Systèmes numériques - Ernest'O'Clock

Paul Fournier, Juliette Schabanel, Samuel Vivien

Premier semestre 2020

Table des matières

1 Utilisation de l'horloge

Il a y 4 commandes dans le makefile qui sont utile pour l'utilisation de l'horloge :

- `make` qui va simplement build l'exécutable de l'horloge.
- `make run` qui va build l'exécutable si il ne l'est pas déjà, puis la lancer.
- `make speed` qui fait la même chose que la commande précédente pas avec le mode hyper-vitesse.
- `make clean` qui nettoie tout (mais nettoie moins que `rm -rf \`).

2 Compilation `.net` → `.c`

2.1 Motivations

De prime abord, nous avions implémenté un simulateur de netlist, codé en `OCaml`, mais nous avons décidé de changer et de produire un code `C` ayant le même comportement que la netlist d'entrée, et ce pour deux raisons.

1. Les performances de `C`, en particulier grâce à l'optimisation apportée par `gcc`.
2. Le côté bas niveau permet de gérer plus facilement les entrées/sorties de l'ordinateur/simulateur/émulateur.
3. Avons-nous mentionné `gcc` ?

2.2 Implémentation

Le compilateur est implémenté en `OCaml`, parce que la plupart du code en amont l'était déjà, ce qui permet d'éviter des interfaces inutiles et lourdes.

L'idée est assez simple finalement, en `C`, on alloue un énorme tableau, dont chaque case est un fil, un tableau par RAM, et un tableau par ROM. Les opérations sur les netlist se transforment alors par un isomorphisme presque immédiat en opération sur les tableaux.

2.2.1 Les fils

Nous donnons ici rapidement les idées derrière la traduction des opérations sur des fils simples.

Opérations sur les fils	Opérations sur les tableaux
Constante (`true` ou `false`)	Constante (0 ou 1)
`not a`	`!a` (nb : plus rapide qu'un `xor`)
`and`	`&`
`or`	`|`
`xor`	`^`
`a nand b`	`!(a & b)`
`mux s b a`	`s?a:b`
`reg`	Assignation

2.2.2 Les RAM

Chaque RAM est un tableau. La lecture est simplement la lecture des cases du tableau, et l'écriture, sans grande surprise, l'écriture dans les cases du tableau.

Les opérations sont effectuées bit à bit, il pourrait donc y avoir quelques optimisations à effectuer de ce côté là (en écrivant par blocs de puissances de deux par exemple).

Pour des questions d'utilisation mémoire du simulateur, et comme nous n'avons pas implémenté de tables de hachage en `C`, la taille de la RAM est bornée, par une quantité prédéfinie, mais modifiable par un argument optionnel au moment de la compilation.

Si une adresse non valide est spécifiée, le CPU lève un `SEGFAULT`.

2.2.3 Les ROM

Étant immuables, les ROM sont définies en dur dans le code `C` généré par le compilateur (par exemple, aux alentours des lignes ~ 150 du code produit pour notre CPU, on peut trouver la ROM comprenant le code à exécuter, telle les disquettes des anciens).

On peut donc y lire, comme dans une RAM, et si une adresse invalide est spécifiée (ex : fin du programme), l'exécution s'arrête proprement.

3 Microprocesseur

3.1 Le design

Le microprocesseur a été construit de façon très simple, ayant dans l'esprit que l'on essayait de faire quelque chose de simple avec peu d'instructions et beaucoup de registres à la `RISC-V`. Nous avions un code `minijazz` fonctionnel pour la multiplication, cependant nous avons décidé de ne pas l'ajouter à notre microprocesseur pour gagner en efficacité.

Le microprocesseur a donc été construit autour d'un n-adder. Qui nous permettait à l'aide d'une série de multiplexeurs de faire les opérations d'addition, soustraction, incrémentation, décrémentation. Nous avons rajouté à côté de cela les opérations booléennes et les décalages à droite et à gauche.

Nous avons aussi pris la décision d'opter pour 32 registres de 32 bits, ainsi que 4 flags (carry, overflow, isNeg et notZero).

De plus afin de nous simplifier la vie nous avons décider de séparer la RAM permettant de stocker des données, de la ROM qui avait pour but de contenir le code.

