Projet
Synthèse d'architecture pour FPGA à partir d'un programme Go

Zakaria Sabhi

Brest
8 février 2021

Erwan Fabiani

Table des matières

1 Présentation du projet :

Selon la loi de moore [1], l'évolution des solutions proposées par les processeurs classiques conventionnels atteignent leurs limites en terme d'efficacité / consommation d'énergie. Une architecture hétérogène est de plus en plus utilisée pour surmonter ce problème, voire mieux, améliorer l'efficacité tout en limitant la consommation. Dans ce domaine les FPGA jouent un rôle prépondérant, grâce à leur degré de flexibilité (capacité d'être re-configuré), degré de parallélisme, et l'efficience en terme d'énergie offert. La plupart la programmation des FPGA reposent sur un langage de description matériel **HDL** *(VHDL par exemple)*, ce qui en résulte qu'une connaissance de le domaine des circuits électronique restent nécessaire.

L'apparition du HLS *High Level Synthesis* a permit aux développeurs de contourner ces exigences, permettant à ces derniers d'utiliser les FPGAs plus facilement en utilisant un langages haut niveau comme C ou C++.
Les CSP **« Communicating sequential processes »** ont permit aussi de réduire la complexité qu'on programme un FPGA. Un CSP représente un concept qui décrit l'ensemble des interactions d'un système sous forme des messages qui circule via des canaux entre entités. Une forme de communication qui va parfaitement avec la fonctionnalité de parallélisme qu'offre un FPGA.

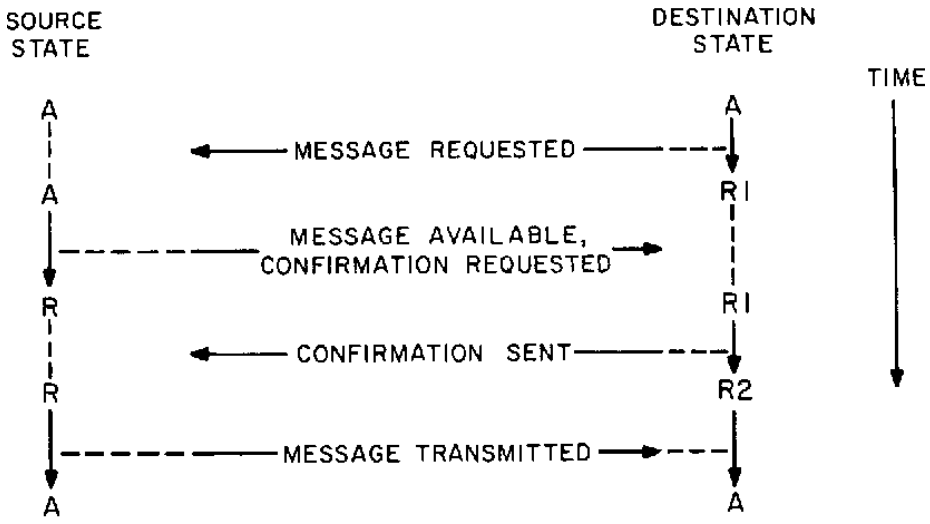

Fig. 3.

FIGURE 1 – Communicating Sequential Processes

Bien que l'objectif de ce projet est de profiter des caractéristiques du modèle CSP en programmant un FPGA en partant d'un programme **« Go »** comme input, ce rapport se focalise sur l'implémentions des canaux synchrones en

FPGA [2].

1.1 Go Langage de programmation :

Go représente un langage de programmation concis, son efficacité est garanti par les fonctionnalité qu'il offre à savoir :

— Go Routine ;
— Canaux ;

1.1.1 Go Routine

Les go routines représentent une tâche légères, un thread qui exécute une fonction. Dans l'exemple ci-dessous, la création d'une go s'effectue avec l'instruction **go f_hello()** :

```
package main

import "fmt"

func f_hello(){
    fmt.Printf("hello, world\n")
}

func main() {
    go f_hello();
    fmt.Printf("main_thread\n")
}
```

FIGURE 2 – Implementation d'une routine en Go

En FPGA, les **go routines** peuvent être utilisées pour exprimer explicitement le parallélisme et indiquer au compilateur quelle portion du programme va être exécuté parallèlement.

