EPFL

Thèse n° 7689

Low-Latency High-Bandwidth Circuit and System Design for Trigger Systems in High Energy Physics

Présentée le 26 février 2021

Faculté des sciences et techniques de l'ingénieur
Laboratoire de systèmes microélectroniques
Programme doctoral en génie électrique

pour l'obtention du grade de Docteur ès Sciences

par

Marcos Vinícius SILVA OLIVEIRA

Acceptée sur proposition du jury

Prof. G. De Micheli, président du jury
Prof. Y. Leblebici, Dr A. Vachoux, directeurs de thèse
Prof. S. Al-Haj Baddar, rapporteuse
Dr S. Haas, rapporteur
Prof. A. P. Burg, rapporteur

École polytechnique fédérale de Lausanne

2021

"I suspect that whatever cannot be said clearly
is probably not being thought clearly either."
— Peter Singer

To my mother Hosana...

Acknowledgements

First and foremost, I would like to express my appreciation and gratefulness to my supervisors Prof. Yusuf Leblebici and Dr. Alain Vachoux, for giving me the opportunity to pursue my doctoral degree at EPFL. In particular, I would like to thank Dr. Alain Vachoux for his encouragement, endless support, and constant guidance, that have been decisive for completing with great success this Ph.D.

I am also very grateful to Dr. Stefan Haas, Dr. Ralf Spiwoks, Dr. Thilo Pauly, and Dr. Nick Ellis for making possible my stay at CERN and participation in the Level-1 Central Trigger group, where I found the people that helped me to enhance my engineering knowledge. In particular, I would like to express my sincere gratitude to Dr. Stefan Haas and Dr. Ralf Spiwoks. Their substantial experience, support and guidance have been an extraordinary contribution to the completion of this task.

I would like to extend my gratitude to the members of my thesis committee: Prof. Giovanni de Micheli, Prof. Sherenaz Al-Haj Baddar, Dr. Stefan Haas, and Prof. Andreas Peter Burg for offering their precious time and positive insights. Moreover, I would like to acknowledge Prof. Sherenaz Al-Haj Baddar for her guidance in a field in which I had no previous experience.

I would like to thank my beloved friends Edinei Santin, Johanie Uccelli, Blerina Gkotse, Moritz Horstmann, Eduardo Brandão, and Elia Conti. Their amity and cooperation have been a source of great encouragement.

To my beloved family for being my inspiration and motivation for everything, my thanks for supporting me and allowing me to pursue my ambition throughout my childhood. Without your support, enduring love, constant guidance, and encouragement, I would never have made it this far.

Finally, I would like to thank God for all the blessings that He has given me, all the special people surrounding me, and the gift of being able to do what I deeply love.

Geneva, November 1, 2020 Marcos Oliveira.

Abstract

The increasing luminosity in HEP (High Energy Physics) colliders demands trigger systems to be more selective. First, more information from the detector is routed to the trigger system. Second, larger parts of this information are processed together. These two requirements introduce new challenges, such as higher bandwidth and higher integration, in terms of data transfer and processing of trigger systems. Both problems have to be addressed, ensuring that hardware and firmware have low and fixed latency, and are reliable. Low latency is essential due to the limited storage available in the detector front-end pipelined memories. Fixed latency is needed because the trigger processing is pipelined, and the inputs need to be time-aligned at every processing step. Reliability is important for high trigger efficiency. If the trigger is not reliable, rare events can be discarded, and uninteresting events accepted.

This Ph.D. thesis presents the upgrade of part of the trigger system of ATLAS (A Toroidal LHC AparatuS). ATLAS is one of the detectors at the world's largest and most powerful particle collider, the LHC (Large Hadron Collider) at CERN (European Center of Nuclear Research). The online trigger system of ATLAS is segmented in two levels. The first level is implemented with custom electronics. For the parts of the first level trigger not exposed to radiation, digital processing is primarily implemented using FPGA (Field Programmable Gate Array) devices. FPGAs offer high processing capacity with low-latency and re-programmability, i.e., the capability of changing the implemented logic. The second level is built from commercial computers, network switches, and custom software. The rate of bunch of particles crossing in the interaction point is 40 MHz, and the first level (Level-1) trigger needs to reduce the rate down to 100 kHz with a very low latency of 2.5 us. Part of the Level-1 trigger system, the MUCTPI (Muon to Central Trigger Processor Interface) connects the output of the barrel and endcap muon trigger to the CTP (Central Trigger Processor), which takes the final Level-1 accept decision.

The first part of this Ph.D. thesis addresses the work on the data transfer part of the MUCTPI. Latency optimized FPGA MGT (Multi-Gigabit Transceiver) configurations have been found. Moreover, an IP (Intellectual Property) core to synchronize data from 208 SL inputs with low and fixed latency has been developed. The total data transfer and synchronization latency is below 125 ns, corresponding to 60% of the latency budget. All MUCTPI on-board and off-board high-speed serial links have been tested. The Bit Error Rate (BER) values for all links running

at 12.8 Gb/s have been measured as lower than one bit error per day with a confidence level of 95%. This value is acceptable as it corresponds to only one potential fake trigger or lost event per day.

The second part of this thesis covers the development of the MUCTPI sorting network and its FPGA implementation using RTL (Register-Transfer Level) and HLS (High-Level Synthesis) approaches. Both approaches achieved a very low latency value of 31.25 ns, corresponding to only 15% of the latency budget. HLS provided advantages such as requiring much less design effort, enabling early testing, and having slightly higher performance in terms of timing slack and logic resource usage.

Keywords: HEP, ATLAS, low-level trigger, MUCTPI, FPGA, MGT, low-latency, fixed-latency, reliability, BER, statistical eye-diagrams, sorting networks, RTL, HLS.

Résumé

La luminosité croissante des collisionneurs HEP (High Energy Physics) exige que les systèmes de déclenchement soient plus sélectifs. Tout d'abord, davantage d'informations provenant du détecteur sont acheminées vers le système de déclenchement. Ensuite, des parties plus importantes de ces informations sont traitées simultanément. Ces deux exigences introduisent de nouveaux défis, tels qu'une plus grande largeur de bande et une meilleure intégration, en termes de transfert de données et de traitement des systèmes de déclenchement. Ces deux problèmes doivent être résolus, en veillant à ce que le matériel et les microprogrammes aient une latence faible et fixe, et soient fiables. Une faible latence est essentielle en raison du stockage limité disponible dans les mémoires en pipeline du frontal du détecteur. Une latence fixe est nécessaire car le traitement des déclencheurs est en pipeline et les entrées doivent être alignées dans le temps à chaque étape du traitement. La fiabilité est importante pour une efficacité élevée des déclencheurs. Si le déclencheur n'est pas fiable, les événements rares peuvent être rejetés et les événements inintéressants acceptés.