Nous avons eu alors un problème : comment charger des valeurs de 32 bits dans une instruction de 32 bits. Nous avons donc décidé de créer ce que nous avons appelé le Demi-registre. Un registre 16 bits qui contient la moitié de l'instruction du cycle précédent. Permettant ainsi de charger en deux cycle une grande valeur. Pour les petites valeurs, il nous suffisait d'indiquer de l'on devait compléter notre valeur de 16bits par des zéros.

3.2 Le code machine

Une instruction machine se décompose en 5 parties :

1. L'opération codé sur les 7 premiers bits. Il s'agit du parcours dans l'arbre de multiplexeur permettant de nous ramener à la bonne valeur. Nous n'utilisons pas tous les bits pour nos instructions ce qui nous permettrais d'en rajouter plus si besoin.
2. Sur le 8^{e}bit se situe l'ordre d'écrire ou non le résultat du calcul. Permettant ainsi de différencier les **and** et **sub** des instructions **test** et **cmp**.
3. La valeur d'entrée sur 16 bits. Il peut s'agir d'une valeur numérique, ou du numéro d'un registre. Cette valeur est sauvegardé dans le demi-registre pour le cycle suivant.
4. Le type de l'entrée est ensuite implémenté sur 2 bits pour indiquer si il s'agit d'un registre, de la valeur pointée par un registre ou simplement d'une valeur numérique (étendu avec des zéros ou la valeur dans le demi-registre).
5. Ensuite se situe la valeur de sortie sur 5 bits, car il s'agit forcément de l'indice d'un registre.
6. Et pour finir sur le 32^{e}bit, une indication sur si il s'agissait de la valeur du registre ou l'emplacement en mémoire pointé par le registre qui nous intéresse

4 Langage assembleur

4.1 Le langage

Nous avons créé notre langage assembleur en nous inspirant fortement du langage `x86-64`. Il comporte un jeu d'instructions plus restreint qui sont les suivantes :

- `add` $x\ y \to x := x + y$
- `sub` $x\ y \to x := x - y$
- `neg` $x \to x := -x$
- `and` $x\ y \to x := x \&\& y$
- `or` $x\ y \to x := x || y$
- `xor` $x\ y \to x := x^y$
- `not` $x \to x := \neg x$
- `lsl / lsr` $x\ n \to$ logical shift de n
- `asr` $x\ n \to$ arithmetical shift de n
- `incr` $x \to x := x + 1$
- `decr` $x \to x := x - 1$

- `mov[flag]` $x\ y \rightarrow x := y$ conditionné par le drapeau
- `jmp / j[flag]` *label* → saut (conditionnel) vers le label
- `cmp` $x\ y \rightarrow$ donne aux drapeaux les valeurs pour $x - y$
- `test` $x\ y \rightarrow$ donne aux drapeaux les valeurs pour $x \&\& y$

Les registres sont des registres 32 bits et sont au nombre de 32. Ils sont nommés de `r00` à `r31`, les trois derniers ayant pour alias respectivement `rsp, rbp` et `cpp`
Les différents drapeaux autorisés sont :

- `e` : nullité du résultat [ZF]
- `ne` : non nullité [¬ ZF]
- `s` : résultat négatif [SF]
- `ns` : résultat positif [¬ SF]
- `g` : > [¬(SF xor OF) & ¬ ZF]
- `ge` : ⩾ [¬(SF xor OF)]
- `l` : < [SF xor OF]
- `le` : ⩽ [(SF xor OF)|ZF]
- `a` : > non signé [¬ CF & ¬ ZF]
- `ae` : ⩾ non signé [¬ CF]
- `b` : < non signé[CF]
- `be` : ⩽ non signé [CF|ZF]

La syntaxe est calquée sur celle de x86-64 :

- % < *nom du registre* > pour désigner la valeur contenue dans un registre.
- (% < *nom du registre* >) pour désigner la valeur contenue en mémoire à l'adresse contenue dans le registre, un seul de ces accès mémoire est autorisé par instruction
- $ < *entier* > pour les constantes.
- < *label* >: pour poser un label et " < *label* > " pour y sauter, les caractère autorisés pour les noms de labels sont les lettres majuscule et minuscules de l'alphabet latin, '_' et les chiffres hors première caractère (i.e. `(['a'-'z' 'A'-'Z'] | '_') (['a'-'z' 'A'-'Z'] | '_' | ['0'-'9'])*`).
- # pour désigner le début d'un commentaire, tout le texte suivant un # sera ignoré jusqu'au premier retour à la ligne.

