SEE09 – 2ème année

Professeur :
Christophe **JEGO**

PROCEXEIRB*

Projet expérimental de conception de circuit numérique

Mai 2019 – V1.1

David **DEVANT**

Aurélien **TROMPAT**

* Le « X » est propriété de SpaceX, le reste est à nous

SOMMAIRE

L'ensemble du projet (Sources, bag-tools et rapport) est disponible sur notre page GitHub à l'adresse suivante :

https://github.com/Devdevdavid/ProceXeirb

1 Introduction

1.1 Le contexte du projet

Ce projet a été réalisé dans le cadre d'un cours de conception de circuits numériques à l'ENSEIRB-MATMECA. Son objectif est de comprendre comment les processeurs de nos jours fonctionnent au niveau logique.

ProceXeirb est la continuité d'un projet commencé en 2018 par Pierre JOUBERT et Julien BESSE lors de leur seconde année de formation ingénieur. Leur travail avait abouti sur l'implémentation de deux processeurs nommés CPU et GPU. Tandis que le premier générait des points dans un espace 3D, le second, axé sur la partie graphique, les affichait sur un écran VGA en traçant les segments entre les points.

Leur projet était principalement orienté sur l'enrichissement d'un processeur 8-bits à jeu d'instruction limité et à l'ajout de la partie graphique. Un grand effort avait été accordé à la programmation de ces processeurs grâce au développement d'une chaine de compilation spécifique et à la création d'un langage propriétaire, le *Baguette*.

Aujourd'hui en 2019, nous, David DEVANT et Aurélien TROMPAT, choisissons de reprendre ce projet sous le nom de ProceXeirb pour le porter encore plus loin et enrichir ses fonctionnalités.

1.2 Les améliorations apportées

L'amélioration majeure de ProceXeirb réside tout d'abords dans l'ajout des appels de fonction au *Baguette*. En d'autres termes, nous souhaitons modifier la façon dont se déroule l'exécution d'un programme sur les processeurs. Nous voulons offrir plus de possibilité à l'utilisateur du *Baguette* en lui permettant de segmenter ses codes sources dans le but d'optimiser ses applications.

Cette nouvelle fonctionnalité nécessite des modifications tant au niveau matériel que logiciel. En effet, un appel de fonction signifie que le processeur va passer de son contexte courant à un nouveau contexte. Lorsque la fonction est terminée, le processeur doit être capable de revenir à son précédent contexte.

Pour garder l'historique des contextes, nous devons ajouter une pile d'appel. Nous détaillerons son implémentation matérielle et son utilisation logicielle par la suite.

1.3 Note concernant ce rapport

Étant donné que ce projet est une reprise de l'an passé, nous ne décrirons pas l'ensemble des parties dans ce rapport. Seuls les éléments qui ont été ajoutés ou modifiés seront décrit ici. Pour obtenir des informations complémentaires sur ProceXeirb, nous vous invitons à lire le rapport de fin de projet de l'an passé intitulé :

« *Conception et implémentation d'un système de rendu 3D sur FPGA* »
par Pierre JOUBERT et Julien BESSE, SEE08, Mai 2018.

2 Architecture matérielle

2.1 Vue d'ensemble

Notre architecture est basée sur l'utilisation de deux processeurs identiques (CPU et GPU). Chacun contrôle un bus de données propre, lui donnant accès à différents périphériques.

- **Les mémoires RAM de 8ko** : Elles stockent le programme et les variables de leur processeur respectif. Elles peuvent être récrites par le programmateur.
- **La mémoire partagée de 4ko** : Une mémoire double port permettant le transfert d'information entre le CPU et le GPU.
- **La CallStack** : C'est le périphérique qui permet les appels de fonctions
- **Le programmateur** : Il permet de suspendre l'exécution des deux processeurs et de réécrire le contenu de leurs mémoires RAM.
- **Le bloc GPIO** : Un bloc dédié au pilotage des éléments physiques de la carte Nexys 4 (Switch, LED, Afficheurs 7 segments).
- **Le bloc VGA** : Il offre une interface simple d'utilisation pour piloter le périphérique VGA et donc l'affichage à l'écran.
- **Les périphériques annexes** : chaque processeur dispose d'un timer permettant de générer un compteur à une fréquence configurable et un diviseur pour réaliser des opérations sur réels/entiers.

La carte Nexys 4 offre une horloge de 100MHz, nous avons donc cadencé l'ensemble de nos blocs à cette fréquence.

La figure suivante montre une vue d'ensemble de l'architecture matérielle :

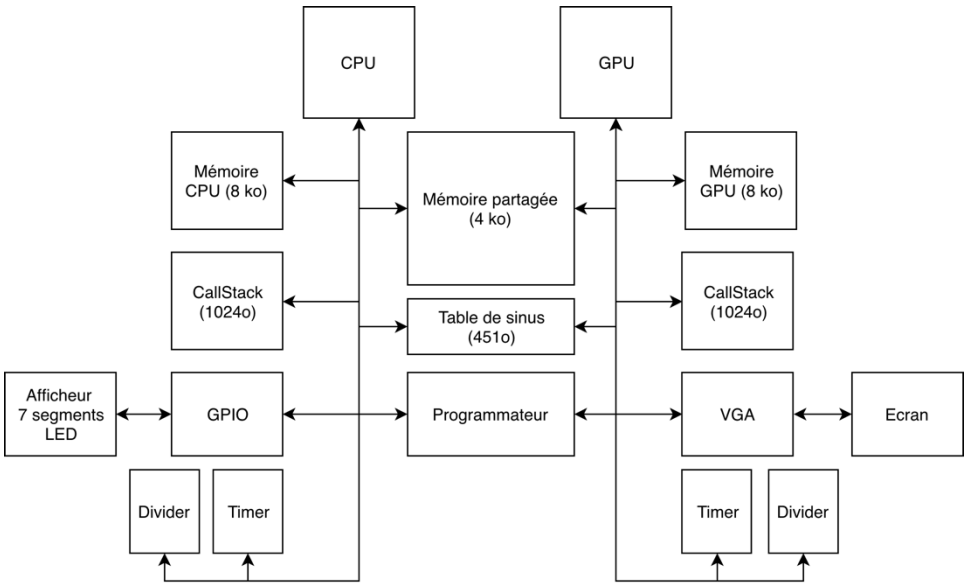

Figure 1. Vue d'ensemble de l'architecture du processeur

2.2 Organisation de la plage d'adressage

Chaque processeur dispose de son propre bus de données. De ce fait, ils ont chacun une plage d'adresse différente. Bien que la majeure partie soit identique, la différence réside dans l'accès aux périphériques dédiés du CPU et du GPU.

Les bus disposent de 20 bits pour encoder les adresses. Ceci nous offre donc une plage allant de 0x00000 à 0xFFFFF.

Les processeurs sont dits 25 bits car les instructions qu'ils utilisent sont composées de 5 bits indiquant l'identifiant de l'instruction et de 20 bits spécifiant une adresse sur le bus de données.

Les zones grisées sur la figure ci-contre indique que la plage n'est pas allouée. Une écriture dans ces plages n'aura aucun effet et une lecture retournera toujours la valeur 0.

Les périphériques et registres ayant été ajoutés ou modifiés par rapport à 2018 sont détaillés dans les parties suivantes.