Cette thèse de doctorat présente la mise à niveau d'une partie du système de déclenchement d'ATLAS (A Toroidal LHC AparatuS). ATLAS est l'un des détecteurs du plus grand et du plus puissant collisionneur de particules au monde, le LHC (Large Hadron Collider) du CERN (Centre européen de recherche nucléaire). Le système de déclenchement en ligne d'ATLAS est segmenté en deux niveaux. Le premier niveau est réalisé avec une électronique personnalisée. Pour les parties du déclencheur du premier niveau qui ne sont pas exposées aux rayonnements, le traitement numérique est principalement mis en œuvre à l'aide de dispositifs FPGA (Field Programmable Gate Array). Les FPGA offrent une grande capacité de traitement avec une faible latence et une reprogrammabilité, c'est-à-dire la possibilité de modifier la logique mise en œuvre. Le deuxième niveau est construit à partir d'ordinateurs commerciaux, de commutateurs de réseau et de logiciels personnalisés. Le taux de groupes de particules au point d'interaction est de 40 MHz, et le déclencheur de premier niveau (Level-1) doit réduire le taux à 100 kHz avec une latence très faible de 2,5 us. Faisant partie du système de déclenchement de niveau 1, le MUCTPI (Muon to Central Trigger Processor Interface) connecte la sortie du canon et le bouchon du déclencheur muon au CTP (Central Trigger Processor), qui prend la décision finale d'acceptation de niveau 1.

Résumé

La première partie de cette thèse de doctorat porte sur les travaux relatifs à la partie transfert de données du MUCTPI. Des configurations FPGA MGT (Multi-Gigabit Transceiver) à latence optimisée ont été réalisées. De plus, un module IP (propriété intellectuelle) permettant de synchroniser les données de 208 entrées SL avec une latence faible et fixe a été développé. La latence totale de transfert et de synchronisation des données est inférieure à 125 ns, ce qui correspond à 60% du budget de latence. Toutes les liaisons séries à haut débit embarquées et non embarquées du MUCTPI ont été testées. Les valeurs du Bit Error Rate (BER) pour toutes les liaisons fonctionnant à 12,8 Gb/s ont été mesurées comme étant inférieures à une erreur sur les bits par jour avec un niveau de confiance de 95%. Cette valeur est acceptable car elle correspond à un seul faux déclencheur potentiel ou événement perdu par jour.

La deuxième partie de cette thèse couvre le développement du réseau de tri MUCTPI et sa mise en œuvre FPGA en utilisant les approches RTL (Register-Transfer Level) et HLS (High-Level Synthesis). Les deux approches ont permis d'atteindre une valeur de latence très faible de 31,25 ns, ce qui correspond à seulement 15% du budget de latence. L'approche HLS présente des avantages tels que le fait de nécessiter beaucoup moins d'efforts de conception, de permettre des tests précoces et d'avoir des performances légèrement supérieures en termes de timing et d'utilisation des ressources logiques.

Mots-clés : HEP, ATLAS, déclenchement de bas niveau, MUCTPI, FPGA, MGT, faible latence, latence fixe, fiabilité, BER, diagrammes oculaires statistiques, réseaux de tri, RTL, HLS.

Estratto

L'incremento della luminosità negli acceleratori di particelle per la fisica delle alte energie (High Energy Physics, HEP) richiede una maggiore accuratezza dei sistemi di trigger, utilizzati per selezionare dati di interesse. In primo luogo, una maggiore quantità di informazione dal rivelatore viene indirizzata al sistema di trigger. In secondo luogo, porzioni più grandi di questa informazione sono elaborati assieme. Questi due requisiti introducono nuove sfide per la progettazione di sistemi di trigger in termini di trasferimento dati ed elaborazione, quali una maggiore larghezza di banda e una maggiore integrazione. Entrambi i problemi debbono essere affrontati, assicurando che sia l'hardware che il firmware abbiano una latenza contenuta e fissa e siano affidabili. Una bassa latenza è essenziale a causa della limitata capacità di archiviazione disponibile nelle memorie pipeline del front-end del rivelatore. Una latenza fissa è, invece, necessaria poiché l'elaborazione del trigger è parallelizzata e i dati in ingresso devono essere allineati temporalmente in ogni fase dell'elaborazione. L'affidabilità è importante per avere un'elevata efficienza del trigger. Se il trigger non è affidabile, degli eventi rari possono essere scartati, mentre quelli non di interesse possono essere accettati.

Questa tesi di dottorato presenta l'aggiornamento di parte del sistema di trigger di ATLAS (A Toroidal LHC ApparatuS). ATLAS è uno dei rivelatori installati presso il più grande al mondo e potente acceleratore di particelle, il Large Hadron Collider (LHC) presso il Centro Europeo per la Ricerca Nucleare (CERN). Il sistema di trigger online di ATLAS è segmentato in due livelli. Il primo livello è implementato con elettronica su misura. Per le parti del trigger di primo livello non esposte a radiazione, l'elaborazione digitale è implementata principalmente con logica programmabile (dispositivi Field Programmable Gate Array, FPGA). Gli FPGA offrono elevate capacità di elaborazione con bassa latenza e riprogrammabilità, che consiste nella possibilità di cambiare la logica implementata. Il secondo livello di trigger è costituito da computer commerciali, switch di rete e software apposito. La frequenza con cui un gruppo (bunch) di particelle attraversa il rivelatore al punto di interazione è 40 MHz, e il trigger di primo livello (Level-1) deve ridurre la frequenza di dati di interesse a 100 kHz con una latenza molto bassa di 2.5 us. Parte del sistema di trigger Level-1, l'interfaccia MUCTPI (Muon to Central Trigger Processor Interface), connette l'uscita del trigger associato alle camere a muoni sul cilindro e alle estremità del rivelatore all'elaboratore centrale del trigger (Central Trigger Processor, CTP),

che prende la decisione finale di accettazione del Level-1.

