

# Système Digital

## Rapport de projet

Elie Michel, Louis Garrigue, Nicolas Jeannerod et Aurélien Delobelle

27 janvier 2013

## 1 Vue d'ensemble

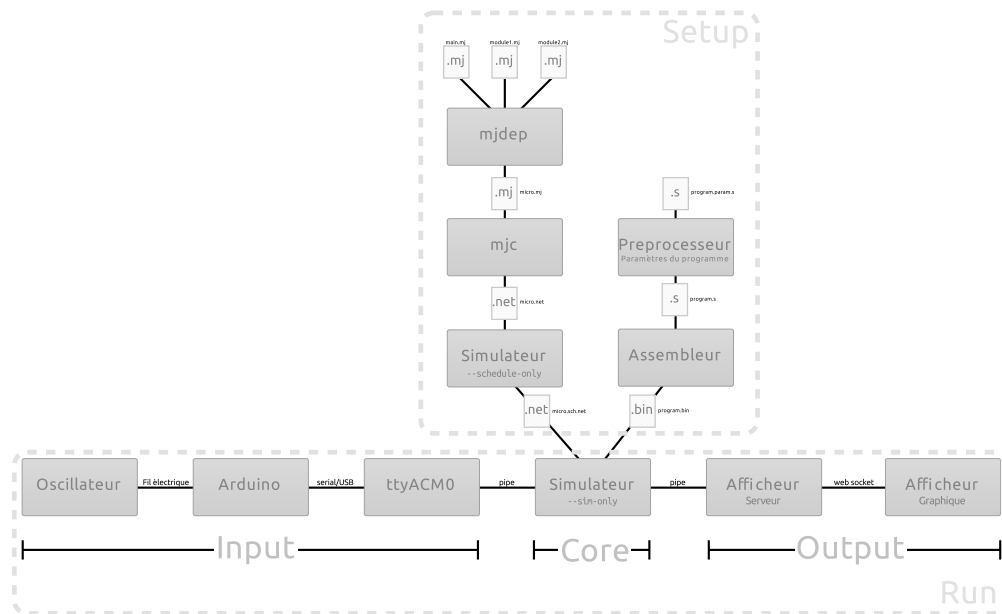


FIGURE 1 – Organisation du projet

Le but principal de ce projet est la conception d'un microprocesseur, mais c'est autour du simulateur que le projet s'organise, comme le montre la figure 1.

Ce simulateur reçoit des informations de trois types différents :

- Le circuit à simuler
- L'état de la ROM (le programme, dans le cas de notre microprocesseur)
- Les entrées du circuit

Il retourne ensuite les sorties du circuit. Il est intéressant de faire la distinction entre les entrées/sorties du circuit, qui se font en temps réel lors de la simulation et le chargement du circuit et de la ROM qui se font à l'initialisation, depuis des fichiers préalablement générés. Ces deux phases sont appelées respectivement **run** et **setup** et possèdent chacune un script éponyme à la racine de l'archive.

Le microprocesseur est construit en utilisant minijazz et un préprocesseur permettant l'agrégation de plusieurs fichiers. Le programme est assemblé par le programme jupiter. L'entrée du circuit est reliée par l'intermédiaire du port série à un oscillateur carré réel et la sortie peut être reliée à deux afficheurs, l'un graphique et l'autre non.

## 2 Simulateur

Les sources et dépendances du simulateur se trouvent dans le répertoire `sim/`

### 2.1 Utilisation

#### 2.1.1 Installation

La compilation du simulateur nécessite `make`, `ocamlpt`, `ocamldep`, `ocamllex` et `menhir`. Le plus simple est d'utiliser Linux, bien que nous ayons également testé sous Windows *via* `cygwin`. Il est probable que le simulateur fonctionne sous MacOS bien que nous n'ayons pas eu à disposition de telle machine.

`make sim` compile uniquement le simulateur alors que `make` (ou `make all`) lance également un exemple d'utilisation. Le programme `sim` ainsi créé est autonome.

#### 2.1.2 Fonctionnement général

Sim prend en entrée les fichiers suivants :

- Un fichier netlist contenant la description du circuit (ordonnée ou non)
- Un fichier décrivant la ROM (facultatif)

Le fichier ROM est facultatif. S'il n'existe pas, un avertissement est affiché dans la console mais la simulation est tout de même effectuée. La ROM est alors identiquement nulle.

#### 2.1.3 Syntaxe des différents fichiers

La syntaxe de la netlist peut se trouver ici : (en seconde partie)

<http://www.di.ens.fr/~bourke/minijazz.pdf>

La syntaxe du fichier de description de la ROM est la suivante :

- Les valeurs sont représentées par des caractères : 1 représente un bit activé tandis qu'un 0 représente un bit désactivé.
- Les espaces et tabulations sont ignorées.
- Les retours à la ligne annoncent une nouvelle adresse mémoire.
- Des commentaires peuvent être ajoutés en fin de ligne après le caractère #.
- Une ligne commençant directement par le caractère # n'est pas considérée comme correspondant à une adresse.
- Les adresses sont écrites par ordre croissant en partant de 0. Si le fichier est plus long que la taille de la rom, le comportement du simulateur n'est pas défini.
- La première valeur entrée détermine la taille des mots de la mémoire. Les lignes suivantes sont automatiquement tronquées ou complétées par des 0. Veillez donc à ce que la première ligne ne soit pas vide!

