Institut **S**upérieur d'**I**nformatique
De **M**odélisation et de leurs
Applications
Complexe des Cézeaux
BP 125
63173 AUBIERE Cedex

Centre **E**uropéen pour la
Recherche **N**ucléaire
CH 1211
GENEVE 23
Switzerland

Rapport de stage de 2ème année d'école d'ingénieur
Option Informatique des Systèmes Embarqués

Conception d'un injecteur de données hardware

Présenté par Vincent Delord

Tuteur entreprise : Jean-Christophe Garnier
Tuteur ISIMA : Emmanuel Mesnard

Remerciements

Je tiens tout d’abord à remercier mon tuteur au CERN, Jean-Christophe Garnier, pour sa grande disponibilité, son soutien et la confiance qu’il m’a donné.

Je remercie également Emmanuel Mesnard pour son implication dans le rôle de tuteur de stage.

Je remercie tout particulièrement Niko Neufeld et John Evans pour leurs appuie dans mes démarches.

Enfin je remercie toute l’équipe LHCb Online pour m’avoir accueilli et pour avoir porté un intérêt à mon travail, et d’une manière générale toutes les personnes avec qui mes rapports furent enrichissants.

Résumé

L’expérience LHCb, menée dans le cadre du CERN, recueille un nombre extraordinaire de données. Le système d’acquisition de ces données est donc démesuré, entièrement dédié à cette tâche. Pour réaliser des tests sur ce système d’acquisition, hors expérience, il existe un injecteur de données qui permet de **simuler** le flot habituel. Dans l’optique d’une future optimisation du réseau de ce système en **Ethernet 10 gigabit**, le LHCb souhaite se doter d’un **injecteur** hardware permettant de fonctionner sur ce nouveau réseau et d’y tester différents types de **protocoles de communication** tels qu’IP, MEP et TCP. Cet injecteur est réalisé au moyen d’une carte de développement Altera munie d’un **FPGA** et de différentes interfaces de communications.

Mots clés : Ethernet 10Gb, Injecteur, Protocoles de communication, FPGA, simuler

Abstract

At CERN, experiments must gather huge amounts of data. The Data Acquisition System of the LHCb is therefore very large, and dedicated to this task. A **data injector** allows scientists to test this acquisition system between two Physics activities. Aiming at an upgrade of this network up to **10Gb Ethernet**, the LHCb wants a new hardware data injector which send **simulated** data on this new interface. This device must also be able to switch between different **network protocols** such as TCP and MEP to test some eventual upgrade of the actual protocol. This injector will work on an Altera Development board which contains an **FPGA** chip and many different high speed output interfaces.

Key words: 10Gb Ethernet, data injector, network protocols, FPGA, simulated.

Glossaire

Accélérateur : Un accélérateur est un tube sous vide dans lequel des particules sont accélérées par des champs électriques et dont la trajectoire est guidée par des champs magnétiques.

Antimatière : La matière est constituée de particules, l’antimatière d’antiparticules. Une antiparticule possède une charge électrique opposée à celle d’une particule. L’antimatière est maintenant rare dans l’univers alors que sa quantité était égale à la quantité de matière lors du Big Bang. Lorsque de l’antimatière rencontre de la matière elles s’annihilent pour se transformer totalement en énergie.

Architecture matérielle : Décrit l’agencement de composants électroniques ainsi que leur interaction. Par défaut de langage elle désigne aussi l’agencement interne des processeurs et autres microcontrôleurs.

Big Endian : Quand certains ordinateurs enregistrent un entier sur 32 bits en mémoire, par exemple 0xA0B70708 en notation hexadécimale, ils l'enregistrent dans des octets dans l'ordre qui suit : A0 B7 07 08, pour une structure de mémoire basée sur une unité atomique de 1 octet et un incrément d'adresse de 1 octet. Ainsi, l'octet de poids le plus fort (ici A0) est enregistré à l'adresse mémoire la plus petite, l'octet de poids inférieur (ici B7) est enregistré à l'adresse mémoire suivante et ainsi de suit

Bus : Ensemble de liaisons physique qui peuvent être exploitées en commun par plusieurs éléments pour communiquer.

Chronogramme : Frise chronologique permettant de suivre les variations de plusieurs sorties logique d’une architecture.

Désencapsulation : Procédé inverse de l’encapsulation qui consiste à enlever l’entête d’un paquet, interpréter les informations qu’il contient et transmettre les donnée seules à l’utilisateur de sorte que la gestion de la transmission lui soit totalement transparente.

Evènement :

Encapsulation : Procédé consistant à accompagner des données destinées à transiter sur un réseau d’un entête contenant des informations propre à son transport.

FIFO : L'acronyme FIFO est l'abréviation de l'expression anglaise First In, first Out, que l'on peut traduire par « premier arrivé, premier servi » (littéralement « premier entré, premier sorti »). Ce terme est employé en informatique pour décrire une méthode de traitement des données. Cette méthode correspond à une méthode de traitement des éléments d'une file (calculs d'un ordinateur, stocks). Par extension il désigne est composants mémoire qui gèrent leurs données selon cette méthode.

FPGA (Field-Programmable Gate Array) : Circuit intégré reprogrammable, capable de réaliser des traitements complexes.

JTAG (Join Test Action Group) : Utilisé au lieu du terme générique Boundary-Scan, c’est une norme pour faciliter et automatiser les tests sur les circuits intégrés. Il a désormais d’autres applications comme la programmation des composants logiques programmables (FPGA).

Little Endian : Les autres ordinateurs enregistrent 0xA0B70708 dans l'ordre suivant : 08 07 B7 A0 (pour une structure de mémoire basée sur une unité atomique de 1 octet et d'un incrément d'adresse de 1 octet), c'est-à-dire avec l'octet de poids le plus faible en premier

Méson : Particule non élémentaire composée d’un nombre pair de quarks et d’antiquarks.

Module : Un module est une architecture dédié à une fonction précise comprise dans une architecture plus générale.

Multiplexeur : Composant logique qui permet de sélectionner parmi plusieurs entrées celle qui sera connectée à la sortie.

PLL : Phase Locked Loop, en français, Boucle à phase asservie, est un montage électronique permettant d'asservir la phase instantanée de sortie sur la phase instantanée d'entrée, mais elle permet aussi d'asservir une fréquence de sortie sur un multiple de la fréquence d'entrée.

Quark : Le constituant élémentaire de la matière. Il existe plusieurs quarks ayant pour propriétés une fraction de charge électrique élémentaire. Ainsi deux quarks de charges +2/3 et un quark de charge -1/3 forment un proton.

Remplissage : (Padding en anglais) Procéder consistant à compléter le corps d’un paquet avec des « 0 » jusqu’à atteindre la taille minimale de ce dernier.

Unité de Contrôle (UC) : Sous partie d’une architecture chargée de piloter l’unité de traitement. Elle gère l’ordonnancement des différentes tâches et pilote leurs réalisations par l’unité de traitement.

Unité de Traitement (UT) : Ensemble d’éléments de stockage, de traitement de l’information et de calcul d’une architecture matérielle.

Contents

Table des illustrations

Introduction

Le CERN (Organisation Européenne pour la Recherche Nucléaire) fournit aux scientifiques les moyens de réaliser des expériences très poussées dans le domaine de la physique des particules. Quatre expériences s'articulent autour de l'accélérateur de particules LHC et étudient ce qui se passe lors de collisions de particules. Chaque expérience repose en premier lieu sur un détecteur qui collecte une grande quantité de données qu'il faut analyser et enregistrer. Une partie de l'analyse est effectuée alors que les données transitent sur le système d'acquisition pour réduire le volume à enregistrer.

Pour réaliser des tests fonctionnels sur le système d'acquisition du LHCb hors expérience réelle, ce dernier dispose d'un injecteur de données. Cet injecteur simule le flux de données qui est sensé transiter par le système d'acquisition et se comporte, aux yeux des autres composants de ce système, comme des cartes d'acquisition de données.

Dans l'optique d'une amélioration future du réseau de ce système d'acquisition, qui passerait de l'Ethernet 1 Gigabit à l'Ethernet 10 Gigabit, les informaticiens de l'équipe LHCb Online envisagent aussi faire évoluer le protocole réseau vers TCP pour permettre plus de contrôle de flux.

Dans un premier temps il faudra donc réaliser un injecteur hardware capable de s'interfacer avec un réseau Ethernet 10 Gigabit. Ensuite il faudra qu'il gère différents protocole de couche application pour permettre de réaliser des comparatifs.

1. Présentation générale

1.1 Le CERN

Le CERN est l'Organisation Européenne pour la Recherche Nucléaire. C'est le plus grand centre de recherche en physique des particules du monde et il se situe sur la frontière franco-suisse. Il est le résultat d'un des premiers projets de coopération européenne et compte de nos jours 20 états membres européens.

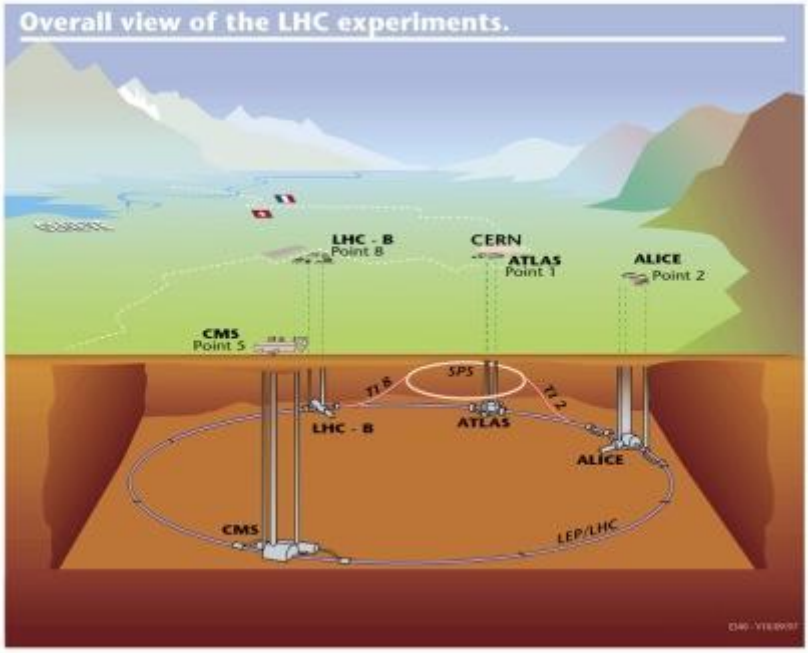


Figure 1 : Vue d'ensemble du LHC

De nombreux autres Etats participent aussi à ses expériences. On trouve ainsi 6500 scientifiques, de 80 nationalités différentes, représentant environs 500 universités et instituts de par le monde. Ces scientifiques sont appuyés dans leurs tâches par environs 2500 personnes, ingénieurs, techniciens, ouvriers, administrateurs ou secrétaires, qui sont chargés de concevoir et de construire les appareils sophistiqués mis en œuvre dans chaque expérience.

Le CERN permet aussi à qui le souhaite, au travers d'internet, d'apporter sa contribution au travail de traitement des données. Cela va du super calculateur d'université à votre ordinateur de bureau.

1.2 L'expérience LHCb

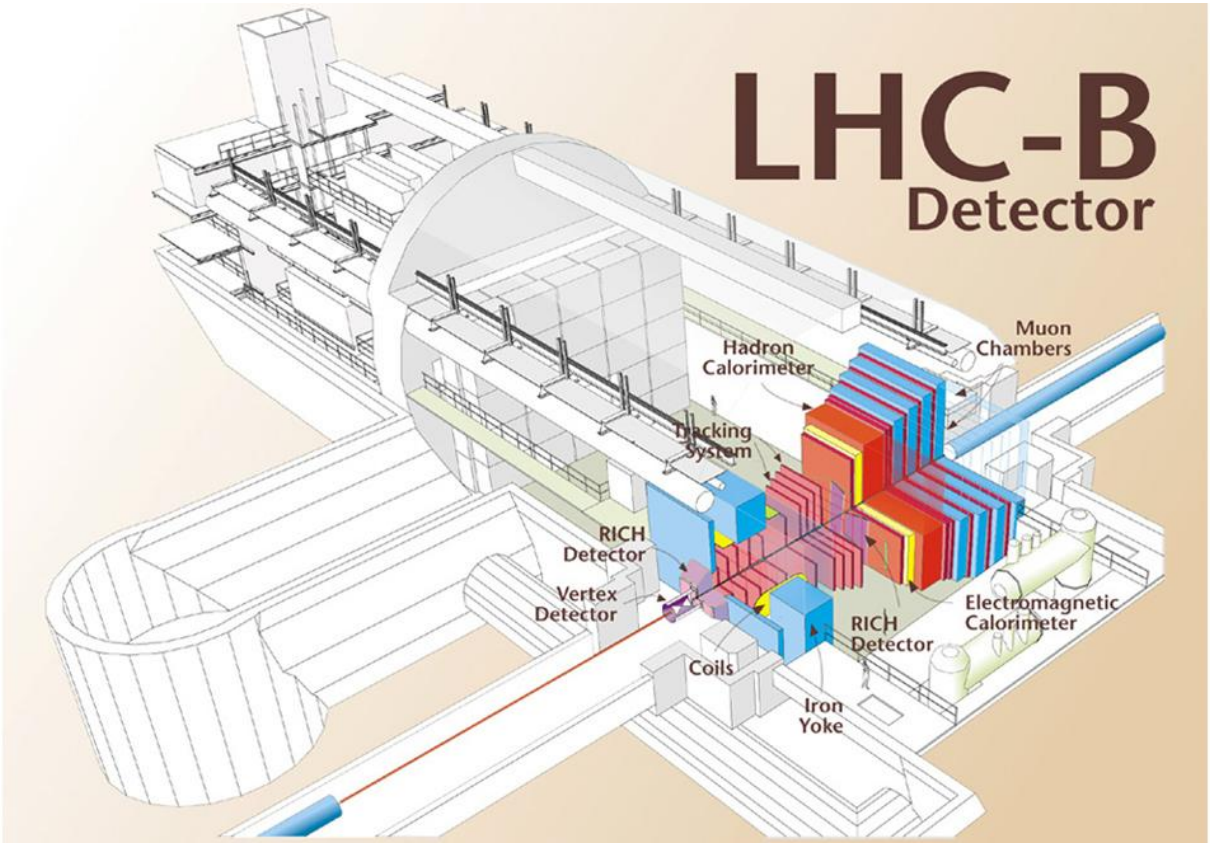


Figure 2 : Schéma du détecteur LHCb

L'acronyme LHCb signifie Large Hadron Collider beauty. Ce dernier mot « beauty » vient du nom de la particule que ce détecteur va étudier, le méson B.

L'univers est en apparence formé uniquement de matière, de particules. Les physiciens pensent que lors du Big Bang, la matière et l'antimatière ont été créées dans des proportions égales mais ils ne savent pas expliquer ce que serait devenue l'antimatière.

La rencontre entre une particule et une antiparticule, entre la matière et l'antimatière, annihile les deux masses pour les convertir en énergie pure selon célèbre formule $E=mc^2$. Comment expliquer alors la présence plus importante de matière que d'antimatière dans l'univers ?

La collision entre deux protons permet de produire de nombreuses particules. Les scientifiques du LHCb prévoient de détecter des mésons B et d'observer leur désintégration. En effet un méson est composé d'un nombre égal de quark et d'antiquark (les quarks étant les particules élémentaires composants les électrons, les protons et les neutrons). Ainsi, en comprenant comment, lors de la décomposition d'un méson, la matière l'emporte sur l'antimatière, ils pourront expliquer par extension pourquoi l'univers n'est composé que de matière.

Le détecteur est composé de nombreux sous-détecteurs, chacun est destiné à déterminer un type de particule en fonction de leurs propriétés physiques :

- VELO (VErtex LOcator) : situé dans l'axe du faisceau, ce détecteur permet de fournir des mesures précises de la trajectoire des particules à proximité de la zone de collision. Il est également utilisé pour reproduire la création et l'affaiblissement des mésons Beauté et Charme afin de fournir des données précises de leur durée de vie.

- RICH 1&2 : les deux « Ring Image CHerenkov counters » permettent d'identifier les particules chargées passant dans le détecteur, ainsi que leur quantité de mouvement.

- Spectromètre : dipôle magnétique situé à proximité de la zone de la collision pour en minimiser la taille.

- Tracking System : ce détecteur détermine la trajectoire des particules entre le VELO et les calorimètres, ainsi que la quantité de mouvement des particules.

- Calorimètres : en absorbant l'énergie des particules, les calorimètres permettent d'identifier les hadrons, les électrons et les photons, qui sont parfois le résultat de l'affaiblissement des mésons B. Ils fournissent aussi des informations sur leur énergie.

- Muon : ce détecteur utilise la capacité de pénétration des muons pour les détecter précisément. Les muons sont aussi un résultat de l'affaiblissement des mésons B.

Tous ces détecteurs envoient leurs données brutes sur un réseau Ethernet chargé de les prétraiter et de les stocker pour ensuite permettre aux physiciens de les analyser, c'est le système d'acquisition.

1.3 Le système d'acquisition

Le système DAQ (Data AcQuisition) s'étend de l'électronique qui interface les sous-détecteurs de l'expérience LHCb jusqu'au système de stockage des données potentiellement intéressantes. Tout au long de ce système, les données sont sélectionnées par différents mécanismes pour ne conserver que ce qui est potentiellement pertinent et intéressant pour l'expérience. On différencie ainsi LHCb Online, qui regroupe tout ce qui concerne le traitement effectué au cours de l'acquisition, de LHCb Offline, qui regroupe donc le traitement effectué sur les données stockées.

Le LHC produira des collisions, ou événements, à une fréquence de 40 MHz. Seulement les collisions intéressantes ne seront que de l'ordre de 10 MHz, mais seulement 1% de ces collisions produiront des paires de quark et d'antiquark Beauty. De plus, seulement 15% de ces collisions produiront des mésons B, dont la désintégration en sera intéressante que dans un cas pour mille. Au final, la fréquence des collisions intéressantes sera de 15 Hz.