La prima parte di questa tesi di dottorato affronta il lavoro sulla parte di trasferimento dati del MUCTPI. Sono state trovate delle configurazioni di ricetrasmettitori a multi-gigabit (Multi-Gigabit Transceiver, MGT) su FPGA ottimizzate per la latenza. Inoltre, è stato sviluppato un blocco funzionale di proprietà intellettuale (IP) per sincronizzare i dati provenienti da 208 ingressi SL a bassa e fissa latenza. La latenza totale di trasferimento dati e sincronizzazione è inferiore a 125 ns, che corrispondono al 60% del bilancio totale della latenza. Sono stati testati tutti i collegamenti seriali ad alta velocità del MUCTPI sulla scheda e fuori dalla scheda. Per tutti i collegamenti a 12.8 Gb/s è stata misurata una incidenza di errori per bit (bit error rate, BER) inferiore a un errore su bit al giorno con un livello di confidenza del 95%. Questo valore è accettabile poiché corrisponde a solo un potenziale trigger falso o evento perduto al giorno.

La seconda parte di questa tesi affronta la progettazione della rete di ordinamento del MUCTPI e la sua implementazione su FPGA in RTL (Register Transfer Level) e con HLS (High Level Synthesis). Entrambi gli approcci hanno ottenuto un valore di latenza molto basso di 31.25 ns, che corrispondono al 15% del bilancio complessivo di latenza. L'approccio HLS ha fornito vantaggi quali un minore sforzo di progettazione, favorendone il test anticipato, nonché delle prestazioni leggermente maggiori in termini di slack temporale e utilizzo di risorse logiche.

Parole chiave: HEP, ATLAS, trigger a basso livello, MUCTPI, FPGA, MGT, bassa latenza, latenza fissa, affidabilità, BER, diagrammi oculari statistici, reti di smistamento, RTL, HLS.

Kurzfassung

Die zunehmende Luminosität in HEP (High Energy Physics)-Beschleunigern erfordert eine höhere Selektivität der Triggersysteme. Erstens werden mehr Informationen vom Detektor an das Triggersystem gesendet. Zweitens werden größere Teile dieser Informationen gemeinsam verarbeitet. Diese beiden Anforderungen bringen neue Herausforderungen mit sich, wie eine höhere Bandbreite und eine höhere Integration, was die Datenübertragung und die Verarbeitung durch die Triggersysteme betrifft. Beide Probleme müssen angegangen werden, wobei sichergestellt werden muss, dass Hardware und Firmware eine niedrige und feste Latenzzeit haben und zuverlässig sind. Niedrige Latenzzeiten sind aufgrund der begrenzten Kapazität, die in den Warteschlangen-Speichern des Detektor-Front-Ends zur Verfügung steht, unerlässlich. Eine feste Latenz ist erforderlich, da die Triggerverarbeitung in einer Warteschlange erfolgt und die Eingänge bei jedem Verarbeitungsschritt zeitlich ausgerichtet werden müssen. Zuverlässigkeit ist wichtig für eine hohe Triggereffizienz. Wenn der Trigger nicht zuverlässig ist, können seltene Ereignisse verworfen und uninteressante Ereignisse akzeptiert werden.

In dieser Doktorarbeit wird das Upgrade eines Teils des Triggersystems von ATLAS (A Toroidal LHC AparatuS) vorgestellt. ATLAS ist einer der Detektoren am größten und leistungsstärksten Teilchenbeschleuniger der Welt, dem LHC (Large Hadron Collider) am CERN (Europäisches Zentrum für Kernforschung). Das Online-Triggersystem von ATLAS ist in zwei Level aufgeteilt. Der erste Level ist mit anwendugsspezifischer Elektronik implementiert. Für diestrahlungsunbelasteten Teile des Triggers des ersten Levels wird die digitale Verarbeitung hauptsächlich mit FPGA (Field Programmable Gate Array)-Geräten realisiert. FPGAs bieten eine hohe Verarbeitungsleistung mit geringer Latenz und bieten durch Reprogrammierbarkeit die Möglichkeit, die implementierte Logik zu ändern. Der zweite Triggerlevel wird aus kommerziellen Computern, Netzwerk-Switches und anwendungsspezifischer Software aufgebaut. Die Rate der sich im Interaktionspunkt kreuzenden Teilchenpakete beträgt 40 MHz, und der Trigger des ersten Levels (Level-1) muss diese Rate auf 100 kHz während der sehr geringen Latenz von 2,5us reduzieren. Als Teil des Level-1-Triggersystems verbindet das MUCTPI (Muon to Central Trigger Processor Interface) den Ausgang des Barrel und Endcap-Muon-Triggers mit dem CTP (Central Trigger Processor), der die endgültige Level-1-Annahmeentscheidung trifft.

Kurzfassung

Der erste Teil dieser Dissertation befasst sich mit der Arbeit am Datentransferteil des MUCTPI (Muon to Central Trigger Processor Interface). Es wurden latenzoptimierte FPGA MGT(Multi-Gigabit Transceiver) Konfigurationen ermittelt. Darüber hinaus wurde ein IP (Intellectual Property)-Kern entwickelt, um Daten von 208 SL-Eingängen mit niedriger und fester Latenz zu synchronisieren. Die gesamte Datenübertragungs- und Synchronisationslatenz liegt unter 125 ns, was 60% des Latenzbudgets entspricht. Alle on- und off-board seriellen Hochgeschwindigkeits-MUCTPI-Verbindungen wurden getestet. Die Werte der Bitfehlerrate(BER) für alle Verbindungen, die mit 12,8 Gb/s betrieben werden, wurden als niedriger als ein Bitfehler pro Tag mit einem Konfidenzniveau von 95% gemessen. Dieser Wert ist akzeptabel, da er nur einem potenziellen falschem Auslösen oder verlorenen Ereignis pro Tag entspricht.

Der zweite Teil dieser Arbeit befasst sich mit der Entwicklung des MUCTPI-Sortiernetzwerks und seiner FPGA-Implementierung unter Verwendung von RTL- (Register-Transfer-Level) und HLS- (High-Level-Synthesis) Ansätzen. Beide Ansätze erreichten einen sehr niedrigen Latenzwert von 31,25 ns, was nur 15% des Latenzbudgets entspricht. HLS bot Vorteile, wie z.B. einen wesentlich geringeren Designaufwand, die Möglichkeit frühzeitiger Tests und eine etwas höhere Leistung in Bezug auf Timing Slack und Nutzung der Logikressourcen.

Schlüsselwörter: HEP, ATLAS, Low-Level-Trigger, MUCTPI, FPGA, MGT, niedrige Latenz, Feste Latenz, Zuverlässigkeit, BER, statistische Augendiagramme, Sortiernetzwerke, RTL, HLS.