#### 2.1.4 Syntaxe de la commande sim

Le seul paramètre obligatoire est le fichier netlist à simuler. Sim s'appelle alors comme suit :

```
sim [filename].net
```

Par défaut, le Sim utilise le fichier `rom.sim` pour la rom. Pour modifier le fichier de description de la rom, utiliser l'option `-rom [filename].sim`.

L'opération d'ordonnement de la netlist peut s'avérer très coûteuse en temps, c'est pourquoi on peut choisir de sauvegarder la netlist ordonnée à l'aide de l'option `-schedule-only`. Dans ce cas, l'étape de simulation n'est pas effectuée et la netlist ordonnée est enregistrée dans `[filename].sch.net`.

Lorsque l'on veut utiliser un fichier netlist déjà ordonné, il faut utiliser l'option `-sim-only`. À ce moment, si la netlist n'a pas préalablement été ordonnée, il peut y avoir des erreurs diverses. Le plus simple est de n'utiliser cette option que lorsque le fichier termine par `.sch.net` afin d'éviter les confusions.

Sim lit les valeurs des entrées du circuit sur l'entrée standard et écrit celles des sorties sur la sortie standard (sans retours à la lignes ni espaces dans les deux cas). L'entrée est lue bloc par bloc sans perdre une seule valeur, et donc si elle est générée plus vite qu'elle n'est traitée, un délais s'établit entre la génération et son envoi à la netlist simulée. Si ce comportement n'est pas souhaité, le paramètre `-async` peut être spécifié au lancement de Sim afin de vider l'entrée après l'avoir lue à chaque itération de la simulation. Ainsi la valeur lue en entrée a toujours été écrite pendant la dernière itération.

## 2.2 Fonctionnement interne

Le simulateur peut se décomposer en plusieurs phases :

- Le parseur de netlist
- L'optimizer, qui ordonne et simplifie les netlists
- Le parseur de fichier de ROM
- Le lecture/écriture des entrées/sorties du circuit
- L'initialisation, convertissant l'AST en structure plus adaptée à la simulation
- Le cœur du simulateur

### 2.2.1 Parseur de netlist

Nous avons globalement repris celui donné en TP au début de l'année. Nous avons cependant modifié les types `ty` et `value` pour abandonner les tableaux de booléens, trop peu efficaces et gourmands en mémoire, au profit d'une représentation basée sur la représentation binaire d'entiers. Le type est un entier spécifiant la taille de la nappe, ou 0 pour un bit simple, et la valeur est un entier. Le seul problème est qu'on est alors limité dans les nappes par la taille des entiers caml (31 bits sur une architecture 32 bits) mais ce problème est résolu par la phase d'optimisation (sauf pour les accès mémoire).

### 2.2.2 Optimizer

Les netlist générées depuis minijazz étant relativement longues, nous avons cherché à les optimiser. Nous nous sommes alors posé la question de l'utilité des nappes : leur implémentation n'apporte aucune fonction spécifique (sauf RAM et ROM), et n'accélère donc en rien la simulation. De plus, de part l'utilisation des circuits définis récursivement, le nombre de fonctions SELECT, SLICE et CONCAT était vraiment élevé dans la netlist. Dès lors, il y avait deux choix possibles :

- conserver l'idée de nappe, et adapter les fonctions de base, AND OR NOT MUX à leur utilisation
- supprimer les nappes (si cela pouvait apporter de la rapidité)

La première solution n'ayant pas vraiment de sens physique, nous avons opté pour la deuxième : supprimer la notion de nappe. En fait, nous n'avons pas tout à fait fait ça, dans la mesure où nous ne voulions pas supprimer les primitives ROM et RAM, qui utilisent fondamentalement les nappes. Presque toutes les nappes ont été supprimées, sauf localement autour des primitives RAM et ROM, où elles sont reconstruites.

C'est le travail du module Optimizer du simulateur de faire cela. Ce module améliore le travail de Scheduler :

- il prend une valeur de type `Netlist_ast.program` en entrée
- il ressort une valeur de type `Netlist_ast.program`, avec les équations triées dans un ordre topologique au niveau des dépendances.

Il fait cependant mieux que ça, et ce pour plusieurs raisons. En premier lieu, la complexité du travail de Optimizer est en  $\mathcal{O}(n)$  ou  $\mathcal{O}(n \log n)$ , quand celle du Scheduler était de  $\mathcal{O}(n^2)$ . Cela est rendu possible par le fait que l'Optimizer stocke dans son graphe (qui représente la netlist) plus d'informations (notamment l'équation associée au noeud), ce qui permet donc de retrouver l'équation associée à un noeud en temps constant, au lieu de devoir chercher dans une liste (en temps linéaire donc). Sur une netlist de 25000 portes, cela multiplie l'efficacité par (au moins) 1000, ce qui se ressent quand on passe de 50s de préparation à la simulation à moins d'une seconde.

De plus, lors de la construction du graphe représentant la netlist, l'Optimizer supprime les nappes. Une variable `a :3` se verra remplacer par `an0, an1, an2` (le double tiret bas n'étant pas toléré?). L'ordre des variables étant important au niveau des inputs/outputs, ce détail est bien sûr contrôlé. Cela permet alors la suppression de la plupart des fonctions SELECT, SLICE et CONCAT, qui deviennent simplement des égalités bit-à-bit. À ce stade, le graphe comporte donc beaucoup plus de noeuds que la netlist d'équations. Lors du retour à la netlist, on supprime les branches inutiles de la netlist, à savoir

toutes les équations qui ne servent pas au calcul des outputs. La netlist est aussi triée par ordre topologique à ce moment. Elle comporte beaucoup plus d'équations que l'original.

