Bit = 1: The input is high

Note: Writing to Read-Only Registers PxIN

Writing to these read-only registers results in increased current consumption while the write attempt is active.

11.2.2 Output Registers PxOUT

Each bit in each PxOUT register is the value to be output on the corresponding I/O pin when the pin is configured as I/O function and output direction.

Bit = 0: The output is low

Bit = 1: The output is high

11.2.3 Direction Registers PxDIR

Each bit in each PxDIR register selects the direction of the corresponding I/O pin, regardless of the selected function for the pin. PxDIR bits for I/O pins that are selected for other module functions must be set as required by the other function.

Bit = 0: The port pin is switched to input direction

Bit = 1: The port pin is switched to output direction

Exercice 24 Retrouvez l'adresse de P5DIR dans le tableau de la page 413 de msp430x4xx.pdf. On trouve en bas de la p. 409 qu'un bit à 1 configure le port en sortie alors qu'un bit à 0 le configure en entrée. Ainsi au reset du MSP430 toutes les IO sont configurées en entrées et la puce ne risque pas d'agir sur son environnement tant qu'elle n'est pas programmée pour.

Exercice 25 Qu'est-ce qui est mappé à l'adresse 49 ? Voici en principe démystifiés les deux `mw` du début du TP. Si ce n'est pas encore tout clair, posez des questions...

Exercice 26 Et maintenant passons aux deux autres diodes LED1 et LED2 : cherchez-les sur le schéma de la page 3, trouvez à quel bit de quel port elles sont reliées, enfin allumez-les séparément puis *toutes les deux ensemble* (faites vos tests en utilisant des `mw` dans `mspdebug` plutôt que des itérations de compilation / prog / run...)

Exercice 27 Le buzzer (cherchez-le sur les schémas) se commande comme une diode : pour jouer une note, il suffit de le faire «clignoter» à la bonne fréquence. Essayez de jouer une note. Vous pouvez faire monter et descendre la note pour faire des bruits de sirènes... faites un concours de sons et lumières.
Vous voudrez peut-être définir l'équivalent assembleur d'une procédure : l'instruction `CALL #toto` fait un saut à l'étiquette toto, tout comme `JMP toto` (mais attention, il faut un `#` devant l'étiquette). En plus l'adresse suivant l'instruction `CALL` est mémorisée : ainsi, l'instruction `RET`, à placer à la fin du code de votre procédure, transfère l'éxécution après le `CALL`. Et si vous voulez passer un paramètre à votre procédure, vous le passez dans un registre...

7 S'il reste du temps

Exercice 28 Écrivez (et testez) un programme qui lit un argument dans R15, et le sort en binaire sur nos deux diodes : une diode clignote comme une horloge, et une autre diode s'allume pour les bits à 1 (en commençant par les bits de poids faible).

Exercice 29 Écrivez une procédure qui divise R14 par R15 et renvoie le résultat dans R15.

Partie II: MSP430 - Pile, appels de fonctions, interruptions

Le mécanisme des interruptions matérielles est un rouage essentiel dans le fonctionnement des ordinateurs. L'objectif de ce dernier TP est de le mettre en œuvre sur la plateforme MSP430. En particulier, on va s'intéresser aux notions suivantes : requête d'interruption (IRQ), vecteur d'interruption, priorités, masquage d'interruption, routine de traitement d'interruption (ISR), sauvegarde de contexte, acquittement d'interruption.
Nous en avons vu une implémentation matérielle lors du dernier TP micromachine. Il utilisait un mécanisme appelé *branch and link*, dans lequel l'adresse de retour était stockée dans un registre du processeur. Dans le MSP430, l'adresse de retour est stockée sur une pile, ce qui permet par exemple à une routine de traitemement d'interruption d'être elle-même interrompue par une interruption plus prioritaire. Nous allons d'abord observer ce mécanisme de pile et les instructions associées.
Mais avant ça, on va prendre en main les boutons, sans s'occuper des interruptions.

1 Boutons et attente active

Exercice 30 Ressortez un programme qui fait clignoter une LED. Testez-le.

Exercice 31 Sur la carte (voir schéma dans le sujet de TP précédent) trouvez à quelles broches du msp430 sont connectés les deux boutons-poussoirs S1 et S2. Identifiez le port GPIO correspondant Px.