Resumo

A luminosidade crescente dos colididores HEP (High Energy Physics) exige que os sistemas de filtragem sejam mais seletivos. Primeiro, mais informações do detector são encaminhadas para o sistema de filtragem. Em segundo lugar, partes maiores destas informações são processadas em conjunto. Estes dois requisitos introduzem novos desafios, tais como maior largura de banda e maior integração, em termos de transferência de dados e processamento dos sistemas de filtragem. Ambos os problemas devem ser tratados, assegurando que o hardware e o firmware tenham latência baixa e fixa, e sejam confiáveis. A baixa latência é essencial devido ao armazenamento limitado disponível nas memórias implementadas dentro do detector. A latência fixa é necessária porque o processamento de filtragem é executado em cadeia, e as entradas precisam ser alinhadas no tempo em cada etapa do processamento. A confiabilidade é importante para a alta eficiência da filtragem. Se a seleção não for confiável, eventos raros podem ser descartados e eventos desinteressantes podem ser aceitos.

Esta tese de doutorado apresenta a atualização de parte do sistema de filtragem do ATLAS (A Toroidal LHC AparatuS). ATLAS é um dos detectores do maior e mais potente colisor de partículas do mundo, o LHC (Large Hadron Collider) no CERN (Centro Europeu de Pesquisa Nuclear). O sistema de filtragem online do ATLAS é segmentado em dois níveis. O primeiro nível é implementado com eletrônica personalizada. Para as partes do sistema de filtragem do primeiro nível não expostas à radiação, o processamento digital é implementado principalmente usando dispositivos FPGA (Field Programmable Gate Array). Os FPGAs oferecem alta capacidade de processamento com baixa latência e reprogramabilidade, ou seja, a capacidade de alterar a lógica implementada. O segundo nível é construído a partir de computadores comerciais, switches de rede e software personalizado. A taxa de colisão de partículas no ponto de interação é de 40 MHz, e o primeiro nível (Nível-1) de filtragem precisa reduzir a taxa para 100 kHz com uma latência muito baixa de 2,5 us. Parte do sistema de filtragem de Nível 1, o MUCTPI (Muon to Central Trigger Processor Interface) conecta a saída dos sistemas de filtragem de muons ao CTP (Central Trigger Processor), que toma a decisão final de aceitação do Nível 1.

A primeira parte desta tese de doutorado aborda o trabalho na parte de transferência de dados do MUCTPI. Foram encontradas configurações otimizadas de latência para dispositivos MGT

(Multi-Gigabit Transceiver) implementados em FPGA. Além disso, foi desenvolvido uma IP (Propriedade Intelectual) para sincronizar dados de 208 entradas SL com latência baixa e fixa. A latência total de transferência e sincronização de dados está abaixo de 125 ns, o que corresponde a 60% do limite de latência. Todos os links seriais de alta velocidade MUCTPI on-board e off-board foram testados. Os valores de Bit Error Rate (BER) para todos os links operando em 12,8 Gb/s foram medidos como erros inferiores a um bit por dia com um nível de confiança de 95%. Este valor é aceitável, pois corresponde a apenas um falso disparo ou evento perdido por dia.

A segunda parte desta tese aborda o desenvolvimento da rede de classificação do MUCTPI e sua implementação FPGA usando abordagens RTL (Register-Transfer Level) e HLS (High-Level Synthesis). Ambas as abordagens alcançaram um valor de latência muito baixo de 31,25 ns, o que corresponde a apenas 15% do limite de latência. O HLS proporcionou vantagens como, por exemplo, exigir muito menos esforço de projeto, permitir testes iniciais e ter um desempenho ligeiramente superior em termos de desempenho e uso de recursos lógicos.

Palavras-chave: HEP, ATLAS, filtragem de baixo nível, MUCTPI, FPGA, MGT, baixa latência, latência fixa, confiabilidade, BER, diagramas estatísticos de olho, redes de classificação, RTL, HLS.

Contents

Acknowledgements **i**

Abstract (English/Français/Italiano/Deutsch/Português) **iii**

List of Figures **xix**

List of Tables **xxiii**

Acronyms **xxv**

1 Introduction **1**
1.1 Low-latency high-throughput circuit applications 2
1.2 CERN, LHC and ATLAS 3
1.3 The trigger and data acquisition system 4
1.4 The L1 trigger system 6
1.5 ATLAS muon trigger 8
1.6 MUCTPI 9
1.7 Thesis motivation 10
1.8 Thesis organization 11

2 MUCTPI Upgrade **13**
2.1 Motivation 13
2.2 MUCTPI architecture 16
2.2.1 Muon Sector Processor 17
2.2.2 Trigger, Readout, and TTC processor 18
2.2.3 System on chip 19
2.2.4 On-board connectivity 19
2.2.5 Off-board connectivity 19
2.3 Summary 20

I Data transfer **21**

3 High-speed serial link testing **23**
3.1 Introduction . . . 23
3.1.1 Bit Error Rate . . . 24
3.1.2 Eye-diagram . . . 26
3.1.3 Statistical eye-diagram . . . 27
3.1.4 Eye mask compliance . . . 28
3.2 MUCTPI demonstrator . . . 29
3.3 Bit-error-rate test firmware . . . 30
3.4 Bit-error-rate test software . . . 31
3.5 Test laboratory results . . . 34
3.5.1 BER test . . . 34
3.5.2 High-speed oscilloscope eye diagram . . . 35
3.5.3 Statistical eye-diagram . . . 35
3.5.4 Eye opening area study . . . 36
3.5.5 Eye-diagram mask compliance test . . . 40
3.6 Integration test results . . . 40
3.6.1 RPC and TGC sector logic modules . . . 40
3.6.2 L1Topo . . . 43
3.7 Summary . . . 46

4 FPGA transceiver latency optimization **49**
4.1 Introduction . . . 49
4.2 FPGA transceivers . . . 49
4.3 Latency optimization . . . 51
4.3.1 Latency evaluation test system . . . 51
4.3.2 Data path latency test results . . . 52
4.3.3 Clock fabric latency uncertainty test results . . . 53
4.4 Summary . . . 57

5 Synchronization and Alignment **59**
5.1 Introduction . . . 59
5.2 Data frame format . . . 60
5.3 Requirements . . . 62
5.4 Firmware . . . 64
5.5 Functional simulation . . . 66
5.5.1 Work environment . . . 67
5.5.2 Unit test . . . 67
5.5.3 Reference and running phase offset test . . . 67