En dernier lieu, on supprime toutes les équations de la forme  $y=x$  (sauf celles qui concernent une output) et on remplace la variable  $y$  par la variable  $x$  dans la netlist. Cette phase supprime énormément de portes, ce qui permet à la fin d'obtenir une netlist bien plus petite que l'original. Par exemple :

- mjc.byte donne une netlist de 22143 portes
- après la suppression des nappes, on passe à 48384 portes
- après la suppression des doublons, 8132 portes.

### 2.2.3 Parseur de la ROM

La lecture des fichiers de description de la ROM est suffisamment simple pour tenir dans un lexer (pas besoin de menhir ici). Ce parseur servait à l'origine à charger le fichier décrivant les entrées au cours du temps.

## 2.3 Entrées/sorties du circuit

Les entrées étaient au début écrites à l'avance dans un fichier lu au démarrage du simulateur, mais cette méthode atteint ses limites lorsque les informations envoyées à l'entrées devaient d'une part être infinies et d'autre part ne pouvait pas être prévues à l'avance. Nous avons alors choisi de dialoguer sur l'entrée/sortie standard (bien que ce soit en fait là aussi un fichier), en vue de la communication avec les autres parties du projet.

Les entrées étaient initialement séparées d'un retour à la ligne entre chaque série et typées (les nappes mises entre crochet) pour vérifier qu'elles correspondent à l'entrée attendu. Pour des raisons de performances, nous nous sommes contenté de lire les entrées successives bout à bout, sans aucune séparation. Dans la pratique, il n'y a en entrée qu'un seul bit et ce choix divise donc par deux la taille des fichiers échangés. Le même allègement est appliqué aux sorties.

### 2.3.1 Initialisation

Nous avons eu énormément de problèmes d'efficacité avec le simulateur et n'avons de ce fait pas conservé la structure telle qu'elle est chargée pour la simulation. Plutôt que d'indiquer les différentes équations par leur nom dans le fichier de netlist, elles sont numérotées et un tableau les contenant est initialisé. Nous avons également tenté d'encoder l'arbre de syntaxe abstraite dans un autre tableau d'entier afin de se rapprocher au maximum du bas niveau mais les gains de performance n'étaient finalement pas intéressants.

### 2.3.2 Simulation

Le simulateur se doit d'être le plus rapide possible. Nous avons commencé avec une fréquence d'à peu près 15Hz, ce qui ne permettait vraiment pas de simuler correctement une horloge, surtout avec notre jeu d'instruction extrêmement bas niveau.

Les raisons de cette lenteur n'ont pas été simples à déterminer. Une première amélioration fut d'abandonner l'utilisation de maps au profit de tables de hachage (passage à 20-25Hz) mais la simulation restait étrangement lente. Ne trouvant pas d'où venait le problème, nous avons, avec l'aide de Tim, fait le profiling d'une simulation. Le profiling fourni par ocaml ne donne que le nombre d'appels à chaque fonction et nous n'avons rien repéré d'anormal. Nous avons cependant constaté que plus de la moitié des opérations étaient des concat/select/slice et c'est ce qui a motivé notre idée de retirer les nappes.

Un profiling plus poussé à l'aide de `gprof` et du module `Gc` a montré que la représentation des nappes sous forme de tableaux de booléens mobilisait un peu trop la mémoire. Nous avons donc décidé de les représenter par des entiers malgré la perte de généralité engendrée (limite de taille des nappes). Un autre point que le profiling a montré est l'utilisation un très grand nombre de fois du hashage, c'est pourquoi nous avons numéroté les équations au lieu de les représenter par une chaîne.

Les performances se sont améliorées jusqu'à avoisiner les 200Hz mais cette valeur nous semblait encore très faible au vu des capacités d'un ordinateur. Ne pouvant optimiser plus le simulateur sans tout recommencer, nous avons alors entrepris d'optimiser la netlist. Nous avons alors pu dépasser les 400Hz dans certains cas.

Une dernière optimisation intéressante fut de couper le code du microprocesseur en deux et de simuler les deux parties séparément. Nous n'avons rien eu à modifier sur le simulateur et il a ainsi pu profiter des architectures multi cœurs.

## 3 Microprocesseur

### 3.1 Introduction

Nous avons commencé par nous baser sur le fonctionnement des microprocesseurs MIPS étant donné que c'était la seule architecture que nous connaissions. Nous avons cependant tenu à faire quelque chose de différent.

#### 3.1.1 Ligne de conduite

Notre premier choix important a été de décider que chaque fonction du microprocesseur prendrait ses arguments dans les mêmes registres prévus à cet effet (à savoir les \$a\*), et renverrait ses résultats dans les registres prévus à cet effet (les \$r\*), à l'instar des appels système de MIPS qui regardent toujours \$v0 et \$a0.

#### 3.1.2 Densité

Ce choix nous a dirigé vers un objectif de densité. Nous avons alors constaté qu'il était possible de ne conserver que 4 registres<sup>1</sup>, réduisant par là énormément le nombre d'instructions de gestion de la mémoire puisque nous n'avons donc plus besoin que de 2 bits pour choisir un registre.