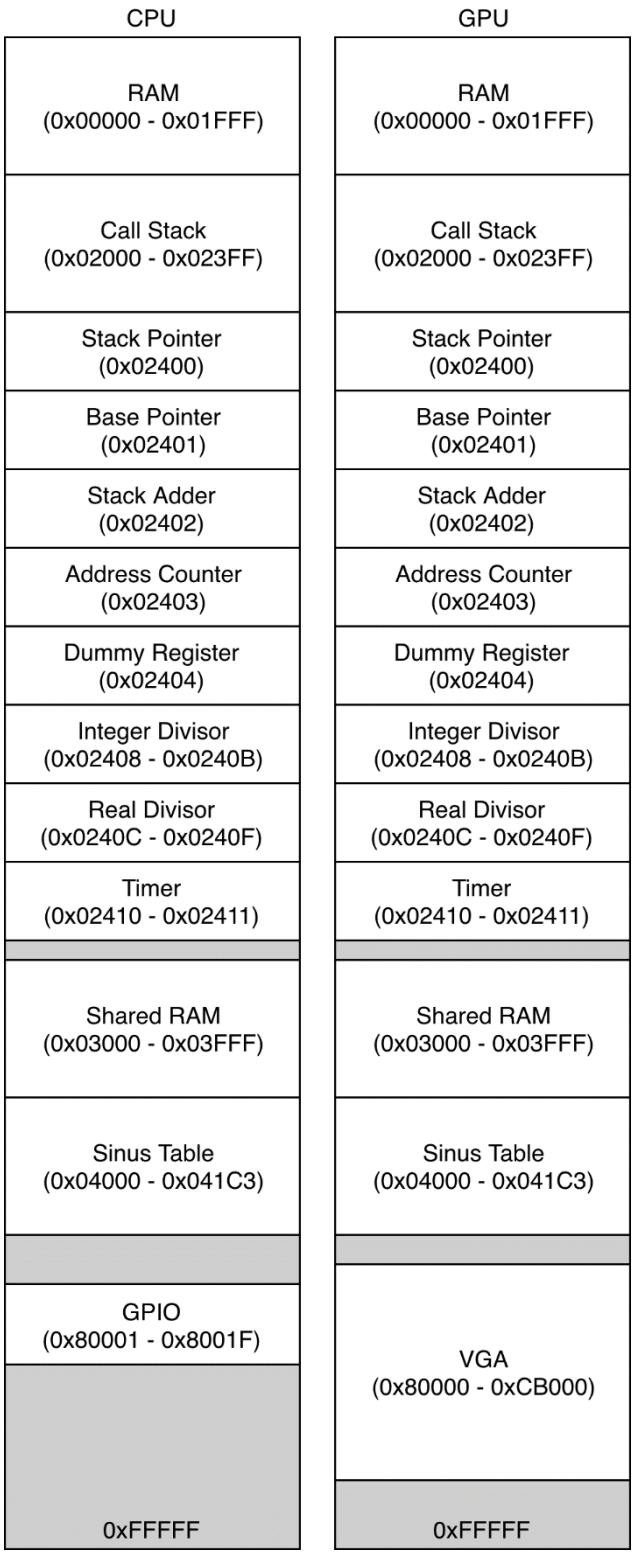

Figure 2. Plages d'adresse des bus de données des CPU et GPU

2.3 Détails sur les périphériques

2.3.1 La mémoire RAM

C'est un périphérique de 8192 espaces mémoire de 25 bits qui est complètement écrasé au moment de la programmation. Chaque espace permet de stocker au choix, une instruction processeur ou bien une variable globale.

Après un reset, le compteur d'instruction démarre à l'adresse 0x00000 et déroule le programme en fonction des instructions.

2.3.2 Le système de Call Stack

Le système de Call Stack a été ajouté pour instaurer les appels de fonction. Il est notamment constitué d'une pile d'appel, de deux registres et d'un additionneur. Voyons en détails chacun d'entre eux.

2.3.2.1 Pointeur de pile et pointeur de base

Ces deux pointeurs sont en réalité de simples registres capables de mémoriser une valeur sur 25 bits. Ils permettant de naviguer dans la CallStack.

Le pointeur de pile est appelé ESP (*Extended Stack Pointer*) tandis que le pointeur de base est nommé EBP (*Extended Base Pointer*). Nous avons conservé les mêmes notations que celles utilisées par les concepteurs de microcontrôleur afin de faciliter l'analogie avec des modèles célèbres tels que les ATMEGA ou STM32.

A l'initialisation, l'ESP et l'EBP pointent tout deux sur le haut de la pile d'appel (Adresse les plus élevée). L'ESP peut être incrémenté ou décrémenté grâce aux instructions PUSH et POP pour pouvoir se déplacer dans la pile. A contrario, la valeur de EBP est directement écrite par le processeur en passant par le bus.

2.3.2.2 La pile d'appel

C'est une pile disposant de 1024 espaces mémoire. Bien qu'aucune protection ne soit active, le processeur n'écrit pas directement dans cette pile, il passe par l'intermédiaire du pointeur de pile (ESP) et du pointeur de base (EBP). Pour écrire dans la pile, le processeur écrit à l'adresse indiquée par ESP.

ESP joue donc le rôle de pointeur. Pour l'exploiter, nous utilisons les instructions GAD et SAD qui permettent non pas de lire/écrire à l'adresse passée en paramètre mais de d'agir sur l'espace mémoire indiqué à l'adresse passée en paramètre.

Le schéma suivant reprend le principe de pointeur des instructions GAD et SAD :

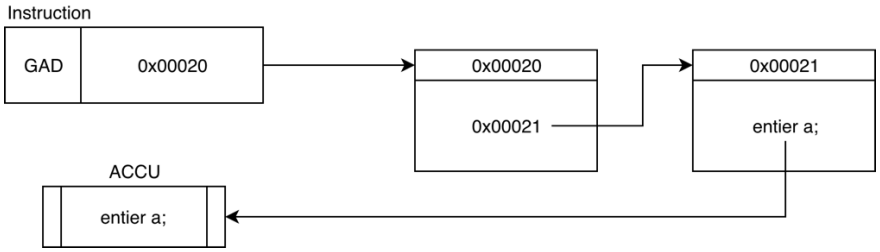

Figure 3. Principe des instructions SAD et GAD

2.3.2.3 L'additionneur de pile

Le *Stack Adder* est un additionneur entre le pointeur de base (EBP) et un registre nommé *OffsetParam* accessible par le processeur grâce à l'instruction CSA. Ce système permet de calculer l'adresse d'une variable locale stockée en pile. Nous détaillerons son utilité dans la partie logicielle de ce rapport.

Voici le schéma de l'additionneur de pile :

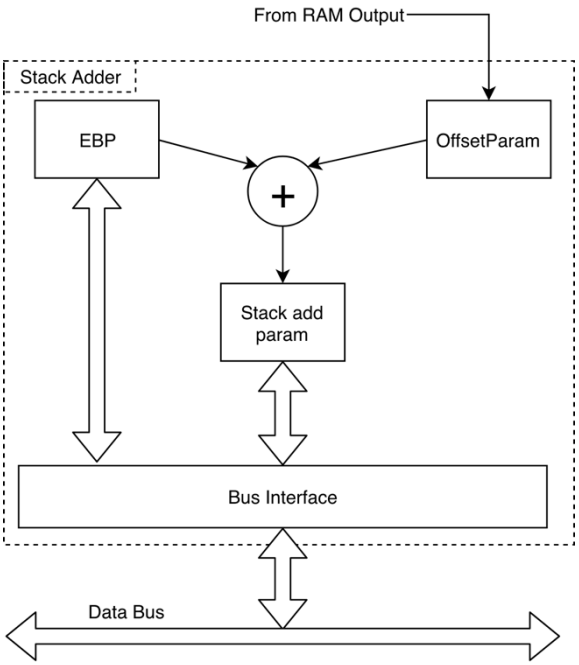

Figure 4. Additionneur de la CallStack pour calculer un pointeur de pile

Ce schéma est encadré en pointillé car il fait partie intégrante du CPU (Partie UT), il n'est pas un périphérique dédié au même titre que la CallStack ou les timers. Vous le retrouverez dans le schéma complet du CPU situé annexe de ce document.