La grammaire est décrite simplement par les règles suivantes :

$$
\begin{array}{rcl}
< mem > & = & \%< register > |(\%< register >) \\
< param > & = & \%< register > |(\%< register >)|\$< entier > \\
< instr > & = & < operateur >< mem >< param >? \\
 & | & mov < flag >? < mem >< param > \\
 & | & jmp < flag >?" < label > " \\
 & | & < label >:
\end{array}
$$

4.2 Compilation

Nous avons codé un assembleur qui assemble notre langage assembleur vers notre langage machine en utilisant OCaml, et en particulier `ocamllex` pour le lexer et `menhir` pour le parser. Notre compilateur vérifie que la limite d'une accès mémoire par instruction n'est pas dépassée et la présence d'un registre en argument quand celle-ci est requise.

5 Interface homme machine

Nous allons ici détailler la façon dont nous avons implémenté l'interface entre le microprocesseur, qui, après calcul, donne une représentation du temps sous format sept segments.

Commençons par expliquer comment nous avons réalisé l'affichage en lui-même.

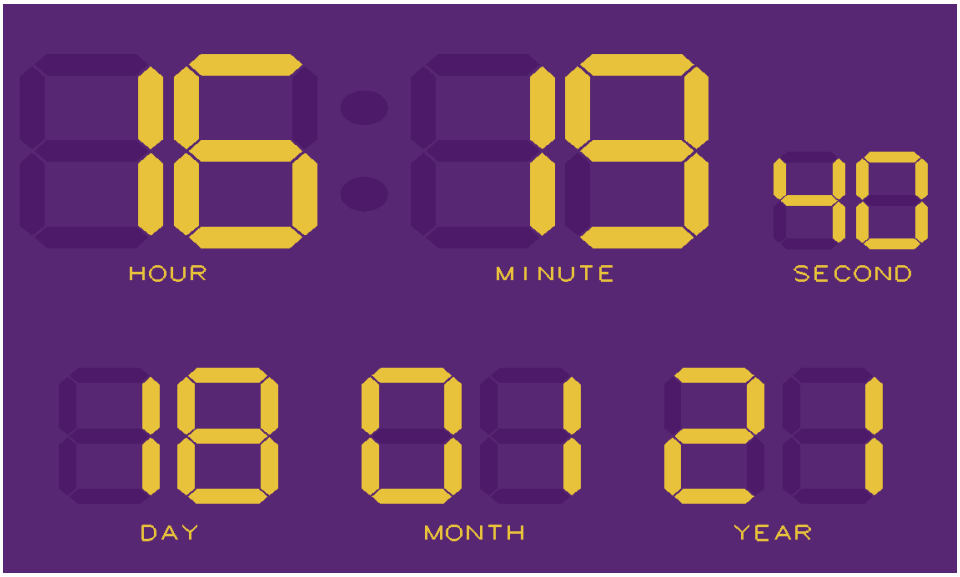


FIGURE 1 : Capture d'écran de la sortie de l'horloge

5.1 Programmation graphique de l'horloge

Ils a donc d'abord fallu décider d'une façon d'encoder en mémoire les segments. Pour cela, chaque chiffre est représenté par un mot de 7 bits (donc en pratique par un octet). La figure ci-dessous détaille quel bit correspond à quel segment (avec le mot indicé comme $b_0b_1b_2b_3b_4b_5b_6b_7$).

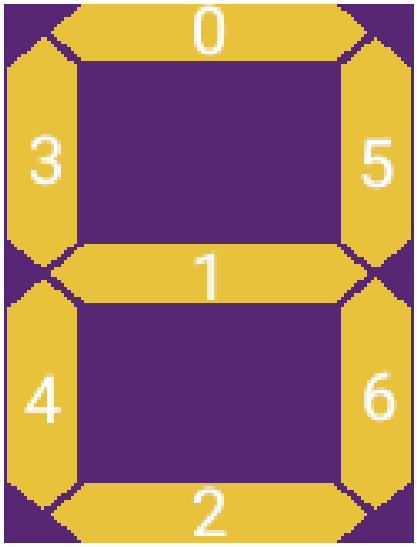


FIGURE 2 : Encodage des segments

L'interface s'est effectuée directement par une lecture dans la RAM. Une section "MMIO" a été définie et réservée pour cela. C'est concrètement du repiquage de fil de la mémoire vers un afficheur à cristaux liquide, mais virtualisé dans le code `C`.

