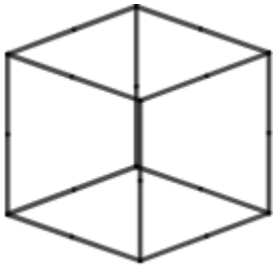
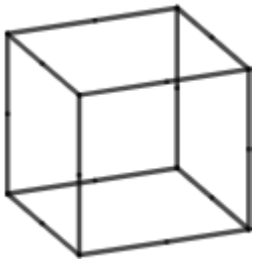
Conception de circuit numérique

Conception et implémentation d'un système de rendu 3D sur FPGA


Pierre JOUBERT
Julien BESSE
SEE08

Professeur :
Christophe JEGO

1. Table des matières

2. Introduction

Le projet que nous avons choisi est la réalisation d'un système capable de générer des images de synthèse à partir de scènes 3D très simples.

Dans un ordinateur grand public, le calcul de ces images que nous appellerons *rendus* se fait avec deux unités de calcul :

- Le CPU, pour *Central Processing Unit*, qui est un processeur généraliste fait pour exécuter des programmes. Un CPU est une pièce qui se trouve dans tous les ordinateurs car elle est nécessaire, c'est elle qui est capable d'accéder aux périphériques et exécuter un système d'exploitation.
- Le GPU, pour *Graphics Processing Unit*, qui est un processeur assurant les fonctions de calcul de l'affichage. Ce type de processeur est en général organisé pour faire des calculs parallélisés car les tâches graphiques le permettent ; contrairement à un CPU qui exécute le plus souvent des tâches composées d'un seul thread.

Nous pouvons constater que la solution la plus commune pour effectuer un rendu avec de bonnes performances est de déléguer une partie du travail à un processeur dédié et optimisé pour cela ; pour notre projet, nous avons décidé de suivre de principe : bien qu'il soit identique au CPU, nous choisissons d'utiliser un second processeur pour dessiner à l'écran.

3. Architecture du système

3.1. Vue d'ensemble

Notre système est composé de deux processeurs, équipés de leurs périphériques. Le CPU est relié via un bus à :

- Une RAM contenant le programme et les variables du CPU,
- Une RAM partagée avec le GPU,
- Une mémoire en lecture seule pour les calculs de sinus et cosinus,
- Un contrôleur GPIO (General Purpose Input Output) permettant de communiquer avec l'utilisateur,
- Un périphérique SPI lié à un accéléromètre était prévu, mais celui-ci n'a finalement pas été implémenté.

Le GPU, quant à lui, est relié à :

- Une RAM contenant le programme et les variables du GPU,
- Une RAM partagée avec le CPU,
- Une mémoire en lecture seule pour les sinus et cosinus,
- Un périphérique générant des signaux pour un écran VGA.

Pour faciliter le développement, un programmeur via port série a été implémenté. Celui-ci peut modifier le contenu des RAM CPU et GPU afin d'y mettre un code généré avec notre chaîne d'outils.

L'architecture globale choisie est présentée ci-dessous.

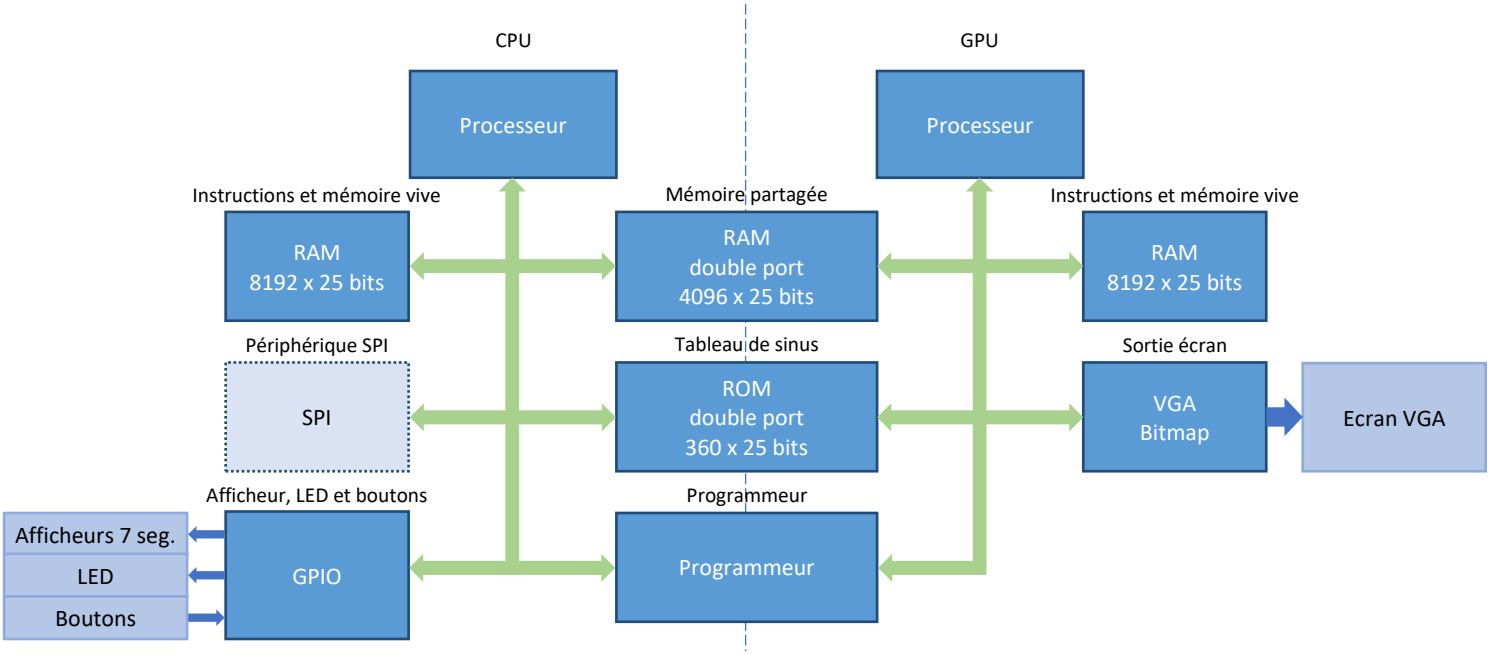


Figure 1 architecture globale du système

Nous pouvons voir que chaque processeur possède son bus, les communications ne peuvent se faire que via la mémoire partagée. A l'image d'un système moderne, le CPU a accès aux périphériques généraux, et le GPU a accès au périphérique d'affichage.