2.3.2.4 Le compteur d'adresse

Le compteur d'adresse EIP (*Extended Instruction Pointer*) est un pointeur sur la mémoire RAM qui indique la prochaine instruction que le processeur doit exécuter. Un appel de fonction va agir sur ce compteur grâce à une instruction de saut : JMP.

Toutefois, lorsque la fonction est terminée, il faut être capable de revenir à la position de ce compteur avant le saut. Pour cela, nous rendons accessible en lecture le compteur d'adresse de façon à pouvoir sauvegarder sa valeur dans la pile.

Nous avons aussi autorisé l'écriture de la valeur de ce registre car, bien qu'une instruction de saut soit déjà présente dans le jeu d'instruction, elle ne nous permet pas de sauter à une adresse calculée. Nous avons besoin de cette permission pour sortir d'une fonction et revenir à l'endroit où nous y sommes entrés.

2.3.3 Diviseurs d'entiers et réels

Dans le projet de 2018, nos prédécesseurs avaient rencontré des difficultés sur l'implémentation de la division dans l'UAL. En effet, il était nécessaire de réaliser l'opération en un seul coup d'horloge (10 ns). Dans le cas des entiers, ils avaient choisi de tronquer les opérandes pour diminuer le chemin critique. La division sur réels avait, elle, été complètement abandonnée car jugée trop complexe pour fonctionner à 100MHz.

Dans cette nouvelle version, nous avons choisi d'externaliser les divisions en créant un périphérique dédié. Nous avons donc supprimé les instructions DIV et FDI et modifié le compilateur de manière adéquate. Voici le schéma de notre « Diviseur Optimisé » :

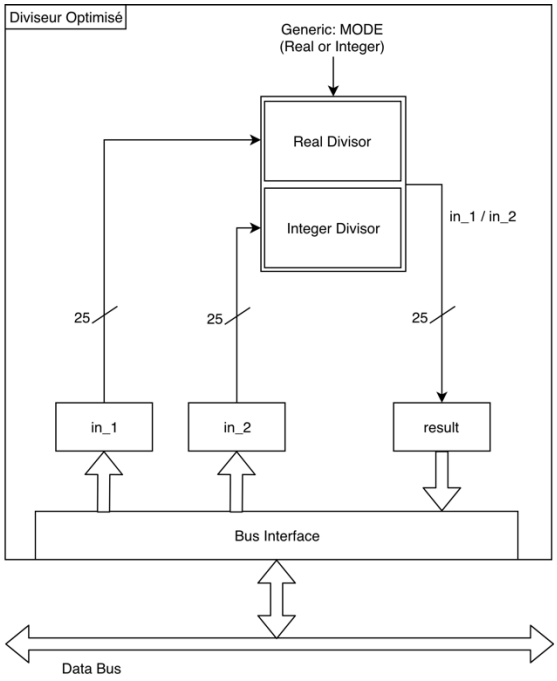

Figure 5. Diviseur externalisé pour entiers et réels

Nous avons 3 registres : `result` contient le résultat de l'opération `in_1 / in_2`.

Pour simplifier le développement, nous avons créé un seul bloc pour les deux divisions. Un GERNERIC est utilisé pour indiquer le type des valeurs d'entrée (Réel ou entier) et ainsi permettre le choix de l'algorithme à utiliser pour l'implémentation.

Le fait que nous laissions Vivado s'occuper de l'implémentation de la division nous apporte des problèmes de dépassement de Slack. En effet, l'outil essais de générer une division qui puisse fonctionner à 100MHz du fait que ses entrées peuvent potentiellement évoluer à chaque front d'horloge (car connectées au bus de données).

Or, étant donné la façon dont nous l'avons implémentée dans notre projet, nous devons changer les entrées du diviseur en exécutant une instruction du processeur. Ceci implique des délais de 6 coups d'horloge.

En d'autres termes, le délais laissé au diviseur entre le moment de l'écriture de son entrée N°2 et le moment de la lecture de sa sortie, suffit à obtenir un calcul terminé. (Voir 6.1 Pourquoi le « Slack » est-il négatif ?)

2.3.4 Table de sinus

Cette table contient les valeurs de sortie d'un sinus pour un θ allant de $0°$ à $450°$. Chaque valeur est codée en virgule fixe allant de 0x0000100 (+1.0) à 0x1FFFF00 (-1.0).

Cette table nous sert aussi de référence pour la fonction cosinus. La seule différence est l'ajout de $90°$ à θ avant la lecture de la table de sinus. C'est pourquoi nous l'avons prolongée de $360°$ à $450°$. De cette façons, la même table est utilisée pour les sinus et cosinus.

2.3.5 Les timers

Les deux processeurs sont équipés d'un timer chacun. Ce bloc permet d'obtenir un compteur qui s'incrémente à une période configurable. L'horloge de 100MHz est divisée par 100 pour obtenir une période 1 us. Ensuite, un second diviseur d'horloge compte les fronts montants de ce signal. Il divise l'horloge par la valeur du registre *period*.

Finalement, un dernier bloc compte les fronts d'horloge de ce dernier signal et incrémente la valeur du registre *cptValue*.

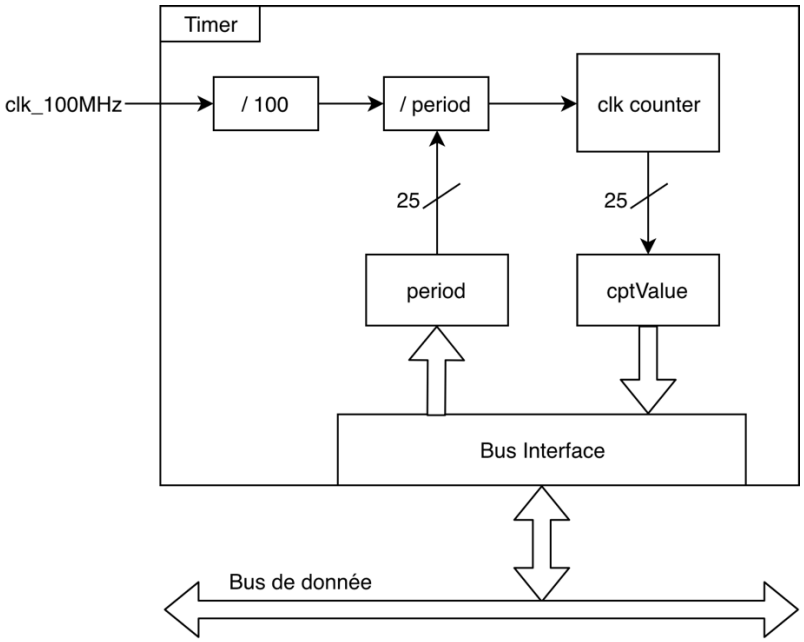

Figure 6. Timer configurable pour générer un compteur incrémenté périodiquement

Le timer peut être utilisé dans les programmes écrits en *Baguette* pour réaliser des attentes ou synchroniser des actions dans le temps. La période maximum de ce timer est de 2^{25} microsecondes soit 33.5 secondes.