Comme il n'existe aucune technologie permettant de transférer et de stocker les informations relatives à toutes les collisions, qui représentent tout de même 40 To/s, le LHCb a mis au point, conjointement au système DAQ, un système de sélection des données, le Trigger System. Ce système de sélection est aussi bien basé sur de l'électronique que sur du logiciel. Ses différents niveaux de sélection permettent de déterminer quels sont les événements à conserver. La première sélection s'effectue au niveau du L0 Trigger. Les calorimètres et le détecteur MUON sont les plus rapides et permettent de déterminer si la collision est potentiellement intéressante. La fréquence des événements acquis devient donc de 1 MHz à la sortie du L0 Trigger.

Après L0, les données sont envoyées vers les cartes TELL1, le composant principal de ce sélecteur.

Supervisée, elle effectue une sélection, elle met en mémoire tampon les données, puis elle les transfère via un réseau IP assez complexe vers les ordinateurs qui appliqueront ensuite des algorithmes de sélections pour encore réduire la fréquence d'événements intéressants et donc réduire la taille des données à mémoriser. Cette ferme de calcul correspond au HLT (High Level Trigger).

L'ensemble du DAQ est supervisé par un chef d'orchestre qui s'occupe de synchroniser les différents modules et de gérer la consommation des ressources mémoire et CPU des différentes fermes. Ce superviseur est le TFC (Time and Fast Control).

L'injecteur de données remplace les cartes TELL1 sur le réseau du DAQ et envoie des données fictives sur celui-ci.

1.4 Réseau Ethernet et TFC

Dans cette partie, nous allons plus en détail le fonctionnement du réseau Ethernet du DAQ ainsi que le rôle du TFC au sein de celui-ci. La connaissance des différents protocoles et voies de communication est nécessaire à la compréhension du cahier des charges du projet.

Les données transitent depuis les cartes TELL1 jusqu'au fermes de calcul par le biais d'un protocole à 3 couches, Liaison / Réseau / Transport (modèle OSI). Dans un premier temps les données brutes sont conditionnées en paquets MEP. Ce paquet se compose d'un en-tête de 12 Octets et d'un corps. Ensuite, ce paquet MEP est encapsulé dans un ou plusieurs paquets IP selon sa taille. Ces paquets IP sont à leur tour encapsulés en paquets Ethernet puis envoyés sur le média physique.

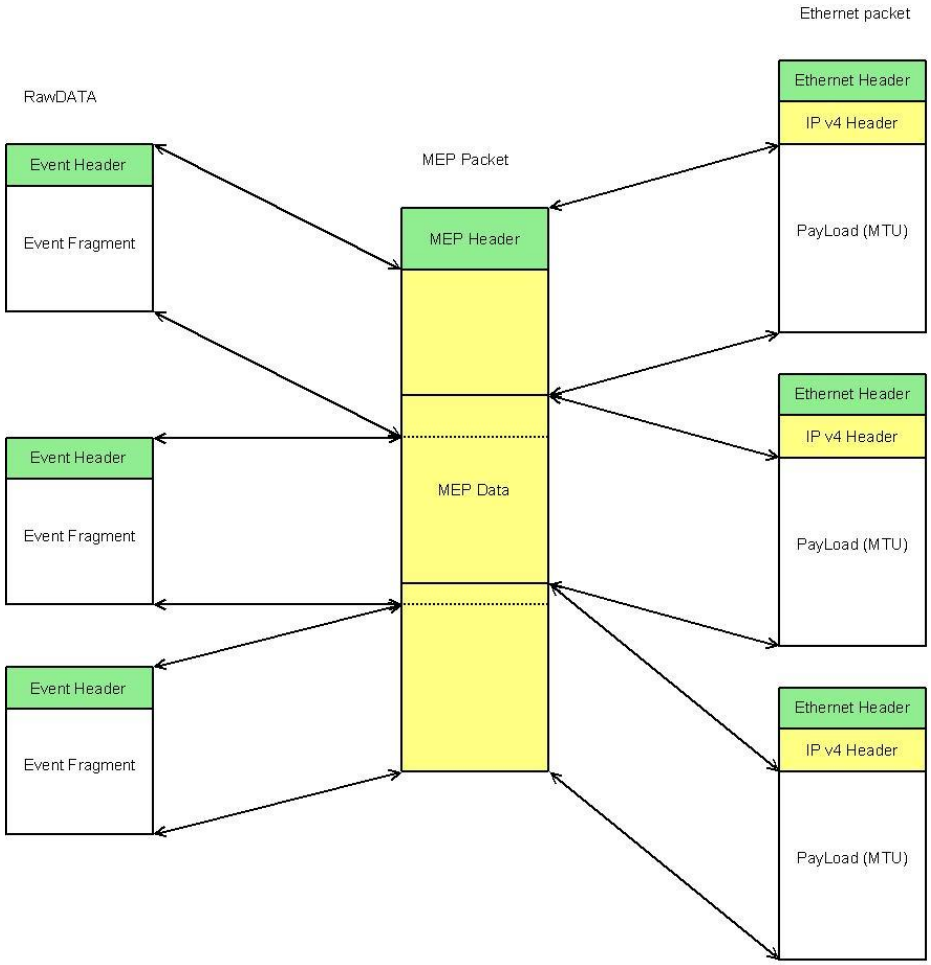


Figure 3 : Empaquetage des données

Les en-têtes des différents types de paquet sont tous construit grâce aux informations provenant du TFC sauf certains qui doivent être calculé par le pilote de l'interface réseau.

Le TFC quant à lui, communique ses informations par un autre type de réseau. Comme il est entre autre chargé de gérer les flux de données ainsi que de synchroniser les différentes partie du DAQ, il est doté d'un réseau de communication dédié à très haute vitesse en fibres optiques. Ce réseau est appelé TTC (Timing, trigger and control).

2. Analyse fonctionnelle

Après avoir compris dans les grandes lignes le fonctionnement du DAQ et compris la fonction précise de l'injecteur de donnée, il a fallu dans un premier temps définir exactement toutes les fonctions que devrait remplir l'injecteur hardware, les outils à ma disposition pour le réaliser, ainsi que les mécanismes nécessaires à son intégration dans le DAQ.

2.1 Cahier des charges

Il existe actuellement au CERN un injecteur de données. C'est un injecteur software qui fonctionne sur le réseau actuel, Ethernet 1 Gigabit.

Cet injecteur étant un software il n'a pas accès à l'interface TTC du TFC. Pour palier à ce manque il reçoit certaines informations via le réseau Ethernet et doit reconstituer les autres lui-même.

Dans le futur, l'expérience LHCb prévoie de se doter d'un réseau Ethernet 10 Gigabit Ethernet. A l'occasion de cette mise à neuf du réseau, ils envisagent de reconsidérer leurs choix de protocole de communication. En effet le protocole MEP est très basique et ne permet aucun contrôle de flux. Ils envisagent donc de le remplacer par un protocole dérivé du TCP.

Ainsi, l'expérience LHCb souhaite se doter d'un nouvel injecteur de donnée capable de recevoir les informations du TFC via le TTC, capable de produire des données sur un réseau 10 Gigabit Ethernet et permettant de choisir différents types de protocole de la couche transport.

Pour des raisons de vitesse de fonctionnement et d'interfaçage matériel avec la carte de réception du TTC ce nouvel injecteur devra être non plus software mais hardware.

Il sera programmé sur une carte autonome munie d'une puce FPGA (Réseau de portes programmables). C'est un processeur qui, au lieu de posséder une architecture fixe, définie lors de la construction de la puce, peut être programmé et reprogrammé à volonté via un logiciel de description architecturale et une interface USB qui permet de charger l'architecture souhaitée sur la puce. De plus cette carte devra comprendre une interface grande vitesse permettant de gérer l'Ethernet 10 Gigabit et la carte d'acquisition optique, ainsi qu'un port permettant de communiquer avec un espace de stockage contenant les données.

Etant le premier étudiant à travailler sur ce projet, il m'a été demandé, lors de mon séjour au CERN, d'avancer au maximum sa conception.

2.2 Outils et technologies

Pour réalisé ce projet, j'ai eu à ma disposition, un kit de développement Altera Arria GX. Ce kit comporte une carte Arria GX PCI Express et un logiciel de description architecturale et de programmation de FPGA, Altera Quartus II v8.2 ainsi qu'un logiciel de simulation d'architecture Altera Modelsim.

La Carte possède un FPGA de type Arria GX possédant 2 banques de broches (sur 6) dédiées à la gestion de flux. C'est-à-dire qu'elle propose des broches haut débit pour les données, des alimentations typiques des couches physiques des réseaux Ethernet ainsi que des horloges de références offrant une plus grande fiabilité. Ces banques de broches sont routées vers une HSMC (Interface de communication haute vitesse). C'est sur cette HSMC que seront connecté la carte d'acquisition optique ainsi que la fiche destinée à recevoir le câble Ethernet 10 gigabit. De plus elle possède une interface PCI Express qui permet de communiquer avec un disque dur.

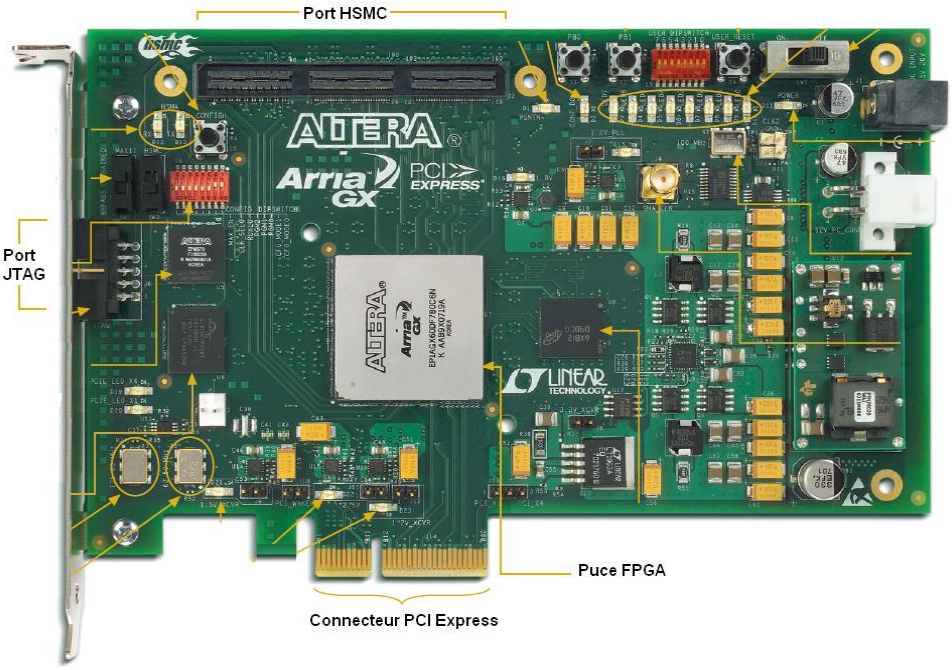


Figure 4 : Photo de la carte Altera

Pour la conception et la validation des architectures développées j'ai eu accès à deux logiciels de la compagnie Altera, Qartus II pour la conception et ModelSim pour la simulation.

Quartus II est un logiciel de conception d'architecture assisté. La création d'une architecture se déroule le plus souvent en 6 étapes.

- Dans un premier temps l'Unitée de Traitement (UT) de l'architecture est développée en mode « Schematics » grâce aux composants matériels disponibles dans la librairie Altera (FIFO, RAM, Multiplexeur, Bus, Bascules …).

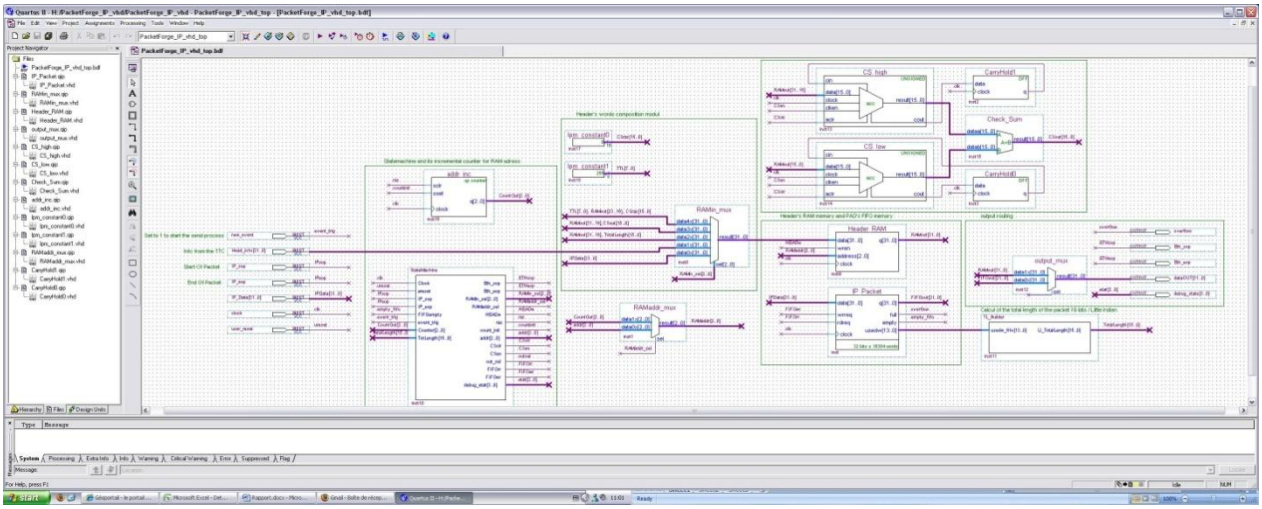


Figure 5 : Capture d'écran de Quartus II en mode schématique

- Ensuite l'Unitée de Contrôle (UC) de l'architecture est décrite sous la forme d'une machine d'état en VHDL, compilée et intégrée à la bibliothèque de composants. Ainsi, une fois terminée, elle peut être intégrée dans la feuille schématique de l'unité de traitement.

- L'architecture un fois créée, il faut la tester et la valider avec le logiciel ModelSim. Ce logiciel permet de piloter les valeurs en entrée de l'architecture et de suivre son comportement au travers d'un chronogramme.

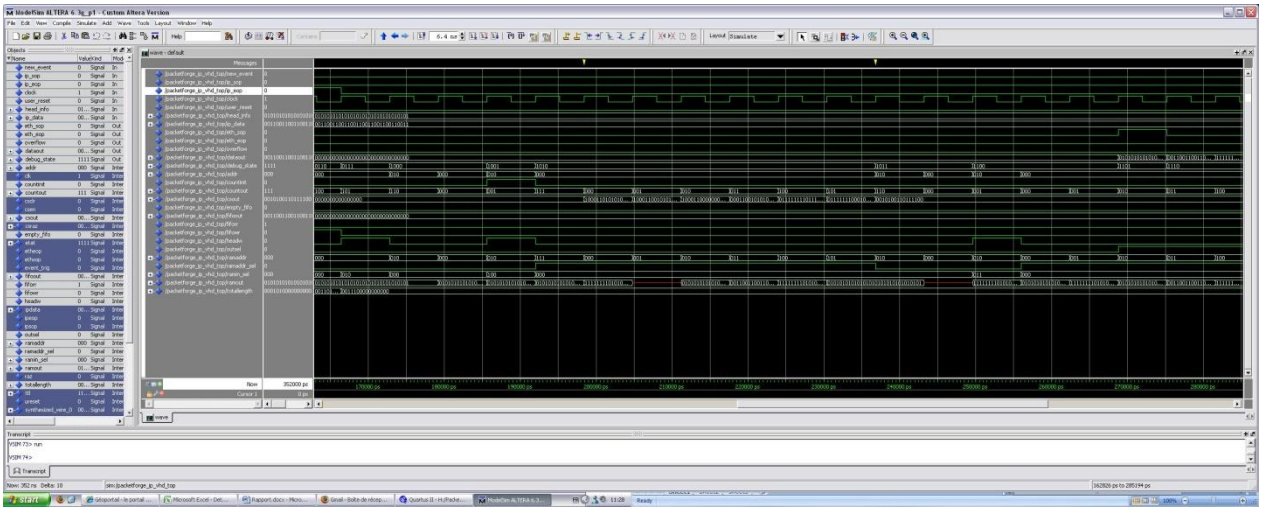


Figure 6 : Capture d'écran ModelSim

- Maintenant que l'architecture fonctionne, il faut associer les broches du FPGA aux entrée et au sorties puis mettre en place les PLL pour obtenir les horloges voulues.

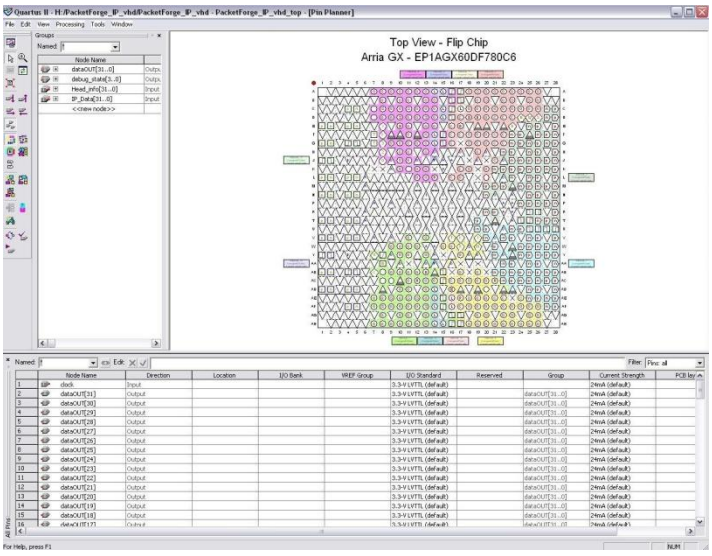


Figure 7 : Capture d'écran Quartus II mode association des broches

- Une fois ces associations effectuées la conception de l'architecture est terminée. On lance donc une compilation complète du projet. Durant cette étape, le logiciel établi le routage réel des portes du FPGA dans un fichier.