1.1.2 Canaux

Un canal est une liaison qui connecte deux go routines concurrentes, en FPGA, cette liaison exprime une dépendance de données entre les différent go-

routine :

```go
package main

import "fmt"

func calculus(input chan int, output chan int ){
    data := <- input
    data = data * 2
    output <- data
 }

func main() {
    in := make (chan int)
    out := make(chan int)
    data := 2
    in <- data
    go calculus(in , out)
    data = <- out
    fmt.Println(data)
}
```

FIGURE 3 – Implementation d'un canal en go

Le programme étudié est représenté sous forme de des noeuds communiquants entre eux, les noeuds ici représentent des go-routines, des threads *light*, qui communiquent via des canaux inspirés du modèle CSP.

Synthétiser un programme en Go pour l'implanter au sein d'un FPGA, revient à traiter la problématique concernant le choix du moyen pour synthétiser les canaux de communication synchrones du programme [3], plusieurs travaux on été proposer pour répondre à cette problématique, bien que chacun propose des manières différents pour Parser le code du programme, la majorité choisissent la méthode FIFO *(First In First Out)* pour représenter les canaux synchrones.

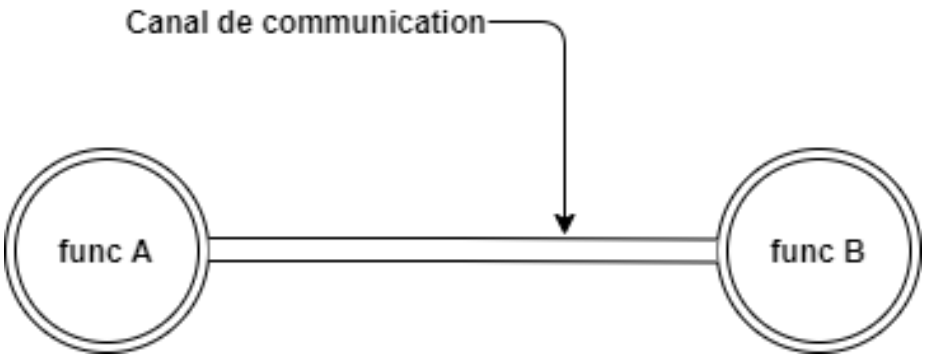

FIGURE 4 – Communication entre deux noeuds

2 Implementation des canaux synchrones :

2.1 État de l'art

2.1.1 L'outil `Reconfigurel0`

`Reconfigurel0` est une plate-forme pour programmer un accélérateur matériel à partir d'une application. Avec les outils proposés dans cette solution *commerciale*, l'application peut être développer en langage **GoLang**. La plate-forme en question se base sur les FPGAs, permettant ainsi de profiter au maximum de l'aspect parallélisme d'un programme en Go.

`Reconfigure.io` [4] représente **PaaS** *platform as a service*, qui compile le programme Go l'optimise et le déploie sur des instances FPGA. La plateforme utilise son propre compilateur **rio** qui permet de derouler automatiquement les boucles basics d'une part, d'autre part, pour avoir un programme pipeliné sur FPGA, le passage par des go-routine est obligatoire.

Pour les communication entre noeuds(*go-routines)*, le compilateur favorise l'utilisation du protocole AXI.