Pour simplifier le développement, le système entier est cadencé sur une horloge de 100 MHz. Cette fréquence est imposée par le périphérique VGA qui doit respecter une fréquence de rafraichissement d'écran de 60 Hz.

3.2. Bus et organisation des espaces mémoire

Pour que les processeurs puissent communiquer avec les périphériques, ceux-ci sont interconnectés via un bus. Ce bus est constitué de plusieurs signaux :

NOM	SENS		
Adresse	Processeur	→	Périphérique
Données d'entrée	Processeur	←	Périphérique
Données de sortie	Processeur	→	Périphérique
Activation du bus	Processeur	→	Périphérique
Mode écriture	Processeur	→	Périphérique

Figure 2 signaux composant le bus

De par le nombre réduit de signaux composant ce bus, celui-ci est facile à implémenter. Si le maître du bus veut lire une adresse, il suffit de valider le signal d'activation, donner la valeur souhaitée au signal d'adresse, et le périphérique aura retourné la donnée correspondante sur le signal « données d'entrée » avant le front montant suivant de l'horloge.

Comme détaillé dans la section 3.6, le périphérique d'affichage VGA nécessite un espace mémoire de 307201 adresses à lui tout seul ; il faut donc 19 bits d'adresse au minimum, ce qui fait

524287 adresses possibles ; 217086 si on retire celles occupées par le buffer du VGA. Par sécurité et pour être sûrs de ne pas manquer de place, nous avons décidé que la largeur du signal d'adresse serait de 20 bits, donc un espace mémoire de 1048575 adresses.

Le système de rendu suivant une architecture *Von Neumann*, la largeur du bus des données correspond à la largeur du code d'instruction suivi de l'adresse associée. Nous avons choisi d'utiliser 5 bits pour coder l'opérande ; ce qui fait un total de 25 bits de données.

L'organisation de l'espace mémoire est schématisée ci-dessous :

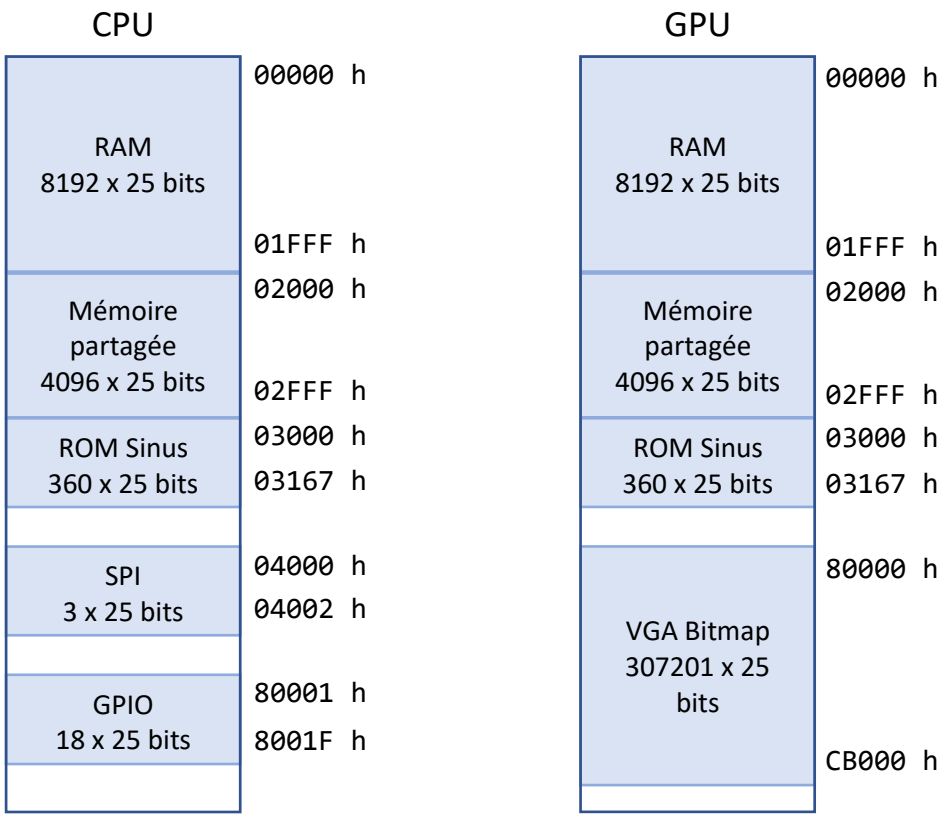


Figure 3 organisation des espaces mémoire

Les RAM, la mémoire partagée ainsi que la ROM étant des composants communs aux deux processeurs, l'adressage est identique. Les périphériques qui ne sont pas présents sur les deux bus ont été mis à la suite selon un placement que nous avons jugé optimal.

3.3. Les processeurs

Le CPU et le GPU du système sont deux instances d'un même bloc : un processeur 25 bits venant de la modification du processeur 8 bits réalisé lors du cours EN217 portant sur la conception d'un processeur à jeu d'instructions élémentaires. Un schéma se trouvant en annexe à la page 0 représente de processeur modifié.

Les modifications apportées à ce processeur sont les suivantes :

3.3.1. Passage de 8 bits de données à 25 bits

Pour nos besoins, les signaux de données, d'adresse et de codes d'instruction sont passés de 8 bits, 6 bits et 2 bits à 25 bits, 20 bits et 5 bits respectivement.

La modification de la description des blocs logiques du processeur a été anticipée lors de la conception de celui-ci. La largeur des signaux est donc modifiable via des valeurs génériques ; cette opération a donc été courte et réussie du premier coup.

3.3.2. Ajout d'opérations arithmétiques et logiques

Pour améliorer les performances du processeur, des instructions ont été ajoutées. Ci-dessous se trouve une comparaison des instructions arithmétiques et logiques du processeur avant et après modification :

Instructions processeur 8 bits original	*Instructions processeur 25 bits*
Non ou	Non ou
	Ou
	Et
	Ou exclusif
Addition	Addition sur entiers
	Soustraction sur entiers
	Division sur entiers
	Multiplication sur entiers
	Modulo sur entiers
	Addition sur virgule fixe
	Division sur virgule fixe
	Multiplication sur virgule fixe
	Charger

Figure 4 comparaison des opérations arithmétiques et logiques

Bien que les instructions originales nous permettaient d'effectuer la plupart des calculs nécessaires, cela serait lent car il faudrait plusieurs cycles de calcul pour obtenir un résultat. Toutes ces opérations sont exécutées en un seul cycle ; la multiplication est donc beaucoup plus rapide avec le nouveau processeur.