Nous avons au début cherché la densité maximale, avec un très petit jeu d'instructions (sur 5 bits seulement!). Mais pour des raisons pratiques (pour que certaines manipulations ne prennent pas trop d'instructions) nous l'avons enrichi. Finalement, les instructions sont codées sur 6 bits.

## 3.2 Architecture générale

### 3.2.1 Taille des valeurs

Le choix de la taille en bit des valeurs manipulées par le microprocesseur est un choix important et l'un des premiers que l'on ait fait. Il nous fallait trouver un compromis entre petite taille et maniabilité.

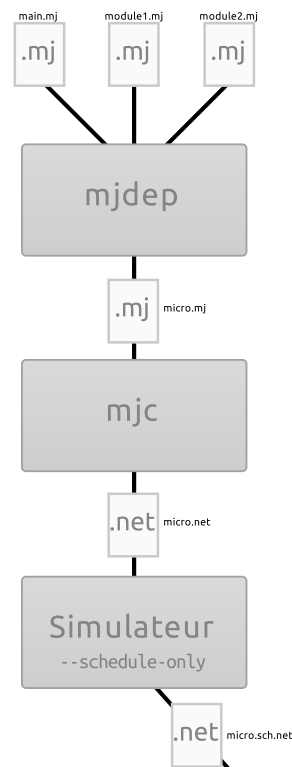
Les valeurs à afficher dans l'horloge sont toutes, à l'exception de l'année, de taille inférieure à 256 et le choix de mots de 8 bits nous ont semblés être un choix raisonnable. Pour la gestion des années, nous avons choisi de découper en deux bytes selon son écriture décimale et non binaire afin de simplifier la conversion en 7 segments.

Ce choix a par la même occasion déterminé le nombre de cases mémoire puisque nous utilisons un registre pour choisir l'adresse. Une RAM de 256 adresses est largement suffisante pour programmer une horloge. Nous sommes cependant revenu sur ce choix par la suite. (cf 3.3)

**Dans la ROM** Il n'était par contre pas possible de se limiter à des programmes de 256 instructions seulement, d'autant plus que notre jeu d'instructions extrêmement bas niveau nous oblige à produire de longs programmes. Le curseur d'instructions est donc représenté sur 16 bits, permettant des programmes de 65 536 instructions. Cette disparité entre la taille des mots machine et celle du curseur d'instructions nous a contraint à utiliser des sauts de taille relative plutôt qu'absolue. Il est cependant toujours possible de spécifier des tailles absolues dans des cas où c'est nécessaire en utilisant deux registres.

---

1. \$a0, \$a1, \$r0, \$r1



### 3.2.2 Séparation en unités

- Nous avons séparé les instructions en 4 unités en fonction de leurs deux premiers bits en langage machine :
- SYS qui s'occupe des entrées et sorties de la machine (depuis l'horloge, et vers l'affichage).
  - ALU qui fait les opérations de base, arithmétiques et logiques (add, sub, mult, div<sup>2</sup>; and, or, not, shift).
  - MEM qui gère les accès mémoire, que ce soit entre registres ou entre un registre et la RAM.
  - JUMP qui contrôle l'adresse de lecture des instructions.

**Inconvénients** La séparation en quatre unités impose un nombre fixe d'instructions possibles par unité, ce qui a amené quelques choix discutables :

- Par manque de place, le move d'un  $\$a[i]$  vers un  $\$a[j]$  (resp.  $\$r[i]$  vers  $\$r[j]$ ) n'existe pas. On doit pour cela utiliser une étape intermédiaire.
- Nous avons absolument besoin du LI (load immediate), et ne trouvons pas de place pour cette instruction. Finalement, elle appartient à l'ALU.
- Pour certaines instructions, on ne peut regarder que dans  $\$a0$  (et  $\$a1$  si deux arguments sont exigés), alors que pour d'autres on peut choisir le(s) registre(s) d'argument(s). Cela est dû au fait que certaines unités manquaient de place par rapport à d'autres.
- L'unité JUMP a accueilli les commandes WCA (Write Current Address) et END (termine le programme)

### 3.2.3 Notes sur le load immediat

Le LI est une instruction qui nous a posé problème car elle est contradictoire avec la petite taille de nos instructions. En effet, l'instruction doit contenir la valeur que l'on veut charger, ce qui n'est pas possible avec des instructions plus petites que les mots machine.

Une première idée fut d'interpréter l'instruction suivante non pas comme une instruction mais comme une valeur. En utilisant les 2 derniers bits de l'instruction LI puis les 6 bits de l'instruction suivante, on pouvait charger toute valeur.

Le problème est que cela revenait en fait à faire des instructions de taille variable et complexifiait le processeur. Gérer des tailles variables pour une seule instruction manquait de cohérence et nous avons préféré faire un chargement partiel, ne gérant pas toutes les valeurs possibles.

Il s'est ensuite posé la question du choix valeurs à gérer. La réponse la plus immédiate est de charger les bits de poids faible seulement. Une autre solution aurait pu être de choisir des valeurs que l'on est plus susceptible d'utiliser pour l'horloge, comme 24 ou 60, mais nous préférons conserver un minimum de généralité. Il était également possible de choisir les nombres premiers pour simplifier la décomposition en sous-étapes du chargement de valeurs plus importantes, mais nous n'avions la place dans le jeu d'instruction de ne choisir que 8 valeurs et en ajoutant 0 et 1 on ne pouvait aller que jusqu'à 11 (ou 13 sans 0), ce qui ne permet pas de décomposer tout nombre représenté sur 8 bits. Finalement, nous avons donc conservé la première solution.