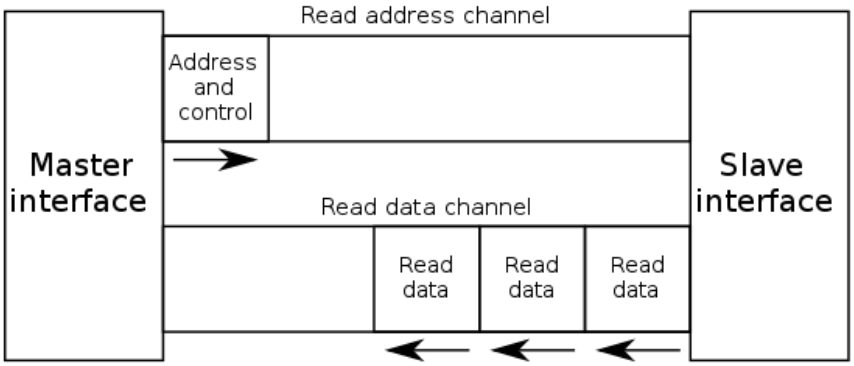

FIGURE 5 – Protocole de communication AXI

2.1.2 Compilateur `Merlin`

`Merlin` génère les bibliothèques des fonctions les Kernel accélérées/optimisées (en C ou C++) ainsi que leurs fichiers binaires pour programmer le FPGA [5].

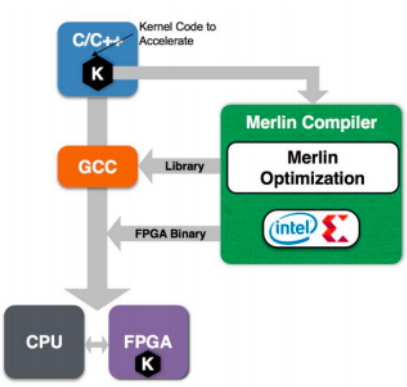

Figure 1 Merlin Compiler

FIGURE 6 – Merlin Compiler User Guide V2.2.4 September 12, 2018

Pour la partie `dataflow HLS`, l'application est exprimé sous forme d'un DFG avec des noeuds de computations des canaux de communications entres eux. Pour les communication entres ces noeuds, les canaux sont de types synchrones, et sont définis sous forme de `buffer FIFO`. L'utilisation du compilateur `Merlin`

pour des application en Go, nécessite une interface qui permet de *parser* le programme vers une arbre de syntaxe abstraite (AST) avant de le synthétiser au sein du FPGA, le `parser` utilisé ici est celui du OpenCL :

```
$ make mcc_bitgen
merlincc vec_add_kernel.mco -o vec_add_kernel_20180813_094029.aocx  --
platform=aocl::a10gx

Generating vec_add_kernel_20180813_094029...

aocl 17.1.1.273 (Intel(R) FPGA SDK for OpenCL(TM), Version 17.1.1 Build 273,
Copyright (C) 2017 Intel Corporation)

aoc: Environment checks are completed successfully.
You are now compiling the full flow!!
aoc: Selected target board a10gx
aoc: Running OpenCL parser....
aoc: OpenCL parser completed successfully.
aoc: Compiling....
aoc: Linking with IP library ...
aoc: First stage compilation completed successfully.
aoc: Hardware generation completed successfully.
*****Warning: This feature has been deprecated. It will be
removed in the next release. Please see reports/report.html
in the project directory for the new report.

Area report successfully created: kernel_top.aoco-area-report.html
```

FIGURE 7 – Merlin Compiler User Guide V2.2.4 September 12, 2018

2.2 Travail effectué

2.2.1 Implementation *« Hardware »* du canal

La figure ci_dessous montre le circuit de contrôle pour un canal de communication limité à un échange d'input/output à la fois [1]. Les signaux de contrôle :

- `Start / Finish` :
  Chaque ensemble *input/output*, dispose d'un pair de signal *Start/Finish*. *« Start »*, permets de déclencher l'exécution du sous circuit relative à la bascule « SR », afin de signaler l'existence d'une demande de communication *(en attente)*. La porte ou est une optimisation qui permets de gagner un cycle d'horloge pour l'exécution de la dernière communication en attente.
- `Ready / Transfer` :
  Le signal **« `Ready` »** est activé une fois qu'une demande de demande de faire un **« input »** du canal. Une fois qu'une demande de récupérer un **« output »** du canal existe, le signal **« `Transfer` »** s'active si, et seulement si, une demande d'écriture et une demande de lecture du/dans

1. Code VHDL en annexe

le même canal sont prêtes. Le circuit garantit l'exécution qu'une seule opération « Lecture / Ecriture » à un temps « t ». le signal « `Transfer` » a pour rôle aussi de mettre a zéro les signaux de contrôle des bascules qui initient le transfert.