Pour des raisons de performances de l'implémentation de ce système dans un FPGA, le calcul d'une multiplication et d'une division sur des nombres entiers ne se fait pas sur les 25 bits de données : les opérandes de la multiplication sont tronqués, seuls les 16 bits de poids faible sont lus. La division ne prend que les 7 premiers bits du numérateur et les 4 premiers bits du dénominateur.

Pour le calcul des coordonnées, employant des multiplications avec des nombres réels, la gestion des nombres à virgule fixe ainsi que les opérations d'addition, de division et de multiplication ont été ajoutées au processeur.

Ces nombres à virgule fixe sont constitués de 11 bits pour la partie entière et 8 bits pour la partie décimale. Le choix de cette taille de partie entière a une raison : ces nombres seront utilisés pour le calcul des coordonnées des segments affichés à l'écran. Le résultat final sera donc un nombre compris entre 0 et 639. La taille de l'affichage est détaillée à la section 3.6.

Le choix de la partie décimale s'est fait après plusieurs tests d'implémentation sur FPGA. Un équilibre a été trouvé entre précision et délai de propagation du résultat des opérations.

La valeur maximale d'un nombre à virgule fixe est donc environ 1023.996 et la valeur minimale – 1024. La valeur d'un bit de poids faible est environ 0.0039, ce qui est suffisant pour les calculs que nous allons exécuter.

Le calcul du modulo sur nombre entier ainsi que la division sur les nombres à virgule n'ont pas été implémentés, car ces calculs nécessitent beaucoup de ressources sur un FPGA, et les temps de propagations ne permettaient pas de cadencer le processeur à la vitesse fixée. Ces calculs ne sont pas utiles pour effectuer des rendus.

Enfin, une opération « charger » a été ajoutée ; elle permet de charger une valeur dans le registre d'accumulation. Cette opération évite d'avoir à vider le registre en faisant une opération NON-OU d'un registre dont tous les bits sont à 1.

Des opérations de comparaison entre deux nombres ont été ajoutées : le processeur peut déterminer si un nombre est plus petit, égal ou plus grand que celui contenu dans le registre de sortie du résultat précédent. Le résultat binaire de ces opérations est transmis à la machine à états du processeur via le registre carry.

3.3.3. Ajout d'accès à la mémoire selon une adresse calculée

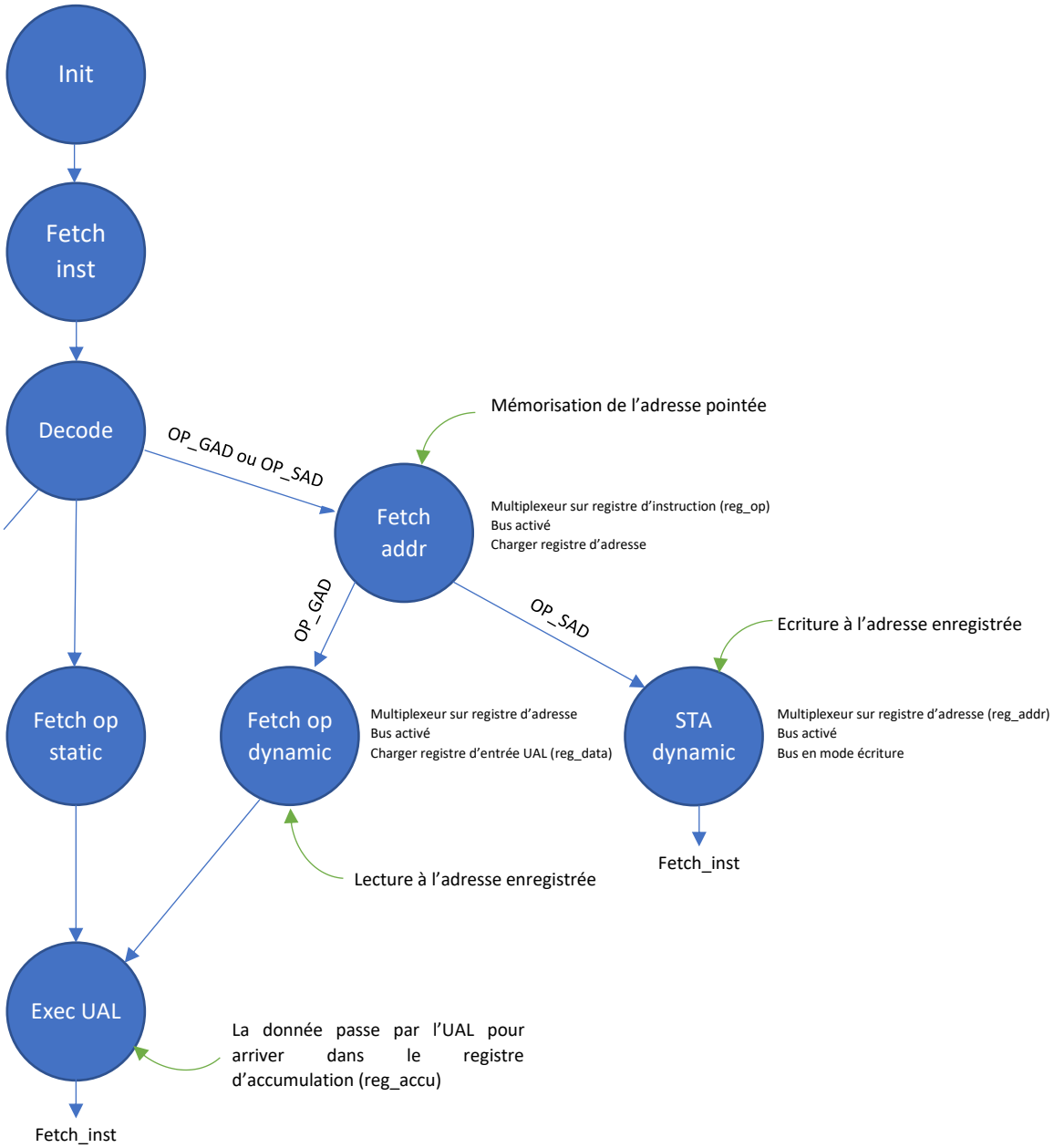
D'autres instructions ont été ajoutées au microprocesseur pour répondre à un besoin de notre projet : pouvoir accéder à un pixel selon ses coordonnées. Le périphérique VGA est en effet vu comme une grande zone mémoire donc chaque case représente un pixel par le GPU.