- Une fois ce fichier obtenu il ne reste plus qu'à programmer la puce par le biais du module USB.

2.3 Intégration dans le DAQ

Une fois le cahier des charges établi, j'ai du me familiariser en détail avec le fonctionnement de ce système d'acquisition. Cela passe par l'étude des différents protocoles réseau utilisés au CERN ainsi que par la compréhension des mécanismes de synchronisation de l'envoie des données. Je vais donc dans cette partie vous présenter les trois protocoles réseau avec lesquels j'ai pu travailler ainsi que les différentes étapes d'une simulation.

2.3.1 Protocoles de transmission des données

Dans un premier temps, je vais présenter les deux protocoles réseaux permettant d'encapsuler les données brutes émise par l'injecteur sur le réseau Ethernet. Ce sont les protocoles MEP (Multi Event Protocol), spécifiquement mis au point pour le système d'acquisition de l'expérience LHCb, et le protocole standard IP.

Le protocole MEP regroupe dans un paquet un certain nombre d'évènement s (NbEvent) et lui associe un en-tête de 12 octets comprenant, un identifiant unique, le nombre d'évènements compris dans le paquet, la longueur totale du paquet ainsi que l'identifiant de la partition à laquelle appartiennent ces évènements.

Voici un tableau où l'on peut voir en bleu ciel l'en-tête du paquet MEP suivit des données brut des différents événements.

L0ID/L1ID[7 :0]	L0ID/L1ID[15 :8]	L0ID/L1ID[23 :16]	L0ID/L1ID[31 :24]
NbEvent[7 :0]	NbEvent[15 :8]	TotLength[7 :0]	TotLength[15 :8]
PartID[7 :0]	PartID[15 :8]	PartID[23 :16]	PartID[31 :24]
EventID1[7 :0]	EventID1[15 :8]	Len1[7 :0]	Len1[15 :8]
Data[7 :0]	Data[15 :8]	…	…
EventID2[7 :0]	EventID2[15 :8]	Len2[7 :0]	Len2[15 :8]
Data[7 :0]	Data[15 :8]	…	…

L0ID/L1ID : 32 bits

Ce champ contient l'identifiant du premier évènement du paquet MEP. Ce champ est rempli par le TFC.

TotLength : 16 bits

Taille total du paquet MEP, en-tête inclus, en octets. Ce champ est rempli par le driver.

NbEvent : 16 bits

Nombre d'évènements contenus dans le paquet MEP. Ce champ est rempli par le TFC.

PartID : 32 bits

Définit à quelle partition ces données appartiennent. Ce champ ne change pas durant la totalité d'un « RUN », d'une série de collisions.

Un paquet MEP est ensuite encapsulé dans un ou plusieurs paquets IP comme décrit plus haut. Ce paquet IP est composé d'un corps d'une taille maximum de 8 Mo et d'un en-tête de 20 octets. Cet en-tête contient plusieurs champs dont, la longueur totale du paquet, son identifiant, dans le cas ou le paquet MEP est séparé en plusieurs paquets IP (fragmentation) il y a un bit permettant de savoir si le paquet et le dernier du fragment et un champ « décalage » (offset en anglais) qui permet de reconstituer le paquet MEP à sa réception, ensuite il y a la durée de vie du paquet sur le réseau ainsi que l'adresse source et l'adresse destinataire.

Voici un tableau récapitulatif présentant l'en-tête d'un paquet IP (en jaune) contenant le début d'un paquet MEP (en bleu ciel et rose).

Version/IHL	Type of service	TotLength[15 :8]	TotLength[7 :0]
ID[15 :8]	ID[7 :0]	Flags[2 :0]/FO[12 :8]	FragOffset[7 :0]
Time To Live	Protocol	CheckSum[15 :8]	CheckSum[7 :0]
IP-Source[31 :24]	IP-Source[23 :16]	IP-Source[15 :8]	IP-Source[7 :0]
IP-Dest[31 :24]	IP-Dest[23 :16]	IP-Dest[15 :8]	IP-Dest[7 :0]
L0ID/L1ID[7 :0]	L0ID/L1ID[15 :8]	L0ID/L1ID[23 :16]	L0ID/L1ID[31 :24]
NbEvent[7 :0]	NbEvent[15 :8]	TotLength[7 :0]	TotLength[15 :8]
PartID[7 :0]	PartID[15 :8]	PartID[23 :16]	PartID[31 :24]
EventID1[7 :0]	EventID1[15 :8]	Len1[7 :0]	Len1[15 :8]
Data[7 :0]	Data[15 :8]	…	…
EventID2[7 :0]	EventID2[15 :8]	Len2[7 :0]	Len2[15 :8]
Data[7 :0]	Data[15 :8]	…	…

Version : 4 bits

Version du protocole IP utilisé. Ce champs vaut 4 (IPv4) et est déterminé par le TFC.

IHL : 4 bits

Longueur de l'en-tête du paquet IP (Internet Header Length) en mots de 32 bits. Ce champs vaut 5 actuellement et est déterminé par le TFC.

Type of Service : 8 bits

Indique le type de service désiré pour ce paquet. Il diffère pour les paquets à destination du Level1 (L1 trigger) et ceux à destination des fermes (HLTrigger). Ce champs est rempli par le TFC.

TotLength : 16 bits

Longueur totale du paquet, en-tête IP inclus, mesuré en octets. Ce champs est calculé et rempli par le pilote.

ID : 16 bits

Identifiant du paquet IP. Dans le système d'acquisition du LHCb, ce champ est rempli avec les 15 bits de poids faible du champ L0ID/L1ID du paquet MEP et le bit de poids fort est mis à 0. Défini par le pilote.

Flags : 3 bits

Bit 0 : toujours nul.

Bit 1 : toujours nul.

Bit 2 : vaut 0 si ce paquet est le dernier d'une série de fragments (ou s'il est le seul fragment) et vaut 1 s'il y a d'autres fragments d'un même MEP. Défini par le pilote.

FragOffset : 13 bits

Ce champs permet de savoir où le fragment se situait dans le paquet d'origine. Le premier fragment a ainsi un décalage de 0 et les suivants ont des décalages croissants, mesurés en octets.

Time To Live : 8 bits

Temps en secondes durant lequel le paquet peut rester sur le réseau avant d'être rejeté. Mis à 0xFF par le pilote à l'envoi du paquet.

Protocol : 8 bits

Indique le protocole de la couche supérieure, ici MEP.

CheckSum : 16 bits

Ce champ comporte la « somme de contrôle » de l'en-tête du paquet. Elle est calculée en sommant tous les bits de l'en-tête sur 16 bits en remettant la retenue sortante sur la retenue entrante.

2.3.2 Déroulement d'une simulation

Lors d'un évènement normal, le TFC déclenche l'envoi des données depuis les 300 cartes TELL1 vers la fermes HLT qu'il à choisis selon ses disponibilités. Concrètement, le TFC envoie aux cartes TELL1 les informations nécessaires à la construction des différents en-têtes de paquet dont l'adresse IP destinataire de la ferme choisie.

Dans le cas de l'injecteur, celui-ci reçois les informations en provenance du TFC à la place des cartes TELL1. Cela déclenche l'envoie de 300 paquets MEP contenant les données simulées. Chaque paquet MEP doit posséder une adresse IP source différente, correspondant aux adresses IP des cartes TELL1 et non pas à l'adresse IP de l'injecteur, ceci dans le but de donner l'impression à la ferme de recevoir des données provenant des cartes TELL1.

2.4 Vue d'ensemble de l'injecteur

Maintenant que le cahier des charges du projet est défini et que les outils sont pris en mains, la conception de l'injecteur peut débuter. Dans un premier temps, il me faut définir dans les grandes lignes de quoi cette architecture sera composée.

L'injecteur doit être interfacé avec 3 différentes entités : le système de stockage des données, le TFC pour recevoir les ordres, et le réseau Ethernet des fermes de calcul vers lequel les données sont envoyées. J'ai choisi de me pencher sur la partie de l'architecture qui se charge d'envoyer les informations récoltées de la mémoire vers le réseau Ethernet. Voici un schéma qui présente l'architecture globale de l'injecteur avec les différentes fonctions qu'il doit réaliser.

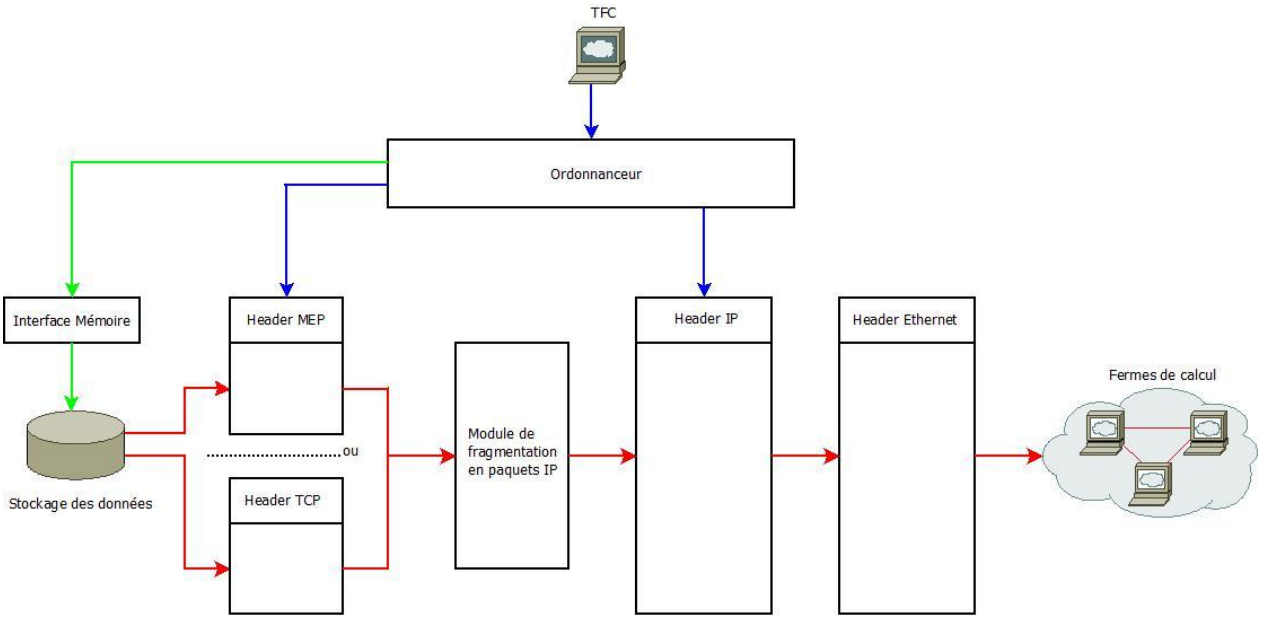


Figure 8 : Schéma d'ensemble de l'injecteur

Il m'a semblé logique de commencer par réaliser une architecture qui fonctionne sur le réseau actuel avant de la faire évoluer plus tard. Ainsi, pour facilité l'évolution future de l'architecture, j'ai réalisé plusieurs modules indépendants les uns des autres pour qu'ils puissent être remplacés sans effort.

Je devais donc réaliser 5 modules qui encapsulent les données provenant l'interface mémoire pour permettre leur diffusion sur le réseau. Ces cinq modules sont : le module MEP, le module de partitionnement, le module IP, le module Ethernet, et pour finir l'ordonnanceur de ces 4 premiers modules.

Seulement cette modularité à un revers, au cours de leurs transits de module en module, les données doivent être transmises puis stockées de nouveau dans le module suivant ce qui multiplie les espaces de stockage et les temps d'accès à ces derniers ralentissent l'architecture. Pour trouver un compromis j'ai donc cherché à éliminer les modules inutiles. Le module Ethernet et le module IP ne peuvent être rassemblés car le module Ethernet est forcément seul puisqu'il est licencié, il ne peut pas être modifié. Le module MEP doit être conservé puisqu'il doit pouvoir être remplacé par le module TCP. Par contre le module de partitionnement peut être éliminé. En effet il est facile de gérer le partitionnement directement à la sortie du module MEP. Cela implique qu'il faudra intégrer un nouveau module de partitionnement dans le futur module TCP.

Je vais donc vous détailler la réalisation de ces différents modules du plus proche de la couche physique, l'Ethernet, au plus haut niveau, le MEP et l'ordonnanceur.

3. Conception

Une fois l'analyse globale du sujet effectué et la prise en main des outils de développement terminé, j'ai pu me lancer dans la conception des différents modules qui composent l'architecture de cet injecteur matériel. Je vais vous présenter dans cette partie ceux que j'ai eu le temps de réaliser, à savoir, le module de communication Ethernet 10 gigabit, le module d'encapsulation IP, le module d'encapsulation MEP gérant aussi la fragmentation, pour finir par le module de gestion du TTC et ordonnanceur du système.

Au début de mon stage, pour me familiariser avec l'environnement Altera, j'ai commencé par développer le module de plus bas niveau : le module Ethernet.

3.1 Module Ethernet

3.1.1 Choix du module

Ce module Ethernet 10Gigabit demandant une utilisation optimale des ressources des composants Altera, il a été décidé que nous allions utiliser un module Ethernet développé par Altera et optimisé pour leurs cartes. En effet Quartus II, le logiciel de conception propose à l'utilisateur d'intégrer dans son architecture des IPs (Intellectual Properties / Propriétés Intellectuelles). Elles prennent la forme d'un composant normal sur les schémas de conception. La différence étant que le code source du composant ne peut-être ni édité ni consulté par l'utilisateur. Bien entendu l'utilisation prolongée de ces propriétés intellectuelles est soumis au paiement d'une licence.

Dans un premier temps, j'ai donc utilisé la version d'évaluation de différents composants permettant de gérer l'Ethernet 10 gigabit pour tester leurs performances sur le kit de développement ainsi que leurs compatibilité avec la norme physique choisie pour la connectique Ethernet.

En effet, si l'Ethernet que nous avons l'habitude d'utiliser (100M à 1Gbit) est standardisé en connectique RJ45, l'Ethernet 10 Gbit n'étant pas encore commercialisé à grande échelle aucun média physique n'a encore été normalisé et il en existe plusieurs qui se dispute la première place.

Pour ce projet, le CERN à choisi d'adopter la norme CX4. Ce type de média physique se compose de 4 bits de réception codés sur 8 broches, et 4 bits de transmission codés sur 8 broches. Pour chaque bit B on envoie la valeur de B sur la broche + et -B sur la broche - . Cette norme permet une plus grande fiabilité des données transmises du fait de la redondance des broches pour chaque bit et permet aussi d'envoyer et de recevoir des données en même temps.

Ainsi, pour être compatible avec ce type de connectique, le module Ethernet choisi doit proposer, dans ses options de configuration, la norme de codage physique des données XAUI (10Gbit Attachment Unit Interface) qui permet, entre autre, de gérer 4 bit de réception et 4 bits d'envoie de manière indépendante.

Une fois la propriété intellectuelle appropriée trouvée, j'ai pris contact avec le service de licence du CERN pour en obtenir les droits pour mon travail.

3.1.2 Implantation du composant

Pour tester la compatibilité des différents modules Ethernet que j'ai étudié, j'ai du comprendre leurs fonctionnement pour pouvoir les programmer sur le FPGA.

En lisant la documentation, on apprend que ce module Ethernet 10 Gb se compose de 3 grandes parties indépendantes :

- Une RAM permettant de configurer le module. Celle si permet de paramétrer des options telles que l'adresse MAC source, les requêtes ???, les trames de pauses, le remplissage et toutes les autres subtilités du protocole Ethernet.
- Un système d'envoi composé d'un système de calibration des horloges et d'une FIFO stockant les données à envoyer.
- Un système de réception qui désencapsule les données reçues et les affiche sur le bus de sortie.

Pour étudier son fonctionnement, j'ai tout d'abord réalisé une architecture simple qui permettait de le mettre en œuvre. Cette dernière était composée d'une PLL permettant de générer les bonnes fréquences d'horloges, nécessaire à son fonctionnement ainsi qu'une machine d'état qui se charger de l'initialisation des registres internes du module et de commander l'envoi d'un paquet type.

Une fois cette architecture théorique réalisée, le challenge fut de comprendre comment associer les entrée/sorties de l'architecture aux broches du processeur. Pour ce faire Quartus II dispose d'un outil d'association des broches assez intuitif, comme présenté sur la figure 6. Ainsi, Cette architecture minimaliste utilisait 19 broches du FPGA :

- Les 16 broches haut débit dédiées au transfert de données : 2*2*4 pour la norme XAUI, qui code 4 bits de réception et 4 bit d'envoi sur 2 broches chacun.

- Les 2 broches de l'horloge de référence provenant de la même banque que les broches de transfert. Cette horloge de référence est à transmettre directement au module et elle lui permet de gérer les fonctions d'envoi et de réception à la bonne vitesse.
- Une broche d'horloge standard qui après modification par la PLL sert à synchroniser les fonctions d'accès à la RAM de configuration.

Le système d'envoi des données doit fonctionner à la vitesse de l'horloge de référence et communiquer au module le début et la fin d'un paquet Ethernet. Cette communication se fait grâce à deux bits, l'un permet de signaler que le mot placé sur le bus de transmission est le premier d'un paquet, l'autre permet de signaler quel mot est le dernier du paquet. Il y a la possibilité de configurer le module de sorte qu'il vérifie que la taille du paquet soit correcte ou non.

La conception de cette petite architecture m'a permis de me familiariser avec le logiciel de conception et avec l'utilisation du FPGA.

