ENSEIRB-MATMECA
BORDEAUX-INP

3A SYSTÈMES EMBARQUÉS

Rapport de Projet Avancé

Accélération d'un algorithme de FFT sur processeur CV32A6

Élèves :
Germain ANCEY
Célian COSSEC
Léa KELEKE
Martin OVAERE

Enseignants :
Camille LEROUX
Mathieu ESCOUTELOUP

27 janvier 2026

Table des matières

Introduction

Ce rapport s'inscrit dans le cadre du projet de neuvième semestre (S9) de notre cursus d'ingénieur en Systèmes Embarqués. Ce projet a pour objectif de réaliser les premières étapes de développement nécessaires à la participation au *6th National RISC-V Student Contest* (édition 2025-2026), un concours sponsorisé par Thales, le GDR SOC^2 et le CNFM.

Présentation du concours

Le défi proposé cette année consiste à optimiser et accélérer l'exécution d'un algorithme de Transformée de Fourier Rapide (FFT) traitant 2^n échantillons sur une architecture de processeur ouvert RISC-V. Le cœur de processeur de référence utilisé est le CV32A6, une implémentation compacte de l'architecture CVA6 développée par l'OpenHW Group à partir du design original ARIANE de l'ETH Zürich.

Objectifs et contraintes de conception

La participation à ce concours impose le respect de contraintes matérielles et logicielles strictes, visant à tester notre capacité à innover tout en préservant l'intégrité de l'architecture système :

- **Exactitude des résultats** : Toute solution développée doit produire des résultats correspondant bit à bit aux motifs de référence fournis dans le kit de développement.
- **Intégrité de l'architecture** : La solution doit reposer principalement sur le processeur CV32A6 ou sur un coprocesseur étroitement couplé via l'interface CV-X-IF.
- **Gestion de la mémoire** : Il est interdit d'augmenter la taille totale des mémoires caches de données ou d'instructions.
- **Performance temporelle** : La fréquence opérationnelle du CV32A6 sur FPGA ne doit pas être dégradée de plus de 20% par rapport au design de base, telle que mesurée par l'analyse temporelle sous Vivado.
- **Compatibilité logicielle** : La solution doit maintenir une compatibilité totale avec le jeu d'instructions RV32IM_Zicsr, bien que l'ajout d'extensions ISA personnalisées soit autorisé.
- **Non-régression** : Le matériel modifié doit être capable d'exécuter sans erreur les applications de test MNIST et CoreMark fournies.

Dans ce contexte, notre travail au cours de ce semestre s'est focalisé sur l'analyse de l'algorithme FFT fourni et sur la mise en œuvre de premières optimisations logicielles et architecturales visant à réduire le nombre de cycles d'horloge nécessaires au calcul, critère principal de classement du jury.

1 Principe de la transformée de Fourier et implémentation en C

L'objectif principal du projet est l'accélération matérielle de la Transformée de Fourier sur une architecture RISC-V. Cette opération mathématique est fondamentale pour le traitement numérique du signal.

1.1 Transformée de Fourier discrète (DFT)

La DFT permet de transformer un signal temporel discret $x[n]$ de N échantillons en sa représentation fréquentielle $X[k]$. Elle est définie par l'équation suivante :

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot e^{-j\frac{2\pi}{N}kn}$$

Cette implémentation propose des avantages comme des inconvénients. Malgré que l'algorithme soit simple à implémenter et ne nécessite aucune contrainte sur la taille N du signal, sa complexité algorithmique est en $O(N^2)$. Pour notre projet de 512 points, cela représenterait environ 262 144 multiplications complexes, ce qui n'est pas optimal pour un processeur embarqué.

1.2 Transformée de Fourier rapide (FFT)

La FFT, et plus particulièrement l'algorithme de Cooley-Tukey utilisé dans la bibliothèque `KissFFT` de notre projet, réduit considérablement cette complexité en exploitant les symétries des coefficients de rotation (*twiddles*).

Cette approche permet d'avoir une complexité réduite à $O(N \log_2 N)$. Pour $N = 512$, on passe de N^2 à environ 4 608 opérations, offrant un gain de performance massif. Toutefois cette approche nécessite généralement que N soit une puissance de 2 (ou factorisable en petits nombres premiers) et consomme davantage de mémoire pour le stockage des *twiddle factors* précalculés.

1.3 Le principe du *butterfly* (papillon)

Le "*Butterfly*" est l'unité de calcul élémentaire de la FFT. Il consiste à combiner deux échantillons de l'étage précédent pour en produire deux nouveaux via une multiplication complexe par un coefficient de rotation W_N^k (*Twiddle*) et des additions/soustractions complexes.

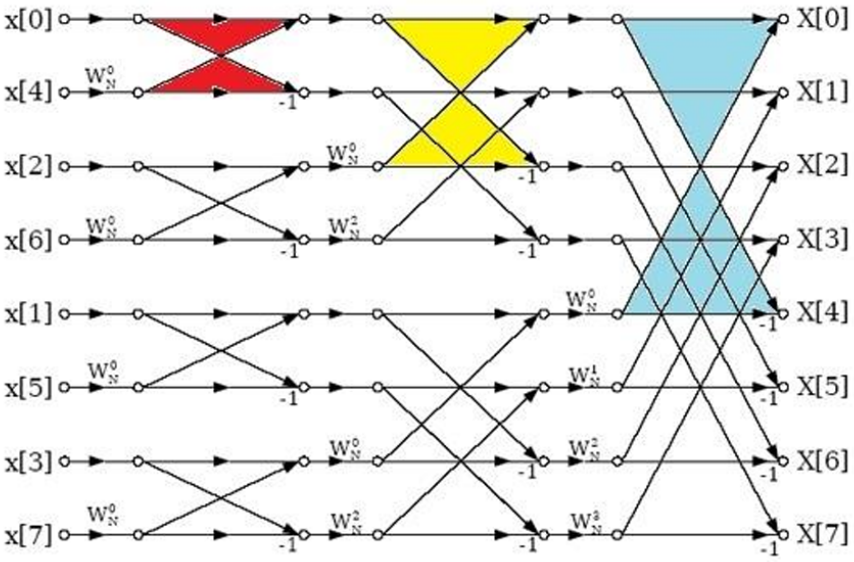

FIGURE 1 – Schéma du calcul du papillon sur 3 étages récursifs

Utilisation et structure

Dans une architecture comme celle implémentée dans notre fichier `kiss_fft.c`, le *butterfly* permet de diviser récursivement le problème en DFT de taille 2 ou 4. Chaque papillon effectue l'opération suivante sur deux données A et B :

— $A' = A + (B \times W_N^k)$

— $B' = A - (B \times W_N^k)$

Application au projet : FFT 512 points

Pour notre implémentation sur 512 points, la structure du calcul est optimisée de plusierus manières.