Les opérations STA et les calculs faits requièrent de connaître l'adresse de source ou de destination lors de l'écriture du code. De ce fait, le seul moyen de faire une écriture à un endroit déterminé à l'exécution est de générer une instruction puis l'exécuter.

Les instructions nommées SAD et GAD pour *Set at ADdress* et *Get at ADdress*, prennent en paramètre l'adresse d'une variable contenant un pointeur vers la case mémoire visée. Ainsi, si le programme exécute l'instruction SAD 0A, le processeur va chercher la valeur de l'adresse 0A, l'enregistrer dans un registre dédié aux adresses, et enfin écrire la valeur du registre d'accumulation à l'adresse enregistrée.

Pour implémenter ces instructions, des modifications ont été faites dans la machine à états finis : deux nouvelles branches ont été ajoutées pour l'étape de mémorisation de l'adresse cible, et des états « fetch op dynamic » et « STA dynamic » ont été créés.

Le registre dédié aux adresses a été rajouté dans l'unité de contrôle, il prend en entrée la donnée de la RAM, et sa sortie est dirigée vers une troisième entrée du multiplexeur d'adresse. La partie modifiée de la machine à états se trouve ci-dessous :



Dans le cas de l'opération GAD, la valeur lue à l'adresse pointée passe par l'UAL, qui, lorsque son code d'opération correspond à GAD, la transmet à sa sortie.

3.3.4. Ajout d'une entrée d'inhibition

Pour permettre de « libérer le bus » du processeur, ce qui est utile lorsque le programmeur est utilisé pour modifier le contenu la RAM, une entrée « cpu_init » a été ajoutée. Ce signal, synchrone à l'horloge, bloque le processeur dans un état d'initialisation et ses sorties vers le bus sont mises en haute impédance.

3.4. Les mémoires

Toutes les mémoires du système sont des instances d'IP RAM générées avec l'outil de Xilinx.

Une description de RAM a été faite en VHDL, mais le fait d'utiliser l'IP rend l'implémentation sur FPGA plus rapide et optimisée. En effet, le FPGA est équipé de « blocs de RAM », qui ne sont utilisés que si l'outil de placement-routage juge cela possible. L'emploi de l'IP assure que ce choix sera fait, et l'outil de synthèse ne perd pas de temps à traiter la description des RAM.

Les processeurs possèdent chacun une RAM de 8192 mots de 25 bits faites pour contenir le programme et mémoriser les variables. Il d'agit deux instances de l'IP nommée « blk_mem_gen_0 » ; la configuration du générateur d'IP est la suivante :

Type d'interface	Native
Type de mémoire	RAM simple port
Largeur des données	25 bits
Profondeur des données	8192 mots
Registres de sortie	Désactivés
Fichier d'initialisation	Oui

Figure 5 configuration du générateur d'IP pour la RAM de programme et variables

Les registres de sortie ont été désactivés pour que les données soient disponibles sans attendre un coup d'horloge. Ceci n'est pas optimal pour une implémentation sur FPGA, mais les RAM sont prévues pour fonctionner à des fréquences bien plus élevées ; il n'y a donc pas de problème lors de l'implémentation.

Un fichier d'initialisation décrit l'état de la RAM après configuration du FPGA. Ceci permet d'exécuter un programme avec les processeurs sans avoir à les programmer via le port série (fonctionnement décrit en section 3.5).

Les processeurs ont en commun une RAM de 4096 mots de 25 bits, faite pour partager des données entre le CPU et le GPU et inversement. Cette RAM est reliée aux deux bus, il s'agit donc d'une RAM double port. L'IP instanciée pour cette RAM se nomme « blk_mem_gen_1 » et sa configuration est la suivante :

Type d'interface	Native
Type de mémoire	RAM double port
Horloge commune	Oui
Largeur des données	25 bits
Profondeur des données	4096 mots
Registres de sortie	Désactivés
Fichier d'initialisation	Non

Figure 6 configuration du générateur d'IP pour la RAM de programme et variables

La ROM implémentée permet de calculer le sinus d'un nombre. Il s'agit d'un tableau de 360 valeurs, dont la sortie vaut le sinus de l'entrée si on l'interprète comme un nombre à virgule fixe.

3.5. Le programmeur

Pour modifier le programme exécuté par le CPU et le GPU, un bloc nommé « programmeur » a été conçu.

Ce bloc est capable d'inhiber le microprocesseur du bus auquel il est relié, ce qui libère ce dernier ; le programmeur peut donc accéder aux RAM. Lorsque le programmeur n'est pas actif, celui-ci met ses connexions au bus en haute impédance pour ne pas gêner le fonctionnement normal.

Pour activer le programmeur, l'utilisateur doit appuyer sur un bouton de la carte (BTNL pour le CPU et BTNR pour le GPU), puis envoyer via le port série des mots de 32 bits qui seront tronqués à 25 bits. Le programmeur se désactive après avoir reçu exactement 8192 mots de 32 bits et réactive le processeur en même temps ; il faut envoyer systématiquement tout le contenu de la RAM.

Le port série est lu via une IP VHDL récupérée sur nandland.com. Les échanges se font en 8 bits, 1 bit de stop et pas de bit de parité.

3.6. Le périphérique VGA

Le périphérique VGA est une IP réalisée par Mr Bornat. Nous avons choisi une résolution d'affichage de 640 par 480 pixels, avec 4 bits par pixels codant des niveaux de gris.

Cette IP étant déjà prévue pour être accédée comme une RAM, l'interfaçage au bus est trivial.

Suite aux premiers essais, le besoin d'une synchronisation entre le GPU et les rafraichissements de l'écran s'est fait sentir. En effet, des artefacts apparaissaient car le GPU écrivait sur l'écran pendant le rafraichissement de celui-ci, l'image était donc scindée horizontalement.