5.5.4 Latency variation effect in the memory write side . . . 69
5.5.5 Metastability effect on the memory write side . . . 72
5.5.6 Addressing latency variation in the memory write side . . . 75
5.5.7 Finding the error-free read pointer offsets . . . 80
5.5.8 Addressing latency variation in the memory read side . . . 84
5.5.9 Latency simulation . . . 88
5.5.10 Output phase variation . . . 93
5.5.11 Summary . . . 93
5.6 Integration tests . . . 94
5.7 Summary . . . 97

II Data processing **99**

6 Data processing issues and challenges **101**
6.1 Introduction . . . 101
6.2 Trigger unit . . . 104
6.3 Sorting unit used in Run 2 . . . 105
6.3.1 Comparison . . . 105
6.3.2 Selection . . . 106
6.3.3 Multiplexing . . . 107
6.4 Implementation results . . . 107
6.5 Summary . . . 109

7 Sorting Networks **111**
7.1 Introduction . . . 111
7.2 Introduction to merging and sorting networks . . . 112
7.2.1 Zero-one principle . . . 113
7.3 Batcher merge-exchange sorting algorithm . . . 114
7.4 Batcher odd-even and bitonic merging networks . . . 117
7.5 Odd-even and bitonic mergesort networks . . . 120
7.6 Special sorting networks . . . 122
7.6.1 David C. Van Voorhis 16-key sorting network . . . 123
7.6.2 Sherenaz W. Al-Haj Baddar 22-key sorting network . . . 123
7.7 Network optimisations . . . 123
7.7.1 Input and output optimisation . . . 125
7.7.2 Pre-sorted input and unsorted output optimisation . . . 126
7.8 Batcher sorting methods comparison . . . 129
7.8.1 Delay . . . 129
7.8.2 Number of comparisons . . . 130
7.8.3 Summary . . . 132

7.9 Divide-and-conquer method . . . 132
7.10 MUCTPI sorting network . . . 139
7.11 Validation of MUCTPI sorting network . . . 145
7.12 Summary . . . 145

8 Implementation approaches 149
8.1 Introduction . . . 149
8.1.1 Sorting unit . . . 149
8.1.2 RTL and HLS design flows . . . 150
8.2 RTL implementation . . . 153
8.2.1 Combinational-only sorting networks . . . 153
8.2.2 Pipelined sorting networks . . . 154
8.2.3 Pipelining configurations . . . 156
8.2.4 Hierarchical options . . . 158
8.2.5 Architecture options . . . 158
8.2.6 Generating VHDL code . . . 159
8.2.7 Vendor-specific design flow . . . 160
8.2.8 Design verification . . . 161
8.2.9 Implementation results . . . 162
8.3 HLS implementation . . . 167
8.3.1 Data Structure . . . 167
8.3.2 Comparison-exchange unit . . . 167
8.3.3 Network pairs header . . . 170
8.3.4 Top-level without multiplexor . . . 170
8.3.5 Top-level with multiplexor . . . 172
8.3.6 Exploring different solutions . . . 174
8.3.7 Vendor-specific design flow . . . 174
8.3.8 Implementation results . . . 177
8.4 Comparative study . . . 181
8.4.1 Design exploration effort . . . 182
8.4.2 Performance metrics . . . 183
8.5 Summary . . . 184

9 Conclusions and Outlook 187
9.1 Data transfer . . . 188
9.2 Data processing . . . 190
9.3 Outlook . . . 193

A RTL description of the sorting unit 195

Bibliography **205**

Curriculum Vitae **213**

List of Figures

1.1 Overview of the ATLAS experiment . . . 3
1.2 Overview of the ATLAS TDAQ system for Run 3 . . . 5
1.3 Overview of the ATLAS L1 trigger system for Run 3 . . . 7
1.4 Ph.D. thesis context diagram . . . 10

2.1 LHC plan . . . 13
2.2 Legacy MUCTPI system . . . 15
2.3 MUCTPI architecture . . . 16
2.4 MUCTPI prototype version 1 . . . 18

3.1 Block diagram of the eye diagram measurement . . . 26
3.2 Eye-diagram of two MIOCT outputs operating at 320 Mbps . . . 26
3.3 Statistical eye diagram example . . . 27
3.4 MiniPOD eye-diagram mask . . . 29
3.5 Eye diagram with mask . . . 29
3.6 MUCTPI system demonstrator . . . 30
3.7 IBERT firmware and connectivity block diagram . . . 31
3.8 Serial link test automation . . . 32
3.9 IBERTpy generated report . . . 33
3.10 Oscilloscope eye diagram of one MSP MGT output running at 11.2 Gb/s . . . 35
3.11 MUCTPI V1 eye-diagram running at 6.4 Gb/s . . . 36
3.12 MUCTPI V2 eye-diagram running at 6.4 Gb/s . . . 36
3.13 MUCTPI V3 eye-diagram running at 6.4 Gb/s . . . 36
3.14 MUCTPI V1 eye-diagram running at 12.8 Gb/s . . . 36
3.15 MUCTPI V2 eye-diagram running at 12.8 Gb/s . . . 36
3.16 MUCTPI V3 eye-diagram running at 12.8 Gb/s . . . 36
3.17 OAPH MUCTPI-V1 SL 6.4 Gb/s . . . 38
3.18 OAPH MUCTPI-V2 SL 6.4 Gb/s . . . 38
3.19 OAPH MUCTPI-V3 SL 6.4 Gb/s . . . 38
3.20 OAPH MUCTPI-V1 SL 12.8 Gb/s . . . 39
3.21 OAPH MUCTPI-V2 SL 12.8 Gb/s . . . 39