### 3.2.4 Notes sur le shift

Une autre instruction à laquelle notre choix de densité a donné un comportement particulier est le shift. On ne choisit pas à l'appel de cette fonction le sens dans lequel se fait le décalage et pour cause, il se fait toujours vers les poids forts. Si on veut faire un décalage de  $n$  vers les poids faibles, il suffit en fait de faire un décalage de  $8 - n$  puis de regarder les poids forts du résultat seulement (c'est-à-dire le registre  $\$r1$ ).

### 3.2.5 Notes sur les sauts conditionnels

La densité joue également un rôle dans le fonctionnement des sauts conditionnels. Plutôt que de faire une instruction *branch if equal zero* prenant en argument à la fois une adresse et un registre à tester, on ne regarde qu'un registre et on laisse le choix du type (avant/arrière, absolu/relatif) et de l'amplitude du saut à l'instruction suivante.

Une amélioration que nous pourrions envisager dans les instructions de saut est l'ajout de sauts de taille fixe. Il y a à cela deux raisons :

---

2. La division est euclidienne

- Les instructions de saut relatif permettent de choisir un registre, contrairement à la philosophie que nous voulions adopter. C’est uniquement dû au fait qu’il restait de la place dans le jeu d’instructions de l’unité JUMP mais on pourrait parfaitement envisager de retirer certaines de ces instructions.
- Les sauts relatifs sont le cas le plus courant où il est nécessaire de charger une constante écrite sur plus de 3 bits et est donc fastidieux à programmer. Il serait donc bien plus simple d’avoir une instruction faisant un « grand » saut, même s’il est de taille fixe car on peut alors au besoin ajouter des instructions vides pour ajuster le répéter.

Il est à noter que ce comportement peut être simulé en enregistrant dans une case mémoire facilement accessible (entre 0 et 7, pour qu’un LI suffise à charger son adresse) une valeur élevée dès le début du programme, on peut émuler cette fonctionnalité.

### 3.2.6 Stockage de la date/heure

Nous imaginions d’abord réserver des adresses particulières de la RAM pour stocker les valeurs de la date/heure. Le problème de cette méthode est que l’on ne peut faire qu’un seul accès simultané à la RAM et il n’était donc pas possible d’afficher ces valeurs en même temps. Nous avons donc utilisé une mémoire spécifique : 7 registres en écriture seule branchés à la sortie font office de mémoire graphique.

Ces registres ne peuvent cependant pas être modifiés simultanément, du fait de notre jeu d’instruction réduit et de l’absence de 7 mémoire lisible simultanément. Nous avons donc doublé la mémoire graphique afin de créer un système de *double buffering* : les modifications sont apportées à une mémoire tampon au fur et à mesure du code et lorsque l’on appelle la commande FLIP, tous les registres d’affichage sont modifiés simultanément. La date/heure est ainsi affichée en une seule instruction.

Nous tenions à implémenter l’intégralité de la logique en netlist, c’est pourquoi la sortie de celle-ci est une suite de chiffre sous forme 7 segments, telle que l’on pourrait imaginer la brancher directement sur les segments de l’afficheur sans aucun traitement de la part de ce dernier. Les valeurs enregistrées dans la mémoire graphique passent donc dans un module de conversion en affichage à 7 segments avant d’être envoyées sur la sortie.

### 3.2.7 Interactions entre les différentes unités

On envoie à chaque unité son code opération (les quatre bits qui ne servent pas à définir l’unité), et elle exécute cette opération. L’ALU par exemple, envoie les valeurs \$a0 et \$a1 à chaque opérations, et reçoit tous les \$r0 et \$r1 associés. Elle mux alors selon le code opération qu’elle a reçu. De même, les résultats des unités sont mux pour n’effectuer que l’action voulue. Selon l’unité, ces résultats peuvent changer différents choses (les valeurs des registres pour l’ALU, l’emplacement de la tête de lecture pour JUMP, etc). Les instructions ayant généralement des effets de bord (le flip pour SYS, la modification de la RAM pour MEM, les sauts pour JUMP), on indique à chaque unité si elle doit agir ou pas.

La figure 2 présente l’architecture microprocesseur et l’interaction entre les différentes unités.

### 3.2.8 GPU

Derrière le module SYS se cache en fait un cinquième module : le GPU (*Graphical Processing Unit*, je précise par principe bien que je me doute que vous en connaissiez le sens). Le GPU représente en nombre de portes la moitié de notre processeur et a pour rôle de convertir les entiers codés en binaire en une série de codages 7 segments. Le poids important de cette partie du circuit vient du fait qu’il doit gérer simultanément les 7 entiers affichés.

Les deux actions les plus effectuées par le GPU sont les divisions euclidiennes par 10 et les conversions 4 bits binaires vers 7 segments (le comportement en cas de valeur supérieure à 9 n’étant pas spécifié). Nous avons tenté d’optimiser les divisions par 10, le diviseur étant connu statiquement, mais il s’est avéré que le nombre de portes n’était pas diminué. Il se peut que cette absence de différence entre la division générale et la division par une constante connue d’avance disparaisse lors de la phase d’optimisation du simulateur, mais nos tests datent d’avant l’existence de cette phase.