Nous avons donc implémenté un système de rotation de buffers :

- deux RAM Xilinx de 307199 x 4 bits ont été instanciées, améliorant la vitesse de synthèse et diminuant l'utilisation d'éléments logiques du FPGA,
- un registre à l'adresse CB000h est mis à 0 lorsqu'un buffer est prêt à être rempli par le GPU. Une fois que celui-ci a fini de dessiner, il écrit la valeur 1 dans le registre pour indiquer au périphérique que le buffer est prêt à être affiché,
- lorsqu'un buffer est prêt, le périphérique attend la fin d'un rafraîchissement pour permuter les buffers. Le nouveau buffer est donc affiché. Le buffer précédent est d'abord remis à zéro, puis mis à disposition du GPU.

Le schéma suivant illustre ce fonctionnement (Figure 7).

La remise à zéro des RAM ne pouvant se faire qu'en écrivant 0 dans chaque case une par une, il faut 307200 coups d'horloge pour la vider, soit 3 millisecondes environ. Avant l'implémentation de ce système de remise à zéro, le GPU s'en chargeait, ce qui prenait plusieurs centaines de millisecondes ; le gain en performances est donc important.

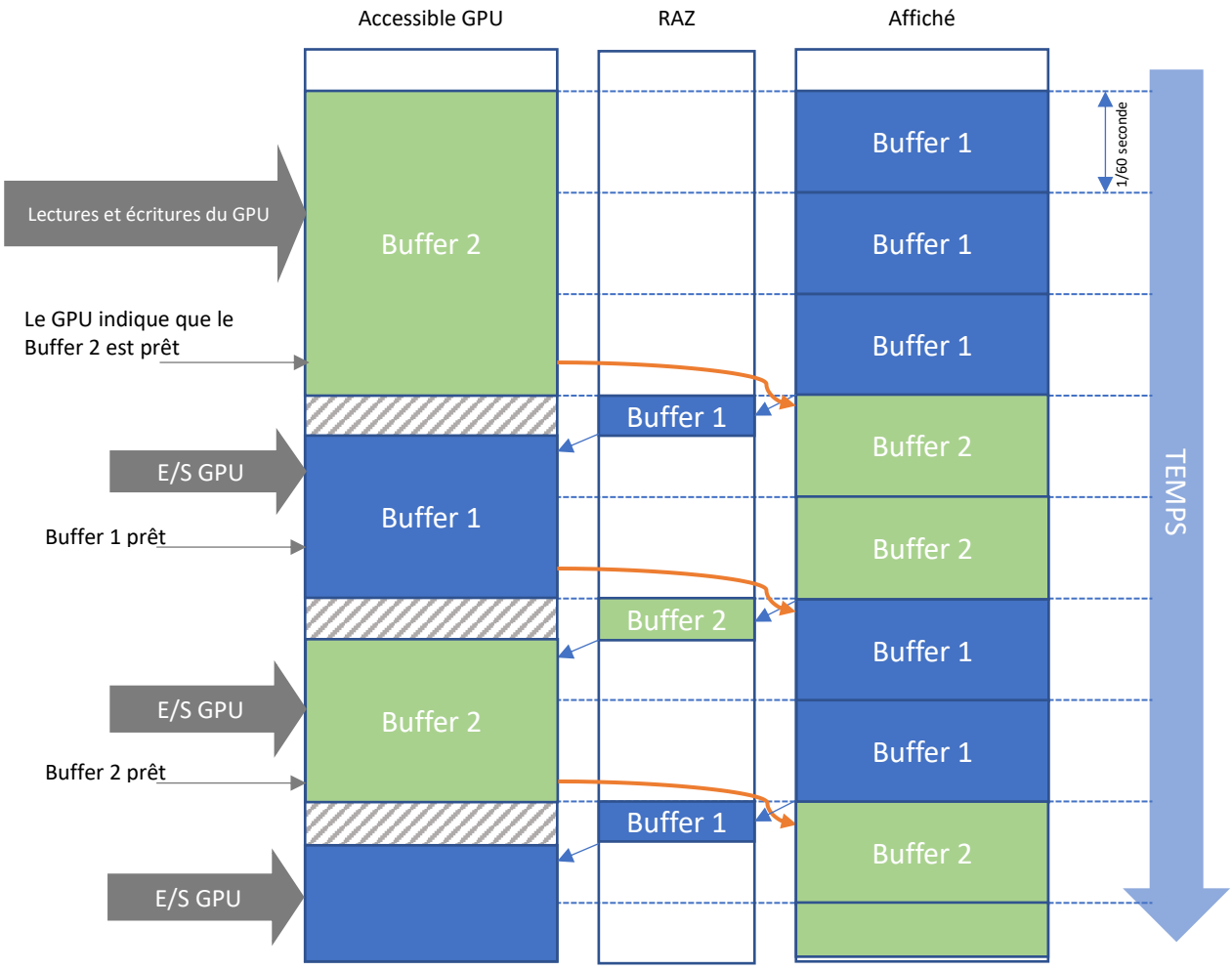


Figure 7 rotation des buffers du périphérique VGA

3.7. Le périphérique GPIO

Pour aider au développement du système et de ses programmes, un contrôleur GPIO est implémenté. Il permet d'afficher une valeur sur l'afficheur 7 segments, de contrôler la luminosité des 16 LED vertes présentes sur la carte, et enfin de lire une valeur codée avec les 16 boutons à glissière.

L'afficheur 7 segments reflète la valeur de la case mémoire `80001h` : Toute valeur écrite est représentée en hexadécimal.

La luminosité de chaque LED peut être contrôlée en écrivant aux adresses `80010h` à `8001Fh` : une valeur de 0 éteint la LED, et une valeur supérieure à 255 (entier) ou 1 (virgule fixe) allume la LED au maximum. Le contrôleur GPIO fait varier le rapport cyclique du temps d'allumage de chaque LED.

La case mémoire `80002h` peut être lue par le processeur pour récupérer l'état des 16 boutons à glissière, le bouton 15 correspondant au bit 15 et le bouton 0 au bit 0. On peut donc choisir une valeur entre 0 et 65535 (entier) ou 0 et 255,996 (virgule fixe).

3.8. Le périphérique SPI

Un périphérique SPI était prévu sur le bus du CPU pour lui permettre de détecter l'orientation de la carte avec l'accéléromètre équipé sur la carte de développement.

Ce bloc n'a pas été décrit, car le budget de temps du projet venait à sa fin. Ce bloc n'est pas nécessaire à la démonstration de rendu 3D ; l'impact est donc minime.