L'adoption d'une architecture itérative confère des avantages structurels déterminants pour une implémentation matérielle. Le premier bénéfice est la **suppression de la pile** (*Stack*). Alors que la récursion impose une gestion dynamique du contexte, souvent complexe et coûteuse en ressources logiques sur un FPGA, l'approche itérative se contente d'une empreinte mémoire fixe et prévisible. Cette caractéristique élimine par conception tout risque de débordement de pile (*Stack Overflow*), garantissant une meilleure stabilité du système.

Sur le plan de la performance, cette structure favorise le **contrôle du pipeline et le parallélisme**. La présence de boucles de calcul explicites (`for`) permet aux outils de synthèse de haut niveau (HLS) d'appliquer efficacement des directives de déroulage (*Unrolling*) et de pipelinage. Le FPGA est ainsi capable de traiter simultanément plusieurs papillons par cycle d'horloge, ce qui maximise considérablement le débit global du traitement (*Throughput*).

En termes d'accès aux données, la méthode assure une **gestion déterministe de**

la mémoire. L'étape de permutation initiale permet de positionner les données à leurs adresses définitives dans le buffer de travail dès le début du traitement. Cette prédictibilité simplifie la conception de l'unité de génération d'adresses (AGU) et optimise l'utilisation des blocs de mémoire RAM internes (BRAM).

Enfin, l'algorithme garantit une **latence prévisible**, caractérisée par un nombre de cycles d'exécution fixe et connu dès la compilation. Ce déterminisme temporel est une exigence absolue pour les systèmes temps réel critiques, tels que le radar ou les télécommunications, que la récursion ne permet pas de satisfaire avec la même rigueur.

1.4 Comparaison des implémentations de la FFT : récursive vs itérative

Dans le cadre de l'optimisation de l'algorithme de Transformée de Fourier Rapide (FFT) pour une cible matérielle (FPGA), deux approches algorithmiques distinctes ont été évaluées : l'approche récursive et l'approche itérative avec permutation explicite.

L'implémentation récursive (`kf_work`)

Cette version traduit la définition mathématique de l'algorithme de Cooley-Tukey. La fonction se divise elle-même en sous-problèmes plus petits via des appels récursifs jusqu'à atteindre la taille de base ($m = 1$), puis recombine les résultats lors de la remontée de la pile d'appels.

```
void kf_work( ... ) {
    // ...
    // Appel récursif pour diviser le problème
    if (m != 1) {
        do {
            kf_work(Fout, f, fstride*p, in_stride, factors, st);
            f += fstride*in_stride;
        } while ((Fout += m) != Fout_end);
    }

    // Recombinaison des résultats (Butterflies)
    switch (p) {
        case 2: kf_bfly2(Fout, fstride, st, m); break;
        case 4: kf_bfly4(Fout, fstride, st, m); break;
        // ...
    }
}
```

FIGURE 2 – Code de la fonction récursive kf_work_nr()

La gestion de l'ordre des calculs est ici implicite, déterminée par la profondeur de la récursion et l'état de la pile système.

L'implémentation itérative (`kf_work_nr`)

Cette version "aplatit" l'algorithme en supprimant toute récursion. Elle restructure le calcul en deux phases distinctes et séquentielles : la permutation des données et le calcul

par étages.

```c
static void kf_work_nr( ... ) {
    // --- Etape 1 : Permutation (Mixed Radix Bit-Reversal) ---
    for (int i = 0; i < n; i++) {
        // Calcul de l'index 'src' inverse a la volee
        // ...
        Fout[i] = f[src];
    }

    // --- Etape 2 : Calcul des Papillons (Butterflies) ---
    // On remonte du plus petit etage (m=1) vers le plus grand
    int m = 1;
    while (f_ptr >= fac) {
        int p = f_ptr[0]; // Radix courant
        size_t fstride = n / (p * m);

        // Boucle explicite sur les blocs de donnees
        for (int i = 0; i < n; i += p * m) {
            if (p == 4) kf_bfly4(Fout + i, fstride, st, m);
            else if (p == 2) kf_bfly2(Fout + i, fstride, st, m);
        }
        m *= p;      // Taille du prochain bloc
        f_ptr -= 2;  // Passage a l'etage suivant
    }
}
```

FIGURE 3 – Code de la fonction itérative kf_work_nr()

2 Architecture matérielle

2.1 La carte Zybo Z7-20

Pour ce projet, nous travaillons sur une carte Digilent, la Zybo Z7-20. C'est une carte de développement basée sur un SoC Xilinx Zynq-7000, combinant sur une même puce un Processing System (PS) et une logique programmable (Programmable Logic, PL). Le Processing System correspond à la partie processeur du SoC. Il met en œuvre notamment un processeur dual-core ARM Cortex-A9. En ce qui concerne la logique programmable, c'est un FPGA Artix-7 qui est intégré au SoC.