Maintenant qu'on a récupéré le statut "on/off" de chaque segment, il est temps de l'afficher !

Nous avons choisi d'utiliser `OpenGL` comme libraire graphique. Ce choix a été motivé principalement parce que nous avions déjà (un peu) d'expérience avec cette librairie et que cela permettait de gagner en efficacité pendant la production. Par ailleurs, c'est une libraire répandue et connue, ce qui nous a évité de chercher pendant des heures comment faire pour afficher quelques bâtons sur un écran. Cependant, bien que nous avons voulu, en gardant le code assez simple, optimiser autant que faire se peut la vitesse d'affichage, `OpenGL` restant une libraire de rendu 3D à l'origine, il est évident que les opérations matricielles à répétition qu'engendrent le positionnement des différents éléments sur l'écran sont probablement sous-optimaux par rapport à une librairie spécialisée dans l'affichage rapide de forme géométrique simples en deux dimensions.

Comme il y a 3 formes à afficher (segment horizontal, segment vertical et point), nous avons décidé d'utiliser les "display list" pour stocker en mémoire les formes et les répliquer plus rapidement par la suite. Nous n'avons pas relevé expérimentalement de différence, mais c'est considéré comme une bonne pratique par l'ouvrage qui nous a servi de référence (OpenGL programming guide - The official guide to learning OpenGL). Ensuite, c'est juste du placement

minutieux de chaque segment. Les détails précis de la position ne seront pas explicités ici (long, moche, pénible, et inutile au propos), ils sont cela dit consultable depuis le code source, pour nos lecteurs masochistes. En fait, en interne, à chaque placement, `OpenGL` fait un produit matriciel 4×4, ce qui pourrait en pratique être simplifié en utilisant autre chose qu'un moteur 3D.

Finalement, à chaque frame, `OpenGL` affiche 7×12 segments, et deux points, de la couleur adaptée (allumé, éteint, selon le seul code couleur qui vaille la peine d'être implémenté) en plus du texte.

5.2 Entrées du microprocesseur

Afin d'interagir avec le monde, le microprocesseur doit aussi prendre des entrées. Pour le projet d'horloge, il s'agit d'une information d'initialisation - l'heure initiale -, et d' une information en temps réel - la parité de la seconde actuelle.

L'heure initiale est donnée sous la forme d'un entier 32 bits, encodé classiquement, contenant le nombre de secondes depuis le premier Janvier 1970 (UNIX epoch). Le microprocesseur fait toutes les conversions nécessaires en interne.

Ensuite, le bit de parité de la seconde en cours (qui sert aussi pour faire clignoter les deux points séparant les minutes des secondes), est calé entre deux paquets de sept bits dans le MMIO, bien au chaud.

5.3 Optimisations pour le mode *hyper-vitesse*

Lors des premiers tests, nous avons remarqué que l'affichage était le facteur limitant en termes de vitesse d'affichage, pas le CPU. C'est pour cela que nous avons voulu passer un peu de temps à réduire le temps passé à afficher des choses à l'écran.

Pour cela, nous avons ajouté un bit dans l'interface MMIO (toujours calé entre deux blocs de sept), qui est envoyé par le CPU lorsque les calculs sont terminés pour une unité de temps, qui dit concrètement au front-end "fin de la pause, il faut mettre à jour l'affichage de l'heure". Une fois que l'affichage est réalisé, ce bit est remis à zéro.

5.4 Lecture des inputs de l'utilisateur

Les entrées clavier de l'utilisateur sont lues lors de l'exécution de l'horloge et que le processus est en avant-plan. Actuellement, tout ce qui est vérifié est l'appui des touches `Q` et `MAJ` en simultané, ce qui ferme le programme.