Exercice 32 Relisez l'encadré page 16 pour vous rafraîchir la mémoire sur l'utilisation des entrées-sorties numériques.

Exercice 33 Au début de votre programme, configurez le port Px pour qu'il agisse comme une entrée (registre PxDIR). Vous trouverez les adresses des différents registres utiles à la page 413 de la documentation msp430x4xx.

Exercice 34 Dans `mspdebug`, utilisez la commande `md` (au besoin, faites d'abord un `help md`) pour lire le registre PxIN :

- quelle est la valeur de PxIN lorsque les deux boutons sont relâchés ?
- quelle est la valeur de PxIN lorsque seul le premier bouton est pressé ?
- quelle est la valeur de PxIN lorsque seul le second bouton est pressé ?
- quelle est la valeur de PxIN lorsque les deux boutons sont pressés ?

Exercice 35 Dans la boucle principale de votre programme, remplacez le ralentissement par une attente sur les boutons. On veut que le bouton soit appuyé puis relâché, comme illustré ci-dessous :

```
.section .init9

main:
   /* initialiser la LED et les boutons */

boucle:

   /* attendre qu'un bouton soit appuye' */

   /* attendre que le bouton soit relache' */

   /* inverser la LED */

   j boucle
```

2 Les instructions call, ret, et un peu de pile

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 36 Lisez les extraits de documentation ci-dessous et page suivante . TOS veut dire *top of stack*.

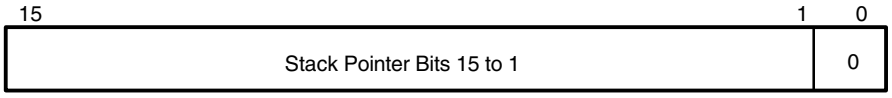
Extrait de la documentation : msp430x4xx.pdf page 45

3.2.2 Stack Pointer (SP)

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

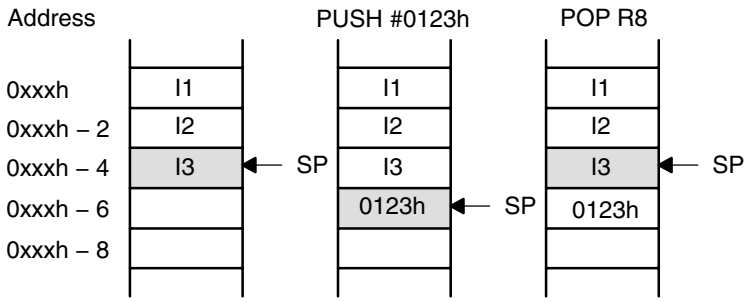
Figure 3–4 shows stack usage.

Figure 3–3. Stack Pointer




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

Figure 3–4. Stack Usage



Extrait de la documentation : msp430x4xx.pdf page 69

CALL	Subroutine		
Syntax	CALL	dst	
Operation	dst	–> tmp	dst is evaluated and stored
	SP – 2	–> SP	
	PC	–> @SP	PC updated to TOS
	tmp	–> PC	dst saved to PC
Description	A subroutine call is made to an address anywhere in the 64K address space. All addressing modes can be used. The return address (the address of the following instruction) is stored on the stack. The call instruction is a word instruction.		
Status Bits	Status bits are not affected.		

Extrait de la documentation : msp430x4xx.pdf page 96

*** RET**	Return from subroutine	
Syntax	RET	
Operation	@SP→ PC SP + 2 → SP	
Emulation	MOV	@SP+,PC
Description	The return address pushed onto the stack by a CALL instruction is moved to the program counter. The program continues at the code address following the subroutine call.	
Status Bits	Status bits are not affected.	

Remarque : Les instructions CALL et RET sont essentiellement des sauts. Les arguments de la fonction/procédure peuvent être passés soit dans les registres, soit en mémoire. Idem pour l'éventuelle valeur de retour. Si ce sont des programmeurs différents qui écrivent l'appelant et l'appelé, il faut qu'il y ait entre eux une règle du jeu commune. Cette règle commune, en général édictée par les concepteurs du système d'exploitation, est appelée la *convention d'appel* (ou "ABI" : pour "Application Binary Interface"). Elle sera étudiée plus précisément en compilation, car elle est aussi indispensable pour permettre la compilation séparée.