— `Reg_Load` :
« `Reg_Load` » n'est pertinent que pour la partie « **Input** » du canal, et permets d'activer le chargement des données dans un registre destinataire.

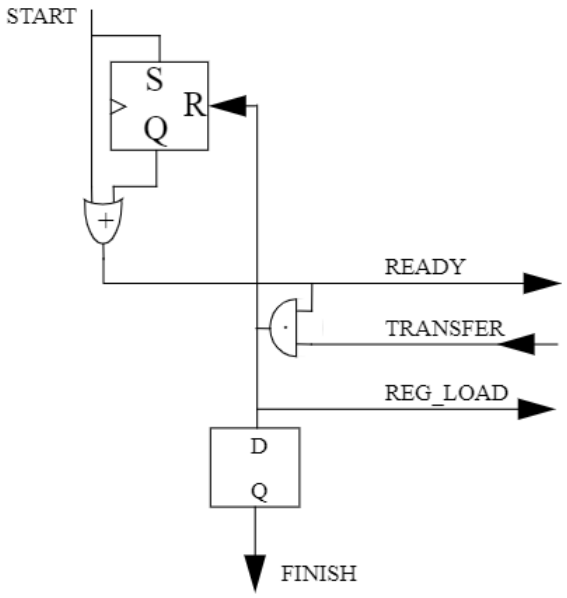

FIGURE 8 – Implementation « Hardware » d'un canal (*« coté Input »*)

2.2.2 AXI4-Stream FIFO

En se basant sur le protocole AXI4 ainsi que l'interface AXI Streaming, l'IP [2] *AXI4-streaming FIFO* permets d'abstraire la complexité du protocole et fournir une « API » simple à utiliser. L'implantation de ce protocole ce fait à travers la bibliothèque / interface C++, fournit par « `Vitis_HLS` » *« hls_stream<> »* [3]. Cette Classe respecte les caractéristique suivant :

— hls_stream<> se comporte comme un « FIFO » avec une capacité infinie.
— La lecture d'une donnée est consommatrice.

2. intellectual property
3. https ://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/hls_stream_library.html

— Les opérations de lectures et d'écritures sont bloquantes [4].

Ces caractéristiques s'alignent parfaitement avec celles d'une application programmée en « Go », de ce fait la bibliothèque « hls_stream<> » peut être utilisée comme interface AXI-Stream [5]. Une tentative d'écriture sur un canal « full ». de lire d'un canal « empty » entraîne un blocage des transactions au niveau **« RTL »** (envoie ou réception des données par exemple) [6].

Language	déclaration de canal	envoie	réception
Golang	input := make(chan int)	output <- v	v := <-input
Vitis_HLS	INT_STREAM input	output->write(v)||output.wirte(v)	input->read(v)||input.read(v)

2.2.3 Mise en oeuvre

L'application repose principalement sur le protocole AXI afin de garantir la synchronisation des échanges entre les différents noeuds, l'interface Stream FIFO offre une implémentation d'un canal disposant d'une capacité « *infini* » pour ce qui est co-simulation en C (la capacité du canal serait adaptée vis à vis les données qui circulent dans celle-ci lors de la synthèse).

```
#include <stdlib.h>
#include "ap_axi_sdata.h"
#include "hls_stream.h"

struct ap_axis{
    ap_int<D> data;
    ap_unit<D/8> keep;
    ap_uint<D/8> strb;
    ap_uint<U> user;
    ap_unit<1> last;
    ap_unit<TI> id;
    ap_unit<TD> dest;
}
```

FIGURE 9 – Les attributs de la structure ap_axis

La donnée circulant dans le canal doit respecter le type qu'on a définit sur l'interface : *« AXI_ VALUE »*(une type définit qui renvoie à la structure

4. Possibilité de les rendre non-bloquantes
5. AXI-Stream ne supporte pas les communications non bloquantes
6. voir annexe 4

« api_ axis » proposé par vitis_hls). Le canal en question, circule un *Stream* de donnée de type *« AXI_ VALUE »*.