3.22 OAPH MUCTPI-V3E SL 12.8 Gb/s . . . 39
3.23 MUCTPI V3 worst-case eye-diagram mask check 6.4 Gb/s . . . 41
3.24 MUCTPI V3 best-case eye-diagram mask check 6.4 Gb/s . . . 41
3.25 MUCTPI V3 worst-case eye-diagram mask check 12.8 Gb/s . . . 41
3.26 MUCTPI V3 best-case eye-diagram mask check 12.8 Gb/s . . . 41
3.27 TGC integration test block diagram . . . 42
3.28 TGC channel 0 to MUCTPI eye-diagram running at 6.4 Gb/s . . . 42
3.29 TGC channel 1 to MUCTPI eye-diagram running at 6.4 Gb/s . . . 42
3.30 TGC channel 2 to MUCTPI eye-diagram running at 6.4 Gb/s . . . 42
3.31 TGC channel 3 to MUCTPI eye-diagram running at 6.4 Gb/s . . . 42
3.32 TGC channel 4 to MUCTPI eye-diagram running at 6.4 Gb/s . . . 42
3.33 TGC channel 5 to MUCTPI eye-diagram running at 6.4 Gb/s . . . 42
3.34 TGC channel 6 to MUCTPI eye-diagram running at 6.4 Gb/s . . . 42
3.35 TGC channel 7 to MUCTPI eye-diagram running at 6.4 Gb/s . . . 42
3.36 TGC channel 8 to MUCTPI eye-diagram running at 6.4 Gb/s . . . 42
3.37 TGC channel 9 to MUCTPI eye-diagram running at 6.4 Gb/s . . . 42
3.38 TGC channel 10 to MUCTPI eye-diagram running at 6.4 Gb/s . . . 42
3.39 TGC channel 11 to MUCTPI eye-diagram running at 6.4 Gb/s . . . 42
3.40 RPC eye-diagram . . . 43
3.41 RPC eye-diagram 7 dB . . . 43
3.42 Best L1Topo eye-diagram . . . 44
3.43 Worst L1Topo eye-diagram . . . 44
3.44 Best eye-diagram 1.25 dB . . . 44
3.45 Best eye-diagram 5.25 dB . . . 44
3.46 Best eye-diagram 7.25 dB . . . 45
3.47 Best eye-diagram 7.75 dB . . . 45
3.48 Best eye-diagram 8.25 dB . . . 45
3.49 Best eye-diagram 9.25 dB . . . 45
3.50 Worst eye-diagram 5.25 dB . . . 46
3.51 Worst eye-diagram 7.25 dB . . . 46
3.52 Worst eye-diagram 8.25 dB . . . 46
3.53 Worst eye-diagram 9.25 dB . . . 46

4.1 Simplified block diagram of a FPGA-based high-speed data transfer scheme . . . 50
4.2 Latency measurement test system block diagram . . . 52
4.3 GTY TX latency-optimized data path . . . 53
4.4 GTY RX latency-optimized data path . . . 54
4.5 Latency uncertainty measurement before optimization in the clock fabric . . . 55
4.6 Latency variation when TXOUTCLK = TXPROGDIVCLK . . . 55
4.7 Latency variation when TXOUTCLK = TXPLLREFCLK_DIV1 . . . 55

4.8 Latency-fixed transmitter clock fabric configuration 56
4.9 Optimized receiver clock fabric configuration 58

5.1 Block diagram of a FPGA-based high-speed data transfer scheme 60
5.2 Timing diagram with Φ^{s}_{rec} and Φ^{a}_{rec} definition 63
5.3 MUCTPI synchronization block diagram 65
5.4 Unit test block diagram 68
5.5 Reference and running phase offset dataset 70
5.6 BCID error for align delay set to 0 71
5.7 Late Φ^{run}_{rec} waveform 71
5.8 Early Φ^{run}_{rec} waveform 71
5.9 Metastability effect on the write alignment pulse propagation delay 73
5.10 Alignment delay iteration example for a RPC input 77
5.11 BCID change and frame-center values 78
5.12 BCID error for align delay frame-center value 79
5.13 RPC BCID offset and CRC error values 81
5.14 TGC BCID offset and CRC error values 82
5.15 Minimum-latency RPC BCID or CRC error value 85
5.16 Minimum-latency TGC BCID or CRC error value 85
5.17 Maximum-latency RPC BCID or CRC error value 86
5.18 Maximum-latency TGC BCID or CRC error value 86
5.19 Minimum-latency RPC BCID or CRC error with $V_L = -1$ and $V_R = 1$ 89
5.20 Minimum-latency TGC BCID or CRC error with $V_L = -1$ and $V_R = 1$ 89
5.21 Maximum-latency RPC and TGC BCID or CRC error with $V_L = -1$ and $V_R = 1$ . . 90
5.22 RPC and TGC BCID or CRC error with $V_L = -3$ and $V_R = 4$ 90
5.23 RPC synchronization latency minimum latency with $V_L = -1$ and $V_R = 1$ 91
5.24 TGC synchronization latency minimum latency with $V_L = -1$ and $V_R = 1$ 91
5.25 Synchronization unit latency Δ_t 92
5.26 RPC and TGC output phase minimum latency with $V_L = -1$ and $V_R = 1$ 94
5.27 RPC and TGC integration test block diagram 95
5.28 RPC to MUCTPI latency measurement waveform 97
5.29 TGC to MUCTPI latency measurement waveform 98

6.1 MSP block diagram 102
6.2 Online serial link eye-diagram 104
6.3 MSP trigger block diagram 105
6.4 Logic diagram for a 6-input one-hot multiplexor 107
6.5 Number of comparators and LUTs for up to 104 muon candidates 108

7.1 Comparison-exchange module for ascending order output 113

7.2 Comparison-exchange module for descending order output 113
7.3 Single comparison 114
7.4 4-key sorting network 114
7.5 Knuth diagram of the Batcher merge-exchange sorting network with $n = 8$ . . . 116
7.6 Knuth diagram of the Batcher ($m = 4$, $n = 4$) odd-even merging network 119
7.7 Knuth diagram of the Batcher odd-even mergesort network with $n = 8$ 121
7.8 Knuth diagram of the Batcher bitonic mergesort network with $n = 8$ 122
7.9 Knuth diagram of the Voorhis 16-Key sorting network 124
7.10 Knuth diagram of the Baddar 22-Key sorting network 125
7.11 Knuth diagram of a 6-key sorting network 126
7.12 Knuth diagram of 8-key input 2-key output sorting network 127
7.13 Knuth diagram of a particular 8-key permutation network 128
7.14 Delay for Batcher sorting networks 129
7.15 Number of comparisons for Batcher sorting networks 131
7.16 Example of a 352-key input 16-key output sorting network block diagram . . . 133
7.17 Selected 352-key input 16-key output sorting network with $R = 16$ 140
7.18 Knuth diagram of the S-network 141
7.19 Knuth diagram of the M-network 142
7.20 Knuth diagram of the MUCTPI sorting network 144

8.1 RTL and HLS design flows 151
8.2 Comparison-exchange unit 153
8.3 Bypass unit 153
8.4 4-key sorting implementation 154
8.5 4-key sorting network 155
8.6 CR unit 156
8.7 BR unit 156
8.8 Block diagram 8-key merge-exchange sorting network 157
8.9 Xilinx Vivado HLS design flow 175