Par la suite, je me suis largement inspiré du fonctionnement de ce module pour créer les miens (IP et MEP). En effet, je réutilise la méthode consistant à gérer sur deux bits indépendant le début et la fin d'un paquet au lieu de suivre mon idée originale d'utiliser 2 séries de bits dans le flot de donnée lui-même pour marquer le début et la fin de l'envoi. Cette méthode, bien qu'utilisée dans de nombreux protocoles comme l'I2C s'avère plus difficile à réaliser et fait perdre du temps de calcul en essayant d'interpréter chaque mot avant de l'envoyer.

Dans la partie suivante, je vais vous présenter le module permettant de gérer la couche réseau juste au dessus de l'Ethernet, l'IP.

3.2 Module IP

Contrairement à la couche précédente, j'ai réalisé l'intégralité de l'architecture. Il m'a donc fallu suivre des schémas stricts permettant une vision claire et bien construite de mon travail dans le but d'une éventuelle réutilisation.

Dans un premier temps il faut concevoir l'unité de traitement de l'architecture. Cela consiste à placer et router entre eux les différents composants matériels nécessaires à la réalisation des différentes fonctions du module.

La fonction principale de cette architecture est de réceptionner une certaine quantité de données en provenance du module MEP et de l'envoyer au module Ethernet après l'avoir associée à un en-tête IP. Il est donc nécessaire que l'architecture comprenne deux unités de stockage pour recueillir les données et stocker l'en-tête. Comme les données sont juste stockées puis renvoyées, une file semble convenir parfaitement. En effet, lors de la lecture d'une file ou FIFO (First In First Out) le premier mot à sortir est le premier qui est entré, ainsi l'ordre des mots dans le paquet est conservé. Par contre, l'entête du fichier doit être modifié plusieurs fois avant l'envoi du paquet. Ceci implique de lire certaine données pour les conserver et d'écrire plusieurs fois pour modifier celles qui doivent l'être. Ainsi, le type de stockage approprié à l'en-tête du paquet est une RAM.

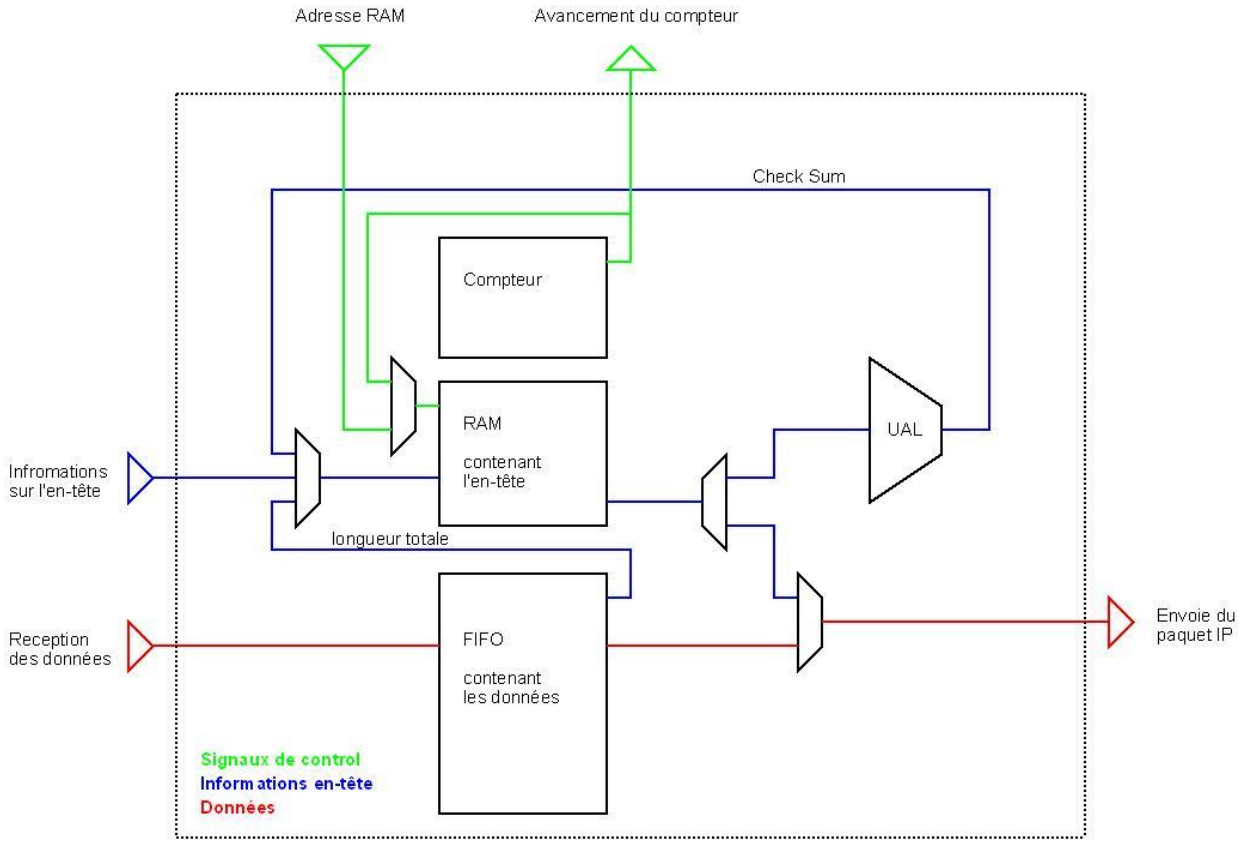


Figure 9 : Schéma simplifié de l'UT du module IP

Par la suite, la question de la taille des mots mémoire s'est posé. En effet, le module Ethernet reçoit les informations sur un bus de 64 bits. J'ai donc pensé dans un premier temps réaliser mes modules (IP et MEP) avec des mots mémoire de 64 bits et donc des bus de 64 bits pour faciliter l'interfaçage entre les différents modules. Cependant l'en-tête du protocole IP comporte 160 bits ce qui n'est pas divisible par 64. Cela impliquait donc un traitement particulier pour la moitié du dernier mot. Ceci était trop contraignant et aurait ralenti considérablement le débit du module. Une gestion du paquet sous la forme de mots de 32 bits m'a semblé alors beaucoup plus naturelle. En plus d'une gestion de l'en-tête sous la forme d'une RAM de 5 mots, cette architecture sur 32 bits offre d'autres avantages que je détaille par la suite.

En étudiant les spécifications de l'en-tête d'un paquet IP on relève seulement 3 champs qui doivent être rempli par le pilote.

Dans un premier temps, les informations de fragmentations qui sont transmises par le module MEP. En vous reportant à la partie 2.3.1, où je détaille le protocole IP, on remarque que ces informations sont réparties en deux champs de 16 bits, l'identifiant et le décalage, qui forme à eux deux un mot de 32 bits dans la RAM. Ainsi, j'ai choisi de faire circuler ces informations sur le bus de données pour économiser un bus de 32 bits qui aurait été dédié à la transmission de ces seules informations. Donc le premier mot transmis sur le bus de donnée n'est pas à stocker dans la FIFO mais bien dans la RAM directement à l'adresse du deuxième mot de l'en-tête.

Ensuite, le protocole IP nécessite de calculer la longueur totale du paquet, en-tête compris, en octets. Or dans la librairie de composant d'altera les FIFO possèdent une sortie « usedw » qui affiche en permanence le nombre exacts de mots présents dans la FIFO. Le nombre d'octet de l'en-tête lui est constant (et vaut 20 octets). Ainsi j'ai construit un petit composant en vhdl qui prend en entré la sortie « usedw » de la FIFO et retourne en sortie la longueur totale du paquet sur 16 bits au format « little endian ».

Une fois la totalité des données reçues et stockées dans la FIFO la sortie de ce composant retourne la valeur exacte de la longueur totale du paquet. Cette valeur doit être inscrite sur les 16 bits de poids faible du premier mot de la RAM sans perdre les informations contenues dans les 16 bits de poids fort. Pour cela j'ai utilisé une fonctionnalité du logiciel Quartus II qui permet de spécifier la provenance de chaque bit d'un même bus. J'ai donc créé un bus dont les 16 bits de poids fort sont reliés aux 16 bits de poids fort de la sortie de la RAM et les 16 bits de poids faibles à la sortie du composant retournant la longueur totale, l'autre extrémité du bus étant elle routée par le biais d'un multiplexeur vers l'entrée de la RAM . Dans un premier temps il faut faire une requête de lecture de la RAM à l'adresse 0 (adresse du premier mot) pour recopier les 16 bits de poids fort sur le bus, à côté des 16 bits de la longueur totale. Puis il suffit de faire une

requête d'écriture de la RAM à cette même adresse pour y placer le nouveau mot possédant la valeur correcte de la longueur totale.

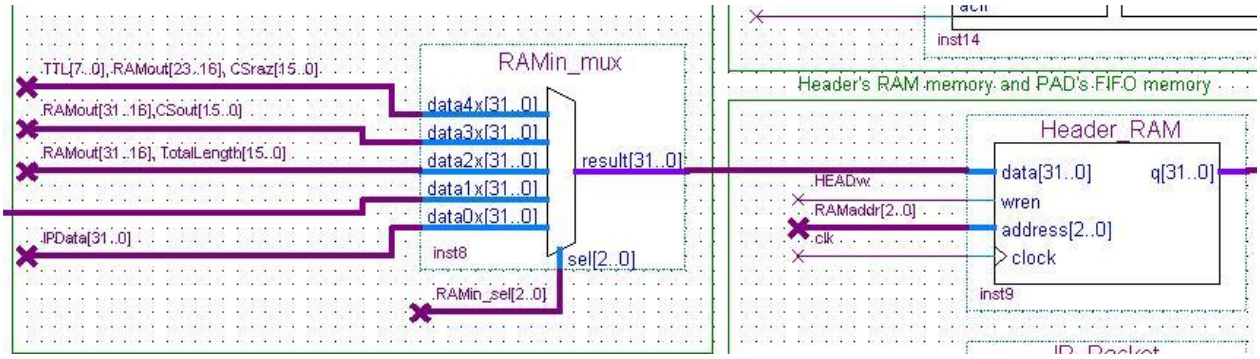


Figure 10 : Multiplexeur permettant de construire l'entête IP

Le troisième et dernier champ à remplir par le pilote est le champ de CheckSum, ou somme de contrôle. Cette somme de contrôle est calculée sur 16 bits en sommant tous les mots de 16 bits de l'en-tête en reportant la retenue sortante sur la retenue entrante. Cependant ce calcul nécessite que le champ destiné à recevoir le CheckSum soit à 0x0000 pendant le calcul de ce dernier. Donc, de la même façon que pour la longueur totale, il est nécessaire d'utiliser un bus dont les 16 bits de poids fort sont reliés à la sortie de la RAM , dont les 16 bits de poids faible sont reliés à la masse et qui est routé vers l'entrée de la RAM. Grâce à ce bus nous allons pouvoir mettre à 0x0000 le champ Check Sum avant le calcul de celui-ci.

Une fois cette initialisation réalisée, il ne reste plus qu'à envoyé séquentiellement tout les mots de l'en-tête vers le module de calcul de la somme de contrôle et remplacer les 0x0000 par la valeur obtenue.

Seulement, un problème subsiste, les mots de l'en-tête sont sur 32 bits et l'algorithme nécessite un calcul sur 16 bits. J'ai donc utilisé pour réalisé le calcul deux accumulateurs arithmétiques de 16bits qui somment respectivement les 16 bits de poids fort et les 16 bits de poids faible de chaque mots au fur et à mesure de leurs sortie de la RAM. Une fois l'accumulation terminée les deux résultats sont sommés à l'aide d'un additionneur 16bits. Le résultat ainsi obtenu en respectant la règle des retenues est renvoyé vers la RAM.

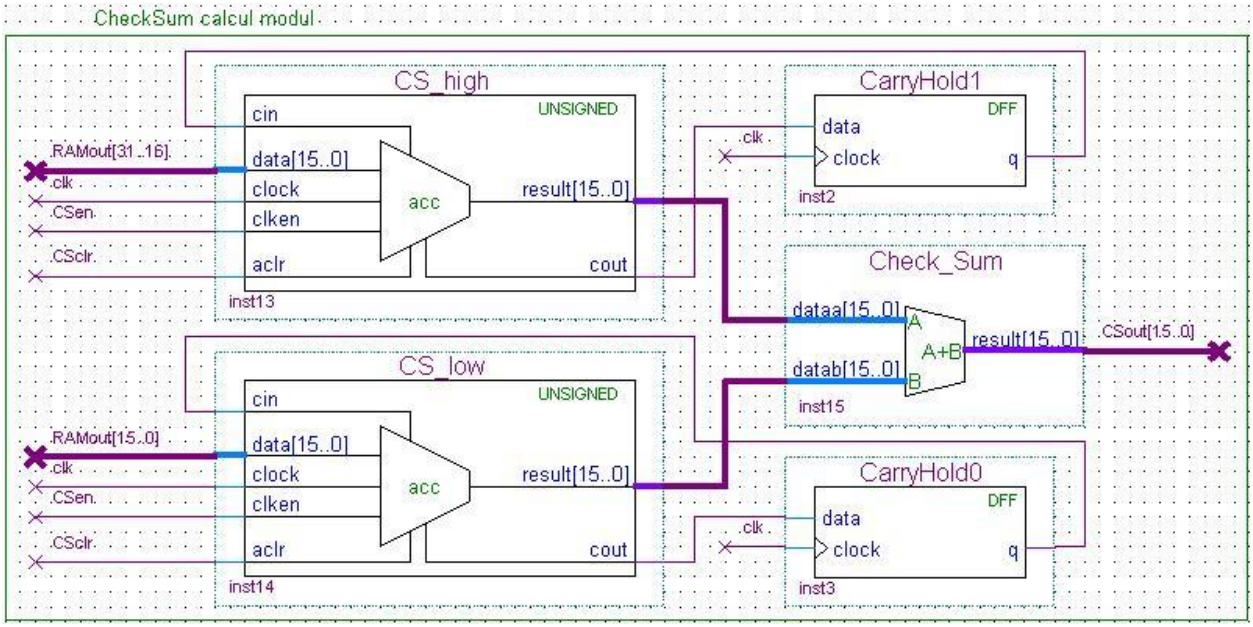


Figure 11 : Module de calcul de la somme de contrôle

En combinant tous ces composants, on obtient l'unité de traitement capable d'exécuter les fonctions du module IP. Le schéma complet et détaillé de cette unité de traitement est disponible en annexe I.

Une fois l'unité de traitement de réalisée, il faut mettre au point l'unité de contrôle qui va piloter cette unité de traitement et ordonnancer ses différentes fonctions. Cette unité de contrôle est réalisée sous la forme d'une machine d'état. Je vais ainsi vous présenter la machine d'état sous la forme d'un graphe.

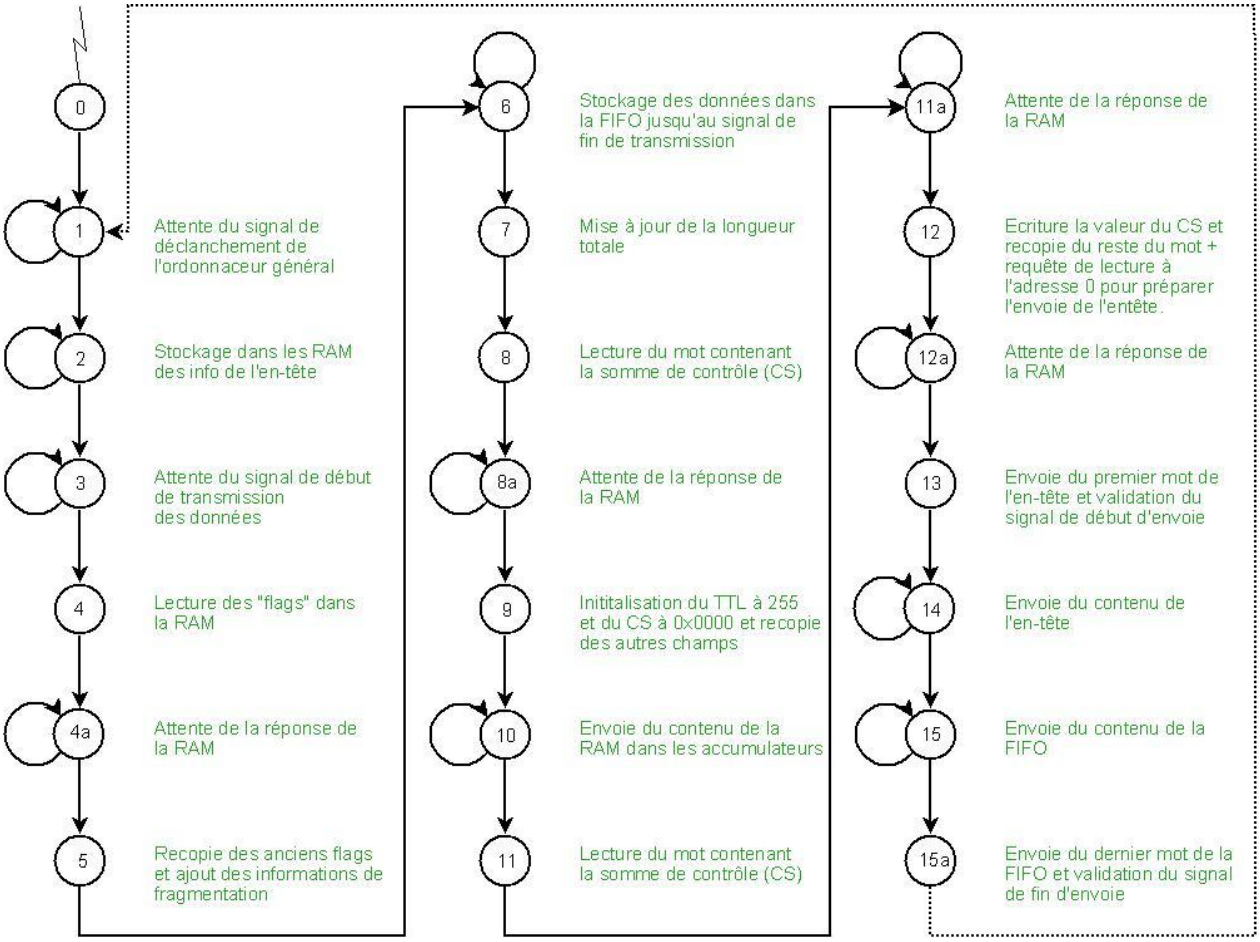


Figure 12 : Graphe de l'UC du module IP

Pour se synchroniser avec les autres modules de l'architecture, l'unité de contrôle dispose de 5 signaux.