Exercice 37 Dans quelle direction grandit la pile du msp430 : vers les adresses croissantes, ou vers les adresses décroissantes ?

Exercice 38 Dans votre programme, à quelle valeur SP est-il initialisé ? Cherchez dans le désassemblage complet du programme l'instruction qui fait cette initialisation.

Exercice 39 Sur le plan mémoire du msp430 (cf sujet du TP précédent) repérez l'emplacement de la pile et sa direction de croissance.

Exercice 40 Ressortez votre boucle de ralentissement et faites-en une procédure `pause` qui attend environ une demi-seconde. On doit pouvoir utiliser cette procédure dans un programme de ce genre :

```
.section .init9
```

```
main:
   /* initialiser la led */

boucle:
   call #pause

   /* inverser la led */

   j boucle

pause:
   /* copiez ici le code que vous avez deja */
   ret
```

Exercice 41 L'intérêt d'une procédure est de l'appeler plus d'une fois... Modifiez votre programme pour qu'il chronomètre la durée d'appui sur le bouton : vous mesurerez ce temps en comptant le nombre de fois que vous pouvez appeler `pause` avant que le bouton soit relâché. Une fois que le bouton est relâché, faites clignoter la LED le même nombre de fois. Vous aurez ainsi deux appels `call #pause` dans ce programme : l'un pour la mesure du temps, et l'autre pour ralentir le clignotement.

Exercice 42 Mettez un breakpoint sur `pause` pour arrêter l'exécution à l'entrée de cette procédure. Observez la valeur du SP lors des deux appels. Observez la valeur pointée par SP lors des deux appels. Expliquez tout celà à un enseignant.

Exercice 43 L'intérêt de la pile apparaît lorsque une fonction appelle elle-même une autre fonction. Modifiez votre programmme pour qu'il utilise une procédure `blinkn` qui fait clignoter la led *n* fois, où *n* est la valeur de R15 à l'entrée de la routine. Votre programme `main` appellera `blinkn` qui appellera lui-même `pause`. Testez-le.

Exercice 44 Toujours à l'entrée de `pause`, retrouvez sur la pile pointée par SP les valeurs des adresses de retour de vos deux `call`.

3 Boutons et interruptions

Un inconvénient majeur de ce qu'on a fait jusqu'ici c'est que le processeur est **entièrement occupé** à attendre que l'on appuie sur un bouton. On parle d'**attente active**, ou **polling** en langue de Shakespeare. Pour remédier à ce problème, on peut configurer le processeur pour qu'il réagisse aux appuis boutons à travers son mécanisme d'interruption. Entre les appuis boutons, le processeur va ainsi être libre d'exécuter autre chose !

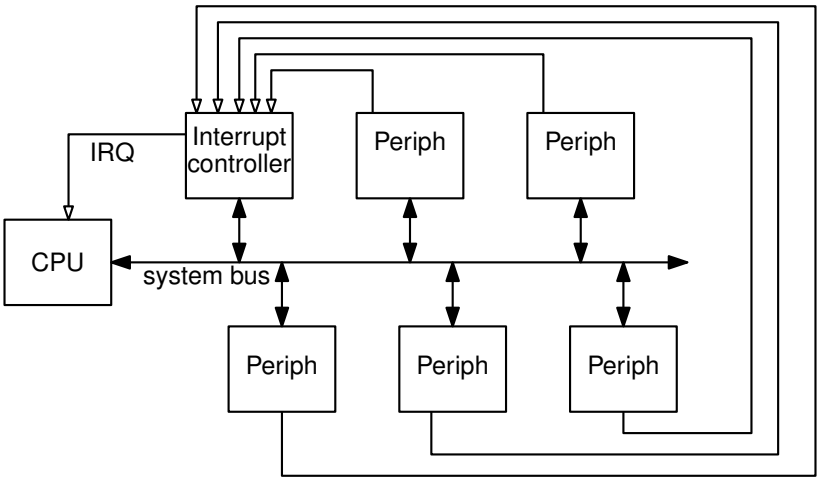
Dans un premier temps, lisez les deux encadrés ci-dessous, pour vous rafraîchir la mémoire sur les différentes notions mises en jeu et pour commencer à comprendre comment les interruptions sont implémentées matériellement dans le cas du msp430 et de notre carte.