2.4 Les instructions du processeur

Le tableau suivant répertorie toutes les instructions connues par les processeurs :

Identifiant	OP Code	Description
Opérateurs logiques		
OP_NOR	0x00	NON-OU Logique
OP_LOR	0x01	OU Logique
OP_AND	0x02	ET Logique
OP_XOR	0x03	OU-Exclusif Logique
Opérateurs sur entiers		
OP_ADD	0x04	Addition d'un entier
OP_SUB	0x05	Soustraction d'un entier
OP_MUL	0x06	Multiplication d'un entier
Opérateurs sur réels		
OP_FAD	0x07	Addition d'un réel
OP_FSU	0x08	Soustraction d'un réel
OP_FMU	0x09	Multiplication d'un réel
Conversion		
OP_FTI	0x0A	Réel vers entier
OP_ITF	0x0B	Entier vers réel
Outils		
OP_PSH	0x0C	Push : Décrémente ESP
OP_POP	0x0D	Pop : Incrémente ESP
OP_STA	0x0E	Sauvegarde de l'ACCU
OP_JCC	0x0F	Saut d'adresse conditionnel
OP_JMP	0x10	Saut d'adresse
OP_GET	0x11	Chargement dans l'ACCU
Tests		
OP_TGT	0x12	Comparaison supérieur ou égal
OP_TLT	0x13	Comparaison inférieur ou égal
OP_TEQ	0x14	Comparaison égal
OP_TNE	0x15	Comparaison différent
Management de mémoire		
OP_CSA	0x16	Calcul d'une adresse dynamique de la pile
OP_GAD	0x17	Chargement de la valeur pointée dans l'ACCU
OP_SAD	0x18	Sauvegarde de l'ACCU à l'adresse pointée
0x18-0x1F : Instructions non utilisées		

Figure 7. Tableau des instructions connues par le processeur

Toutes ces opérations ne sont pas traitées de la même façon par le processeur. Nous avons donc ajouté en annexe le schéma de la machine d'état qui cadence les signaux de commandes.

2.5 Utilisation des ressources du FPGA

A cause des difficultés rencontrées avec les Diviseurs Optimisés, nous avons choisi d'étudier les ressources du FPGA pour mieux comprendre les problèmes de Slack. Les graphiques suivants représentent le nombre de LUT, de DSP et la valeur du WNS pour 4 implémentations différentes des diviseurs.

Dans chacune des expériences, nous prenons en compte les diviseurs des deux processeurs. Ceci implique que les variations sont multipliées par 2 (Un diviseur par processeur).

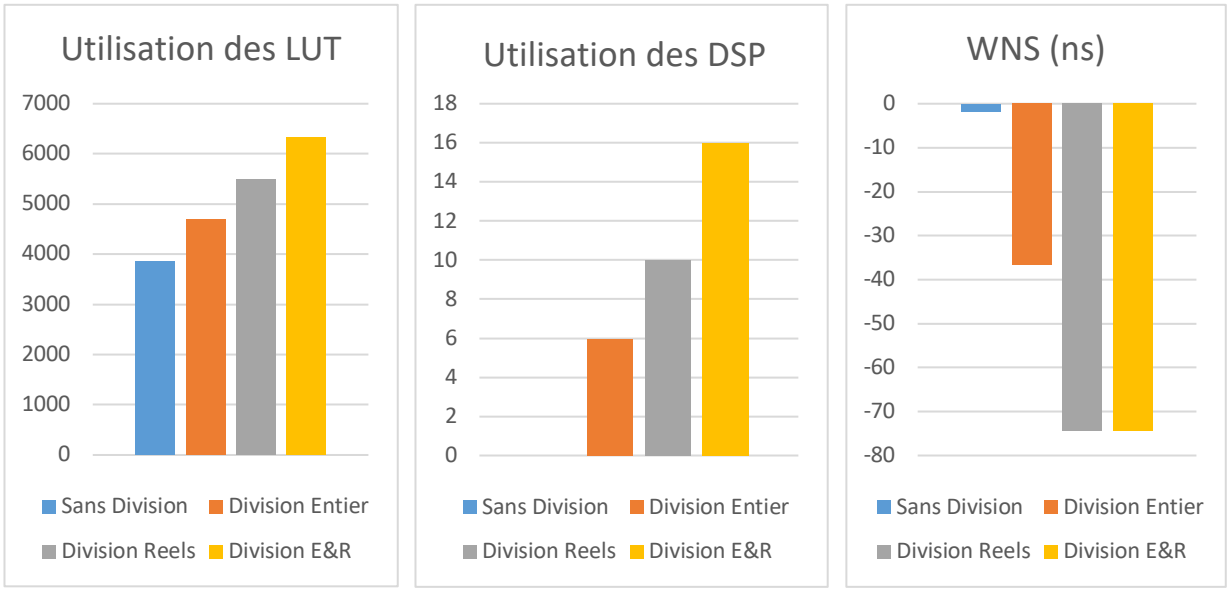

Figure 8. Utilisation des ressources du FPGA par rapport aux diviseurs

Le premier graphique nous montre que l'implémentation des deux diviseurs d'entiers apportent 842 LUT alors que dans le cas des réels à virgule fixe, nous constatons une augmentation de 1635 LUT, soit 42% de LUT supplémentaires. Dans le dernier cas, l'ajout des deux types de diviseurs revient à multiplier les ressources en LUT par 1.60.

Dans notre projet, les DSP sont seulement utilisés dans le Diviseur Optimisé, c'est pourquoi nous obtenons une valeur nulle pour la première implémentation. Nous remarquons que le diviseur réel a besoin de 4 DSP supplémentaire par rapport au diviseur d'entier. L'association des deux résulte simplement en la somme.

Enfin, le WNS qui était inchangé depuis 2018 jusqu'à la venu des diviseurs, tombe entre -35ns et -40ns lorsque l'on utilise les diviseurs entier. C'est en cela que nous pouvons dire qu'il est difficile d'effectuer des divisions à 100MHz. Le WNS chute dramatiquement à -70ns lorsque l'on passe au diviseur de réels. Ceci s'explique par la grande complexité des nombres à virgule fixe qui nécessitent une conversion avant et après la division. Ceci fait alors apparaitre de la séquentialité. Principe que nous essayons souvent d'éviter en conception numérique.

Pour limiter cela, nous avons fait le choix d'implémenter la division d'entier sur 18 bits, au lieu de 25. De cette façon, elle ne dépasse pas les -50ns de Slack et nous permet ainsi de

l'utiliser entre deux instructions processeurs de 60ns minimum. Pour la division de réel, nous avons décidé de retarder la lecture du résultat par notre processeur en ajoutant une instruction sans effet. De cette façon, nous garantissons que le résultat obtenu est le bon.

3 Le langage *Baguette*

Le langage *Baguette* est un langage propriétaire spécialement conçu pour les processeurs de ProceXeirb. En 2018, Il disposait de toute la syntaxe essentiel à un langage de programmation. On pouvait alors déclarer des variables, effectuer des affectations, des calculs et appeler des procédures standards telles que `cos()` ou `afficher_LCD()`.

Voyons ensemble les différentes améliorations qui ont été apportées en 2019.