Tout d'abord un signal venant de l'ordonnanceur général qui déclenche la construction d'un paquet.

Ensuite elle dispose de deux signaux en entrée : IP_sop et IP_eop respectivement Start Of Packet (début du paquet) et End Of Packet (fin du paquet) pour identifier respectivement le premier et le dernier mot de données envoyés par le module MEP.

Et pour finir, lors de l'envoi du paquet IP au module Ethernet, deux signaux en sortie : Eth_sop et Eth_eop fonctionnent de la même façon que les précédents.

Vingt et un états sont nécessaires. Ceux qui sont nommés Xa implémentent un cycle d'attente pour veiller à ce que la RAM soit disponible pour la prochaine opération. J'ai choisi de faire des états d'attente bien distincts pour permettre de paramétrer le temps d'attente de la RAM, car des RAM plus ou moins rapides peuvent être mise en place pour cette architecture. Dans le cas ou l'on voudrait modifier la fréquence de fonctionnement de l'architecture ou la vitesse de réponse de la mémoire ces états permettrons une adaptation plus facile.

Maintenant que les deux couches les plus basses sont réalisées nous allons nous intéresser à la dernière couche de ce protocole de communication, la couche MEP.

3.3 Module MEP

La couche MEP est la couche la plus haute du protocole réseau du système d'acquisition. Le module MEP construit un paquet qui regroupe un nombre constant d'évènements défini par le TFC. Une fois les données collectées dans la mémoire il y ajoute un entête dont vous trouverez les spécifications dans la partie 2.3.1 qui lui est consacrée.

La réalisation de ce module suit les mêmes schémas que le module IP. Nous allons donc commencer par une analyse des différentes fonctions nécessaires à la réalisation d'un paquet et structurer ainsi l'unité de traitement du module. Dans un deuxième temps nous verrons comment piloter et ordonnancer ces différentes fonctions grâce à l'unité de contrôle.

L'entête du module MEP est composé de 2 mots de 32 bits. Nous allons donc les stocker de la même manière, dans une RAM. Les données seront stockées sur 32 bits dans une FIFO de taille supérieure à celle utilisée dans le module IP car un paquet MEP est susceptible d'atteindre une taille de 2^{16} bits.

Pour la réalisation de l'entête, nous avons besoin de la longueur totale du paquet. J'ai donc réutilisé le composant du module IP en le modifiant légèrement car le MEP est en big endian.

Ces quelques composants suffisent à réaliser les fonctions basiques nécessaires à la réalisation de l'entête d'un paquet MEP.

Cependant, lors de l'analyse globale de l'architecture, j'ai choisi de réaliser le partitionnement des MEP en paquets IP à l'intérieur du module MEP. Pour plus de détails vous pouvez vous référer à la partie 2.4.

Il est donc nécessaire d'ajouter quelques composants pour réaliser cette nouvelle fonction. Pour savoir à quel moment arrêter l'envoi des données vers le module IP et demander la création d'un nouveau paquet nous avons besoin d'un composant capable de dire combien de mots nous avons déjà envoyé. Ainsi quand ce nombre atteindra le nombre maximal de mots d'un paquet IP (MTU) nous pourrons gérer le partitionnement.

Les informations de partitionnement n'ont pas besoin d'un traitement particulier car elles se composent d'un seul mot de 32 bit composé des 16 bit de poids faible du champ L0ID/L1ID, de la longueur totale qui représente le décalage du paquet suivant sur 13 bit et des 3 bit de flags signalant la présence d'autre paquet ou non.

Le composant permettant de suivre l'avancement de l'envoi des données se compose d'un registre de 32 bit qui retient la valeur de la longueur totale du paquet et la maintient sur sa sortie. On soustrait de manière continue à cette constante la valeur de la sortie usedw de la FIFO (nombre de mots contenus dans la FIFO). Ainsi nous obtenons le nombre de mots envoyés à l'instant t. J'ai choisi d'utiliser un registre, plutôt que d'aller lire la longueur totale dans la mémoire, pour des questions de temps d'accès qui auraient encore ralenti le module. De plus, l'utilisation de ce registre permet d'avoir la valeur du nombre de mots envoyés rafraichie automatiquement à chaque variation de la sortie usedW.

Le composant permettant de gérer le troisième bit de fragmentation est un comparateur qui prend en entrée la sortie usedW de la FIFO et le compare avec une constante. Cette constante peut prendre deux valeurs. Dans le cas du premier paquet il faut ajouter la longueur de l'entête au nombre de mots contenu dans la FIFO pour obtenir le nombre réel de mots sur le point d'être envoyés. Cependant pour les fragments suivant il ne doit pas être pris en compte. Ainsi, si usedW est supérieur à la constante, alors il y aura d'autres fragments, le bit est à '1', sinon ce fragment est le seul ou le dernier de la série, le bit est à '0'.

Nous en avons terminé avec les spécificités de l'UT du module MEP, voici en résumé un schéma simplifié de celle-ci.

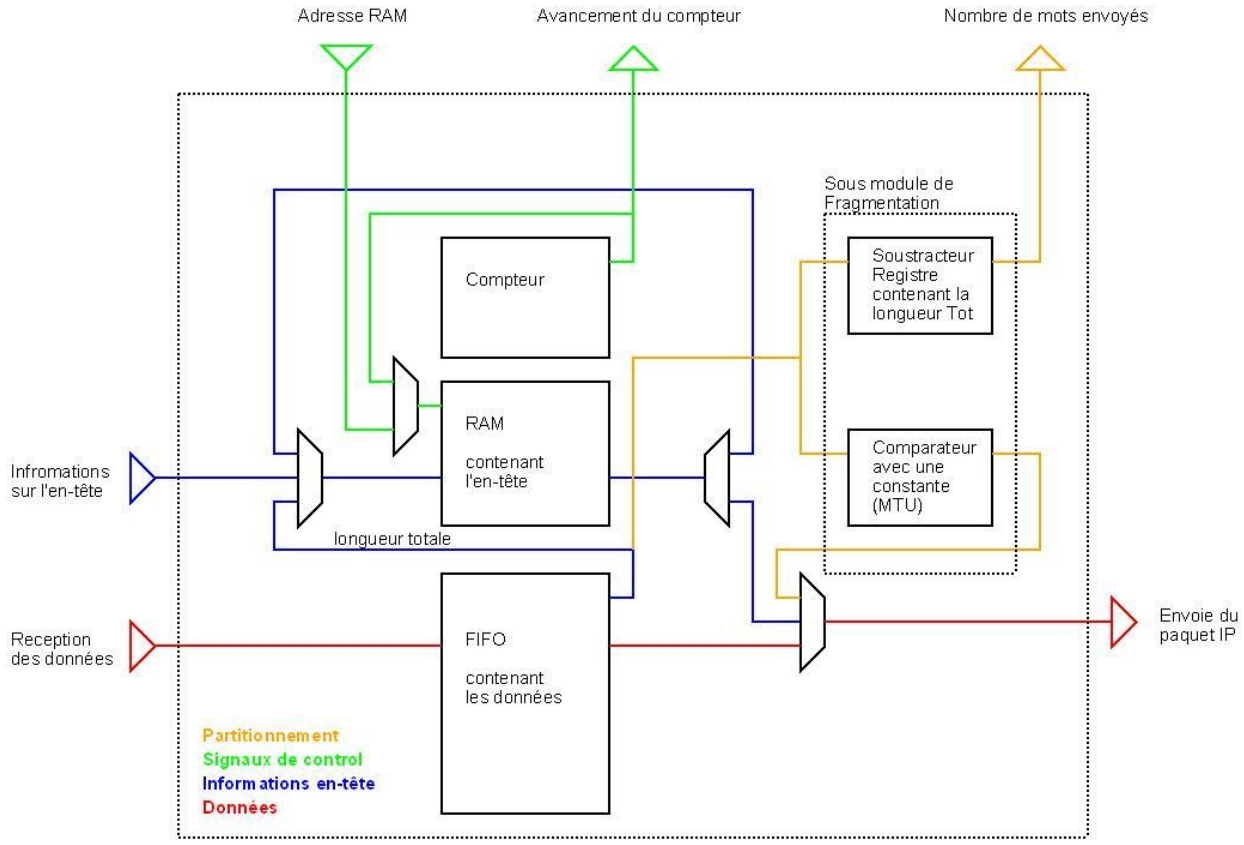


Figure 13 : Schéma simplifié de l'UT du module MEP

Maintenant que tous les composants nécessaires à la réalisation d'un paquet MEP sont réunis, voici l'unité de contrôle que j'ai décrite sous la forme d'un graphe, puis je reviendrais sur certains détails qui m'ont posés plus de problèmes.

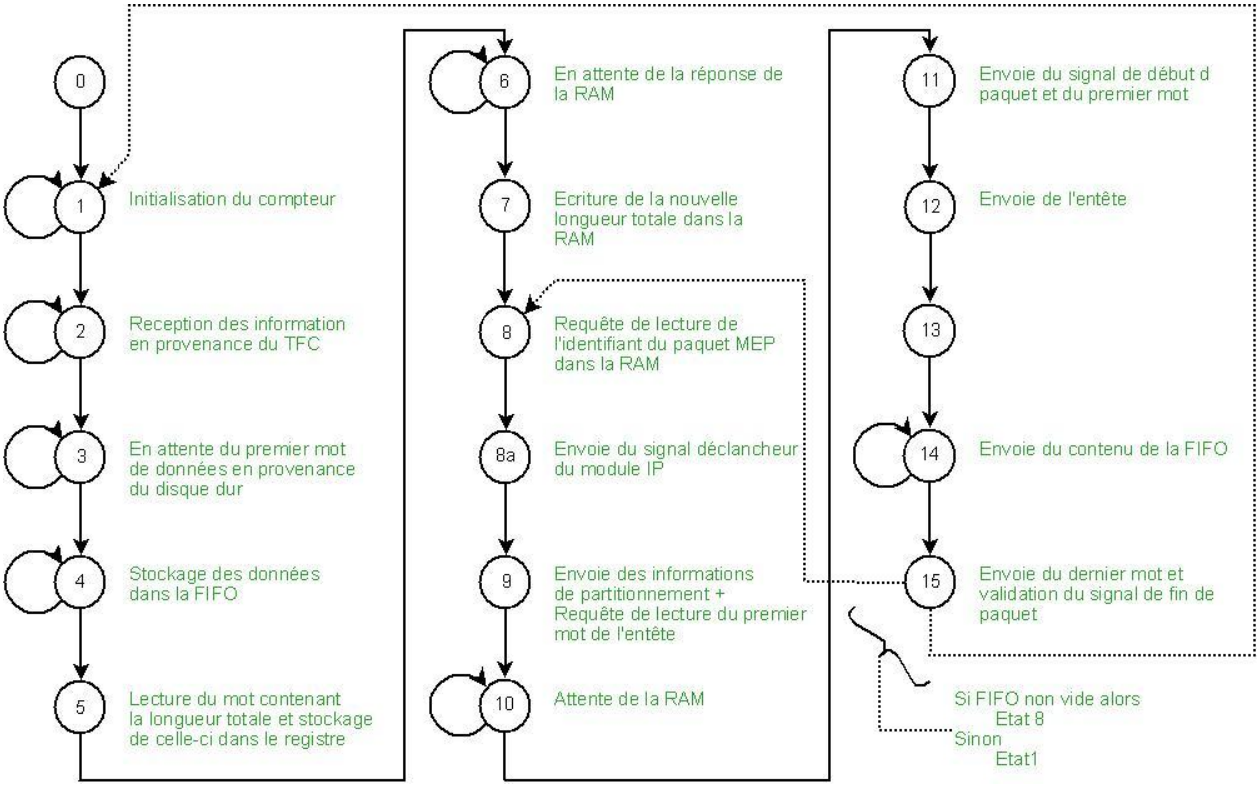


Figure 14 : Graphe de l'UC du module MEP

La construction de l'entête ne pose aucune difficulté. Cependant le partitionnement des données mérite que l'on s'y arrête un instant. En effet celui-ci se fait en plusieurs temps.

Dans l'état 7, la longueur maximale d'un paquet IP moins la taille de l'entête MEP est placée à l'entrée du comparateur. Comme l'état 7 n'est actif que dans le cas où nous sommes au début d'un paquet MEP (en effet la boucle se fait sur l'état 8) cela nous assure de faire une prédiction correcte d'une éventuelle fragmentation du paquet MEP.

Ensuite, les bits de poids faible de l'identifiant du paquet MEP servent d'identifiant à la série de fragments éventuellement générée.

Pour finir, dans le cas où le premier fragment est envoyé mais qu'il reste des informations dans la liste il faut de nouveau effectuer un test sur la quantité de données restantes. Cependant il n'y a plus d'entête à envoyer, c'est pourquoi quand l'état 15 est actif il faut placer à l'entré du comparateur la vrai valeur de la MTU. Dans le cas où il n'y a pus de paquet de ce fragment à envoyer, l'automate repart dans l'état 1.

Ceci marque la fin de la conception de ce module MEP. Maintenant nous disposons des 3 modules d'empaquetage successif des données brutes : MEP, IP, Ethernet. Pour envoyer les données, il ne nous reste plus qu'à ajuster ces briques ensemble et les synchroniser les unes avec les autres. Cette tâche revient à l'ordonnanceur général de l'architecture.

3.4 Ordonnanceur

L'ordonnanceur est un module à part entière. Ce dernier reçoit ses ordres directement du TFC par le biais de la carte d'acquisition optique. Une fois l'ordre de lancer une simulation reçu, il est chargé d'envoyer les requêtes correctes aux différents modules créés plus haut. De plus il doit leurs transmettre les informations qu'il reçoit du TFC nécessaires à la construction des entêtes des différents protocoles.

J'ai supposé pour la réalisation de cette architecture que les données provenant de la carte d'acquisition optique étaient conditionnées sous la forme de mots de 32 bit. Ce qui n'est pas difficile à obtenir par la suite avec un simple registre à décalage avec une sortie en parallèle.

Je vais maintenant vous détailler les fonctions que doit remplir se module et quels ont été mes choix quand à la manière de les réaliser.

Tout d'abord, le module doit recevoir les informations provenant du TFC et les stocker de manière à pouvoir les transmettre au bon moment aux différents modules. Pour cela nous avons besoin d'un RAM de sept (cinq pour IP et deux pour MEP) mots mémoire de 32 bit, ainsi qu'un compteur dédié qui permettra de parcourir les adresses de la mémoire de manière séquentielle lors de la réception et de l'envoi.

Ensuite, nous devons être capables de changer l'adresse IP source des différents paquets MEP que nous transmettrons par celle d'une des cartes TELL1 comme décrit dans le paragraphe 2.3.2. Pour cela, nous devons avoir à notre disposition la totalité des adresses IP des 300 cartes TELL1. J'ai choisi de les stocker dans une RAM de trois cent mots de 32 bits qui dispose elle aussi d'un compteur dédié qui incrémentera l'adresse à chaque nouveau paquet MEP.

Pour finir, l'ordonnanceur doit pouvoir attendre de manière certaine la fin de l'envoi d'un paquet MEP avant de passer au suivant. Cela pose un problème lorsque le paquet a été fragmenté. En effet le module Ethernet d'Altera dispose d'un bit qui reste nul si l'envoie du paquet Ethernet s'est terminé sans erreur. Seulement un paquet MEP peut se trouver sur plusieurs paquets Ethernet. L'ordonnanceur doit donc récupérer le bit de validation du module Ethernet ainsi que le bit de fragmentation produit par le module MEP lors de la prédiction de fragmentation. Si le bit provenant du module Ethernet est à 0 et qu'il n'y a pas ou plus de fragment à envoyé (bit à 0) c'est que l'envoie du paquet MEP est terminé. Cette fonction peut être facilement réalisée par une porte NAND2, qui fait le ET entre ses deux entrées et retourne l'inverse du résultat.

Il ne nous reste plus qu'à assembler ses différentes fonctions pour obtenir l'unité de traitement de ce module. Voici un schéma simplifié qui représente cette unité.