List of Tables

2.1 FPGA used in each of the three prototype versions 17

5.1 RPC SL Data Format 61
5.2 TGC SL Data Format 61
5.3 RPC data frame combinations 83
5.4 TGC data frame combinations 83
5.5 Read pointer offset values 83
5.6 Latency-variation-tolerant read pointer offset values 87
5.7 Latency values for the MUCTPI given in ns 93

6.1 Comparison matrix sorting five elements parallel processing approach 106

7.1 Values of p, q, r, and d mergesort algorithm for N=8 115
7.2 22 divide-and-conquer options 352-to-16-key sorting network 135
7.3 The two fastest divide-and-conquer options 352-to-16-key sorting network . . 137
7.4 22-key input 16-key output baddar22 sorting network pairs 143
7.5 32-key input 16-key output odd-even merging network pairs 143

8.1 Pipelining configurations for $0 \leq D \leq 8$ 157
8.2 RTL implementation options and values 161
8.3 RTL implementation results for $1 \leq L \leq 4$ 163
8.4 RTL implementation results for $5 \leq L \leq 8$ 164
8.5 HLS implementation options and values 175
8.6 HLS implementation results for $1 \leq L \leq 4$ 178
8.7 HLS implementation results for $5 \leq L \leq 8$ 179
8.8 Best RTL and HLS implementation options 184

Acronyms

ATCA Advanced Telecommunications Computing Architecture

ATLAS A Toroidal LHC AparatuS

BC Bunch Crossing

BCID Bunch Crossing Identifier

BCR Bunch Counter Reset

BER Bit Error Rate

CDR Clock Data Recovery

CERN European Organization for Nuclear Research

CL Confidence Level

CMS Compact Muon Solenoid

CRC Cyclic Redundancy Check

CSC Cathode Strip Chambers

CSV Comma-Separated Values

CTP Central Trigger Processor

DAQ Data Acquisition

DRP Dynamic Reconfiguration Port

DUT Device Under Test

EMI Electromagnetic interference

FIFO First In First Out

FMC FPGA Mezzanine Card

FPGA Field Programmable Gate Array

FSM Finite State Machine

GbE Gigabit Ethernet

HEP High Energy Physics

HFT High-Frequency Trading

HL-LHC High Luminosity LHC

HLS High-Level Synthesis

HLT High-Level Trigger

I2C Inter-Integrated Circuit

IBERT Integrated Bit Error Ratio Tester

II Iteration Interval

IP Intellectual Property

JTAG Joint Test Action Group

L1 Level-1

L1A Level-1-Accept

L1Calo Level-1 Calorimeter Trigger

L1Muon Level-1 Muon Trigger

L1Topo Level-1 Topological Trigger Processor

LHC Large Hadron Collider

LS2 Long-Shutdown 2

LS3 Long-Shutdown 3

LUT Look up Table

MDT Monitored Drift Tubes

MGT Multi-Gigabit Transceiver

MIBAK Muon Interface Backplane

MICTP Muon Central Trigger Processor Interface Module

MIOCT Muon Interface Octant Module

MIROD Muon Interface Readout Driver Module

MPO Multi-fiber Push On

MPSoC Multi-Processor SoC

MSP Muon Sector Processor

MTBF Mean Time Between Failure

MUCTPI Muon-to-Central-Trigger-Processor Interface

OAPH Opening Area Percentage Histogram

PCB Printed Circuit Board

PCS Physical Coding Sublayer

PISO Parallel In Serial Out

PMA Physical Medium Attachment

PRBS Pseudo Random Bit Sequence

PVT Process Voltage Temperature

QSFP+ Quad SFP Plus

ROD Readout Driver

RoI Region-of-Interest

ROL Readout Link

ROS Readout Subsystem

RPC Resistive Plate Chamber

RTL Register-Transfer Level

SEU Single Event Upset

SFP Small Form-factor Pluggable

SIPO Serial In Parallel Out

SL Sector Logic

SLR Super Logic Region

SMA SubMiniature version A

SoC System-On-Chip

SONET Synchronous Optical Networking

SPI Serial Peripheral Interface

SSI Stacked Silicon Interconnect

STA Static Timing Analysis

TCL Tool Command Language

TDAQ Trigger and Data Acquisition

TGC Thin Gap Chamber

TNS Total Negative Slack

TOB Trigger Object

TRP Trigger, Readout, and TTC processor

TTC Timing, Trigger and Control

UI Unit Interval

VHDL VHSIC HDL

VME Versa Module Europa

WHS Worst Hold Slack

WNS Worst Negative Slack

1 Introduction

High Energy Physics (HEP) is the field of Physics that studies the nature of the elementary particles that constitute matter and their interactions. Physicists and engineers build particle colliders to test the predictions of different theories of particle physics. The results of these particle collisions are tracked by detectors that use thousands of sensor channels to record information from the collisions, hence producing a large amount of data. Besides, the rate of collisions is designed to be very high in an effort to increase the chances of observing very rare decays.

It is very complicated and sometimes technically impossible to store data and perform physics analyses of the full information extracted from the event in the detector. Additionally, only a tiny part of the events contains interesting information for physics. For this reason, a highly selective process selects data subsets for further detailed analysis. The process of selecting the data extracted from the detector is known as *trigger*.

The trigger systems are subdivided into online and offline triggers. An online trigger uses simplified algorithms to reduce the collision events to an acceptable event output rate that can be processed by the offline trigger. The online/offline terms are given to the fact that the online trigger processes information stored in a time-limited storage system (memory pipelines) immediately after data acquisition, i.e., in real-time. The offline term means that the data are stored in a long-term storage system (hard drives or tapes) and processed later.

The online trigger systems, when required, are subdivided into low-level and high-level triggers. The low-level online trigger system is mainly constrained by the high-bandwidth and the low-latency requirements, given a large number of inputs and limited length of the on-detector memory pipelines. Therefore, the low-level online trigger system uses a reduced set of event data and runs low-complexity trigger algorithms for minimizing latency. It is usually built using custom made electronics optimized for low-latency. For the off-detector

trigger systems, that is, not exposed to radiation, digital processing is primarily implemented using Field Programmable Gate Array (FPGA) devices, given that FPGAs offer high processing capacity with low-latency and re-programmability, i.e., the capability of changing the logic implemented.