2.2 Le core ARIANE

S'intéressant au jeu d'instructions RISC-V, l'ETH Zürich a développé un core open source du nom d'ARIANE. Le groupe OpenHW ayant pour ambition de designer des IPs Risc-V pour un contexte industriel, c'est tout naturellement qu'ils ont intégré le core ARIANE a leur projet. Il s'agit d'un processeur 64-bits : le CV64A6. C'est ensuite Thalès qui a crée la version 32 bits : le CV32A6. En partageant le même code source, ces processeurs forment donc la famille CVA6. Pour ce projet, c'est le processeur CV32A6

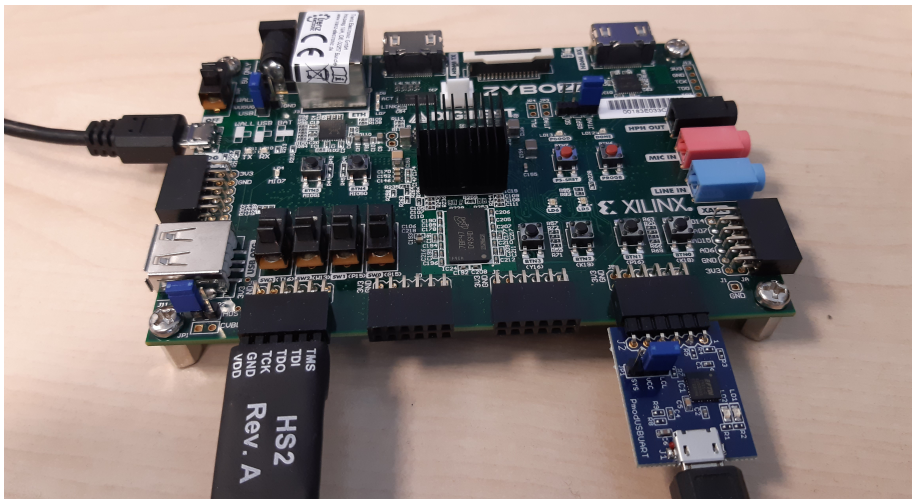

FIGURE 4 – Photo de la carte Zybo

qui va être utilisé tout le long de ce projet. Il est facile de comprendre son nom en le décomposant :

— CV : Core-V. utilisation d'un jeu d'instructions RISC-V
— 32 : données sous format 32 bits
— A : Application grade. Processeur prêt au contexte industriel
— 6 : Pipeline 6 étages

Le processeur utilisé est un softcore. Contrairement à un hardcore, qui est gravé dans le silicium, le softcore est implémenté dans la partie PL du SoC. Ainsi, il est possible dans les améliorations de modifier directement le code source du CV32A6. Celui-ci est écrit en System Verilog, langage de description HDL et réarrange l'implémentation dans la logique programmable pour chaque modification réalisée dans la description.

La pipeline est aussi un élément important du CV32A6. Elle se compose de 6 étapes distinctes.

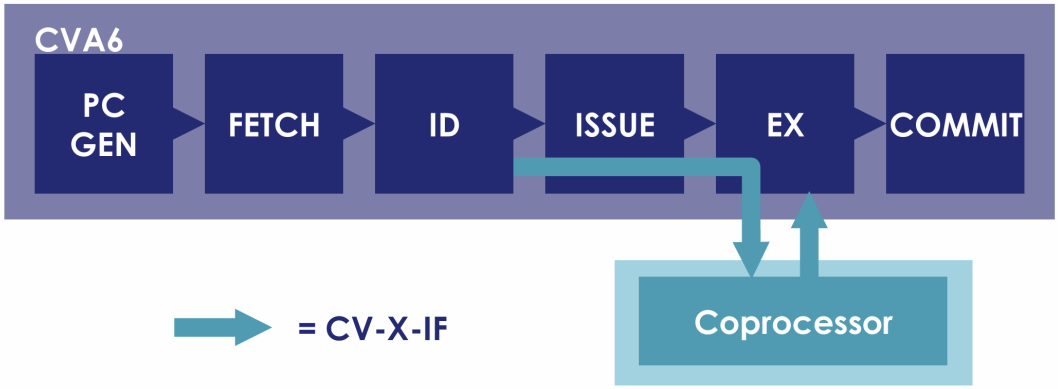

FIGURE 5 – Pipeline du processeur CV32A6 avec interface coprocesseur CV-X-IF

Celle-ci permet le chevauchement de plusieurs instructions simultanément. Il est possible de s'occuper d'une nouvelle instruction dès que la première est passée dans une autre étape de la pipeline, sans attendre son traitement complet.

1. Génération du compteur de programme (PC GEN)

L'étage *PC Generation* est chargé de produire l'adresse de la prochaine instruction à exécuter. Il assure l'incrémentation séquentielle du compteur de programme et prend en compte les instructions de branchement, de saut ou la gestion des exceptions.

2. Récupération des instructions (FETCH)

L'étage *FETCH* récupère l'instruction depuis la mémoire d'instructions à l'adresse fournie par l'étage précédent. Cette phase inclut l'accès mémoire ainsi que les opérations nécessaires à la transmission de l'instruction vers l'étage de décodage.

3. Décodage des instructions (ID)

L'étage *Instruction Decode* analyse l'instruction RISC-V récupérée lors du fetch. Il identifie l'opcode, les registres sources et destination, ainsi que le type d'opération à effectuer. Cet étage est également responsable de la lecture des opérandes depuis le fichier de registres.

4. Émission des instructions (ISSUE)

L'étage *ISSUE* contrôle l'émission de l'instruction vers l'unité d'exécution appropriée. Le CV32A6 étant un processeur *in-order*, les instructions sont émises dans l'ordre du programme. Cet étage vérifie la disponibilité des ressources et des opérandes avant de permettre l'exécution.

5. Exécution (EX)

L'étage *EX* réalise l'exécution effective des instructions. Il englobe les opérations arithmétiques et logiques, le calcul des adresses pour les accès mémoire, ainsi que la résolution des branchements. Dans le CV32A6, cet étage peut également interagir avec un coprocesseur externe via l'interface *CV-X-IF*. Nous parlerons de cette fonctionnalité dans la suite.

6. Validation et écriture des résultats (COMMIT)

L'étage *COMMIT* constitue la dernière phase du pipeline. Il est responsable de l'écriture définitive des résultats dans le fichier de registres et de la validation de l'instruction au niveau architectural.