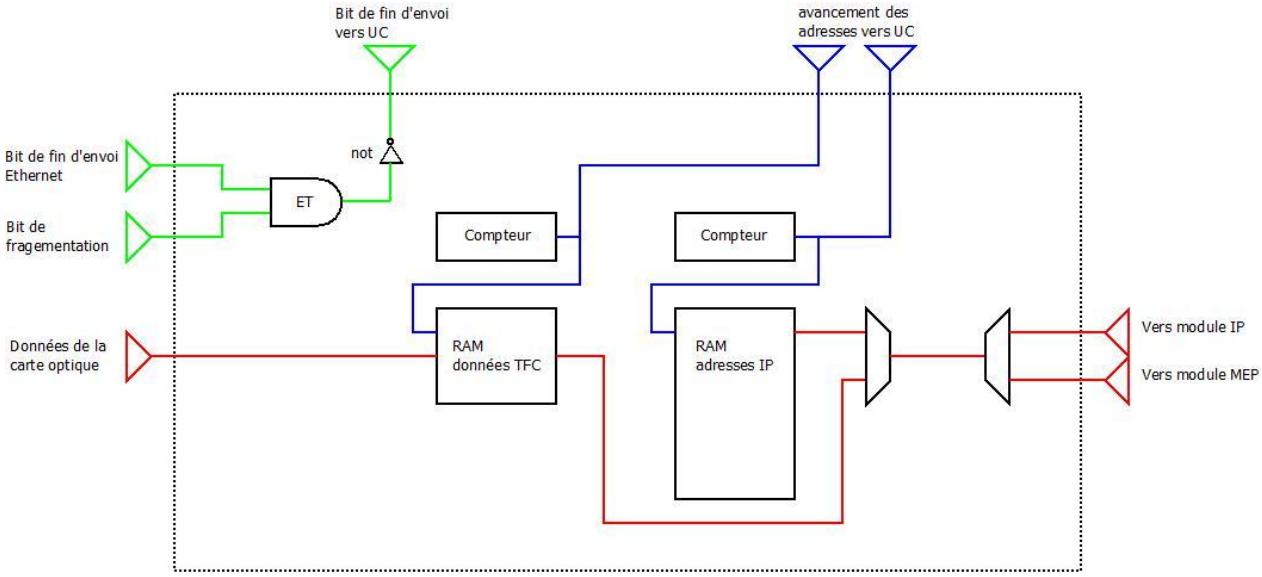


Figure 15 : Schéma simplifié de l'UT de l'ordonnanceur

Maintenant nous allons nous pencher sur l'unité de traitement de cette architecture. Celle-ci est assez simple, l'automate attend le signal du TFC, recueille ses informations, puis il effectue 300 fois, 1 fois par carte TELL1, la boucle qui consiste à envoyer un paquet MEP avant de se remettre en attente du TFC.

Voici en détail le graphe de cet automate.

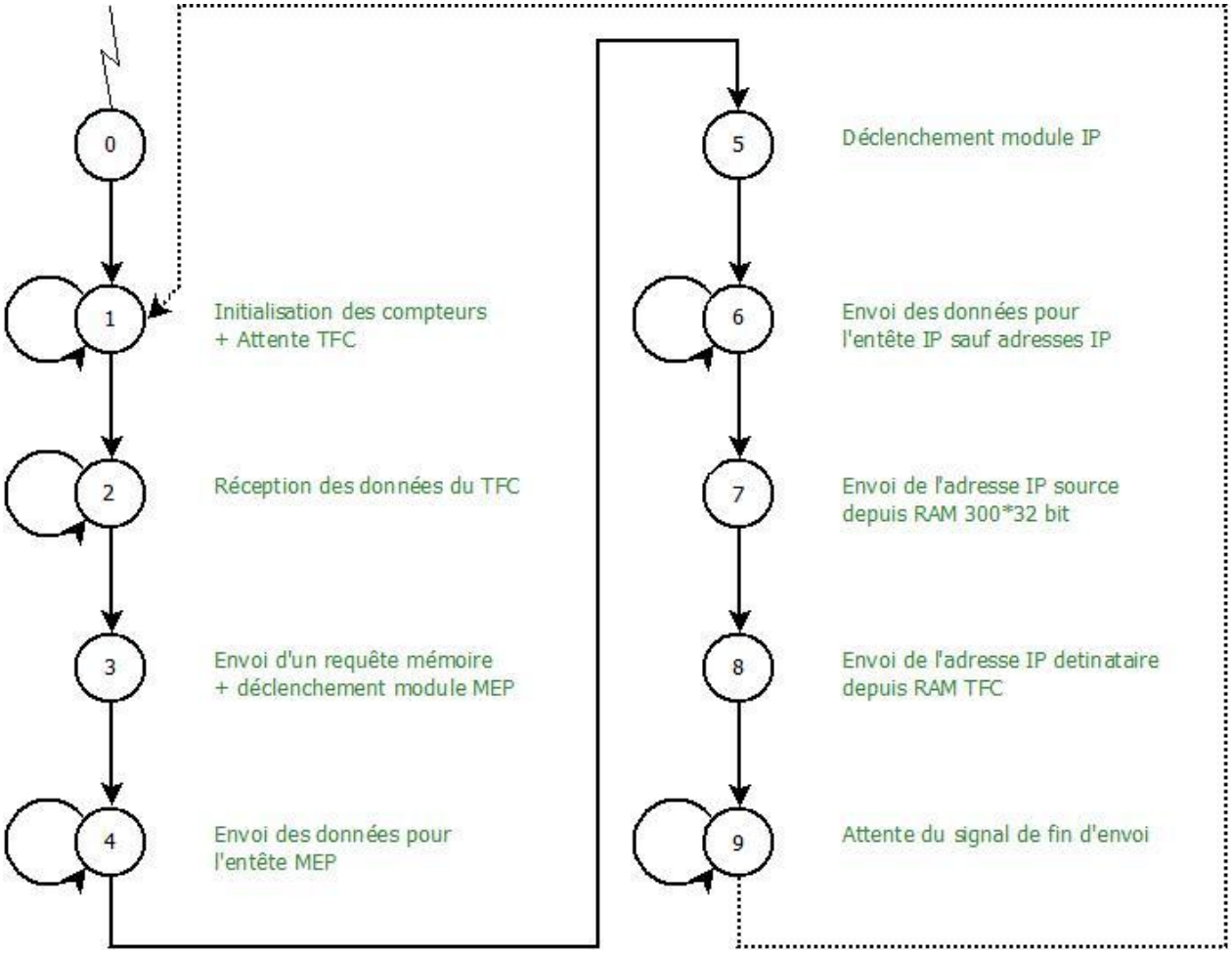


Figure 16 : Graphe de l'UC de l'ordonnanceur

Ce module d'ordonnancement est maintenant terminé et avec lui la partie la partie transmission de données de l'injecteur. En effet grâce au 3 modules d'encapsulation des données et à l'ordonnanceur qui joue le rôle de chef d'orchestre de cette mécanique, les données brutes sont conditionnées en 300 paquets MEP semblant chacun provenir d'une carte TELL1 différente et sont envoyées sur le réseau du système d'acquisition en respectant les protocoles standards IP et Ethernet.

La conception de ces modules ne c'est pas faite directement mais à demandé une longue phase de test et de débogage.

4. Résultats

Une fois qu'une architecture a été correctement compilée, il faut vérifier qu'elle remplie correctement les fonctions pour lesquelles elle a été crée. Dans cette partie, je vais présenter les différentes fonctionnalités des modules testés en simulation. Une fois l'architecture validée, celle-ci est prête pour la diffusion. Le CERN me permet de diffuser mes travaux sur différents médias que je détaillerais.

4.1 Simulations

Dans cette partie, j'ai décomposé les différents modules selon les fonctions qu'ils utilisent. Ainsi je vais présenter les fonctions, communes ou spécifiques à un module en analysant son chronogramme de simulation.

4.1.1 Fonctions communes

Une des fonctions les plus utilisée dans les différents modules est celle qui consiste à lire un mot en mémoire, à le modifier, puis à le réécrire dans la mémoire. En effet ceci intervient à chaque modification de l'entête d'un paquet avant son envoi. Je rappelle ici que la modification du contenu d'un mot mémoire s'effectue grâce à un multiplexeur à l'entrée de la RAM comme décrit dans la partie Conception. Cette fonction se déroule en 2 étapes : l'étape 1 consiste à envoyer la requête de lecture à la mémoire et l'étape 2 à écrire le résultat à la même adresse.

Voici un exemple sous la forme d'un chronogramme :

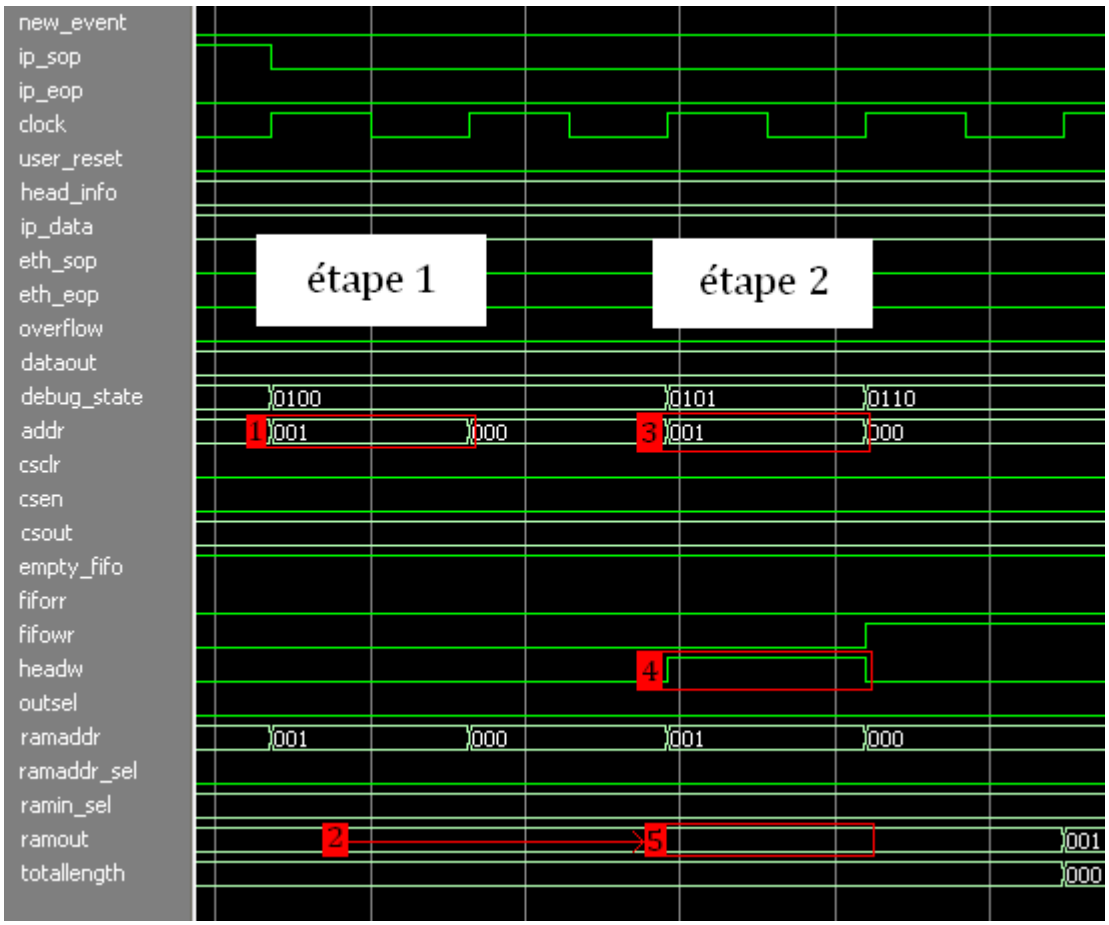


Figure 17 : Chronogramme Lecture/Ecriture

Lors de l'étape 1, en 1 on place l'adresse que l'on souhaite atteindre dans la RAM sur le bus d'adresse (ici 001). La marque 2 met en évidence les 2 périodes d'horloge nécessaires à la réponse de la RAM. Au bout de se temps d'attente le mot mémoire arrive enfin sur le bus de sortie (marque 5). C'est à ce moment là qu'il faut l'écrire en mémoire. On replace donc l'adresse 001 sur le bus d'adresse (marque 3) tout en activant le mode écriture de la RAM (marque 4). Ainsi on a bien lu le contenu de l'entête, modifié son contenu (cette phase est instantanée c'est pourquoi elle ne figure pas sur le chronogramme) puis réécrit le mot mis à jour dans l'entête.

L'autre fonction commune aux différents modules et la fonction d'envoi. Cette fonction se déroule le plus souvent en deux temps : tout d'abord l'envoi de l'entête puis l'envoi des données. L'envoi de l'entête se résume à la lecture séquentielle du contenu de la RAM. Cependant l'envoi des données de manière synchrone avec la fin de l'envoi de l'entête nécessite d'anticiper le temps de réponse de la FIFO à la requête de lecture.

Cette fonction d'envoi peut donc être divisée en 3 parties : la première étape consiste à envoyer les mots de l'entête. Lors de la deuxième étape on envoi le dernier mot de l'entête tout en envoyant une requête de lecture à la FIFO et dans une dernière étape on envoi les données.

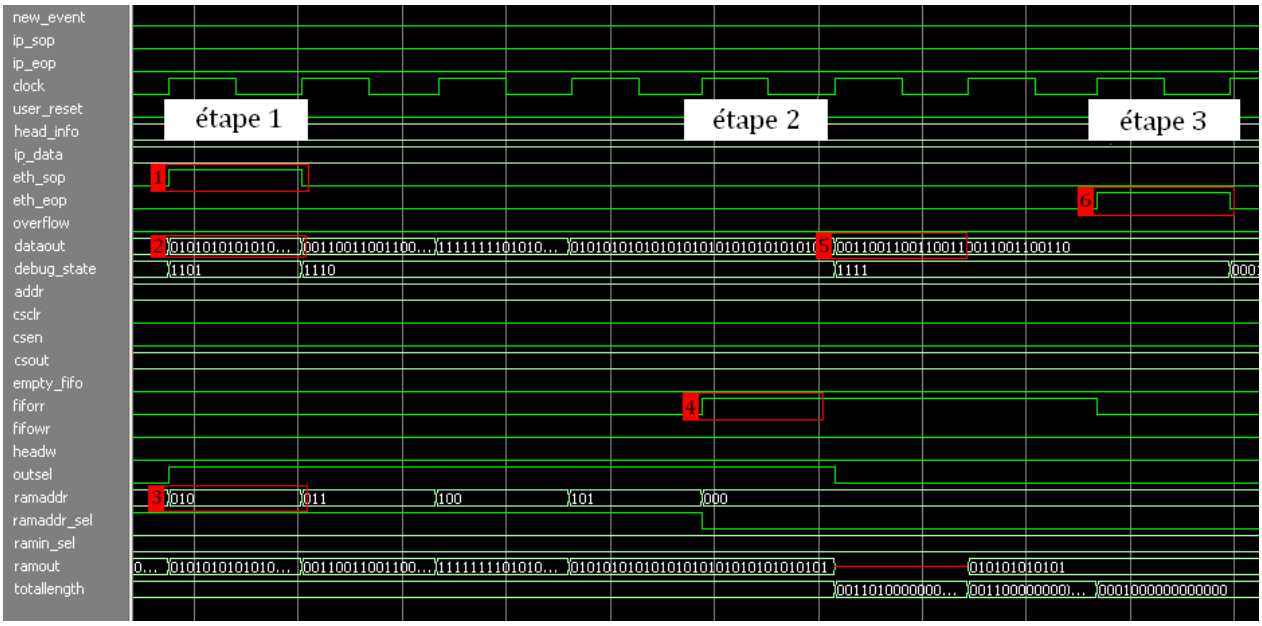


Figure 18 : Chronogramme fonction d'envoi

Comme on peut le voir à la marque 3 l'adresse en entrée de la RAM est déjà de '2' au niveau de l'étape 1, ceci pour anticipé les deux périodes d'horloge de décalage de la RAM. Les données présentes sur le bus de sortie de la RAM (ramout) et de sortie du module (dataout) correspondent donc au premier mot de l'entête (marque 2). Au niveau de la marque 1 on voit que le bit signalant le début d'un paquet, est à 1.

Lors de l'étape 2, on envoi le dernier mot de l'entête et on envoi un requête de lecture à la FIFO (marque 4) de sorte que le premier mot de la FIFO arrive sur le bus de sortie juste après la fin de l'entête (marque 5).

Pour finir, dans l'étape 3, on annule la requête de lecture de la FIFO et on valide le bit de fin de paquet (marque 6).

Ces deux fonctions étant validées, elles peuvent être utilisées dans les différents modules. Cependant elles ne suffisent pas à couvrir l'ensemble des fonctions. Je vais donc présenter les fonctions spécifiques à chaque module.

4.1.2 Fonction spécifique IP

La fonction spécifique au module IP est le calcul de la somme de contrôle. Celle-ci nécessite de combiner les deux fonctions exposées précédemment. En effet tout d'abord, il faut envoyer le contenu de l'entête dans les accumulateurs. Ceci s'apparente à l'envoi de l'entête en routant simplement différemment la sortie de la RAM. Ensuite le résultat envoyé par le module de calcul doit être intégrer dans le bon mot mémoire de la RAM.

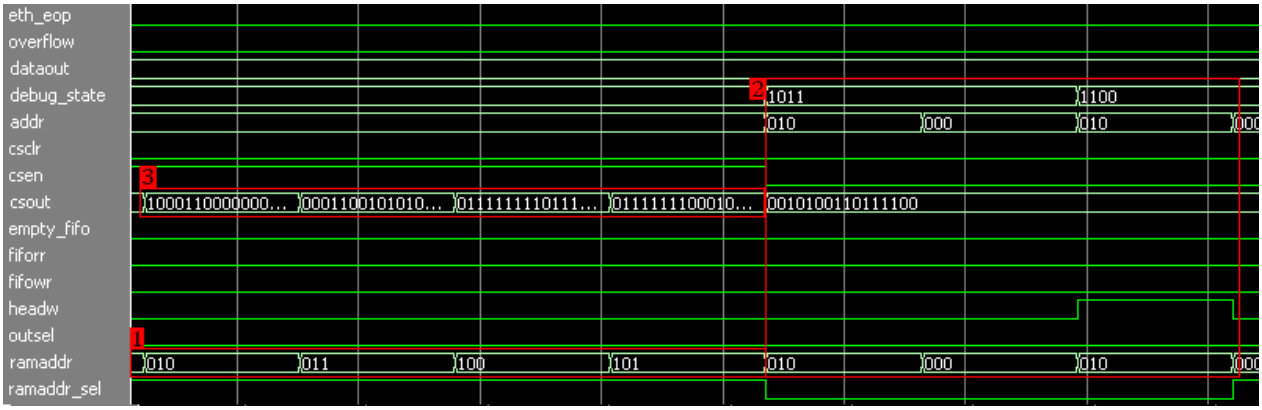


Figure 19 : Chronogramme calcul somme de contrôle

On retrouve dans la zone 1 les requêtes de lecture séquentielle des mots de l'entête dont découle l'évolution de la sortie du module de calcul de la somme de contrôle en zone 3.