À savoir : scrutation VS interruptions, requêtes (IRQ), vecteur, routine de traitement (ISR)

La communication entre un périphérique et le processeur se fait en général au travers des registres matériels. Un composant qui veut transmettre une information au programme place cette information dans un de ses registres, et attend que le processeur vienne lire cette valeur. C'est cette technique, appelée *scrutation* (en anglais *polling*) que vous avez utilisée précédemment pour connaître l'état des boutons. Un inconvénient majeur de cette approche est son incapacité à passer à l'échelle : pour ne pas rater un évènement, le programme doit continuellement aller scruter l'état du matériel, ce qui monopolise le processeur.

L'alternative consiste à mettre en place un mécanisme *d'interruptions* (cf poly page 73). Dans ce cas, le composant qui veut transmettre une information au programme place cette information dans l'un de ses registres, puis envoie au processeur une *requête d'interruption* (en anglais *interrupt request* ou IRQ). Selon les architectures, ces requêtes peuvent transiter sur le bus principal ou sur des fils dédiés appelés des *lignes d'interruptions*. Dans ce cas-là, soit le processeur dispose d'une entrée pour chaque source d'inter-

ruptions, soit comme illustré ci-dessous, les lignes d'interruptions sont concentrées par un périphérique dédié, appelé le *contrôleur d'interruption*.



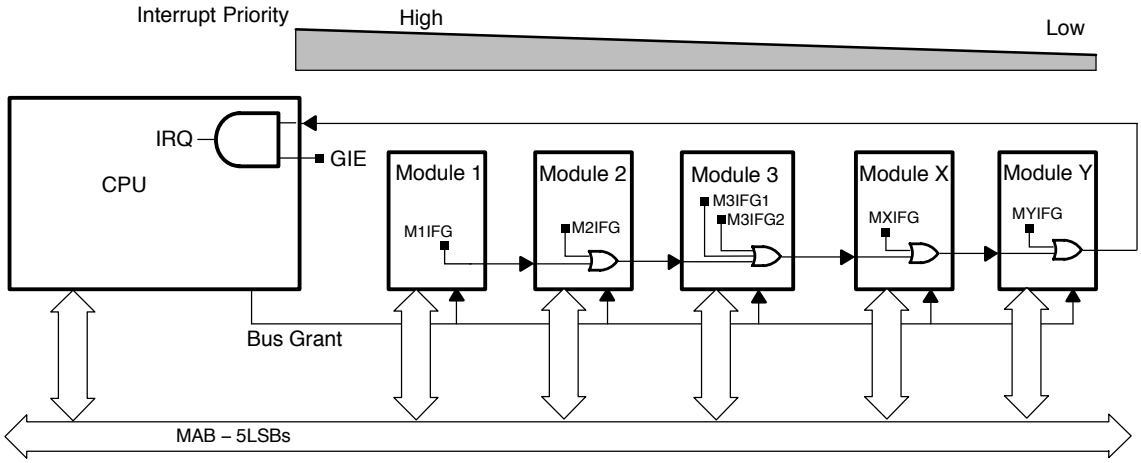
Du côté du processeur, la gestion des interruptions est intégrée au cycle de Von Neumann. Lorsqu'il reçoit une requête, le processeur interrompt automatiquement l'exécution du programme et saute vers une adresse bien connue, à laquelle il s'attend à trouver une *routine de traitement* spécifique (en anglais *interrupt service routine* ISR, ou *interrupt handler*). Chaque ligne d'interruption est ainsi associée à une routine distincte, ce qui permet au programmeur de prévoir un traitement différent pour chaque type d'évènement. La correspondance {requête $n_1 \rightarrow$ routine r_1, requête $n_2 \rightarrow$ routine r_2, etc.} est implémentée par une structure de données appelée la *table des vecteurs d'interruption* (*interrupt vector table* ou IVT). En général il s'agit d'un tableau de pointeurs de fonction, chaque case contenant l'adresse d'une ISR.