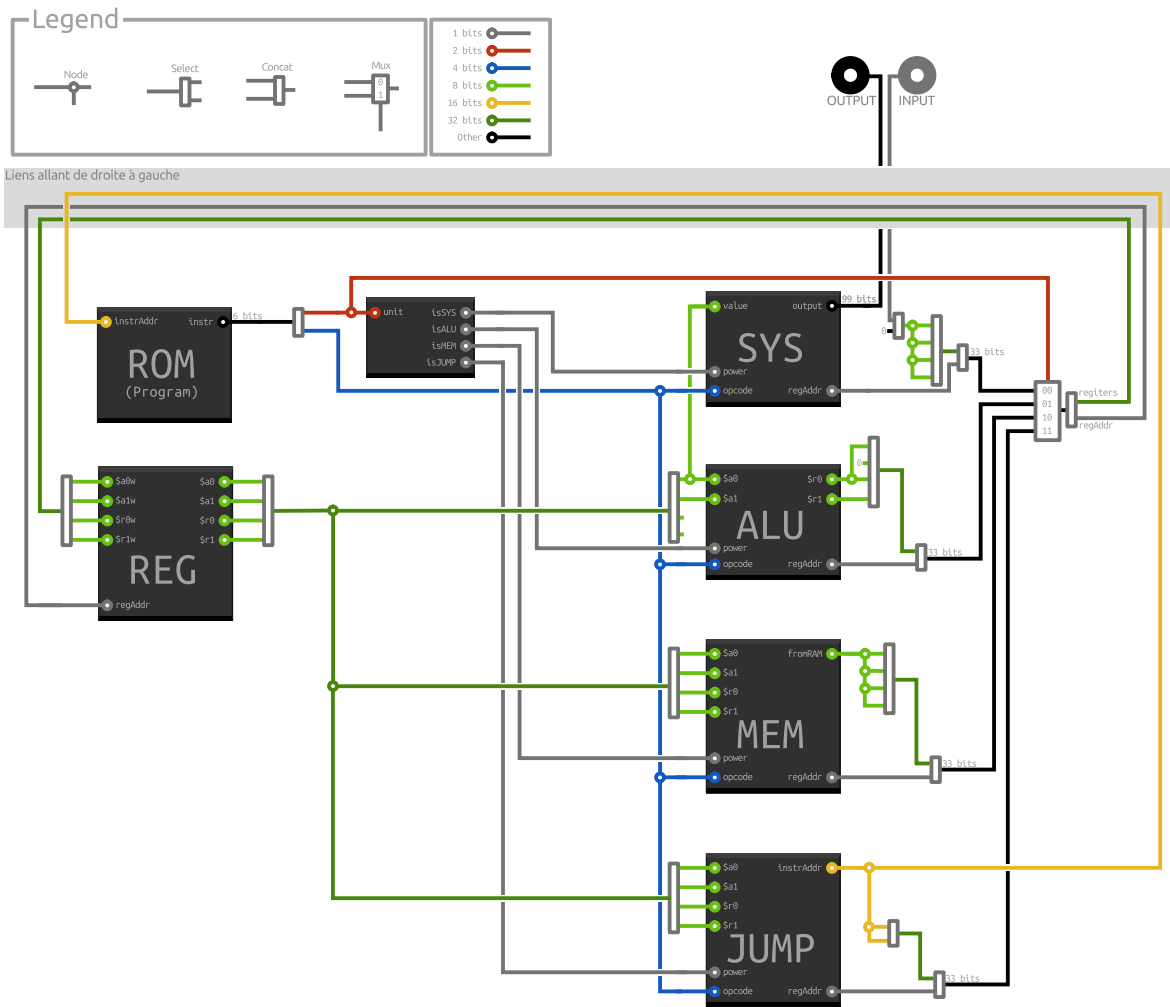


FIGURE 2 – Architecture microprocesseur

La séparation CPU/GPU est un point très intéressant du circuit : il est central puisque le GPU représente la moitié du circuit, mais les liens entre les deux sont d'une part peu nombreux et d'autre part à sens unique. Cela nous a permis de couper la simulation en deux afin d'optimiser l'utilisation de processeurs multi-cœurs en lançant simplement deux occurrences du simulateur.

Notons que nous aurions pu choisir, au vu du son poids important, de gérer la logique du GPU directement dans l'afficheur. Mais nous tenions à jouer le jeu de la simulation jusqu'au bout et imaginer la sortie du simulateur directement branchée fil par fil aux 98 segments de l'afficheur.

### 3.3 Ajout récent

Cette partie du microprocesseur n'était pas présente dans l'ancienne version rendue le 7 janvier mais s'est avéré être indispensable pour limiter la taille du programme.

On a déjà remarqué que les premières adresses mémoire (de 0 à 7) sont relativement simple d'accès (un *load immediat* suffit) alors que les adresses suivantes sont presque impossible à atteindre sans effacer presque tous les registres en pratique et donc inutilisables. L'idée que l'on a eu pour y remédier est d'ajouter un pseudo-registre, noté \$c, qui serait un curseur vers une adresse donnée de la RAM à partir duquel tous les accès mémoire se feraient. Un accès à l'adresse  $x$  se fait ainsi en réalité à l'adresse  $\$c + x$ .



Les opérations possibles sur ce pointeur  $\$c$  sont “ajouter 8” (INCR) et “retirer 8” (DECR), la valeur 8 correspondant au nombre d’adresses faciles d’accès. On enregistre en réalité  $\frac{\$c}{8}$  étant donné que les 3 derniers bits sont toujours à 0. Cette méthode permet de limiter le nombre d’instructions à ajouter.