3.9. Rapport d'implémentation sur le FPGA Xilinx

Lors de l'implémentation sur le FPGA Xilinx, nous avons utilisé une IP pour configurer une PLL du FPGA. Celle-ci prend le signal 100 MHz de l'oscillateur externe et recrée un signal de 100 MHz interne au FPGA avec des caractéristiques connues (temps de gigue, rapport cyclique, etc.).

Le brochage des signaux sortant du FPGA a été spécifié dans le fichier de contraintes *Nexys4_CPU.xdc*

Après placement-routage, l'outil a rapporté un problème de temps de propagation : dans les pires conditions, un signal arrivera 1,27 nanoseconde trop tard en sortie de l'unité arithmétique et logique. La période de l'horloge étant de 10 nanosecondes, cela représente quasiment 13% de dépassement. Nous avons décidé d'ignorer cet avertissement et de prendre le risque d'avoir une erreur de calcul de temps en temps.

Le rapport d'utilisation du FPGA est le suivant :

Type de ressource	*Utilisation*	*Disponible*	*Part utilisée*
LUT (Blocs logiques)	2835	63400	4.47%
Registres FF	1160	126800	0.91%
Blocs de RAM	91	135	67.41%
Entrées/sorties	69	210	32.86%
BUFG (Buffers d'horloge)	2	32	6.25%
MMCM (PLL)	1	6	17%

Figure 8 rapport d'utilisation du FPGA sous forme de tableau

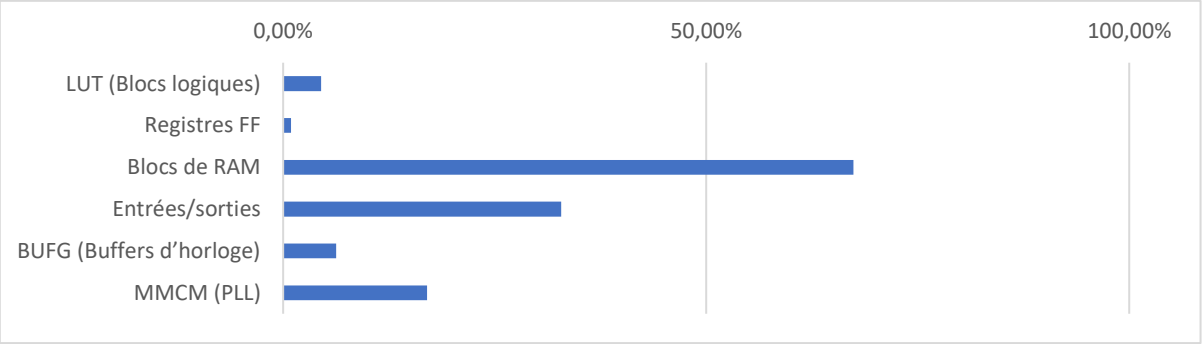


Figure 9 rapport d'utilisation du FPGA graphique

Nous pouvons conclure de ces chiffres que le FPGA est largement suffisant pour nos besoins. Les éléments logiques sont en grande partie inutilisés ; nous pourrions donc augmenter la

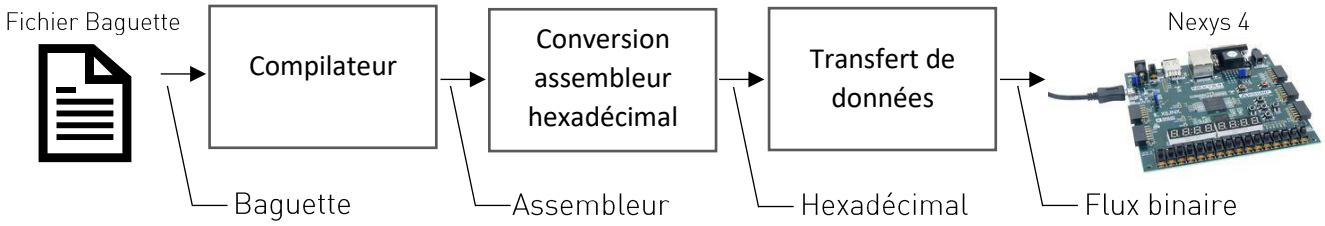
complexité de notre système, ou passer à un FPGA plus petit, quitte à « faire déborder » la RAM dans les LUT, c'est-à-dire utiliser des LUT pour faire de la RAM, ce que Xilinx nomme « LUTRAM ».

4. La toolchain

Afin de nous permettre une génération aisée de programmes, nous avons choisi de réaliser une « toolchain » ou « chaine d'outil », permettant d'utiliser un langage haut niveau. Ainsi, nos outils permettent de s'affranchir de la programmation en binaire ou hexadécimal et de nous rapprocher des langages « modernes » comme le C. Deux langages ont été utilisés pour réaliser cette toolchain : le C pour la conversion assembleur – binaire et le transfert des données, le C++ pour le compilateur.

Notre processeur ne permettant pas de faire toutes les opérations permises sur un processeur standard, nous avons fait le choix d'également créer un langage de programmation pour inciter les utilisateurs à ne pas utiliser une fonctionnalité non implémentée. Ainsi, nous avons conçu un langage, en français, subtilement nommé *le langage baguette*. L'utilisation de ce dernier sera détaillée dans le chapitre 5.

Le schéma décrivant le fonctionnement de la *toolchain* est le suivant :



Les chapitres suivants vont détailler le fonctionnement des blocs de cette *toolchain*.

4.1. Compilateur

Le compilateur est le programme permettant que convertir le langage baguette en assembleur. Son fonctionnement est basé sur la reconnaissance de mots clés dans un texte. Lorsqu'un mot clé est détecté, le programme cherche à déterminer le ou les arguments liés à l'instruction. Le découpage s'effectue de la sorte :

```
Code baguette :
      entier variable1;
      entier variable2;
      variable1 = 20;
      variable2 = variable1 / 5;
```

Avec en rouge les « mots clés », en bleu les variables de retour ainsi qu'en vert, les arguments.

Après avoir découpé le programme d'entrée, le compilateur va réécrire les instructions dans le même ordre que l'initial, mais transcrit en instructions assembleur.