3.1 Les améliorations

Bien que la plupart soient mineures, voici une liste de l'ensemble des améliorations :

- Ajout de l'initialisation des variables globales au moment de la déclaration

```
entier a = 10;
```

- Ajout du « sinon » aux conditions

```
si (a == b)
    r = 1;
sinon
    r = 0;
fin_si;
```

- Ajout des tableaux

```
entier tab[10];
tab[0] = 2;
a = tab[9];
```

- Ajout de l'opérateur d'obtention d'adresse « @ »

```
entier pTab = @tab;
ecrire_a(3, pTab);
pTab = pTab + 1;
lire_a(pTab);
```

- Ajout de la commande de préprocesseur #définition

```
#definition BASE_LED_ADDR (524304)
ecrire_a(255, BASE_LED_ADDR);
```

- Ajout de l'opérateur logique de comparaison « …différent de… »

```
si (a != b)
    r = 1;
fin_si;
```

- Ajout des commentaires « inline »

```
entier result; // This is a result
```

- Enrichissement de la coloration syntaxique de Sublime Text 3

Toutefois, le sujet de ce projet est bel et bien l'ajout des appels de fonction. Commençons donc par la notion de contexte.

3.2 La notion de contexte

Un contexte, au sens de la programmation, est un environnement dans lequel le processeur effectue ses opérations. Il inclut tous les éléments propres à cet environnement. On retrouve alors l'ensemble des registres qui servent à l'exécution du programme tels que le compteur d'adresse (EIP) et les registres de base et de pile (EBP et ESP).

Dans ce contexte, il est possible de déclarer et d'utiliser des variables. Celles-ci sont dîtes locales au contexte. Une variable locale n'est pas accessible depuis un autre contexte.

Un contexte ne peut en contenir un autre mise à part le contexte global. C'est le premier déclaré. Ses variables sont dites globales et sont accessibles dans tous les contextes fils.

Chaque contexte contient des instructions qui permettent de modifier les variables, d'accéder à des périphériques ou de changer de contexte.

Les flèches sur l'exemple suivant indiquent les cas d'utilisation des variables au seins des différents contextes :

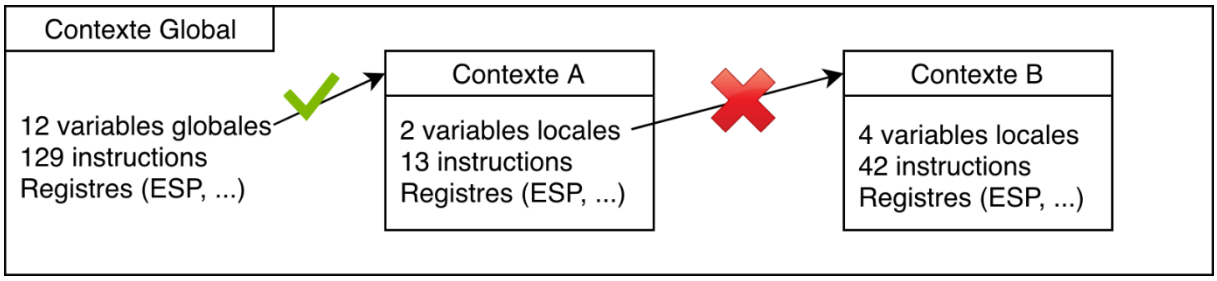

Figure 9. Permissions d'accès entre contexte

Cette notion de contexte a été implémentée en *Baguette* grâce aux fonctions, voyons comment les déclarer.

3.3 La déclaration de fonction

La figure suivante montre un exemple de déclaration de fonction en *Baguette* :

```
1   // Variable globales
2   entier retVal;
3   entier param1 = 42;
4   entier offset = 255;
5
6   // Cette fonction ajoute un offset à 'a'
7   fonction addOffset(entier a);
8       // Variable locale
9       entier result;
10
11      result = a + offset;
12  retourne result;
13
14  retVal = addOffset(param1);
```

Figure 10. Exemple de déclaration de fonction en Baguette

Le contexte global étant déclaré par défaut, les premières lignes de code lui sont donc propres. Ainsi, les trois entiers `retVal`, `param1` et `offset` sont déclarés comme variables globales.

Ensuite, nous déclarons une nouvelle fonction (un nouveau contexte) dont `addOffset` est le nom et les déclarations qui suivent sont ses arguments. Les arguments sont considérés comme des variables locales à la fonction. Seules leur valeurs d'initialisation sont issues du contexte d'où la fonction sera appelée.

Une seconde variable locale est explicitement déclarée pour stocker le résultat. L'opération qui suit est une addition entre une variable locale et une variable globale. Le tout est enregistré dans une variable locale pour être retourné en sortie de la fonction. Nous verrons dans la section dédiée au compilateur que la notion de globale/locale a une réelle importance sur le code assembleur.

La dernière ligne de l'exemple ci-dessus est l'appel de la fonction. L'argument est renseigné, de même que le nom de la variable qui accueillera la valeur de retour. Lors de la compilation, une vérification est effectuée sur la compatibilité des variables grâce au prototype de la fonction.

4 La chaine de compilation

La chaine de compilation est composée de 3 outils qui ont été renommés dans cette version 2019 :

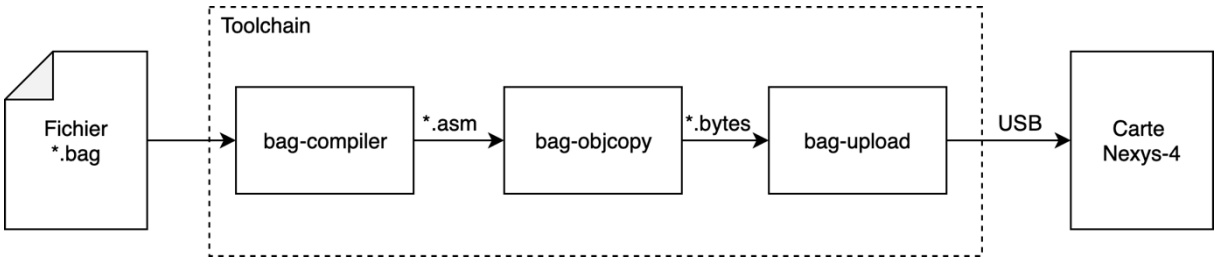

Figure 11. Parcours du fichier Baguette au sein des bag-tools

Cette Toolchain permet de compiler un programme écrit en *Baguette* en un fichier binaire qui est envoyé sur le port série de la carte Nexys 4 afin de la reprogrammer.

4.1 Les améliorations

Voici une liste des améliorations qui ont été apportées en 2019 sur les 3 outils :

- Ajout d'option de lancement pour les 3 outils dans le but d'une utilisation facilité
- Ajout de Makefile pour compiler les outils
- Ajout de Makefile pour lancer la chaine de compilation
- Création d'une bibliothèque commune à tous les outils pour diminuer les efforts de maintenance
- Ajout d'une option de compilation « Debug » pour une meilleure lisibilité du code assembleur généré.
- Ajout de la possibilité de générer un fichier « .coe » pour pouvoir insérer un programme par défaut dans le projet Vivado
- Ajout d'un précompilateur
- Ajout d'un système de LOG avec niveau (Info, Warning, Error)
- Mise en place du système de contexte
- Ajout d'une initialisation matériel en début de contexte global (Init. Afficheur 7 segments, ESP, EBP, …)

4.2 Le compilateur

Le compilateur a été presque entièrement revu et optimisé. Il est désormais plus précis sur les messages concernant les erreurs de syntaxes grâce au nouveau système de LOG. Il est aussi plus facile à modifier depuis que des dizaines de lignes de code ont été factorisées.

L'ajout du précompilateur permet désormais d'effectuer une première passe sur le fichier afin de le mettre en forme pour le compilateur. Les commentaires, les espaces et les saut de lignes inutiles sont supprimés.

Toutefois, la plus grosse mise à jour concerne la mise en place des contextes et la déclarations des fonctions.