2.3 Ajout d'un coprocesseur

Le processeur est basé sur le jeu d'instructions RISC-V, mais pour notre objectif d'accélération de la FFT, il peut être intéressant créer nos propres instructions. Ainsi, au lieu de réaliser plusieurs opérations à la suite dans la partie software, on cherche à réaliser le calcul en matériel, nécessitant bien moins de cycles d'horloge.

Une fonctionnalité utile est la possibilité d'ajouter un coprocesseur. En effet, le CV32A6 peut ainsi envoyer les instructions personnalisées au coprocesseur. Dans la pipeline, cela se traduit par un envoi au niveau de l'étage *Decode*. Lorsque le CVA6 remarque que

l'instruction à décoder est "illégale" (*ie* elle n'est pas contenue dans le jeu d'instructions RISC-V), elle est envoyée directement au coprocesseur.

2.4 Interface de communication CV-X-IF

L'interface CV-X-IF (Core-V eXtension Interface) constitue un élément clé du processeur CV32A6, permettant l'extension et la communication avec des périphériques supplémentaires (le coprocesseur dans ce cas). Elle fournit un moyen de communication standardisé entre le cœur RISC-V et le coprocesseur. Le CV-X-IF se synchronise avec la pipeline pour envoyer les résultats au bon moment et respecter les délais. Cette interface facilite l'implémentation de d'instructions spécifiques, tout en conservant la compatibilité avec l'architecture RISC-V standard.

3 Environnement de développement

L'environnement de travail repose sur le dépôt officiel fourni par Thales (cva6-softcore-contest), qui standardise la chaîne de conception matérielle et logicielle. Cette infrastructure garantit que les développements reproductibles sur la carte cible **Zybo Z7-20**.

3.1 Structure du dépôt et hiérarchie matérielle

Le projet s'appuie sur le dépôt officiel `cva6-softcore-contest`. Son organisation est structurée pour séparer les ressources matérielles, logicielles et les scripts d'automatisation.

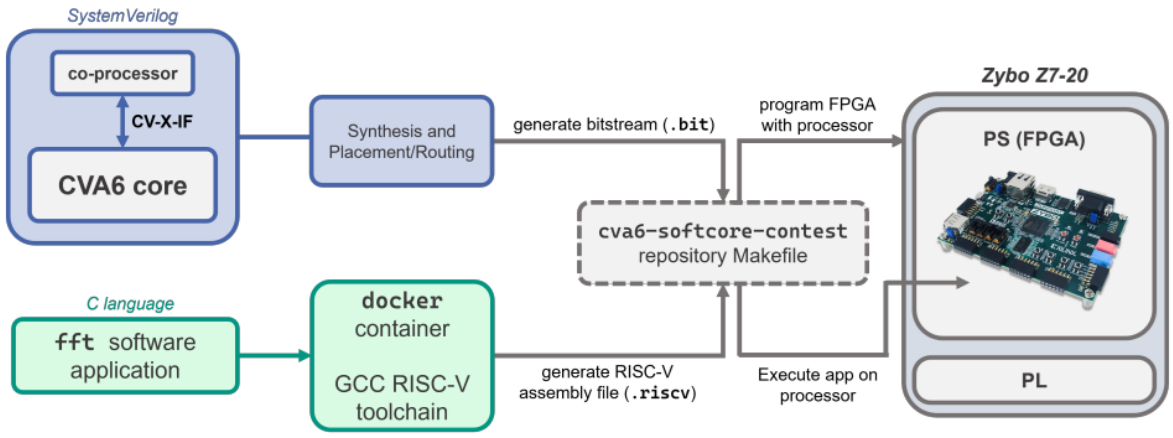

FIGURE 6 – Schéma de la structure du projet

A. Les répertoires matériels (`corev_apu/` et `core/`)

Le design matériel est divisé en deux niveaux de hiérarchie distincts :

— `corev_apu/` **(L'enveloppe système) :** Ce dossier gère l'intégration du processeur dans son environnement. Il contient les bus de communication (AXI, APB), les

périphériques (UART, GPIO) et les composants spécifiques au FPGA (wrappers de mémoire SRAM).

— `core/` **(Le cœur CVA6) :** Ce répertoire contient la logique interne du processeur (pipeline, unités de calcul, gestion des interruptions). Il inclut également le sous-dossier `cvxif_example/`, qui implémente l'interface **CV-X-IF**. Ce port d'extension est crucial car c'est celui qui permet de connecter notre futur accélérateur FFT au pipeline principal sans modifier l'architecture de base du cœur.

B. La couche logicielle (`sw/`)

Le répertoire `sw/` regroupe l'intégralité du code s'exécutant sur le processeur :

— **Les applications (`sw/app/`) :** On y trouve le point d'entrée de nos tests, notamment le fichier `fft_int16_main.c` qui orchestre les mesures de performance.

— **Les bibliothèques (`sw/app/fft/kissfft_lib/`) :** Ce dossier contient le cœur algorithmique. C'est ici que nous interviendrons pour lier le code C à nos nouvelles instructions matérielles.

— **Le BSP (Board Support Package) :** Il contient les pilotes essentiels au démarrage du processeur et à la gestion des entrées/sorties sur la carte Zybo.

C. Flux de synthèse et automatisation (`fpga/` et `Makefile`)

— `fpga/` **:** Contient les scripts Tcl et les fichiers de contraintes (`.xdc`) indispensables à Vivado pour définir le brochage physique et les contraintes d'horloge sur la puce Zynq-7000.

— **Makefile :** Point d'entrée unique du projet. Il organise l'intégralité des flux (simulation, synthèse matérielle, compilation logicielle) garantissant une synchronisation entre le matériel et le logiciel.

3.2 Chaîne de compilation logicielle (Docker)

La compilation croisée pour l'architecture RISC-V nécessite un environnement complexe (GCC spécifique, bibliothèques *newlib*, OpenOCD). Pour s'affranchir des différences de configuration, Thales fournit une image Docker nommée `sw-docker`.

Le fonctionnement de cet environnement repose sur le montage du répertoire `sw/` de la machine hôte directement dans le point de montage `/workdir` du conteneur. Cette architecture permet une grande souplesse : le développeur modifie le code source C sur son système d'exploitation habituel, tandis que la compilation s'effectue instantanément au sein de l'environnement isolé. En exécutant simplement la commande `make fft` à l'intérieur du conteneur, le compilateur RISC-V GCC génère le binaire exécutable au format `.riscv`, prêt à être chargé sur la cible matérielle.