La zone 2 décrit le schéma classique d'une écriture en RAM à l'adresse '2' ou doit se trouver la valeur finale de la somme.

Tout les points cruciaux de l'ordonnancement temporel du module IP on été abordés, un chronogramme complet du module est disponible en annexe. Je vais maintenant faire même pour le module MEP.

4.1.3 Fonction spécifique MEP

Il est intéressant ici de se pencher sur la fonction de partitionnement du module MEP. En effet à la fin de l'envoi d'un premier fragment, et dans le cas ou un second fragment est à envoyer, l'automate doit réamorcer un paquet IP avant de poursuivre l'envoi des données.

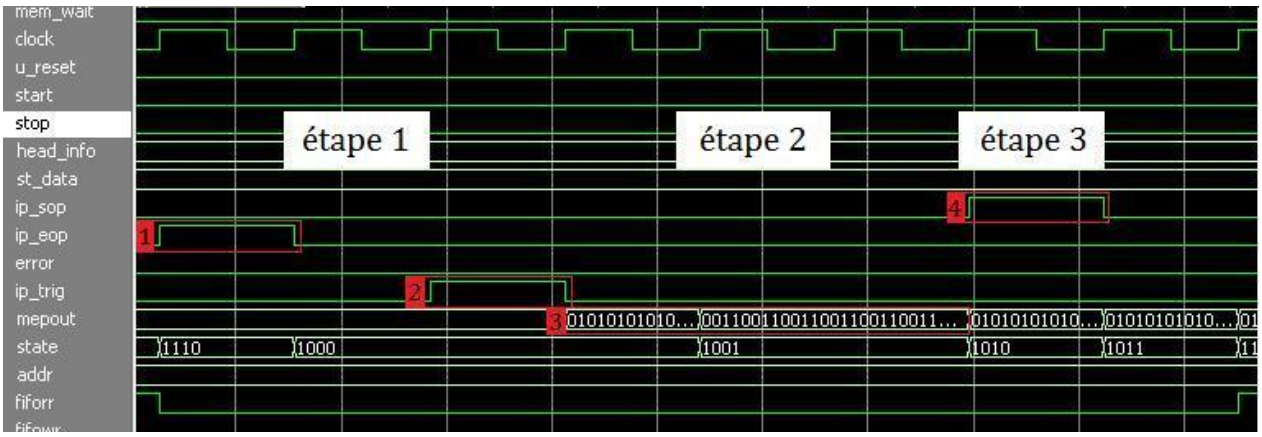


Figure 20 : Chronogramme partitionnement

Cette fonction peut se décomposer en trois étapes. Dans un premier on marque la fin du fragment IP terminé (marque 1) puis on réenclenche le processus de création d'un nouveau paquet IP (marque 2). A cette première étape doit succéder l'envoi des nouvelles informations de partitionnement (marque 3). Et pour finir on envoi la suite des données en signalant le premier mot mémoire de la FIFO (marque 4).

Tout les points cruciaux de l'ordonnancement temporel du module MEP on été abordés, un chronogramme complet du module est disponible en annexe. Le module Ethernet ainsi que le module d'ordonnancement étant tout deux uniquement composés des deux premières fonctions présentées dans cette partie, je ne reviendrais pas dessus.

Maintenant que ces différents modules ont été conçus et testés, ils peuvent être mis à la disposition des personnes qui reprendront la suite de ce projet au sein du CERN.

4.2 Diffusions

Dans cette partie, je vais présenter les différents médias que l'équipe LHCb à mis à ma dispositions pour exposer le fonctionnement de mes travaux.

Fin Septembre, j'ai l'honneur de pouvoir présenter un poster décrivant mes travaux à la conférence TWEPP de Paris. La TWEPP (Topical Workshop on Electronics for Particule Physics) est une conférence annuelle qui se déroule chaque année dans une ville différente. Son but est de présenter des concepts et des résultats originaux de la recherche en électronique en rapport avec les accélérateurs de particules, mais aussi d'encourager les efforts de développement en électronique pour des tâches communes. Tout ceci dans un esprit d'échange et de collaboration entre les mondes de la physique et de l'ingénierie.

A la fin du moi d'avril j'ai donc envoyé un résumé de mon projet pour proposer ma candidature en tant qu'exposant de poster. J'ai été accepté et subventionné par le CERN pour participer à cet évènement.

The LHCb High Level Trigger and Data Acquisition system selects about 2 kHz of events out of the 1 MHz of events, which have been selected previously by the first-level hardware trigger. The selected events are consolidated into files and then sent to permanent storage for subsequent analysis on the Grid.

The goal of the upgrade of the LHCb readout is to lift the limitation to 1 MHz. This means speeding up the DAQ to 40 MHz. Such a DAQ system will certainly employ 10 Gigabit or technologies and might also need new networking protocols: a customized TCP or proprietary solutions.

A test module is being presented, which integrates in the existing LHCb infrastructure. It is a 10-Gigabit traffic generator, flexible enough to generate LHCb's raw data packets using dummy data or simulated data.

These data are seen as real data coming from sub-detectors by the DAQ. The implementation is based on an FPGA using 10 Gigabit Ethernet interface. This module is integrated in the experiment control system.

The architecture, implementation, and performance results of the solution will be presented.

Parallèlement à ceci, je dois fournir une documentation interne au CERN en anglais qui servira aux futurs étudiants qui reprendront le projet par la suite. Cette note devra respecter le guide de style fourni par le CERN pour se genre de document.

Conclusion

Mon projet au sein de l'équipe LCHb a été de travailler à la réalisation d'un injecteur de données hardware pour le système d'acquisition de l'expérience. Une des parties les plus lourdes qui consisté à encapsuler les données et à les envoyer à la bonne ferme de calcul via le réseau Ethernet a été réalisée et éprouvée par la simulation et le test. Cependant le système n'a pas encore été programmé dans son ensemble sur le FPGA

Professionnellement parlant, le fait d'avoir développé la majorité de l'architecture à partir de zéro a été très riche en enseignement. De plus, cela m'a permis de comprendre la nécessité des différentes étapes permettant la réalisation efficace d'un objectif, de la visualisation globale du projet à la conception à proprement parlé des différentes entités, sans oublier les tests. De plus, la recherche des licences pour le module Ethernet m'a permis de rentrer en contact avec différents services du CERN pour négocier les droits avec le fabricant et débloquer les budgets auprès de la hiérarchie.

L'équipe LHCb m'offre la possibilité de présenter un poster résumant mon travail au CERN lors de la TWEPP de Paris fin septembre.

Le projet de l'injecteur est maintenant lancé mais il reste plusieurs problèmes à résoudre. En effet, il reste à concevoir le module qui permet de mettre en forme les données envoyées par le TFC pour l'ordonnanceur. Je compte mettre à profit le temps qu'il me reste au CERN pour le faire. Ensuite il faudra réaliser un module TCP qui pourra au besoin remplacer le module MEP. Et pour finir, il faudra réaliser l'interface FPGA / Disque dur par le port PCI Express de la carte.

Références Bibliographiques

[ALTERA 2007] MNL-01027-1.0, Arria GX Development Board Manuel, 2007

[ALTERA 2009] AN-516-2.2, 10-Gbps Ethernet Reference Design , 2009

[NEUFELD 2004] N.Neufeld and B.Jost, Raw-data transport format, 2004

Annexe I

Regroupement des schémas des unités de traitement des différents modules.

Schéma complet de l'UT du module IP (partie 1)

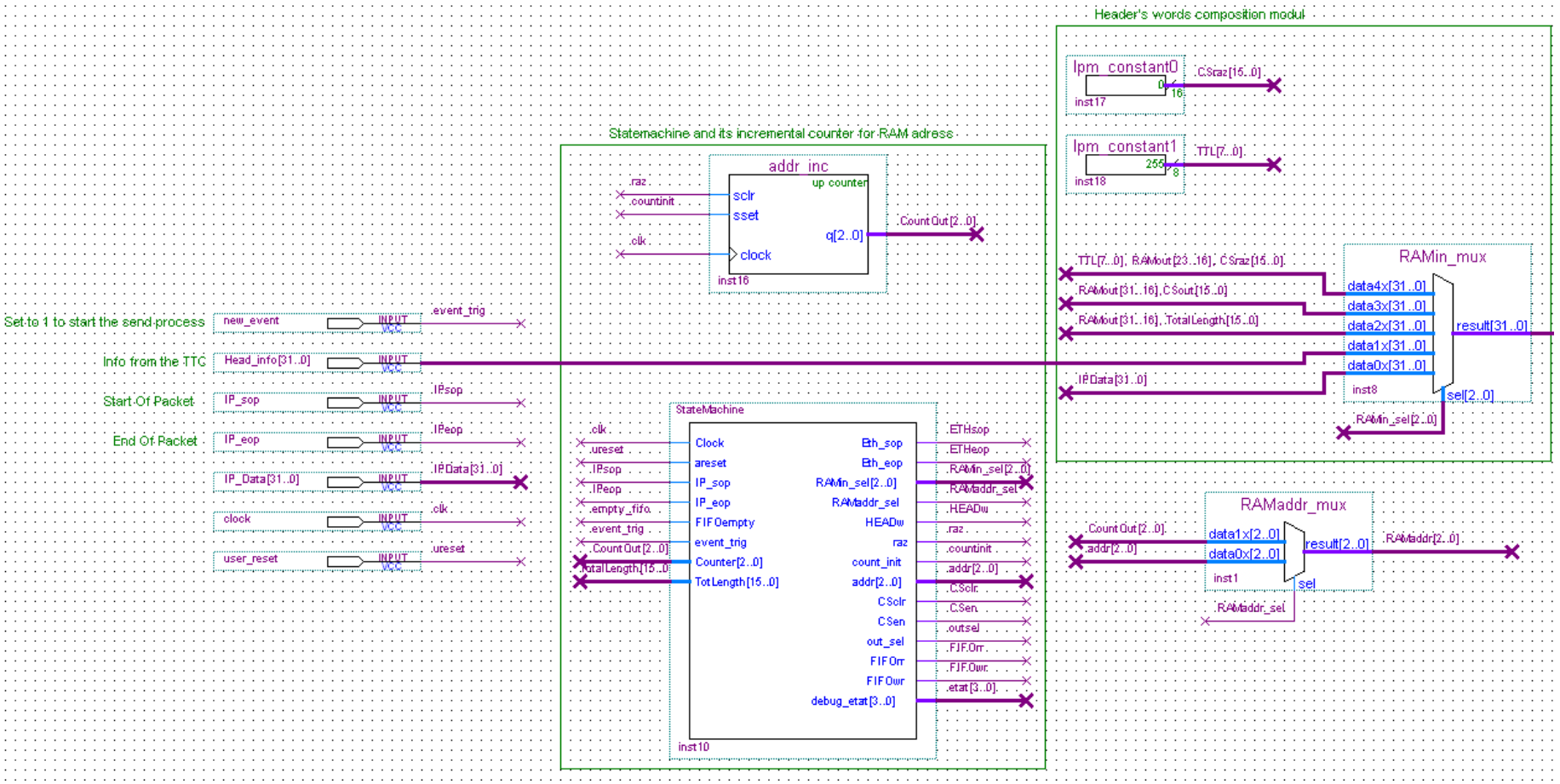


Schéma complet de l'UT du module IP (partie 2)

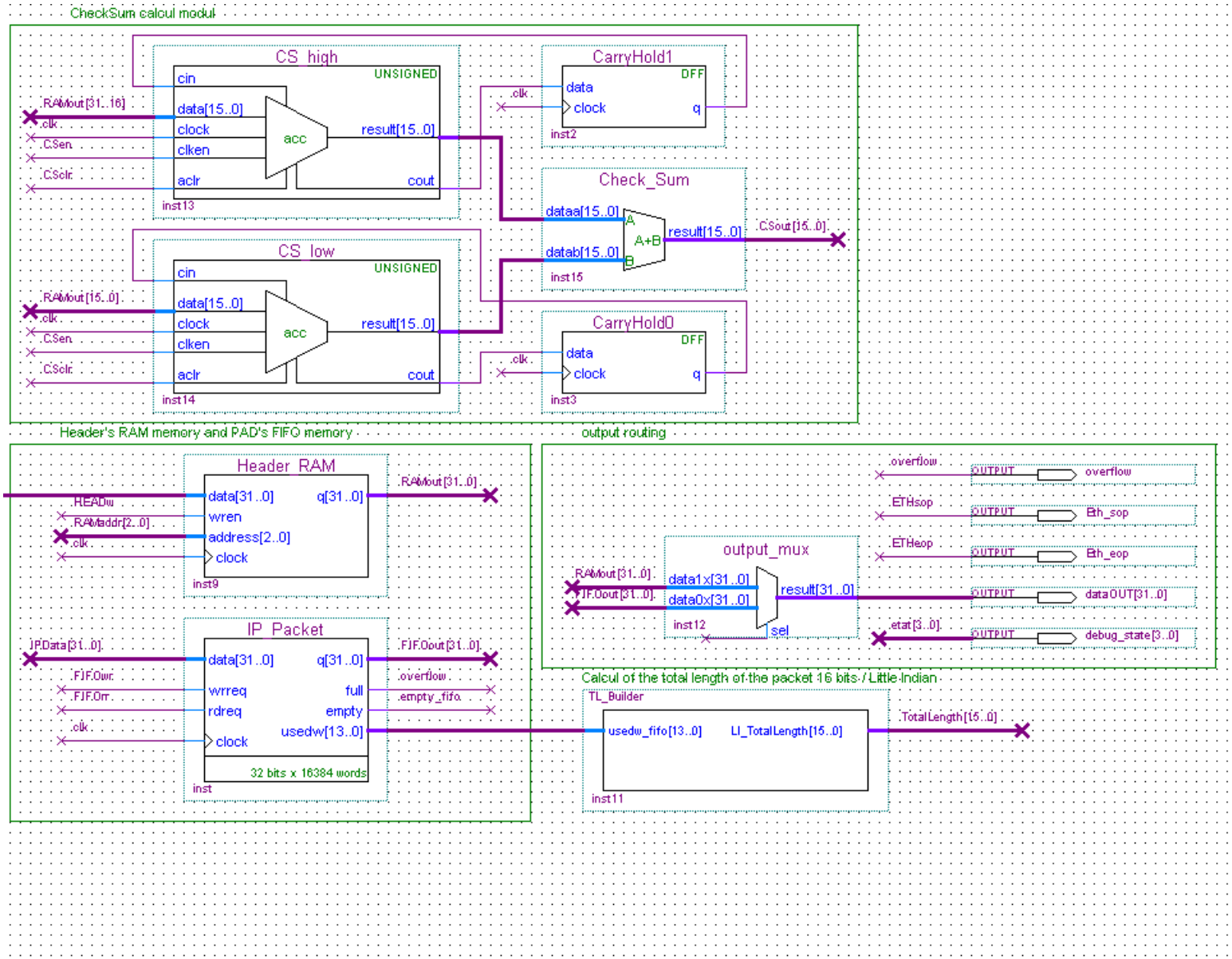


Schéma complet de l'UT du module MEP (partie 1)