Une routine d'interruption est un morceau de code similaire à une fonction, sauf qu'elle est invoquée automatiquement par le processeur, et non pas par un appel explicite. En plus de traiter l'évènement proprement dit en allant lire et/ou écrire dans les registres du périphérique concerné, une ISR devra typiquement *accuser réception* de l'interruption auprès du périphérique et/ou du contrôleur d'interruption, pour qu'il cesse d'émettre la demande. Enfin, terminer une ISR revient à *restaurer le contexte* d'exécution, c'est à dire reprendre l'exécution du programme interrompu, qui ne se sera aperçu de rien.

Utile pour le TP : les interruptions sur le MSP430

Le fonctionnement des interruptions est détaillé au chapitre 2.2 de la documentation (msp430x4xx.pdf pages 29 et suivantes) et nous en reprenons les grandes lignes ici. Le CPU du MSP430 ne dispose pas d'une ligne d'interruption distincte pour chaque périphérique, mais d'une seule ligne partagée par tous les périphériques. Un composant (par exemple le Module 2) qui veut lever une interruption fait passer son *interrupt flag* à 1 (dans notre exemple, il s'agit donc du bit M2IFG). Certains modules ont plusieurs drapeaux, mais le fonctionnement reste similaire (les petits carrés noirs du schéma représentent des bits accessibles en mémoire dans un registre matériel). Le signal traverse les autres périphériques et atteint le processeur. Celui-ci perçoit donc une requête d'interruption (IRQ) lorsque :

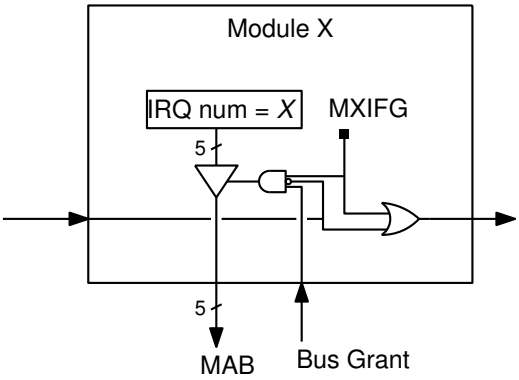
— au moins un des périphériques a un *interrupt flag* levé,
— et le bit GIE du registre SR est vrai (bit *Global Interrupt Enable* du *Status Register*)



Pour savoir de qui vient la requête, le processeur passe alors le signal *Bus Grant* à 1. Mais il se peut que plusieurs périphériques aient des flags levés, et dans ce cas-là on veut les départager par ordre de priorité. Chaque module, grâce à la circuiterie illustrée ci-dessous, émet donc son propre numéro si et seulement si :

— ce module a une interrution en attente (i.e. son *interrupt flag* est levé),
— et aucun module plus prioritaire n'a d'interruption en attente (cf fil de gauche sur le schéma),
— et le signal *Bus Grant* venant du CPU est actif (cf fil venant du bas).

Il y a donc bien un et un seul numéro d'IRQ envoyé au processeur (dans notre exemple, le nombre 2 codé sur cinq bits : 0b00010).



Lorsqu'il reçoit ce numéro, le CPU l'utilise comme indice dans la table des vecteurs, et charge le vecteur dans PC, ce qui revient à sauter à l'adresse de l'ISR.La table est située en mémoire flash à l'adresse 0xFFC0, donc le vecteur numéro *x* est le mot d'adresse 0xFFC0+2*x*.

Dans la suite, vous allez mettre en oeuvre ce mécanisme d'interruption dans votre programme :

— envoi d'une requête sur appui de bouton
— réception des IRQ côté processeur
— saut vers une ISR
— acquittement de l'interruption
— retour au programme principal

Remarque : tant que l'ensemble ne fonctionnera pas correctement (vers la question 49), vous ne pourrez pas tester par vous-même que vos réponses sont correctes. Du coup, soit vous comprenez ce que vous faites et vous avancez jusqu'à la question 49, soit vous avez du mal et alors demandez-nous de l'aide pour vous débloquer. Mais ne perdez pas trop de temps à patauger.