3.3 Flux de synthèse matérielle (Hardware Flow)

La transformation du code SystemVerilog en configuration FPGA est scriptée pour **Xilinx Vivado** via la commande `make cva6_fpga` :

1. **Synthèse :** Traduction du RTL en portes logiques génériques.

2. **Implémentation :** Placement et routage des cellules sur la grille du Zynq-7000.
3. **Génération du Bitstream :** Création du fichier `.bit` final.

Les rapports de ressources (`utilization_placed.rpt`) permettent de surveiller la consommation de LUT, DSP et BRAM.

3.4 Méthodologie de déploiement et de test

Le déploiement sur la carte Zybo Z7-20 s'effectue en deux étapes distinctes via une interface **JTAG-HS2**. On commence par programmer la logique du FPGA avec la commande `make program_cva6_fpga` pour instancier physiquement le processeur.

Une fois le matériel prêt, le binaire logiciel `.riscv` est injecté directement dans la RAM du CVA6 via **OpenOCD** et **GDB**. Cette méthode évite de reflasher le FPGA à chaque modification du code C, ce qui accélère les cycles de test.

La validation fonctionnelle repose sur une liaison **UART**. Les messages de débogage et les résultats de la FFT sont redirigés vers un terminal série (115200 bauds). Le succès du test est prononcé uniquement si la sortie du processeur correspond bit-à-bit à la **"Gold FFT"** de référence.

3.5 Métriques de référence (Baseline)

L'analyse de l'algorithme *KissFFT* sur le cœur standard a permis d'établir les performances de référence (*baseline*) avant optimisation :

Performances logicielles : L'exécution de référence s'établit à **169 000 cycles CPU** (soit 124 000 instructions). Ce score servira de base pour mesurer le gain d'efficacité apporté par nos futures instructions matérielles.

Ressources et Routage : Le design actuel mobilise environ **20 000 LUTs**, occupant ainsi seulement **37%** de la surface disponible sur la Zybo.

Avec une congestion de routage limitée à **10%**, ces indicateurs confirment que nous disposons d'une **marge importante** pour intégrer des coprocesseurs complexes sans risquer de saturer le FPGA ou de dégrader les contraintes de timing.

4 Accélérations apportées

L'objectif principal de ce projet est d'accélérer l'exécution de la FFT en déportant les calculs intensifs du processeur CVA6 vers un coprocesseur matériel dédié implémenté sur le FPGA. Pour ce faire, nous avons utilisé l'interface CV-X-IF (Core-V eXtension InterFace) qui permet d'étendre le jeu d'instructions du processeur sans modifier son cœur interne.

Notre méthodologie de développement pour chaque instruction personnalisée suit trois étapes rigoureuses :

1. **Définition de l'instruction (niveau ISA) :** Choix de l'opcode, du format et configuration du masque de décodage.
2. **Implémentation matérielle (niveau RTL) :** Description de la logique arithmétique en SystemVerilog dans l'ALU du coprocesseur.

3. **Intégration logicielle (niveau application C) :** Modification des macros C de la bibliothèque KissFFT pour appeler l'instruction matérielle via de l'assembleur inline.

Mais avant cela, nous devons également activer l'utilisation du coprocesseur par le CV-X-IF dans le fichier Verilog `cv32a6_im_contest_config_pkg.sv` :

FIGURE 7 – Activation du bit `CVA6ConfigCvxifEn`

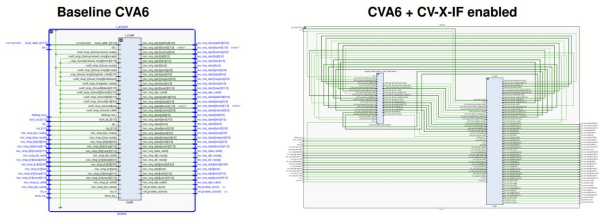

FIGURE 8 – Le processeur CVA6 Ariane avec et sans activation du bit `CVA6ConfigCvxifEn`

4.1 Multiplication complexe

4.1.1 Analyse du goulot d'étranglement

La multiplication complexe est l'opération élémentaire la plus coûteuse de l'algorithme FFT. Mathématiquement, le produit de deux nombres complexes $A = a+ib$ et $B = c+id$ est défini par :

$$P = A \times B = (ac - bd) + i(ad + bc)$$

Sur une architecture RISC-V standard, cette opération nécessite au minimum 4 instructions de multiplication (`mul`), 1 soustraction (`sub`) et 1 addition (`add`), sans compter les instructions de chargement mémoire (`load/store`) et la gestion des décalages pour la virgule fixe. L'objectif est de réduire cette séquence à une seule instruction exécutée en un cycle d'horloge effectif grâce au parallélisme du FPGA.

4.1.2 Encodage de l'instruction (ISA)

Nous avons défini l'instruction `CPLX_MUL` dans le fichier `cvxif_instr_pkg.sv`. Elle utilise le format standard **R-Type** du RISC-V, qui prend deux registres sources (`rs1`, `rs2`) et écrit dans un registre de destination (`rd`).

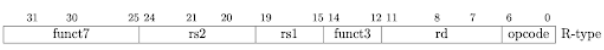

FIGURE 9 – Format de l'instruction R-Type RISC-V

L'instruction est identifiée par l'opcode réservé `custom-3` ($0x7B$ ou 1111011_2) et différenciée des autres extensions par le champ `funct3` fixé à 010_2.

Le décodage repose sur un mécanisme de masque défini dans le package SystemVerilog :

— Les bits du masque à `1` (Opcode, Funct3) imposent une correspondance exacte avec l'instruction modèle.