4.2.1 Appel d'une fonction

Voici les différentes étapes nécessaires à l'appel d'une fonction :

1) Les variables passées en arguments sont copiées vers la CallStack (PUSH)
2) L'adresse de retour est calculée en fonction de la valeur du compteur d'adresse (EIP) puis stockée dans la CallStack.
3) Une instruction de saut vers l'adresse de la première instruction de la fonction est exécutée.

Voici alors l'état de la CallStack à ce moment précis de l'appel :

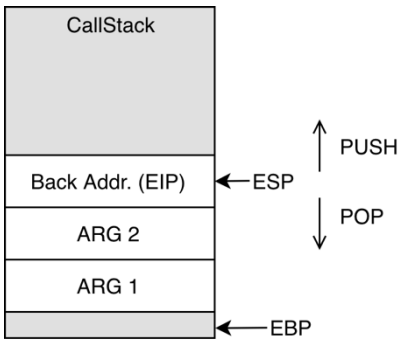

Figure 12. Une CallStack après un appel de fonction

ESP pointe sur le haut de la pile et EBP n'a pas encore été défini.

*** Continuer la lecture de 4.2.2 à 4.2.4 puis revenir ici ***

4) Lorsque la fonction est terminée, la CallStack est revenue à son état antérieur (avant le saut)
5) Les paramètres précédemment ajoutés sont désormais supprimés de la CallStack avec des instructions POP
6) Si nous sommes intéressés par la valeur de retour, nous la copions de DUMMY vers la variable spécifiée

4.2.2 Début d'une fonction

Lors de l'exécution d'une fonction, les instructions définies par le code en *Baguette* ne sont pas immédiatement exécutées. Des étapes préliminaires sont nécessaires pour sauvegarder le contexte :

1) Sauvegarde de l'ancienne valeur de EBP dans la CallStack
2) Sauvegarde de la valeur de l'ESP dans EBP (Les deux pointent au même endroit)
3) Allocation de la mémoire nécessaire aux variables locales dans la pile (Succession de PUSH)
4) Copie des arguments de la CallStack vers les variables locales (Elles-mêmes dans la CallStack)

Voici l'état de la CallStack lorsque l'on s'apprête à commencer l'exécution du corps de la fonction :

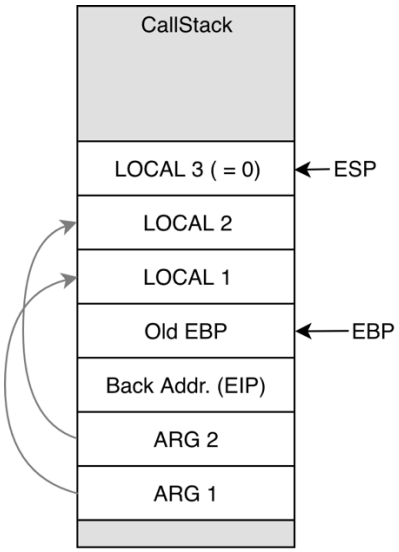

Figure 13. Une CallStack après un début de fonction

Les arguments 1 et 2 sont copiés dans l'espace alloué aux variables locales. Nous pourrions ici optimiser l'espace de la pile car comme nous pouvons le voir, les arguments sont stockés en double dans la CallStack. Nous n'avons pas réalisé cette modification car le fait de changer le comportement de certaines variables locales est plus difficile qu'il n'y paraît.

Nous avons dimensionné la taille de la CallStack à 1024 étages de sorte à être capable d'appeler 10 fonctions les unes dans les autres. Nous nous sommes basés sur un maximum de 100 variables locales par fonction (Pouvant être des tableaux ou les arguments de la fonction). Cela nous permet d'avoir une marge confortable avant d'arriver un dépassement de pile (qui par ailleurs n'est pas contrôlé).

4.2.3 Corps de la fonction

Le corps de la fonction contient les instructions décrites par l'utilisateur en *Baguette*. Ces instructions n'ont pas le même comportement en fonction du type de la variable à utiliser. Dans l'exemple donné en partie 0, nous additionnons une variable globale avec une variable locale.

Dans le premier cas, le chargement est simple car l'adresse des variables globales est connue au moment de la compilation. On peut donc utiliser l'instruction assembleur GET.

Dans le second cas, nous devons utiliser la valeur de la variable stockée dans la CallStack. Or, par définition, la pile d'appel n'est composée que d'éléments temporaires qui dépendent de l'exécution du programme. L'adresse de la variable locales est donc inconnue à l'avance.

Toutefois, ce qui est connu est la différence d'adresse entre le pointeur EBP et l'emplacement de la variable locale dans la pile. Comme le montre la figure de la partie 4.2.2,

Les variables locales sont placées au-dessus de EBP dans l'ordre de déclaration. Pour y accéder, il nous faut calculer leurs adresses dynamiques par rapport au pointeur de Base.

Nous avons développé un additionneur spécifique pour ce calcul.

4.2.3.1 Additionneur d'adresse de la CallStack

Ce bloc permet d'additionner la valeur de EBP avec un registre nommé `OffsetParam`. La valeur de ce registre peut être écrite avec l'instruction assembleur CSA. Suite à cela, la valeur de sortie de l'additionneur est mise à jour et nous pouvons l'utiliser pour accéder à notre variable locale grâce à un GAD. Voici le code assembleur d'une addition entre une variable locale et une globale :

ASM DEBUG

```
CSA addr:(.global::-1)
GAD DYN_ADDI_ADDR
ADD addr:(.global::42)
```

La première ligne indique que l'on écrit « -1 » dans le registre `OffsetParam`. Cette constante est stockée dans le contexte global. L'offset est négatif car dans la CallStack, le remplissage commence dans les adresses de hautes valeurs et avances vers les valeurs basse. Ainsi, EBP aura toujours une adresse supérieure à celle des variables locales.

La seconde ligne charge dans l'ACCU la valeur pointée par le registre de sortie de l'additionneur. Enfin, la dernière permet d'ajouter 42 (constante placée dans le contexte global) à la valeur de l'ACCU.

Toutefois, avec cette méthode, nous ne pouvons pas faire d'opération entre deux variables locales car les instructions assembleur ne peuvent prendre leur variable que depuis la mémoire RAM (il n'est pas possible de calculer une adresse puis d'utiliser sa valeur avec les ADD, SUB, MUL…). La solution que nous avons trouvée consiste à charger la seconde variable en RAM puis placer la première dans l'ACCU pour enfin exécuter l'opération depuis la RAM. Voici le code assembleur d'une soustraction de deux variables locales :

ASM DEBUG

```
// Charge la var locale 2
CSA addr:(.global::-2)
GAD DYN_ADDI_ADDR
// Save en RAM dans DUMMY
STA DUMMY_ADDR
// Charge la var locale 1
CSA addr:(.global::-1)
GAD DYN_ADDI_ADDR
// Effectue l'opération avec DUMMY
SUB DUMMY_ADDR
```

Ceci nous montre que l'utilisation de variables locales est bien moins performante que de simples variables globales. Ici nous effectuons une opération en 6 instructions alors que nous pourrions effectuer le même calcul en seulement 2 instructions en utilisant le contexte global.

4.2.3.2 Appel de fonction successifs

Bien que les appels de fonction récursif ne soient pas supportés (Le risque de dépassement de pile est trop important), il est toujours possible d'appeler une fonction depuis une autre fonction.