5.5 Autres idées non implémentées

Nous avions d'autres projets pour cette partie d'interface. En particulier, nous voulions enrichir l'entrée utilisateur. Des tests en local (`hexdump`, entre autres) nous ont permis de conclure qu'il était, effectivement, possible de faire ce qui est ni plus ni moins qu'un keylogger en lisant directement dans `/dev/input`. Nous avons testé, et il était possible de récupérer les informations de :

- Trackpad
- Clics
- Clavier
- Manette de Nintendo Switch™ type Game Cube (connectique USB)

Le code utilisé pour tester peut être trouvé en annexe. Il aurait alors été possible, avec un peu de traitement, certes, de donner des informations plus complexes au CPU.

Nous aurions aussi voulu modifier le système d'affichage pour pouvoir avoir du contrôle directement sur les pixels mêmes de l'écran. Là aussi, nous aurions voulu aller fouiner du côté du noyau Linux, mais c'est une magie encore trop ésotérique pour nous. Nous aurions aussi pu chercher une librairie qui gère de l'affichage 2D, mais le temps nous a manqué.

L'objectif final aurait été de pouvoir avoir une machine qui a des performances comparables à celles des premières consoles de jeu, et montrer ses performances en implémentant un petit jeu type Snake ou Pong.

De manière orthogonale, nous aurions aussi aimé, quitte à utiliser `OpenGL`, pouvoir faire profiter l'utilisateur de l'horloge d'un environnement 3D composé essentiellement d'une table de chevet sur laquelle repose un radio-réveil (notre horloge).

6 Horloge

6.1 Initialisation

Une partie relativement technique de l'implémentation de l'horloge résidait dans son initialisation. À partir du nombre de secondes depuis le début de l'univers (a.k.a. le premier janvier 1970), il fallait tout décomposer pour avoir l'heure en :

- unité des secondes,
- dizaine des secondes,
- unité des minutes,
- dizaines des minutes,
- et cetera, jusqu'au siècle.

Pour cela le plus simple est de faire une division euclidienne. Une instruction que nous n'avions pas implémenté dans le microprocesseur. Il a donc fallut l'implémenter en assembleur à l'aide de comparaison et décalages logiques.

Si vous vous souvenez de comment vous avez appris à faire des division euclidienne en primaire. Alors vous avez juste à convertir le même algorithme mais qui fonctionne en base 2 pour obtenir notre division euclidienne.

Voici donc le code assembleur de notre division euclidienne qui fait la division de la valeur dans `%r28` par 86400. Avec le reste dans `%r28` et le quotient `%r27`

```
        mov %r26 $86400
        mov %r25 %r26
        jmp "ew1"
bw1:
        lsl %r25 $1
ew1:
        cmp %r28 %r25
        jb "bw1"
        mov %r27 $0
bw2:
        lsl %r27 $1
        cmp %r28 %r25
        ja "ew2"
        add %r27 $1
        sub %r28 %r25
ew2:
        lsr %r25 $1
        cmp %r25 %r26
        jge "bw2"
```

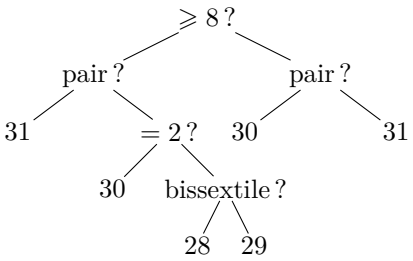
6.2 Incrémentation

Le tic d'horloge est donné sur le bit d'indice 31 de la RAM sous la forme d'un changement de valeur à chaque seconde. Notre programme récupère cette valeur à chaque boucle et la compare à la dernière valeur lue, pour savoir si il doit incrémenter le temps.

Pour faciliter le traitement, chaque chiffre est stocké dans un registre différent. On utilise en plus un registre stockant le numéro du mois en entier et de même pour l'année, ainsi qu'un dernier indiquant si l'année est bissextile ou non.

On commence par incrémenter le chiffre des unités des secondes, le compare à 10, si il est strictement inférieur on saute directement à l'affichage et sinon on incrémente les dizaines, et ainsi de suite jusqu'à la fin de la propagation.