Avec cette structure, non seulement on dispose de la possibilité d'envoyer des données, mais aussi de savoir si la donnée envoyé représente, par exemple, le dernier pixel ou la dernières trame. En effet l'attribut **« last »** désigne le dernier élément récupéré d'un canal.

Les opérations de lecture d'un canal se font avec la méthode `« read »` de la bibliothèque `HLS_Stream` [7] :

— **my_stream.read(dst_var) ;**

Les opérations d'écritures dans un canal se font avec la méthode `« write »` de la même bibliothèque :

— **my_stream.write(src_var) ;**

`Architecture « pipeline »` [8] :

La fonction *application* consiste a implanter une architecture pipeliné qui se compose de trois noeuds :

— Noeud A : effectue une simple multiplication par 2 des données qu'il reçoit dans son canal d'entrée.
— Noeud B : incrémente deux fois la valeur qu'il reçoit dans son canal d'entrée .
— Noeud C : incrémente dune seul fois la valeur qu'il reçoit dans son canal d'entrée.

7. https ://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc
8. Annexe 5

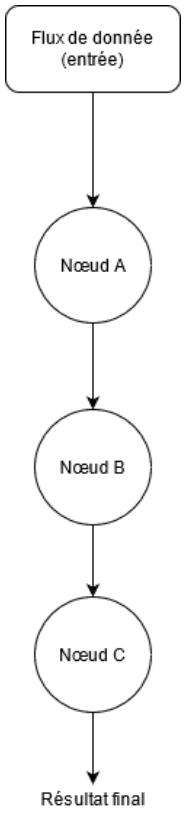

FIGURE 10 – L'architecture de l'application implémentée

`Architecture « broadcast »`[9] :

Dans cette partie, l'application consiste à diffuser des information reçu par un noeuds à un ou plusieurs noeuds :

9. Annexe 6

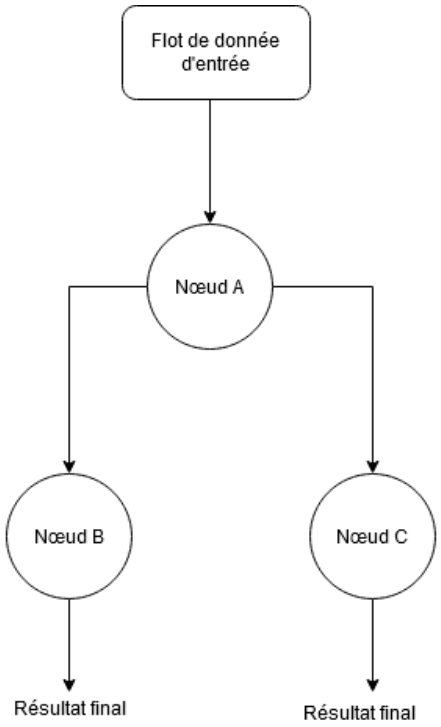

FIGURE 11 – L'architecture d'une application de diffusion

La fonction en question consiste à diffuser sont résultat à deux ou plusieurs noeuds. pour ce faire, l'aspect « Consommation » du échanges via les stream FIFO de type AXI, oblige le fait d'enregistrer les données avant leurs envoie *« buffers »* pour enregistrer les données avant leurs envoie.

Architecture « composée »[10] :