Exercice 45 Commencez par écrire une routine qui inverse l'état de la LED. Ce sera notre ISR. Utilisez temporairement une étiquette inutile, par exemple `temp_isr:` nous verrons plus loin par quoi la remplacer pour que cette routine soit effectivement invoquée par le processeur en réponse à une interruption.

4 Interruption sur bouton poussoir

La première chose à faire consiste à faire en sorte que lorsqu'on appuie sur un des boutons, une interruption soit émise vers le processeur.

Exercice 46 Commencez par lire l'encadré page suivante, qui parle de la gestion des interruptions sur les ports GPIO.

Exercice 47 Dans votre programme, rajoutez les instructions nécessaires pour activer les interruptions du port 1.

11.2.6 P1 and P2 Interrupts

Each pin in ports P1 and P2 have interrupt capability, configured with the PxIFG, PxIE, and PxIES registers. All P1 pins source a single interrupt vector, and all P2 pins source a different single interrupt vector. The PxIFG register can be tested to determine the source of a P1 or P2 interrupt.

Interrupt Flag Registers P1IFG, P2IFG

Each PxIFGx bit is the interrupt flag for its corresponding I/O pin and is set when the selected input signal edge occurs at the pin. All PxIFGx interrupt flags request an interrupt when their corresponding PxIE bit and the GIE bit are set. Each PxIFG flag must be reset with software. Software can also set each PxIFG flag, providing a way to generate a software-initiated interrupt.

Bit = 0: No interrupt is pending

Bit = 1: An interrupt is pending

Only transitions, not static levels, cause interrupts. If any PxIFGx flag becomes set during a Px interrupt service routine or is set after the `RETI` instruction of a Px interrupt service routine is executed, the set PxIFGx flag generates another interrupt. This ensures that each transition is acknowledged.

Note: PxIFG Flags When Changing PxOUT or PxDIR

Writing to P1OUT, P1DIR, P2OUT, or P2DIR can result in setting the corresponding P1IFG or P2IFG flags.

Note: Length of I/O Pin Interrupt Event

Any external interrupt event should be at least 1.5 times MCLK or longer, to ensure that it is accepted and the corresponding interrupt flag is set.

Interrupt Edge Select Registers P1IES, P2IES

Each PxIES bit selects the interrupt edge for the corresponding I/O pin.

Bit = 0: The PxIFGx flag is set with a low-to-high transition

Bit = 1: The PxIFGx flag is set with a high-to-low transition

Note: Writing to PxIESx

Writing to P1IES or P2IES can result in setting the corresponding interrupt flags.

PxIESx	PxINx	PxIFGx
$0 \rightarrow 1$	0	May be set
$0 \rightarrow 1$	1	Unchanged
$1 \rightarrow 0$	0	Unchanged
$1 \rightarrow 0$	1	May be set

Interrupt Enable P1IE, P2IE

Each PxIE bit enables the associated PxIFG interrupt flag.

Bit = 0: The interrupt is disabled

Bit = 1: The interrupt is enabled

Exercice 48 Par défaut, le processeur n'écoute pas les interruptions. Il faut donc activer les interruptions explicitement. Le processeur fournit une instruction pour ça, `eint`. Allez vérifier son comportement et sa syntaxe, page 80 de la doc. msp430x4xx.

Il reste maintenant à étiquetter convenablement votre traitant d'interruption pour que le compilateur l'associe au bon vecteur d'interruption.

Exercice 49 En vous aidant de l'encadré page précédente et de l'extrait ci-dessous, déterminez le numéro du vecteur d'interruption qui nous intéresse (colonne *priority* dans le tableau).
Dans votre code, remplacez l'étiquette `temp_isr` par les deux lignes suivantes, où xx est le bon numéro :

```
.global __isr_xx
__isr_xx:
```

Extrait de la documentation : msp430x4xx.pdf pages 411 et 412

The interrupt vectors and the power-up start address are in the address range 0FFFFh to 0FFC0h. The vector contains the 16-bit address of the appropriate interrupt-handler instruction sequence.