Il devient ainsi très simple d’accéder aux adresses au-delà de 8, bien que le temps d’accès ne soit plus constant. On a également pu augmenter la taille de la RAM. Bien que ce système permette de gérer une RAM de taille quelconque, nous nous sommes contenté de  $2^{16} = 65536$  adresses. Notre programme n’utilise en réalité que 8 chunks (24 adresses) mais par soucis de généralité nous avons voulu montré qu’il était possible de gérer une grande quantité de RAM et donc d’envisager de compiler n’importe quelle programme vers notre assembleur, ce que 256 adresses ne permet pas vraiment.

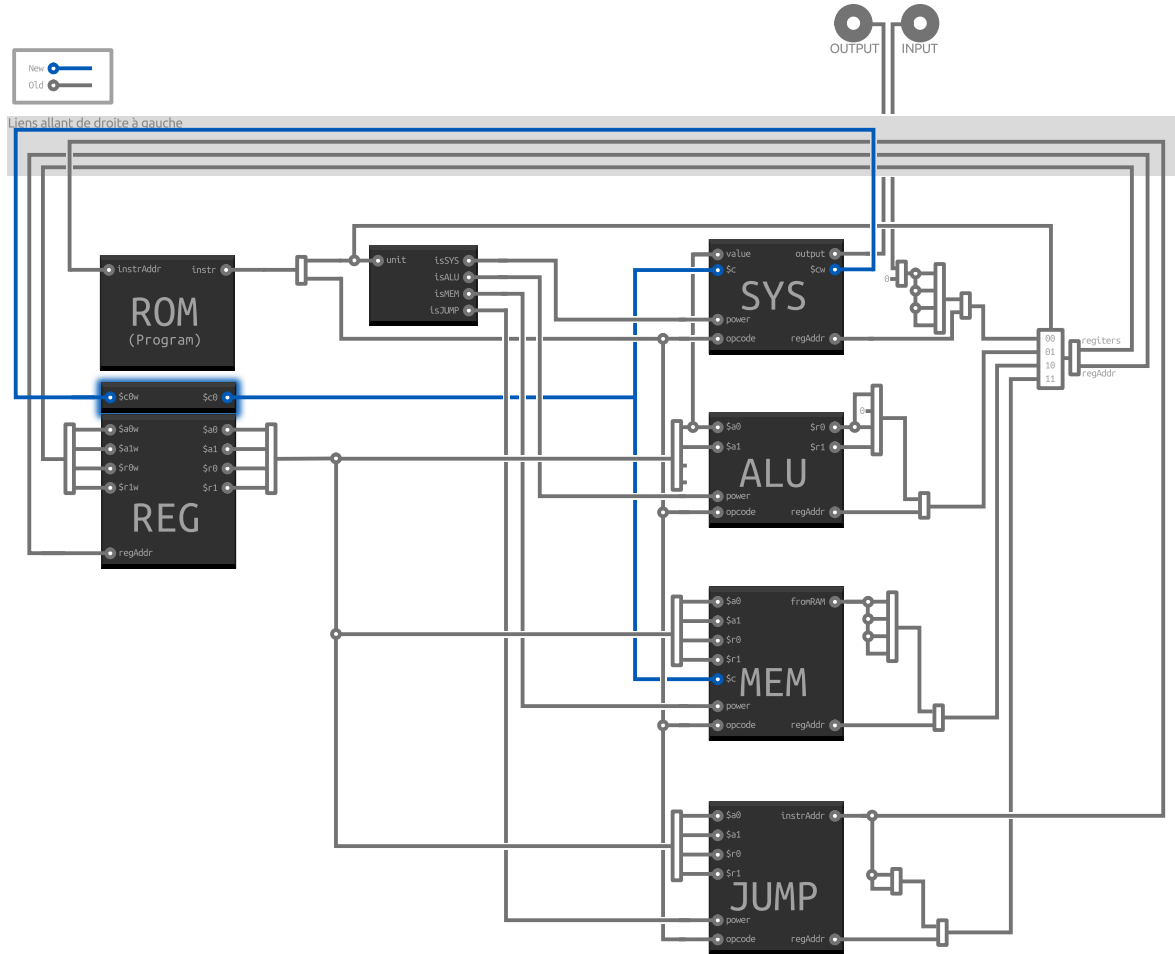


FIGURE 3 – Architecture microprocesseur mise à jour

La figure 3 met en évidence la modification apportée sur le schéma du microprocesseur. On peut noter que nous avons dû faire une entorse à notre partition en différentes unités : le curseur  $\$c$ , bien que relatif aux accès mémoire, est géré par le module système. C’était le seul endroit où il subsistait des adresses d’instruction libres.

### 3.4 Jeu d’instruction détaillé

Code	Nom	Description
00 ** **	SYS	
00 00 10	INCR	Incréméte le curseur de bloc.
00 00 11	DECR	Décréméte le curseur de bloc.