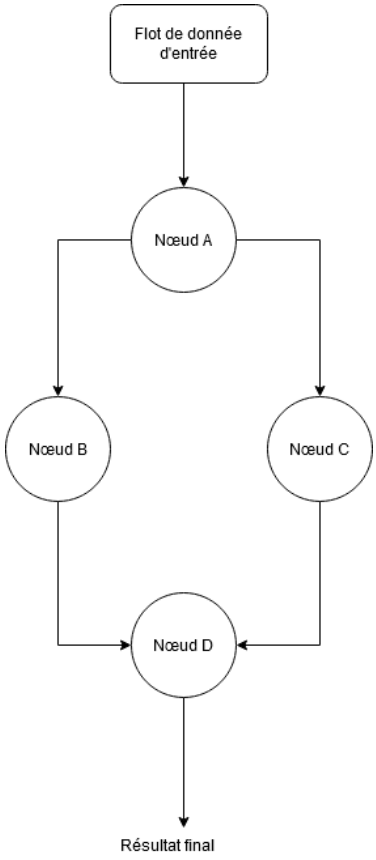

FIGURE 12 – L'architecture d'une application composée

la figure ci-dessus renvoie à l'architecture implanter par l'application, elle se compose de quatre noeuds, chacun effectue un traitement spécifique, **le noeud A** diffuse les données reçues d'une mémoire vers **les noeuds B et C**, chacun de ces noeud effectue une opération puis envoie son résultat au **noeud D**, ce dernier effectue une simple opération d'addition avant d'écrire le résultat final sur son canal de sortie.

10. Annexe 7

3 Conclusion :

Tout au long de ce rapport, nous n'avons traité que l'aspect qui concerne la synthétisions « des canaux » de go vers un langage de haut niveau à savoir le C++ avec les outils proposés par Xilinx [6].

Nous avons vu les différents services offert par Xilinx, qui nous permets de traiter la problématique de la synchronisation des échanges en utilisant des canaux synchrones. Bien que nous avons établie plusieurs tests en se basant sur des architectures Go différentes, il reste à traiter les différents problématique lié au *parsing* du code Go, la conversion et sa compilation. En effet, ce projet ne traite qu'une partie lié au déploiement ce qui laisse plusieurs axes de recherches liés au *proto-typing, parsing, deploiement .... .*

4 Annexes

Annexe 1 :

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


entity input is
Port (
    w_clk : in std_logic;
    start : in std_logic;
    transfer : in std_logic;
    ready : inout std_logic;
    finish : inout std_logic;
    reg_load : inout std_logic
);

end input;

architecture Behavioral of input is
signal  sig_q : std_logic;
signal r_d : std_logic;
component FF
    port (
        ff_clk : in std_logic;
        set : in std_logic;
        reset : in std_logic:='0';
        q : inout std_logic
    );
end component;

begin
    FlipFlop: entity work.FF(Behavioral) port map (
            ff_clk => w_clk,
            set => start,
            reset => r_d,
            q => sig_q
    );
    latch_D : entity work.FF(Behavioral) port map (
            ff_clk => w_clk,
            set => r_d,
            q => finish
    );
    Input : process(start, transfer)
    begin
        ready <= start or sig_q ;
        r_d <= ready and transfer;
        reg_load <= ready and transfer;

    end process;
end Behavioral;
```

FIGURE 13 – Implementation « VHDL » de la partie Input d'un canal

Annexe 2 :

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity output is
Port (
    r_clk : in std_logic;
    start : in std_logic;
    transfer : in std_logic;
    ready : inout std_logic;
    finish : inout std_logic
);

end output;

architecture Behavioral of output is
signal  sig_q : std_logic;
signal r_d : std_logic;
component FF
    port (
        ff_clk : std_logic;
        set : in std_logic;
        reset : in std_logic:='0';
        q : inout std_logic
    );
end component;

begin
    FlipFlop: entity work.FF(Behavioral) port map (
            ff_clk => r_clk,
            set => start,
            reset => r_d,
            q => sig_q
    );
    latch_D : entity work.FF(Behavioral) port map (
            ff_clk => r_clk,
            set => r_d,
            q => finish
    );
    Input : process(start, transfer)
    begin
        ready <= start or sig_q ;
        r_d <= ready and transfer;

    end process;
end Behavioral;
```

FIGURE 14 – Implementation « VHDL » de la partie Output d'un canal

Annexe 3 :

```
#ifndef _AXI_STREAM_EXAMPLE_H
#define _AXI_STREAM_EXAMPLE_H

#include <stdlib.h>
#include "ap_axi_sdata.h"
#include "hls_stream.h"

typedef ap_axis<32,2,1,1> INT_VALUE;
typedef hls::stream <INT_VALUE> INT_STREAM;

#define NUM 10

void print_chan(INT_STREAM* channel);
int initStream(INT_STREAM* streamData);
int traitementA(INT_STREAM* streamData, INT_STREAM* chan);
int traitementB(INT_STREAM* cha, INT_STREAM* data);
int traitementC(INT_STREAM* cha, INT_STREAM* res);
int application(INT_STREAM*  input, INT_STREAM* output );

#endif
```