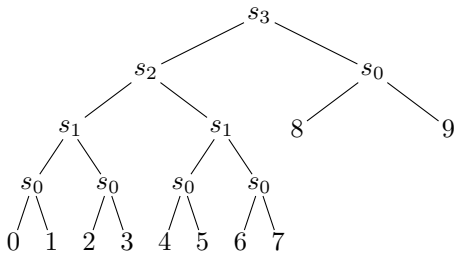
Pour savoir si l'on change de mois on non après avoir incrémenté le jour, nous avons cherché à déterminer le nombre de jours du mois en un minimum de test, ce qui nous à conduit à l'arbre de test suivant où pour chaque nud interne, le fils gauche correspond à un `false` et le droit à un `true` :



6.3 Output

Pour terminer la boucle, on convertit chaque chiffre en une séquence de 7 bits indiquant quels segments doivent être allumés. Pour économiser quelques cycles, nous avons cherché à minimiser le nombre de tests, ce qui nous a conduit à

l'arbre de test suivant, s_i représentant le i-ème bit :



Le résultat est ensuite placé sur la RAM. Pour optimiser l'espace, les chiffres sont regroupés par quatre et le résultat est écrit sur les trois premières cases de la RAM. On termine en donnant au bit 30 la valeur 1 pour signaler à l'affichage qu'il faut effectuer un rafraîchissement.

Listing 1 : Keylogger test

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <linux/input.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <libevdev/libevdev.h>
#include <libevdev/libevdev-uinput.h>

int main(){

        struct libevdev *dev = NULL;
        int fd;
        int rc = 1;

        // keyboard : /dev/input/event3
        //fd : file descriptor
        fd = open("/dev/input/event3", O_RDONLY|O_NONBLOCK);
        // rc : return code
        // modifies *dev such that it is hooked to the /dev/input file
        rc = libevdev_new_from_fd(fd, &dev);
        if (rc < 0) {
                //error happenned
                fprintf(stderr, "Failed to init libevdev (%s)\n", strerror(-rc));
                exit(1);
        }
        printf("Input device name: \"%s\"\n", libevdev_get_name(dev));
        printf("Input device ID: bus %#x vendor %#x product %#x\n",
        libevdev_get_id_bustype(dev),
        libevdev_get_id_vendor(dev),
        libevdev_get_id_product(dev));

        do {
                struct input_event ev;
                rc = libevdev_next_event(dev, LIBEVDEV_READ_FLAG_NORMAL, &ev);
                if (rc == 0)
                printf("Event: %s %s %d\n",
                libevdev_event_type_get_name(ev.type),
                libevdev_event_code_get_name(ev.type, ev.code),
                ev.value);
        } while (rc == 1 || rc == 0 || rc == -EAGAIN);
        0;
}
```

Codification du langage machine, les étoiles signifient que l'on peut mettre librement un 0 ou un 1 sans que cela change l'instruction.

1. Le code sur 7 bits
   - ******1 si calcul
     - *****11 -> arithmétique
       - ****011 -> Additions
         - 00**011 -> add
         - 01**011 -> sub
         - 10**011 -> incr
         - 11**011 -> decr
       - ****111 -> Décalages
         - 00**111 -> lsl
         - 01**111 -> lsr
         - 11**111 -> asr
     - *****01 -> logique
       - ****001 -> calcul booléen
         - 00**001 -> and
         - 01**001 -> or
         - 10**001 -> nand
         - 11**001 -> xor
       - ****101 -> not et minus
         - ***0101 -> not
         - ***1101 -> minus
   - ******0 si mov
     - 0000**0 -> move
     - 1000**0 -> movne
     - 0100**0 -> movs
     - 1100**0 -> movns
     - 0010**0 -> movge
     - 1010**0 -> movg
     - 0110**0 -> movle
     - 1110**0 -> movl
     - 0001**0 -> movae
     - 1001**0 -> mova
     - 0101**0 -> movbe
     - 1101**0 -> movb
     - **11**0 -> mov
2. write_enable

   1 pour écire, 0 sinon
3. Entrée sur 16 bits

   si c'est un entier mettre en binaire avec les bits de poids fort à gauche.
4. Type de l'entrée (imm, reg, mem) :
   - 00 -> valeur binaire avec des 0 au début : 000... param2
   - 01 -> valeur binaire avec le demi-registre au début : demiR.param2
   - 10 -> valeur registre
   - 11 -> valeur pointé par le registre

   Le demi-registre contient la valeur du param2 du cycle précédent.
5. Sortie sur 5 bits

   Valeur en base 2 de l'indice du registre lu
6. Type de la sortie (reg, mem)

   1 si dans la mémoire, 0 sinon