Table 6-3. Interrupt Sources, Flags, and Vectors

INTERRUPT SOURCE	INTERRUPT FLAG	SYSTEM INTERRUPT	WORD ADDRESS	PRIORITY
Power-Up External Reset Watchdog Flash Memory	WDTIFG KEYV (1) (2)	Reset	0FFFEh	31, highest
NMI Oscillator Fault Flash Memory Access Violation	NMIIFG (1) (3) OFIFG(1) (3) ACCVIFG(1) (4)(2)	(Non)maskable (Non)maskable (Non)maskable	0FFFCh	30
Timer_B7	TBCCR0 CCIFG0(4)	Maskable	0FFFAh	29
Timer_B7	TBCCR1 CCIFG1 to TBCCR6 CCIFG6, TBIFG(1)(4)	Maskable	0FFF8h	28
Comparator_A	CAIFG	Maskable	0FFF6h	27
Watchdog Timer+	WDTIFG	Maskable	0FFF4h	26
USCI_A0, USCI_B0 Receive	UCA0RXIFG, UCB0RXIFG(1)	Maskable	0FFF2h	25
USCI_A0, USCI_B0 Transmit	UCA0TXIFG, UCB0TXIFG (1)	Maskable	0FFF0h	24
ADC12	ADC12IFG (1) (4)	Maskable	0FFEEh	23
Timer_A3	TACCR0 CCIFG0(4)	Maskable	0FFECh	22
Timer_A3	TACCR1 CCIFG1 and TACCR2 CCIFG2, TAIFG(1) (4)	Maskable	0FFEAh	21
I/O Port P1 (Eight Flags)	P1IFG.0 to P1IFG.7(1) (4)	Maskable	0FFE8h	20
USART1 Receive	URXIFG1	Maskable	0FFE6h	19
USART1 Transmit	UTXIFG1	Maskable	0FFE4h	18
I/O Port P2 (Eight Flags)	P2IFG.0 to P2IFG.7 (1) (4)	Maskable	0FFE2h	17
Basic Timer 1, RTC	BTIFG	Maskable	0FFE0h	16
DMA	DMA0IFG, DMA1IFG, DMA2IFG(1) (4)	Maskable	0FFDEh	15
DAC12	DAC12.0IFG, DAC12.1IFG(1) (4)	Maskable	0FFDCh	14
Reserved	Reserved(5)		0FFDAh	13
			⋮	⋮
			0FFC0h	0, lowest

(1) Multiple source flags
(2) Access and key violations, KEYV and ACCVIFG, only applicable to FG devices.
(3) A reset is generated if the CPU tries to fetch instructions from within the module register memory address range (0h to 01FFh). (Non)maskable: the individual interrupt-enable bit can disable an interrupt event, but the general-interrupt enable cannot disable it.
(4) Interrupt flags are located in the module.
(5) The interrupt vectors at addresses 0FFDAh to 0FFC0h are not used in this device and can be used for regular program code if necessary.

Exercice 50 Il faut maintenant faire en sorte que votre traitant d'interruption acquite l'interruption. Pour ça, relisez la partie concernant `P1IFG` dans l'encadré de la page page précédente.

Exercice 51 Avez-vous pensé au retour du traitant d'interruption ? Sinon (ou si vous avez utilisé l'instruction `RET`) alors allez vite lire l'encadré ci-dessous et corrigez votre programme de façon adéquat.

Extrait de la documentation : msp430x4xx.pdf pages 97

RETI	Return from interrupt
Syntax	RETI
Operation	TOS → SR SP + 2 → SP TOS → PC SP + 2 → SP
Description	The status register is restored to the value at the beginning of the interrupt service routine by replacing the present SR contents with the TOS contents. The stack pointer (SP) is incremented by two. The program counter is restored to the value at the beginning of interrupt service. This is the consecutive step after the interrupted program flow. Restoration is performed by replacing the present PC contents with the TOS memory contents. The stack pointer (SP) is incremented.
Status Bits	N: restored from system stack Z: restored from system stack C: restored from system stack V: restored from system stack
Mode Bits	OSCOFF, CPUOFF, and GIE are restored from system stack.

Exercice 52 Dans la boucle principale de votre programme, faites de nouveau clignoter une (autre) diode (ou mieux : jouez de la musique sur le buzzer) et constatez avec satisfaction que les deux activités (programme principal et ISR) s'exécutent maintenant en bonne harmonie.