En effet, à partir du moment où l'appel d'une fonction puis son retour ne modifie pas l'état de la pile d'appel (Elle revient à son état précédent), il est possible d'appeler successivement plusieurs fonctions. Voici l'état de la pile après l'appel d'une fonction à un seul argument :

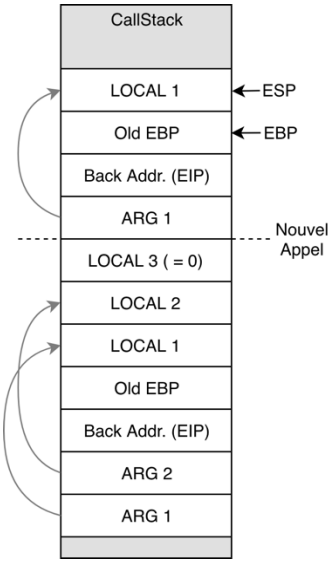

Figure 14. Une CallStack après deux appels successifs

4.2.4 Fin d'une fonction

Enfin, lorsque le corps de la fonction est terminé, quelques instructions assembleur sont ajoutées à la fin pour restaurer le contexte précédent.

1) Sauvegarde de la valeur de retour en RAM dans DUMMY
2) Restauration de la valeur de ESP grâce à EBP (On supprime l'ensemble des variables locales allouées)

A cet instant, voici l'état de la pile :

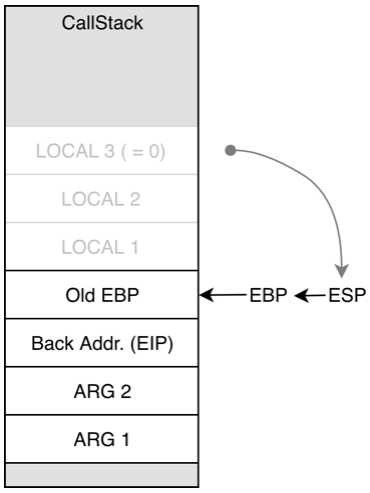

Figure 15. Une CallStack avant le retour au contexte précédent

3) Restauration de la valeur de EBP grâce à la sauvegarde qui avait été faite dans la pile
4) Suppression de cette valeur de la pile (POP)
5) Chargement de l'adresse de retour depuis la pile vers l'ACCU
6) Suppression de cette valeur de la pile (POP)
7) Stockage de la valeur de l'ACCU dans le compteur d'adresse pour effectuer un saut dynamique et revenir à l'ancien contexte

La pile est désormais identique au schéma de la partie 4.2.2.

Maintenant que nous sommes capables de compiler des fichiers *Baguette*, voyons comment les convertir en fichier binaire.

4.3 Le générateur de binaire

bag-objcopy permet de convertir un fichier assembleur en fichier binaire. Il est possible de sélectionner le format de sortie (.bytes ou .coe) grâce au Makefile. Cet outil effectue des opérations de contrôle pour détecter les instructions inconnues et les dépassement de mémoire.

Dans cette nouvelle version 2019, les fichiers .bytes sont désormais écrit en binaire et non pas en ASCII, ceci permet à l'uploader d'être plus efficace lors de la lecture du fichier.

La génération de fichier .coe est utilisée pour programmer la carte Nexys 4 de façon permanente (Même après un reboot). Ce fichier décrit le contenu de la mémoire RAM des processeurs.

4.4 L'uploader de binaire

Pour programmer les processeurs, il est nécessaire d'appuyer sur les boutons poussoirs de la carte pour sélectionner le ou les processeurs à programmer (Voir la disposition des boutons ci-contre).

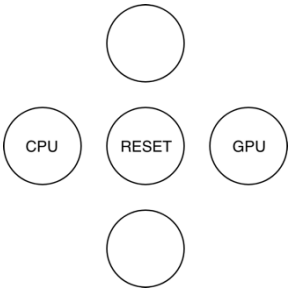

Figure 16. Représentation des boutons de programmation

Ensuite, le lancement de l'uploader permet d'envoyer le binaire par le port série de la Nexys 4. Le programmateur situé sur la carte FPGA se charge alors d'écrire chaque mot de 25 bits dans les mémoires RAM des processeurs.

5 Le programme de démonstration

L'an passé, le programme de démonstration permettait d'afficher un cube en 3D à l'écran. Tandis que les positions des sommets étaient calculées par le CPU, le GPU s'occupait de la génération des images avec le calcul des segments entre les points.

Mise à part quelques changements sur le format des variables (utilisation de tableau), nous n'avons pas revu l'algorithme. Ceci nous montre que le compilateur est totalement compatible avec un code source *Baguette* de 2018. Nous nous étions imposé cette contrainte lors de nos premières réflexions.

Pour montrer l'utilisation de la CallStack, nous avons implémenté un nouvel affichage graphique basé sur la même topographie : le CPU génère des points dans l'espace qui représente une caméra qui avance dans un tunnel et le GPU trace les segments entre ces points.

Ici nous simulons un monde 3D avec des structures multidimensionnelles et une caméra mobile. Pour afficher l'image en 2D, nous utilisons une projection dite « faible » basée sur le théorème de Thales.

6 Notes sur l'état actuel du projet

Dans le but de faciliter la reprise du projet, voici quelques notes sur l'état actuel du projet.

6.1 Pourquoi le « Slack » est-il négatif ?

Lors de la synthèse puis de l'implémentation, Vivado utilise les délais de propagation des blocs qui composent le design pour déterminer si la contrainte en fréquence est validée. Sur chacun de ces délais, Vivado ajoute une légère marge qui lui permet de garantir ses résultats. Ainsi, lorsque le rapport des timings affiche un Slack négatif de 3.7 ns, nous pouvons considérer que nous ne rencontrerons pas de problèmes dans l'utilisation de notre système.

Néanmoins, notre projet indique un Slack négatif variant entre 100ns et 200ns : ceci est contrôlé. En effet, nous avons quelques difficultés avec l'utilisation des diviseurs de réels. Nous savons pertinemment que notre diviseur ne peut pas fonctionner à 100MHz. C'est pourquoi nous l'avons retiré de l'UAL et placé dans un périphérique dédié. De cette façon, nous pensions pouvoir donner tout le temps nécessaire aux opérations pour s'exécuter.

Malgré cela, nous ne sommes pas parvenus à indiquer à Vivado que les contraintes en fréquence n'étaient pas les mêmes dans ce bloc. Comme indiqué dans la partie 2.3.3, ce problème n'impacte pas nos performances.

Toutefois, il faut être très attentif au fait que la valeur indiquée ici est le WNS (*Worst Negative Slack*). En d'autres termes, il est possible que d'autres chemins critiques soient présents mais que le temps de propagation soit inférieur au WNS.

« Un *Slack* peut en cacher un autre »

6.2 Les éléments obsolètes du rapport de 2018

Ce document n'explique pas en détails les éléments du projet qui n'ont pas évolués entre 2018 et 2019. Nous vous invitons donc à lire le rapport de l'an dernier. La liste ci-dessous indique quels éléments sont devenus obsolètes suite à nos modifications :

- Les schémas d'architectures sont incomplets
- L'organisation de l'espace mémoire
- La liste des instructions
- Les divisions ne sont plus faites par l'UAL
- Le schéma séquentiel de la FSM est incomplet

CONCLUSION

Ce projet est l'image même de la formation SEE. Il rassemble nos connaissances en électronique, conception numérique sur FPGA et développement d'application en C/C++. De plus, le fait de reprendre le projet d'anciens SEE nous a permis d'aller au-delà de nos attentes personnelles en repoussant les limites de notre domaine de connaissance.