— Les bits du masque à `0` (les adresses des registres `rs1`, `rs2`, `rd`) sont ignorés par le décodeur, permettant l'utilisation de n'importe quel registre général du processeur.

```
'{
    // Custom Complex Mul: cplx_mul rd, rs1, rs2
    instr: 32'b0000000_00000_00000_010_00000_1111011,
    mask:  32'b1111111_00000_00000_111_00000_1111111,
    resp : '{accept : 1'b1, writeback : 1'b1, register_read : {1'b0,
   1'b1, 1'b1}},
    opcode : CPLX_MUL
}
```

Listing 1 – Définition de l'instruction CPLX_MUL (Extrait de cvxif_instr_pkg.sv)

4.1.3 Implémentation Matérielle (RTL)

L'implémentation a été réalisée dans le fichier `copro_alu.sv`. Contrairement à l'exécution séquentielle du logiciel, le FPGA permet de paralléliser les calculs.

Le chemin de données effectue les opérations suivantes en un cycle :

1. **Unpacking :** Les registres d'entrée 32 bits contiennent deux valeurs 16 bits (Partie Réelle sur les bits [15 :0], Partie Imaginaire sur [31 :16]).
2. **Calcul parallèle :** Les 4 produits partiels ($ar \cdot br$, $ai \cdot bi$, $ar \cdot bi$, $ai \cdot br$) sont calculés simultanément grâce aux blocs DSP du FPGA.
3. **Arrondi :** Nous travaillons en virgule fixe au format Q15. Le résultat d'une multiplication est en Q30. Pour revenir au format Q15 avec un arrondi au plus proche, nous ajoutons $2^{14} = 16384$ (équivalent à 0.5 en Q15) avant le décalage.
4. **Packing :** Les résultats finaux sont tronqués et recombinés dans un seul registre 32 bits.

Le code SystemVerilog ci-dessous illustre cette logique :

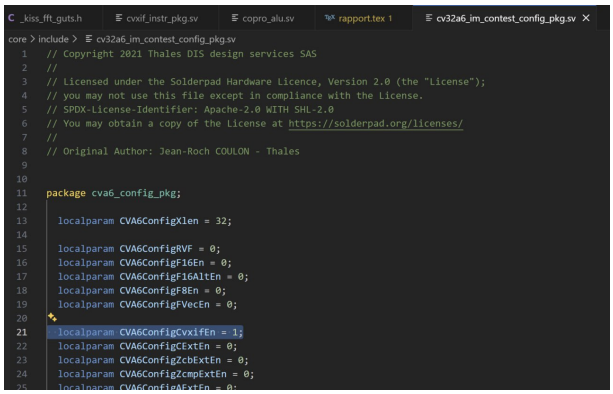

FIGURE 10 – Logique arithmétique de la multiplication complexe (Extrait de copro_alu.sv)

4.1.4 Intégration logicielle

Pour exploiter cette accélération, nous avons modifié le fichier `_kiss_fft_guts.h` de la bibliothèque logicielle. La macro standard `C_MUL`, qui effectuait les calculs séquentiels en C, a été remplacée par un appel assembleur.

La directive `.insn r` permet d'injecter directement l'opcode hexadécimal de notre instruction personnalisée, en mappant les variables C (m, a, b) aux registres processeur correspondants.

```
#define C_MUL(m, a, b) \
    do { \
        __asm volatile ( \
            ".insn r 0x7B, 0x2, 0x00, %0, %1, %2" \
            : "=r"(res) \
            : "r"(a_val), "r"(b_val) \
        ); \
        (m) = res; \
    } while(0)
```

Listing 2 – Macro C_MUL accélérée (Extrait de _kiss_fft_guts.h)

Cette modification est transparente pour le reste du code de la FFT, permettant une accélération immédiate de l'algorithme global sans réécriture complexe.

4.2 Bit reverse

La tentative d'optimisation de la FFT sur le processeur CVA6 avec l'**approche itérative** (non récursive) engendre un certain nombre de bit-reverse dans le software. Pour palier à ça nous avons instancier une instruction dans le coprocesseur.

Définition de l'instruction (Software)

L'instruction **BITREV** a été créée pour effectuer cette permutation instantanément. Elle prend en paramètre la valeur à inverser et le nombre de bits correspondant à la taille de la FFT ($\log_2 N$).

```
// Appel de l'instruction custom 0x7B (funct 0x4)
#define HW_BITREV(val, nbits) ({ \
    int32_t res; \
    asm volatile ( ".insn r 0x7B, 0x4, 0x00, %0, %1, %2" \
        : "=r"(res) : "r"(val), "r"(nbits) ); \
    res; \
})
```

Listing 3 – Macro simplifiée pour BITREV

Implémentation matérielle (SystemVerilog)

Contrairement à une boucle logicielle, le FPGA réalise cette opération par un simple routage combinatoire. L'unité matérielle extrait les N bits de poids faible de l'opérande et les réassigne dans l'ordre inverse.

```
cvxif_instr_pkg::BITREV: begin
    // Récupération dynamique de la taille
    nbits = registers_i[1][5:0];
    bitrev = '0;

    // Logique de renversement (boucle déroulée à la synthèse)
    if (nbits != 0) begin
        for (idx = 0; idx < nbits; idx++) begin
            bitrev[idx] = registers_i[0][nbits-1-idx];
        end
    end
    result_n = bitrev;
end
```

Listing 4 – Cœur de la logique Bit Reversal

4.3 Full Butterfly

L'objectif était de réduire le nombre de cycles en implémentant une instruction matérielle "tout-en-un". Cette unité doit traiter l'opération de **papillon (butterfly)** complète, qui constitue le cœur de l'algorithme FFT.