FIGURE 15 – L'interface de l'API utilisée pour implémenter un communication synchrone en C++

Annexe 4 :

```
WARNING: Hls::stream 'hls::stream<ap_axis<32, 2, 1, 1> >.1' contains leftover data, which may result in RTL simulation hanging.
INFO: [SIM 211-1] CSim done with 0 errors.
INFO: [SIM 211-3] *************** CSIM finish ***************
Finished C simulation.
```

```
WARNING: Hls::stream 'hls::stream<ap_axis<32, 2, 1, 1> >.1' is read while empty, which may result in RTL simulation hanging.
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

FIGURE 16 – trace d'exécutions : Écriture sans lecture (en bleu), lecture sans écriture (en blanc)

Annexe 5 :

```
int application(INT_STREAM* input,INT_STREAM* output ){
    #pragma HLS DATAFLOW
    #pragma HLS INTERFACE axis port=input
    #pragma HLS INTERFACE axis port=output

    int i , ret;
    INT_STREAM chaB, chaC, buffer;
    // Initialisation des données
    initStream(input);
    printf("----------------------------Noeud A (traitementA )----------------------------\n");
    // traitementA
    traitementA(input, &chaB);
    printf("----------------------------Noeud B (traitementB )----------------------------\n");
    // traitementB
    traitementB(&chaB,&chaC);
    // traitementB
    printf("----------------------------Noeud C (traitementC)----------------------------\n");
    traitementC(&chaC, output);
    return EXIT_SUCCESS;

}
```

FIGURE 17 – code source de l'application (traitement pipeline) en C++

Annexe 6 :

```
#ifndef _AXI_STREAM_EXAMPLE_H
#define _AXI_STREAM_EXAMPLE_H

#include <stdlib.h>
#include "ap_axi_sdata.h"
#include "hls_stream.h"

typedef ap_axis<32,2,1,1> AXI_INT;
typedef hls::stream <AXI_INT> INT_STREAM;

#define NUM 10

int initStream(INT_STREAM* input);
int traitementA(INT_STREAM* input, INT_STREAM* output1, INT_STREAM* output2);
int traitementB(INT_STREAM* input, INT_STREAM* output);
void print_chan(INT_STREAM* input);
int application(INT_STREAM* input, INT_STREAM* output1, INT_STREAM* output2);

#endif
```

FIGURE 18 – code source de l'interface de l'application (broadcast) en C++

Annexe 7 :

```
#ifndef _AXI_STREAM_EXAMPLE_H
#define _AXI_STREAM_EXAMPLE_H

#include <stdlib.h>
#include "ap_axi_sdata.h"
#include "hls_stream.h"

typedef ap_axis<32,2,1,1> AXI_INT;
typedef hls::stream <AXI_INT> INT_STREAM;

#define NUM 10

int initStream(INT_STREAM* input);
int traitementA(INT_STREAM* input, INT_STREAM* output1, INT_STREAM* output2);
int traitementB(INT_STREAM* input, INT_STREAM* output);
int traitementC(INT_STREAM* input, INT_STREAM* output);
int traitementD(INT_STREAM* input1,INT_STREAM* input2, INT_STREAM* output);
void print_chan(INT_STREAM* input);
int application(INT_STREAM* input, INT_STREAM* output);

#endif

```

FIGURE 19 – code source de l'interface de l'application composée en C++

Références

[1] [En ligne] URL : https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/hls_stream_library.html

[2] T. Cui, *A Go-to-FPGA Compilation Framework for Streaming Applications.* IEEE, 2017.

[3] W. L. Ian Page, *Compiling occam into Field-Programmable Gate Arrays.* researchgate, 1999.

[4] [En ligne] URL : http://docs.reconfigure.io/

[5] L. B. Cong, J., *High-level synthesis for fpgas : From pro-totyping to deployment.* IEEE Transactions, 2011.

[6] [En ligne] URL : https://github.com/Xilinx/Vitis-Tutorials