Nous remercions Christophe JEGO et Guillaume FERRE pour nous avoir permis de participer à la Soirée Partenaire de l'ENSEIRB-MATMECA de 2019 pour présenter ce projet. Nous sommes ravis d'avoir suscité l'intérêt de notre public pour la formation SEE et nous espérons que plus de projets comme celui-ci, seront rendus possibles pour les futures promotions.

ANNEXES

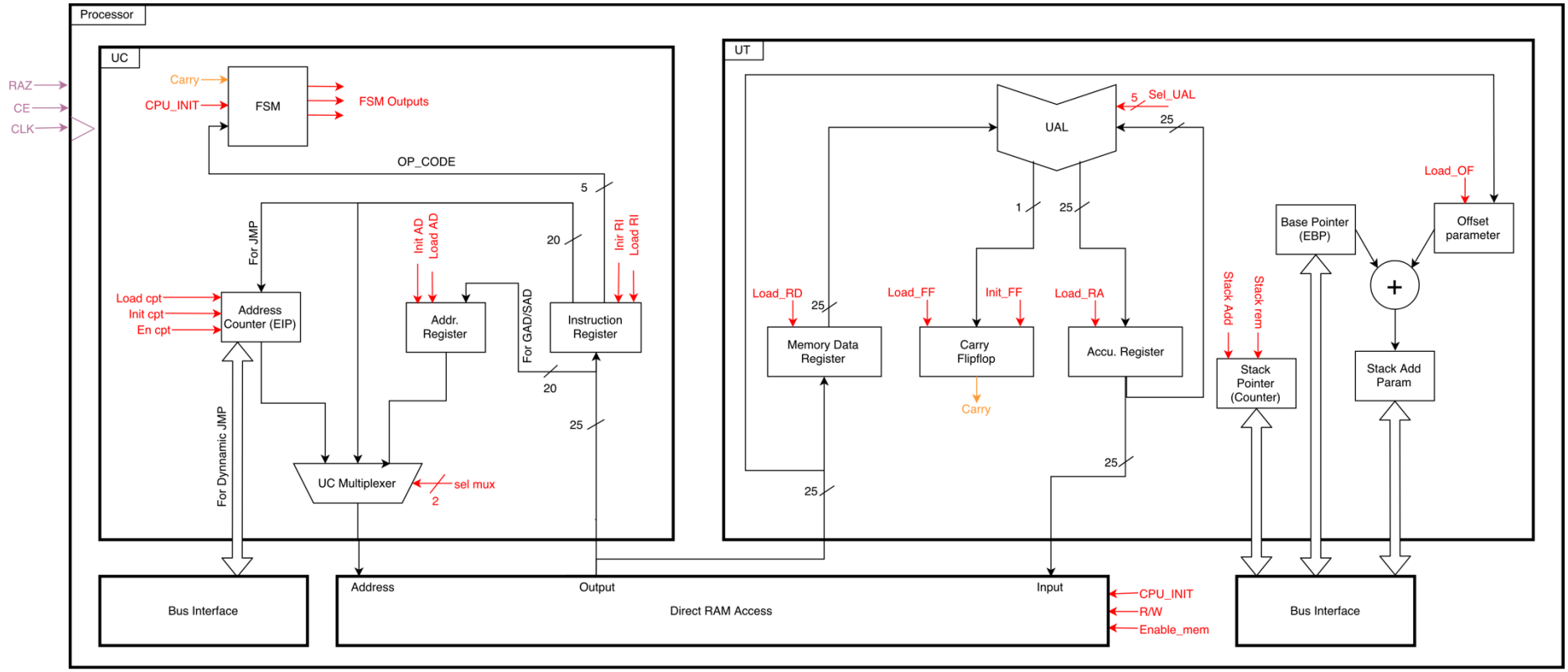

Figure 17. Synoptique de l'architecture du processeur

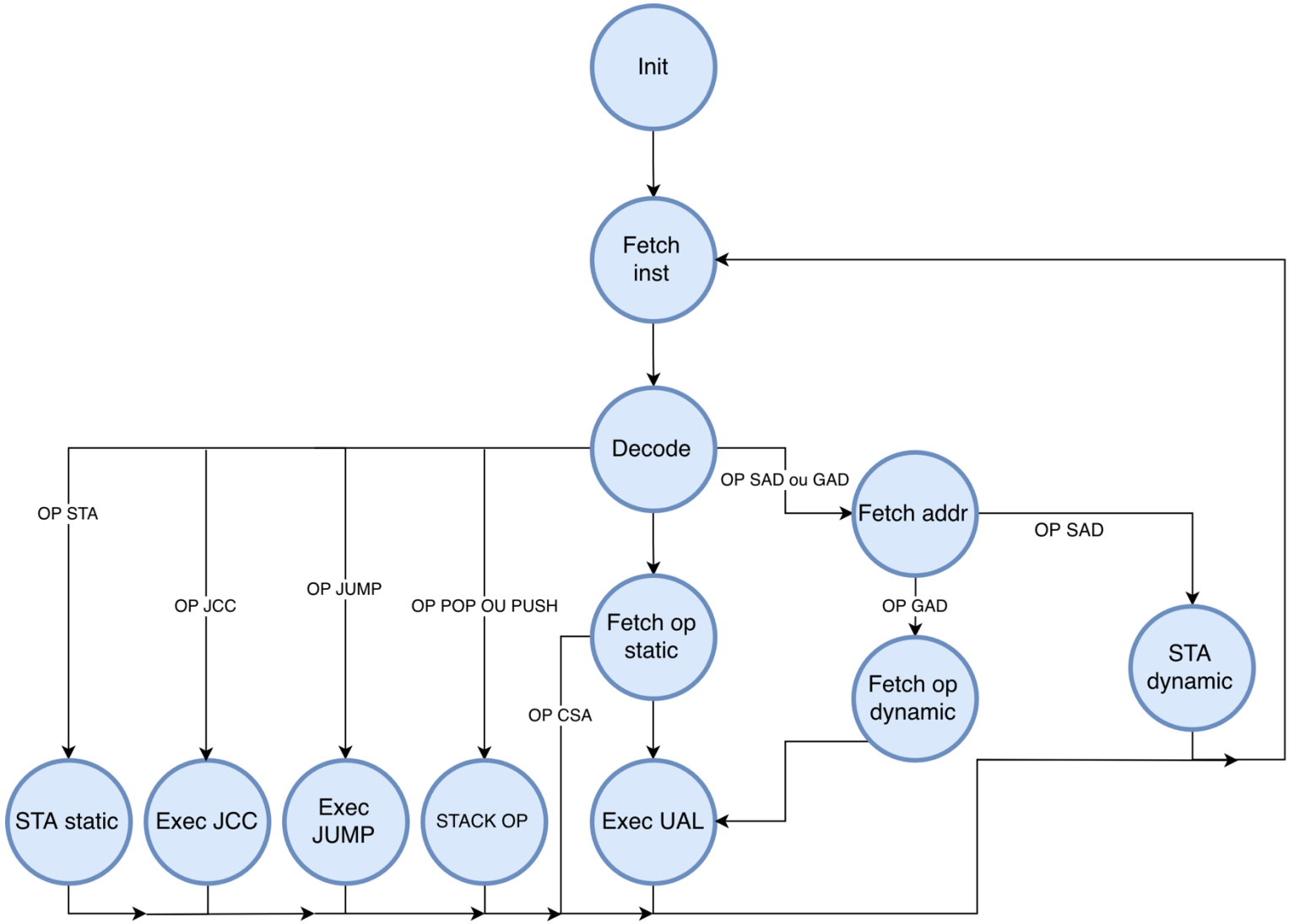

Figure 18. Schéma des différents états de la FSM du processeur