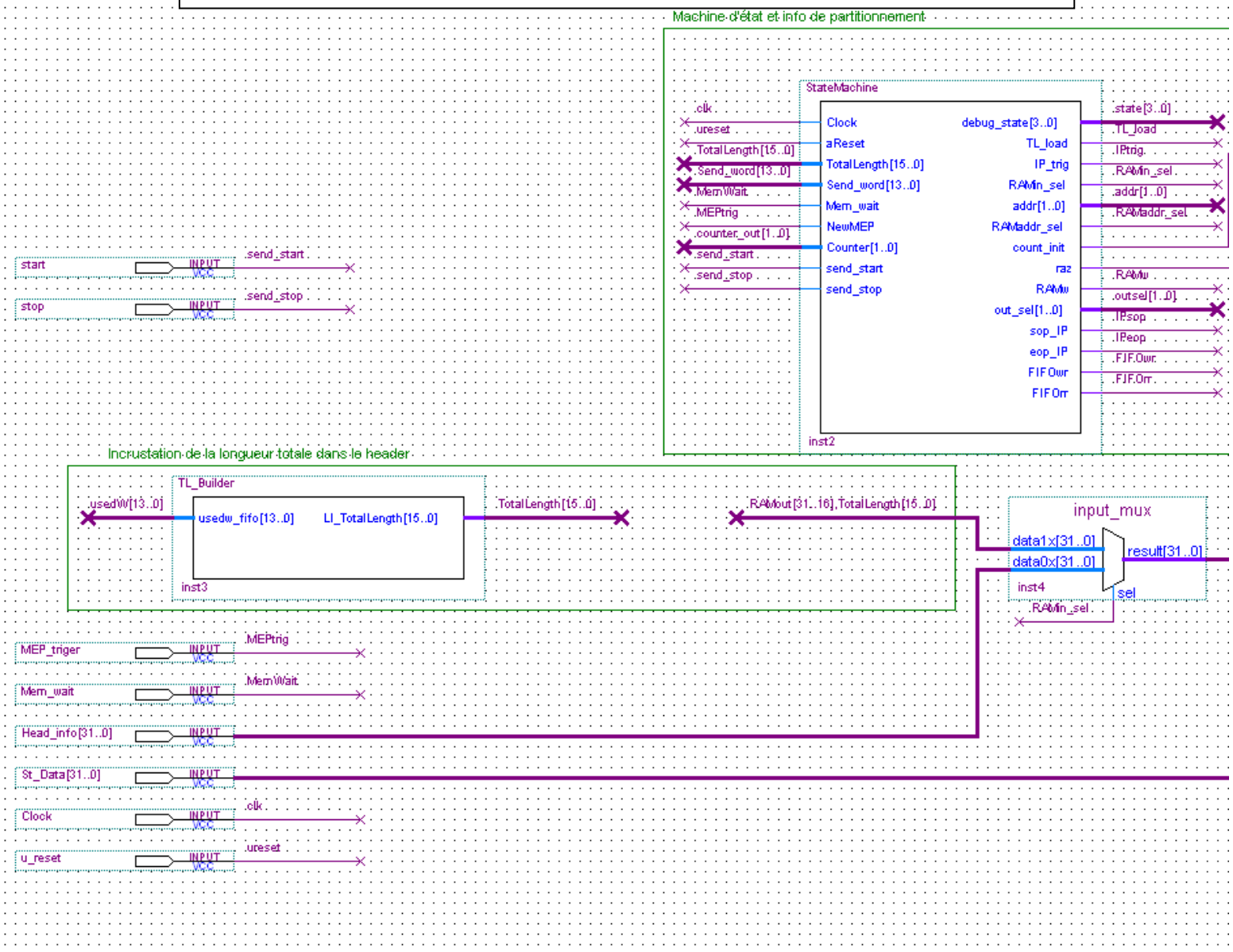
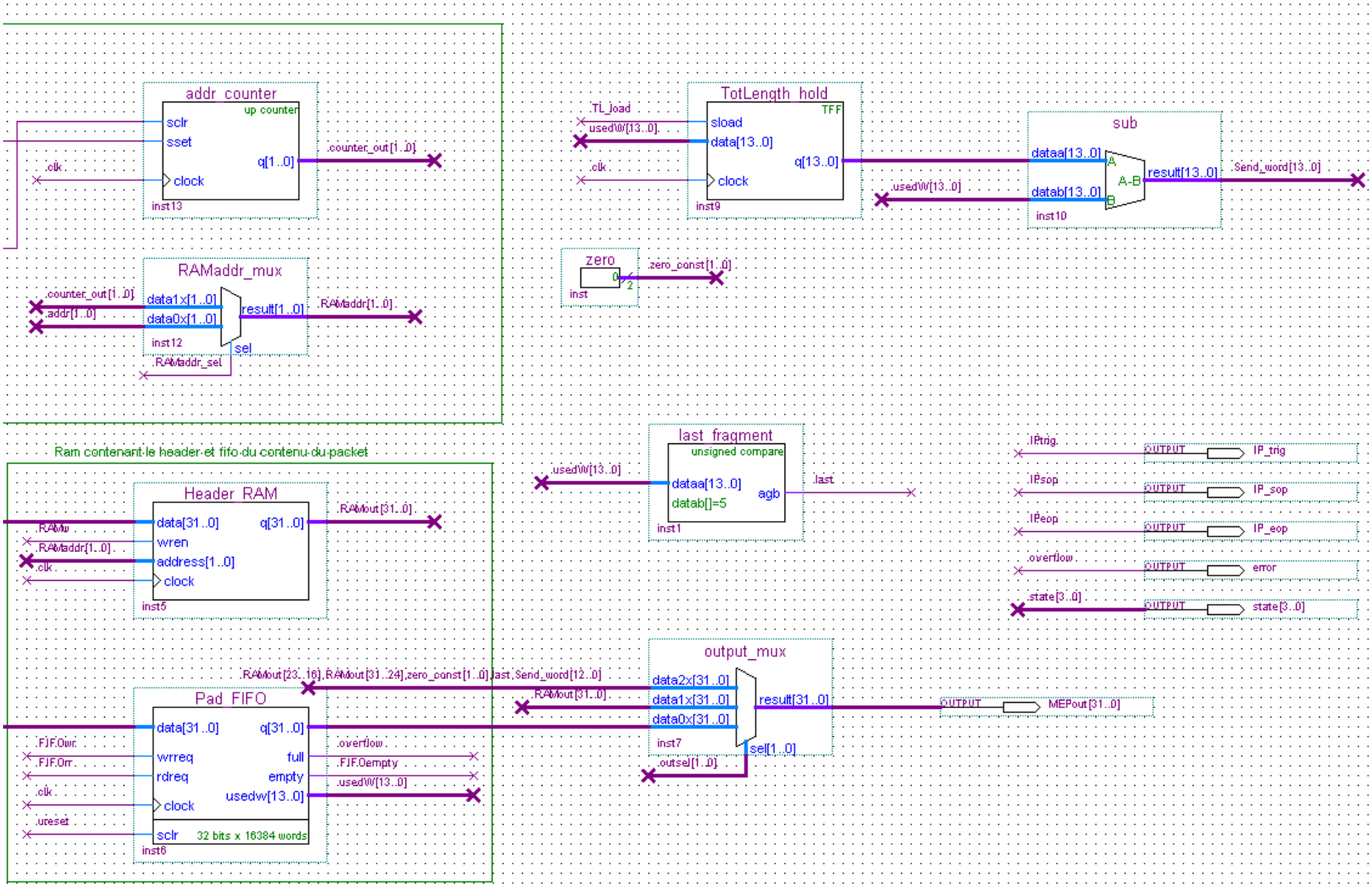


Schéma complet de l'UT du module MEP (partie 2)



Annexe II

Code VHDL des différentes unités de contrôle.

Entité du module MEP

```
--Machine d'etat du module MEP

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity StateMachine is
port (
        Clock       : in std_logic;
        aReset      : in std_logic;
        TotalLength : in std_logic_vector (15 downto 0);
        Send_word   : in std_logic_vector (13 downto 0);
        Mem_wait    : in std_logic;
        NewMEP      : in std_logic;
        Counter     : in std_logic_vector (1 downto 0);
        send_start  : in std_logic;
        send_stop   : in std_logic;

        debug_state : out std_logic_vector (3 downto 0);
        TL_load     : out std_logic;
        IP_trig     : out std_logic;
        RAMin_sel   : out std_logic;
        addr        : out std_logic_vector (1 downto 0);
        RAMaddr_sel : out std_logic;
        count_init  : out std_logic;
        raz         : out std_logic;
        RAMw        : out std_logic;
        out_sel     : out std_logic_vector (1 downto 0);
        sop_IP      : out std_logic;
        eop_IP      : out std_logic;
        FIFOwr      : out std_logic;
        FIFOrr      : out std_logic );
end StateMachine;

```

```
architecture behav of StateMachine is
    type t_state is (q0, q1, q2, q3, q4, q5, q6, q7, q8, q8a, q9, q10, q11, q12, q13, q14, q15);

    signal state        : t_state;
    signal next_state   : t_state;

    begin

    seq_state : process (Clock,aReset)
        begin
            if aReset = '1' then
                state <= q0;
            elsif Clock'event and Clock = '1' then
                state <= next_state;
            end if;
    end process seq_state;

    transition_log : process (Clock, state)
    begin

        case state is
            when q0 => next_state <= q1;
            when q1 =>
                if NewMEP = '1' then
                    next_state <= q2;
                else
                    next_state <= q1;
                end if;
            when q2 =>
                if Counter = "10" then
                    next_state <= q3;
                else
                    next_state <= q2;
                end if;
            when q3 =>
                if send_start = '1' then
                    next_state <= q4;
                else
                    next_state <= q3;
                end if;
            when q4 =>
                if send_stop = '1' then
                    next_state <= q5;
                else
                    next_state <= q4;
                end if;
            when q5 => next_state <= q6;
            when q6 =>
                if Counter = "00" then
                    next_state <= q7;
                else
                    next_state <= q6;
                end if;
            when q7 => next_state <= q8;
            when q8 => next_state <= q8a;
            when q8a => next_state <= q9;
            when q9=> next_state <= q10;
            when q10 =>
                if Counter = "01" then
                    next_state <= q11;
                else
                    next_state <= q10;
                end if;
            when q11 => next_state <= q12;
            when q12 => next_state <= q13;
            when q13 => next_state <= q14;
            when q14 =>
                if TotalLength = X"1400" or send_word = "11111111111110" then
                    next_state <= q15;
                else
                    next_state <= q14;
                end if;
            when q15 =>
                if TotalLength = X"1000" then
                    next_state <= q1;
                else
                    next_state <= q8;
                end if;
            when others => null;
        end case;
    end process transition_log;
```

```
    Output_log : process (state)
    begin
        RAMin_sel   <= '0';
        addr        <= "00";
        RAMw        <= '0';
        out_sel     <= "00";
        sop_IP      <= '0';
        eop_IP      <= '0';
        FIFOrr      <= '0';
        FIFOwr      <= '0';
        RAMaddr_sel <= '0';
        count_init  <= '0';
        raz         <= '0';
        IP_trig     <= '0';
        TL_load     <= '0';

        case state is
            when q1 => -- Counter initialisation
                count_init  <= '1';
                debug_state <= "0001";
            when q2 => -- Header data reception
                RAMw        <= '1';
                RAMaddr_sel <= '1';
                debug_state <= "0010";
            when q3 => -- Waiting for data transmition
                null;
                debug_state <= "0011";
            when q4 => -- storage of received data in the FIFO
                FIFOwr      <= '1';
                debug_state <= "0100";
            when q5 => -- Extraction of the TotalLength RAM word + memorisation of the max length
                addr        <= "01";
                raz         <= '1';
                TL_load     <= '1';
                debug_state <= "0101";
            when q6 => -- RAM responce time
                null;
                debug_state <= "0110";
            when q7 => -- Writing of the new value of the TotalLength
                RAMin_sel   <= '1';
                RAMw        <= '1';
                addr        <= "01";
                debug_state <= "0111";
            when q8 => -- Sending Partition info
                addr        <= "00";
                debug_state <= "1000";
            when q8a =>
                IP_trig     <= '1';
            when q9 =>
                raz         <= '1';
                out_sel     <= "10";
            when q10 => -- Starting the counter for header send
                RAMaddr_sel <= '1';
                debug_state <= "1001";
            when q11 => -- Sending the first word
                out_sel     <= "01";
                sop_IP      <= '1';
                RAMaddr_sel <= '1';
                debug_state <= "1010";
            when q12 => -- Sending the header
                out_sel     <= "01";
                RAMaddr_sel <= '1';
                debug_state <= "1011";
            when q13 => -- Preparing the FIFO words sending
                out_sel     <= "01";
                FIFOrr      <= '1';
                debug_state <= "1100";
            when q14 => -- Sending of FIFO words
                FIFOrr      <= '1';
                debug_state <= "1101";
            when q15 => -- Sending the last word
                eop_IP      <= '1';
                debug_state <= "1110";
            when others => null;
        end case;
    end process Output_log;
end behav;
```

Entité module IP

```
1  -- Machine d'état du module IP
2
3
4  library ieee;
5  use ieee.std_logic_1164.all;
6  use ieee.std_logic_unsigned.all;
7
8  entity StateMachine is
9  port (
10             Clock       : in std_logic;
11             areset      : in std_logic;
12             IP_sop      : in std_logic;
13             IP_eop      : in std_logic;
14             FIFOempty   : in std_logic;
15             event_trig  : in std_logic;
16             Counter     : in std_logic_vector ( 2 downto 0);
17             TotLength   : in std_logic_vector ( 15 downto 0);
18
19             Eth_sop     : out std_logic;
20             Eth_eop     : out std_logic;
21             RAMin_sel   : out std_logic_vector( 2 downto 0);
22             RAMaddr_sel : out std_logic;
23             HEADw       : out std_logic;
24             raz         : out std_logic;
25             count_init  : out std_logic;
26             addr        : out std_logic_vector( 2 downto 0);
27             CSclr       : out std_logic;
28             CSen        : out std_logic;
29             out_sel     : out std_logic;
30             FIFOrr      : out std_logic;
31             FIFOwr      : out std_logic;
32             debug_etat  : out std_logic_vector( 3 downto 0)
33             );
34 end StateMachine;
```

```
architecture behav of StateMachine is
    type t_state is (q0, q1, q2, q3, q4, q4a, q5, q6, q7, q8, q8a, q9, q10, q11, q11a, q12, q12a, q13, q14, q14a, q15, q15a);

    signal state        : t_state;
    signal next_state   : t_state;

    begin

    seq_state : process (Clock,aReset)
        begin
            if aReset = '1' then
                state <= q0;
            elsif Clock'event and Clock = '1' then
                state <= next_state;
            end if;
    end process seq_state;

    transition_log : process (Clock, state)
    begin

        case state is
            when q0 => next_state <= q1;
                debug_etat<="0000";
            when q1 =>
                if event_trig = '1' then
                    next_state <= q2;
                else
                    next_state <= q1;
                end if;
                debug_etat<="0001";
            when q2 =>
                if Counter = "101" then -- peut être changé en 100 si l'@IP viens d'ailleurs.
                    next_state <= q3;
                else
                    next_state <= q2;
                end if;
                debug_etat<="0010";
            when q3 =>
                if IP_sop = '0' then
                    next_state <= q3;
                else
                    next_state <= q4;
                end if;
                debug_etat<="0011";
            when q4 => next_state <= q4a;
                debug_etat<="0100";
            when q4a =>
                if Counter = "000" then -- "000" for 1 clock periode of waiting the RAM output, "001" for 2 ... etc
                    next_state <= q5;
                else
                    next_state <= q4a;
                end if;
            when q5 => next_state <= q6;
                debug_etat<="0101";
            when q6 =>
                if IP_eop = '0' then
                    next_state <= q6;
                else
                    next_state <= q7;
                end if;
                debug_etat<="0110";
            when q7 => next_state <= q8;
                debug_etat<="0111";
            when q8 => next_state <= q8a;
                debug_etat<="1000";
            when q8a =>
                if Counter = "000" then -- "000" for 1 clock periode of waiting the RAM output, "001" for 2 ... etc
                    next_state <= q9;
                else
                    next_state <= q8a;
                end if;
            when q9 => next_state <= q10;
                debug_etat<="1001";
```

```
            when q9 => next_state <= q10;
                debug_etat<="1001";
            when q10 =>
                if Counter = "101" then
                    next_state <= q11;
                else
                    next_state <= q10;
                end if;
                debug_etat<="1010";
            when q11 => next_state <= q11a;
                debug_etat<="1011";
            when q11a =>
                if Counter = "000" then -- "000" for 1 clock periode of waiting the RAM output, "001" for 2 ... etc
                    next_state <= q12;
                else
                    next_state <= q11a;
                end if;
            when q12 => next_state <= q12a;
                debug_etat<="1100";
            when q12a =>
                if Counter = "001" then -- "000" for 1 clock periode of waiting the RAM output, "001" for 2 ... etc
                    next_state <= q13;
                else
                    next_state <= q12a;
                end if;
            when q13 => next_state <= q14;
                debug_etat<="1101";
            when q14 =>
                if Counter = "101" then -- when Counter = "110" the word displayed on RAMout is the last word of the header (5-2=3)
                    next_state <= q14a;
                else
                    next_state <= q14;
                end if;
                debug_etat<="1110";
            when q14a =>
                next_state <= q15;
            when q15 =>
                if TotLength = X"1400" then -- when TotLength = 0x1400 it means there is only 1 word left in the fifo.
                    next_state <= q15a;
                else
                    next_state <= q15;
                end if;
                debug_etat<="1111";
            when q15a =>
                next state <= q1;
            when others => null;
        end case;
    end process transition_log;
```

```
    Output_log : process (state)
    begin
        RAMin_sel   <= "000";
        RAMaddr_sel <= '0';
        addr        <= "000";
        HEADw       <= '0';
        CSclr       <= '0';
        CSen        <= '0';
        out_sel     <= '0';
        FIFOrr      <= '0';
        FIFOwr      <= '0';
        raz         <= '0';
        count_init  <= '0';
        Eth_sop     <= '0';
        Eth_eop     <= '0';

        case state is
            when q1 => -- Waiting for a new event
                count_init  <= '1';
            when q2 => -- Storage of incomming header info
                RAMin_sel   <= "001";
                HEADw       <= '1';
                RAMaddr_sel <= '1';
            when q3 => -- Waiting for the start of packet
                null;
            when q4 => -- Point in the RAM on the Fragmentation info segment
                addr        <= "001";
                raz         <= '1';
            when q5 => -- Storage of the Frag info and reset of the RAM adresse
                addr        <= "001";
                RAMin_sel   <= "000";
                HEADw       <= '1';
            when q6 => -- Storage of Data in the fifo until eop
                FIFOwr      <= '1';
            when q7 => -- Rebuilding of the first word of the RAM with the new TotalLength
                RAMin_sel   <= "010";
                HEADw       <= '1';
            when q8 => -- RAM output now carry the word with the CheckSum (CS) field
                addr        <= "010";
                raz         <= '1';
            when q9 => -- TTL is set to 255 and CS is set to 0x0000, accumulators are reset
                addr        <= "010";
                RAMin_sel   <= "100";
                HEADw       <= '1';
                CSclr       <= '1';
                count_init  <= '1'; -- Memory pointer is set to "111"
            when q10 => -- Header words are send to accumulators
                CSen        <= '1';
                RAMaddr_sel <= '1';
            when q11 => -- Selecting the CS word
                addr        <= "010";
                raz         <= '1';
            when q12 => -- The new CS is put into the word
                addr        <= "010";
                RAMin_sel   <= "011";
                HEADw       <= '1';
                raz         <= '1';
            when q12a => -- RAMaddr_sel had to be set that early because of the RAM latency
                RAMaddr_sel <= '1';
            when q13 => -- Sending of header words
                out_sel     <= '1';
                Eth_sop     <= '1';
                RAMaddr_sel <= '1';
            when q14 =>
                out_sel     <= '1';
                RAMaddr_sel <= '1';
            when q14a => -- this state keeps out_sel up to let the last header word out and set FIFOrr
                out_sel     <= '1';
                FIFOrr      <= '1';
            when q15 => -- Sending of PAD words
                FIFOrr      <= '1';
            when q15a => -- Set up the Eth_eop for the last fifo word
                Eth_eop     <= '1';
            when others => null; -- error catching
        end case;
    end process Output_log;
end behav;
```