4.2. Conversion assembleur – binaire

La conversion assembleur – binaire consiste en la retranscription des instructions en assembleur vers le code binaire associé à l'instruction dans le processeur. Le code d'association est le suivant :

Opération	Valeur	Description
Logical operands		
NOR	0x0	
LOR	0x1	logical OR
AND	0x2	
XOR	0x3	
Mathematical operands		
ADD	0x4	
SUB	0x5	
DIV	0x6	
MUL	0x7	
Fixed precision operands		
FAD	0x9	addition
FDI	0xA	division
FMU	0xB	multiply
Casts		
FTI	0xC	fixed to int
ITF	0xD	int to fixed
Utils		
STA	0x10	
JCC	0x11	
JMP	0x12	jump
GET	0x13	
Tests		
TGT	0x14	greater than
TLT	0x15	lower than
TEQ	0x16	equal
Memory movements		
GAD	0x18	Get @ address
SAD	0x19	Set @ address

Ainsi, voici un exemple de conversion effectuée par le programme :

Instructions assembleur	Instructions binaires
NOR 00019	0000019
ADD 00017	0400017
STA 0001c	100001c
NOR 00019	0000019
ADD 0001c	040001c
STA 80001	1080001
NOR 00019	0000019
ADD 0001c	040001c
TGT 0001d	140001d
JCC 00013	1100013
NOR 00019	0000019
ADD 0001c	040001c
ADD 00018	0400018
STA 0001c	100001c
NOR 00019	0000019
ADD 0001c	040001c
STA 80001	1080001
JCC 00006	1100006
JCC 00006	1100006
NOR 00019	0000019
ADD 0001c	040001c
STA 80001	1080001
JCC 00016	1100016
VAR 0000000	0000000
VAR 0000001	0000001
VAR 1ffffff	1ffffff
VAR 0003000	0003000
VAR 0002000	0002000
VAR 0000000	0000000
VAR 0000064	0000064

Nous voyons que les instructions en 3 lettres cités auparavant sont converties en instructions hexadécimales de deux caractères selon la table de conversion citée auparavant. Une instruction supplémentaire a été ajoutée : VAR. Cette instruction permet d'indiquer que la valeur qui suit n'est pas à concaténer avec des caractères hexadécimaux, mais à traiter comme une variable et ainsi à recopier à l'identique.

4.3. Transfert des données

Le transfert des données est basé sur un envoi octet par octet à travers une liaison série. La première étape consiste en la lecture du fichier binaire. La seconde étape consiste en l'envoi des données les unes après les autres à travers la liaison série spécifiée en argument du programme.

5. Le langage *Baguette*

5.1. Syntaxe

Le langage baguette est sensible à la casse. Ainsi, les variables « var » et « VAR » seront traitées différemment. Tous les caractères alphanumériques peuvent-être utilisés pour nommer les variables. Il est demandé de ne pas combiner les calculs. Ainsi, pour effectuer le calcul A = A * (B+2), il faudra effectuer le calcul suivant :

```
Code baguette :
      entier A;
      entier B;
      entier temp;
      A = 20;
      B = 30;
      temp = B + 2;
      A = A * temp;
```

5.2. Gestion des variables

Pour créer une variable, il est nécessaire de connaitre son type, soit entier non signé, soit réel à virgule fixe. La création de variables s'effectue de la manière suivante :

Type de variable	Syntaxe
Entier	entier A;
Réel virgule fixe	reel A;

5.3. Opérations arithmétiques

Les opérations arithmétiques disponibles avec le langage *baguette* sont les suivantes :

Opération	Syntaxe
Addition	A = A + 1;
Soustraction	A = A – 1;
Multiplication	A = A * 3;
Division	A = A / 3;

5.4. Opérations logiques

Les opérations logiques disponibles avec le langage *baguette* sont les suivantes :

Opération	Syntaxe
NOR	A = A ~| 1;
OR	A = A | 1;
AND	A = A & 3;
XOR	A = A ^ 3;

5.5. Les opérations spéciales

Les opérations logiques disponibles avec le langage *baguette* sont les suivantes :

Opération	Syntaxe
sin	A = sin(15);
cos	A = cos(40);

Le sinus et le cosinus fonctionnent en degrés, et retournent une variable en virgule fixe.

5.6. La conversion de types

Pour utiliser une opération, il est nécessaire d'utiliser les mêmes types de variables. Pour convertir une variable dans un autre type, le langage *baguette* permet les conversions suivantes :

Opération	Syntaxe
Réel vers entier	A = rve(B);
Entier vers réel	A = evr(B);

5.7. Les conditions

Une boucle s'écrit de la manière suivante :

```
si (A < 3)
      //faire quelque chose
fin_si;
```

Les conditions disponibles sont les suivantes :

Opération	Syntaxe
Egale à	if(A == B)
Supérieur à	if(A > B)
Inférieur à	if(A < B)

5.8. Les boucles

Une boucle s'écrit de la manière suivante :

```
tant_que (A < 3)
      //faire quelque chose
fin_tant_que;
```

Les tests disponibles sont les mêmes que pour les conditions.

5.9. Opérations mémoires

Opération	Syntaxe
Ecrire une valeur à une adresse	ecrire_a (valeur, adresse);
Lire a une adresse	lire_a(adresse);
Ecrire sur les 7 segments	afficher_LCD(valeur);

5.10. La coloration syntaxique

Afin de nous permettre une édition de programme plus aisée, nous avons développé un script de coloration syntaxique pour le logiciel *SublimeText3*. Le rendu de ce script est le suivant :

```
3   reel x1_trace_f;
4   reel y1_trace_f;
5
6   entier x0_trace;
7   entier y0_trace;
8   entier x1_trace;
9   entier y1_trace;
10
11  entier nb_seg;
12
13  entier cpt_seg;
14  entier cpt_add;
15  entier adresse;
16
17  entier base;
18  entier retour;
19
20  base = 524288;
21
22  //vidage de l'écran
23  adresse = base;
24  tant_que(adresse < 831487)
25    ecrire_a (0, adresse);
26    adresse = adresse + 1;
27  fin_tant_que;
28  ecrire_a (1, 831488);
29
30  retour = 1;
31  tant_que(retour > 0)
32    retour = lire_a(831488);
33  fin_tant_que;
34
35  //vidage de l'écran
36  adresse = base;
37  tant_que(adresse < 831487)
38    ecrire_a (0, adresse);
39    adresse = adresse + 1;
40  fin_tant_que;
41  ecrire_a (1, 831488);
```

Figure 10 : Capture d'écran du logiciel SublimeText3 pour illustration de la coloration syntaxique