Définition de l'instruction (Software)

Dans le fichier `kiss_fft_guts.h`, nous avons remplacé les calculs complexes standards par des macros appelant une instruction assembleur personnalisée. L'instruction utilise le format **R-type custom** (opcode $0x7B$) avec deux variantes :

- **CPLX_BFLY0** : Effectue $A + (B \times W)$
- **CPLX_BFLY1** : Effectue $A - (B \times W)$

Chaque registre de 32 bits contient un nombre complexe packé : 16 bits pour la partie réelle et 16 bits pour la partie imaginaire.

```
#define HW_BFLY0(a, b, w) ({ \
    int32_t res; \
    asm volatile ( ".insn r 0x7B, 0x3, 0x00, %0, %1, %2, %3" \
        : "=r"(res) : "r"(a), "r"(b), "r"(w) ); \
    res; \
})
```

Listing 5 – Macro C pour l'instruction custom

Implémentation matérielle (SystemVerilog)

Le calcul est intégré via l'interface **CV-X-IF** et s'exécute théoriquement en un seul cycle d'horloge. La logique matérielle suit quatre étapes :

1. **Unpacking** : Séparation des parties réelles et imaginaires des opérandes A, B et W.
2. **Multiplication complexe** : Calcul de $B \times W$ générant quatre produits partiels.

3. **Réduction et Arrondi** : Application d'un décalage à droite (>>> 15) avec un arrondi (+16384) pour maintenir la précision.
4. **Finalisation** : Addition ou soustraction avec le registre A pour obtenir le résultat final.

```
cvxif_instr_pkg::CPLX_BFLY0, cvxif_instr_pkg::CPLX_BFLY1: begin
    // Unpacking : rs1=A, rs2=B, rs3=W
    ar = registers_i[0][15:0];  ai = registers_i[0][31:16];
    br = registers_i[1][15:0];  bi = registers_i[1][31:16];
    wr = registers_i[2][15:0];  wi = registers_i[2][31:16];

    // Multiplication complexe B * W
    p_rr = br * wr; p_ii = bi * wi;
    p_ri = br * wi; p_ir = bi * wr;

    // Produit réduit (shift 15 avec arrondi)
    prod_r = (p_rr - p_ii + 32'd16384) >>> 15;
    prod_i = (p_ri + p_ir + 32'd16384) >>> 15;

    // Addition ou Soustraction finale avec A
    if (opcode_i == cvxif_instr_pkg::CPLX_BFLY0) begin
        result_n[15:0]  = ar + prod_r[15:0];
        result_n[31:16] = ai + prod_i[15:0];
    end else begin
        result_n[15:0]  = ar - prod_r[15:0];
        result_n[31:16] = ai - prod_i[15:0];
    end
    valid_n = 1'b1; we_n = 1'b1; rd_n = rd_i;
    hartid_n = hartid_i; id_n = id_i;
end
```

Listing 6 – Extrait de l'unité de calcul SystemVerilog

Résultats et analyse des erreurs

Malgré une compilation réussie, les tests sur carte Zybo révèlent des écarts systématiques avec la **Gold FFT**. Deux facteurs techniques expliquent cet échec :

— **Échec du Timing (Chemin critique) :** L'enchaînement des multiplications et des additions en un seul cycle crée un chemin logique trop profond pour la fréquence cible du FPGA. Les signaux n'ont pas le temps de se stabiliser.

— **Saturation et Débordements :** L'absence de logique de **saturation matérielle** lors de l'addition finale provoque des dépassements de capacité sur 16 bits, corrompant les résultats.

Cette approche "Full Butterfly" en un cycle est trop **ambitieuse**. Une solution viable nécessiterait de segmenter cette opération dans un pipeline multi-cycles.

5 Résultats obtenus

Lors de l'exécution du programme, on communique sur un terminal via **minicom**. Les valeurs calculées sont comparées avec des golden values qui indiquent si les résultats sont corrects ou non. On obtient alors cette fenêtre.

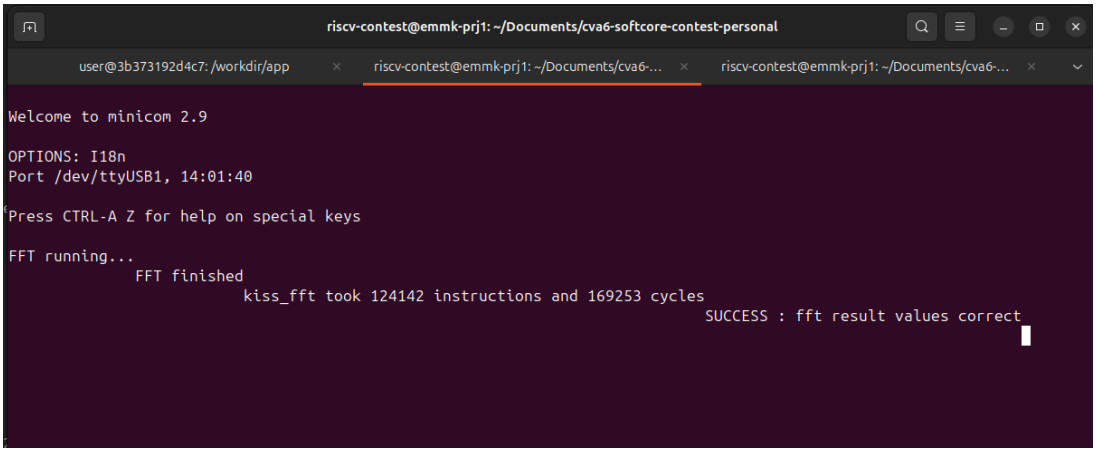

FIGURE 11 – Sortie de minicom lors de l'exécution de l'algorithme FFT fourni par Thales

En cas de mauvais calculs, le terminal affiche **fail** et renvoie les sorties incorrectes. Nous avons divisé les solutions en 2 parties : la version récursive ainsi que la version itérative.

Pour le cas récursif, une fois l'instruction personnalisée de la multiplication complexe implémentée, le nombre de cycles d'horloge chute de 35 000 environ. On passe de 170 000 pour la version sans optimisations à 135 000 cycles.

Dans le cas itératif, on remarque une légère augmentation du temps de traitement, cela est dû au fait que des calculs s'ajoutent pour trouver les indices des butterflies. C'est pour pallier ce problème que le *bit_reversal* est implémenté.