Code	Nom	Description
00 01 00	INPUT	Écrit l'input dans un registre donné.
00 1* **	OUTPUT x	Écrit la valeur de \$a0 dans la mémoire graphique x, avec $0 \leq x \leq 7$
00 11 11	FLIP	Actualise l'affichage du timer avec ce qu'il y a dans sa mémoire.
01 ** **	ALU	
00 00 **	Arithmétique	
01 00 00	ADD	Ajoute \$a0 et \$a1. Le résultat est dans \$r0, la retenue dans \$r1.
01 00 01	SUB	Soustrait \$a1 à \$a0. Le résultat est dans \$r0 <sup>3</sup> , la retenue dans \$r1.
01 00 10	MULT	Multiplie \$a0 et \$a1. Le résultat est dans la concaténation de \$r0 et \$r1.
01 00 11	DIV	Divise \$a0 par \$a1. Le quotient est dans \$r0, et le reste dans \$r1.
00 01 **	Logique	
01 01 00	AND	Calcule le AND bit-à-bit de \$a0 et \$a1 dans \$r0.
01 01 01	OR	Calcule le OR bit-à-bit de \$a0 et \$a1 dans \$r0.
01 01 10	NOT	Calcule le NOT bit-à-bit de \$a0 dans \$r0, et de \$a1 dans \$r1.
01 01 11	SHIFT	SHIFT \$a0 de min(\$a1,n) vers les poids forts. Le résultat est dans la concaténation de \$r0 et \$r1.
01 1* **	LI x	Charge une constante x à trois bits <sup>4</sup> dans \$a0.
10 ** **	MEM	
10 0* **	entre registres	
10 00 ij	MOVE 0ij	Déplace un \$a[i] vers un \$r[j].
10 01 ij	MOVE 1ij	Déplace un \$r[i] vers un \$a[j].
10 1* **	avec la RAM	
10 10 ij	LOAD ij	Charge la valeur en RAM(\$r[i]) dans \$a[j].
10 11 ij	SAVE ij	Enregistre dans RAM(\$r[i]) la valeur \$a[j].
11 ** **	JUMP	
11 00 ij	JFRA ij	Ajoute \$[i][j] à l'adresse de lecture (saut en avant relatif).
11 01 ij	JBRA ij	Retranche \$[i][j] à l'adresse de lecture (saut en arrière relatif).
11 10 ij	IIO ij	Saute une instruction si le registre donné est non nul.
11 11 0i	JAA i	Saut à une adresse absolue donnée par \$[i]0.\$[i]1
11 11 10	WCA	Écrit l'adresse courante dans \$a0.\$a1.
11 11 11	END	Termine le programme.

TABLE 1: Instructions microprocesseur

On peut constater qu'il manque 2 instructions dans ce tableau, les 2 premières. La toute première, l'instruction 00 00 00, a volontairement été laissée inactive car c'est la valeur que prennent toutes les cases de la ROM par défaut. Il était important pour aider au débogage qu'elle ne fasse rien et peut s'avérer utile si on fait des sauts de taille approximative et que l'on a donc besoin d'ajuster, ce qui peut être le cas lorsque l'on cherche à optimiser le code en profitant du fait qu'une certaine valeur soit déjà chargée dans un registre.

Pour l'instruction restante, elle ne fait rien non plus mais uniquement car on ne lui a pas trouvé de rôle. Elle fait partie de l'unité système et est trop peu nombreuse pour ajouter par exemple une fonction de lecture des registres graphiques. On ne pouvait lui donner de rôle sans aboutir à un jeu d'instruction complètement incohérent.

### 3.5 Outils de développement

Le microprocesseur doit être décrit sous forme de netlist. On se rend cependant très vite compte en faisant quelques essais que rédiger directement en netlist est extrêmement fastidieux (et de fait, on manipulera par moment des netlist de près de 30 000 lignes). De nombreuses portions du code se répètent, certaines sont paramétriques et si on décide de changer la taille des mots machine par exemple on est bien embêté.

C'est pourquoi il est impératif d'utiliser une description plus haut niveau. Il y a pour cela deux possibilités :

- utiliser un programme générant directement la netlist pour le micro

3.  $\$r0 = \$a0 - \$a1$  est certifié uniquement si  $\$a0 > \$a1$

4. Le reste est rempli de 0

- compiler en netlist depuis un langage de description de circuits plus haut niveau.

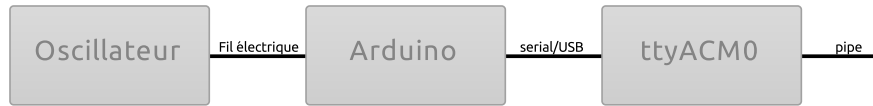
Un compilateur minijazz étant fourni, nous avons choisi la seconde option. Ce choix a cependant posé plusieurs problèmes.

Premièrement, minijazz étant un langage très simple, il ne gère pas l'import de fichiers les uns dans les autres. Cette fonctionnalité nous semblait pourtant essentielle, d'une part pour pouvoir utiliser une même portion de circuit dans plusieurs tests distincts (par exemple le gpu et l'alu) et d'autre part pour pouvoir travailler sans se gêner sur le circuit (git gère bien mieux des fichiers séparés).

Deuxièmement, les messages d'erreur de mjc étaient assez obscures, que ce soit les « The following constraint is not satisfied :  $((8 = 0) ? 1 : (8 = (((0 \leq (8 - 1)) ? (((8 - 1) - 0) + 1) : 0) + ((0 \leq (8 - 8)) ? (((8 - 8) - 0) + 1) : 0))))$  » ou les erreurs localisées « Quelque part entre la première et la dernière ligne du fichier ». Le débogage consistait donc en une longue dichotomie sur tout le code du microprocesseur.

## 4 Programme

## 5 Oscillateur



## 6 Afficheurs