6. Exemple de programme en *Baguette*

```
entier a;
entier b;
reel c;
entier base;
entier y;
base = 524288;

entier adresse;
//commentaire
a = 0;

tant_que (1 < 2)
  //vidage écran
  adresse = 524288;
  tant_que(adresse < 831487)
    ecrire_a (0, adresse);
    adresse = adresse + 1;
  fin_tant_que;

  retour = 1;
  tant_que(retour > 0)
    retour = lire_a(831488);
  fin_tant_que;

  tant_que (a < 640)
    c = sin(b);
    b = b + 1;
    si(b > 360)
      b = 0;
    fin_si;
    c = c + 1.;
    c = c * 200.;
    y = rve(c);
    y = y + 50;
    adresse = 640 * y;
            adresse = adresse + a;
            adresse = adresse + base;
            ecrire_a (15, adresse);
    a = a + 1;
  fin_tant_que;
  a = 0;
  ecrire_a (1, 831488);
fin_tant_que;
```

Figure 11 : Programme GPU permettant de tracer un sinus à l'écran

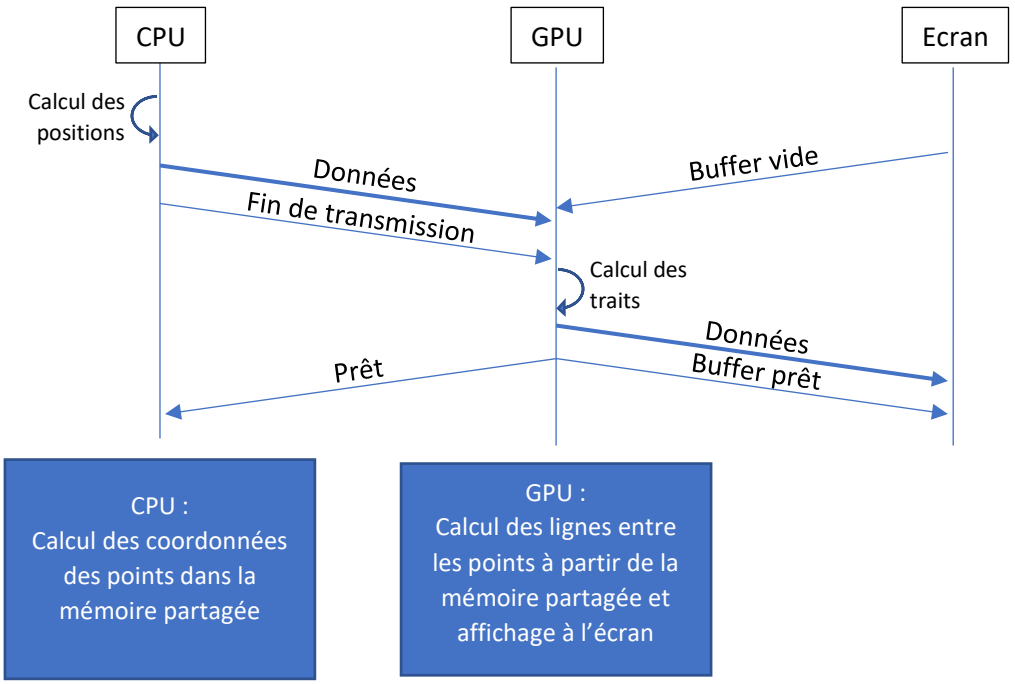
7. Programme de démonstration

La RAM est initialisée par défaut avec un programme de démonstration. Ce dernier consiste en l'affichage d'un cube tournant sur l'écran, l'affichage d'un compteur sur les afficheurs 7 segments ainsi qu'un signal sinusoïdal sur les leds de la Nexys 4. Il permet de montrer les fonctionnalités suivantes :

- Affichage sur un écran à l'aide du VGA
- Capacité de calcul avancé (Multiplication, sinus …)
- Utilisation du périphérique GPIO (Leds, 7 segments)

Sachant que les RAM du CPU et du GPU sont deux instances de la même IP, leurs valeurs initiales sont obligatoirement identiques ; le programme est le même pour le CPU et le GPU. Le programme détecte s'il est exécuté sur le CPU ou le GPU en faisant un test d'écriture/lecture sur une adresse disponible seulement sur le bus du GPU (le dernier pixel de l'écran). Si le test réussit, la partie GPU est exécutée, si le test échoue, la partie CPU est exécutée.

Le fonctionnement de la partie du programme affichant le cube est le suivant :



8. Conversion *baguette*-assembleur

Dans le tableau ci-dessous est détaillé quel code assembleur est associé à chaque instruction en baguette :

```
Addition d'entiers:
    GET variable1
    ADD variable2
    STA variable_retour
Affectation:
    GET variable
    STA variable_retour
Addition de réels:
    GET variable1
    FAD variable2
    STA variable_retour
Division d'entiers:
    GET variable1
    DIV variable2
    STA variable_retour
Division de réels:
    GET variable1
    FDI variable2
    STA variable_retour
Soustraction:
    GET variable1
    FDI variable2
    STA variable_retour
Multiplication d'entiers:
    GET variable1
    MUL variable2
    STA variable_retour
Multiplication  de réels:
    GET variable1
    FMU variable2
    STA variable_retour
Condition:
    GET variable1
    TGT variable2
    JCC addresse_fin_de_condition
Boucle:
    GET variable1
    TGT variable2
    JCC addresse_fin_de_boucle
Fin de boucle:
    JMP addresse_début_de_boucle
```

```
Afficher sur les 7 segments:
  GET variable1
  STA 80001
Ecrire dans la mémoire:
  GET variable1
  SAD variable2
Lire dans la mémoire:
  GAD variable1
  STA variable_retour
OR:
  GET variable1
  LOR variable2
  STA variable_retour
NOR:
  GET variable1
  NOR variable2
  STA variable_retour
XOR:
  GET variable1
  XOR variable2
  STA variable_retour
AND:
  GET variable1
  AND variable2
  STA variable_retour
Conversion vers entier:
  FTI variable1
  STA variable_retour
Conversion vers reel:
  ITF variable1
  STA variable_retour
Sin:
  GET variable1
  ADD index_constante_addresse_sinus
  STA var_temporaire
  GAD var_temporaire
  STA variable_retour
Cos:
  GET variable1
  ADD addresse_constante_90
  ADD index_constante_addresse_sinus
  STA var_temporaire
  GAD var_temporaire
  STA variable_retour
```

9. Annexe : Schéma du processeur