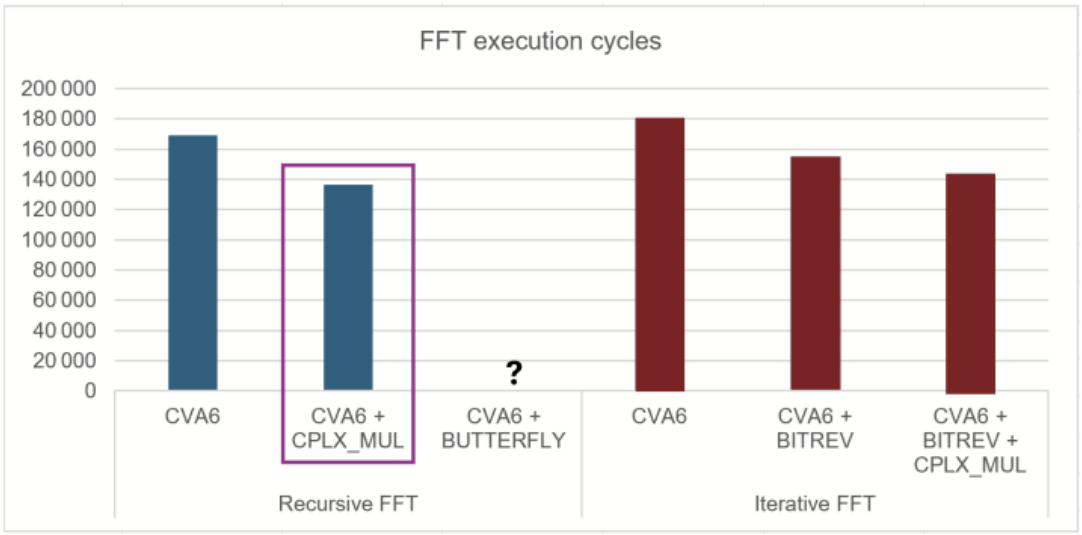

FIGURE 12 – Histogramme des résultats des optimisations

Pour les modifications architecturales (multiplication complexe), on obtient aussi l'utilisation des ressources suivantes :

	CVA6	CVA6 + complex multiplication	Variation
Software performance			
Cycles (FFT)	**169 253**	**136 445**	**-19.4%**
Instructions	124 142	103 224	
FPGA timing constraints			
F_{max}	40.3 MHz	38.5 MHz	-4.5%
Timing slack	+0.175 ns	-0.936 ns	
FPGA ressources			
Logic (LUTs)	9 839	9 814 **+ 150 (CV-X-IF)**	
DSP blocks	4	4 **+ 5 (CV-X-IF)**	

FIGURE 13 – Utilisation des ressources sur le FPGA

6 Perspectives et Optimisations Futures

L'implémentation d'une instruction "Full Butterfly" en un seul cycle, bien que fonctionnelle en simulation comportementale, a révélé ses limites physiques sur FPGA (problèmes de timing et de saturation). Les travaux futurs devront donc se concentrer sur la correction de cette architecture et l'optimisation de la mémoire.

6.1 Pipeline et Saturation du "Full Butterfly"

L'échec de l'instruction **CPLX_BFLY** mono-cycle (décrit précédemment) met en évidence la nécessité de repenser l'architecture matérielle :

— **Passage au Multi-Cycles (Pipelining) :** Pour résoudre le problème de chemin critique (timing violation), l'opération doit être segmentée. Au lieu de tenter le calcul $A \pm (B \times W)$ en un cycle, il faudrait le diviser en 2 ou 3 étages de pipeline (ex : Étage 1 = Multiplication, Étage 2 = Addition/Soustraction). Cela permettrait de remonter la fréquence de fonctionnement vers 50 MHz.

— **Gestion de la Saturation :** L'ajout d'une logique matérielle de **clipping** (saturation) est impératif pour corriger les erreurs de débordement observées sur la Zybo. Cela garantira que les résultats dépassant 16 bits soient maintenus aux valeurs maximales ($+32767$ ou -32768) au lieu de boucler, stabilisant ainsi la précision numérique.

6.2 Approche Alternative et Débogage

Si le "Full Butterfly" pipeliné s'avère trop complexe à intégrer via l'interface CV-X-IF actuelle, une approche intermédiaire consisterait à implémenter des instructions atomiques

accélérées pour l'addition et la soustraction complexes (**CPLX_ADD/SUB**). Cela permettrait de paralléliser le *Fetch* d'une instruction avec le *Calculate* de la précédente.

De plus, l'accès à un simulateur de niveau RTL avancé comme **QuestaSim** (actuellement non disponible dans notre environnement) serait un atout décisif pour visualiser précisément les délais de propagation et valider le comportement temporel avant la synthèse.

6.3 Optimisation Mémoire : Stockage des Twiddle Factors

Une part significative des cycles processeur est actuellement consommée par la latence d'accès à la RAM pour charger les coefficients trigonométriques (*Twiddle Factors*). Une optimisation architecturale standard consisterait à pré-stocker ces tables de Sinus et Cosinus directement dans des **LUTs (Look-Up Tables)** ou une ROM locale au sein du coprocesseur.

Cette approche supprimerait totalement les délais d'attente mémoire en permettant un accès aux coefficients en un seul cycle d'horloge. Toutefois, cette technique de stockage local n'est **pas autorisée** dans le cadre strict du concours, qui impose l'utilisation de la mémoire principale, nous privant ainsi d'un levier d'accélération majeur.

Conclusion

Ce projet nous a permis d'aborder concrètement une architecture de la famille CVA6 dans sa globalité. Nous avons pu travailler sur une accélération matérielle qui modifie directement le code source du core CV32A6 et qui se joint à notre compréhension de l'architecture des processeurs.

Nous avons eu l'occasion de mettre en œuvre une démarche complète de conception conjointe matériel / logiciel, depuis l'analyse de l'algorithme de FFT jusqu'à l'intégration d'instructions personnalisées. L'implémentation de la multiplication complexe et de l'instruction de bit-reverse a permis d'obtenir des gains de performance mesurables.

Enfin, ce projet nous a permis de nous familiariser avec l'ensemble de la chaîne de développement RISC-V, de la modification de l'ISA jusqu'à la validation sur carte FPGA. Il constitue une base solide pour la poursuite des optimisations dans le cadre de la compétition et pour de futurs travaux en accélération matérielle et systèmes embarqués.