In a second stage, known as *high-level trigger*, where latency is less critical, off-the-shelf electronics are used to perform an additional step in the event selection using results from the first-level trigger, and complete event and calibration information. High-level trigger systems also allow one to use the same or at least similar trigger algorithms as in the offline trigger, adding the flexibility of using algorithms further up to the detectors or further down towards the computing.

As a result of the increasing luminosity[1] in HEP colliders, trigger systems have to become more selective in order to keep event acceptance rates manageable. The following two actions are taken to improve selectivity: First, more event information from the detector is used, which leads to a higher number of sensor channels and/or higher bandwidth. Second, trigger systems are required to process larger parts of the detector information together in earlier stages. The increase of the channel count and higher data processing concentration introduces higher integration as the new critical requirement in online trigger systems. In this work, novel low-latency and high-bandwidth architectures for highly integrated trigger systems, and optimized design flows are presented for advancing state of the art in first-level trigger systems.

1.1 Low-latency high-throughput circuit applications

There are low-latency digital circuits in several applications, but they are often constrained to different time scales of processing. For instance, High-Frequency Trading (HFT) requires to access market data and execute orders faster than other investors. In this application, a millisecond reduction in latency can improve profitability by $100M a year [1]. High definition image processing is considered to have low-latency if processing time takes tens of microsecond [2]. First-level trigger systems for the Large Hadron Collider (LHC) require very low-latency. As an example, the A Toroidal LHC AparatuS (ATLAS) first-level trigger subsystems that receive event data every 25 ns are constrained in the order of hundreds of nanoseconds. ATLAS and these subsystems are described in more detail below because they are where the ideas resulting from this research have been deployed.

[1] Luminosity is the number of possible collisions per square centimeter and per second, $cm^{-2}s^{-1}$

1.2 CERN, LHC and ATLAS

The European Organization for Nuclear Research (CERN) was created by 12 countries in Western Europe in 1954. It is based in Meyrin, Canton of Geneva, Switzerland. CERN is devoted to the study of HEP. Currently, it hosts the largest particle physics laboratory in the world and has the participation of 23 member states, three countries with observer status, and 35 countries with cooperation agreements.

Many activities at CERN involve operating the LHC, which is the largest and most powerful particle collider and the biggest machine in the world. The LHC is built inside a circular tunnel 100 meters beneath the ground and consists of a 27-kilometer ring of superconducting magnets with several accelerating structures to boost the energy of the particles. The LHC delivers heavy-ion and proton-proton collisions spaced by a bunch crossing period of 25 ns, corresponding to a 40MHz bunch crossing frequency.

Figure 1.1 shows an overview of the ATLAS detector highlighting its proportions when compared to four human beings. Two are indicated by a red flag, and the remaining two are at the bottom of the experiment. ATLAS is the largest particle physics experiment at the LHC. It measures about 45 meters long, more than 25 meters high, and weighs about 7,000 tons.

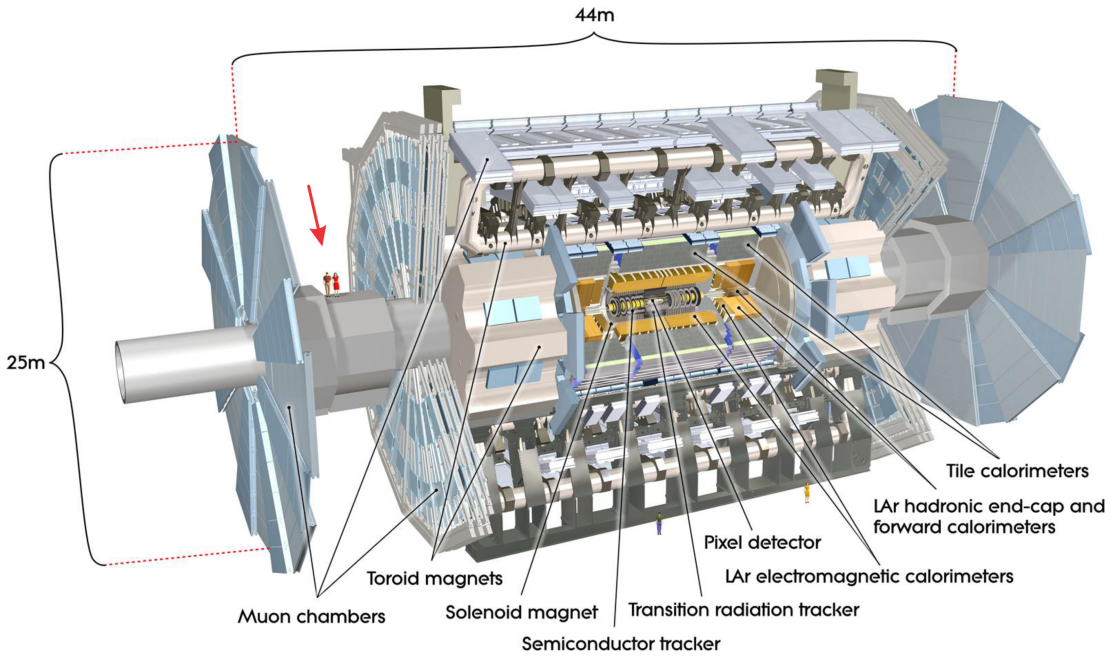


Figure 1.1 – Overview of the ATLAS experiment (from [3])

In order to identify all particles produced by the interactions, the detector is designed in layers. The layers consist of different types of detectors designed to observe specific types of particles. The various tracks that the particles leave in each layer of the detector allow particle identification and measurements of energy and momentum.

The inner detector [3] (pixel detector, semiconductor tracker, and transition radiation tracker) observes charged particles such as electrons or charged pions. After the particles have crossed the tracking system, they reach the calorimeters, where their energy is deposited and measured by the inner electromagnetic and outer hadronic calorimeter systems.

The Electromagnetic Calorimeter [3] measures the energy of electrons and photons as they interact with the electrically charged particles in matter. It uses lead plates, as absorber material, and copper-kapton electrodes (in an accordion shape) kept in a cold Liquid Argon (LAr) vessel, as the active material.

The Hadronic Calorimeter [3] measures the energy of hadrons, such as protons, neutrons, and pions by reading the deposited energy on absorbent material. The Hadronic End-Caps [3] uses LAr technology located inside the same cold vessels as the Electromagnetic Calorimeter and copper plates as absorber material. The Tile Calorimeter [3] surrounds the Electromagnetic and Hadronic End-Caps calorimeters and uses steel plates as absorber material interleaved with scintillator tiles (as active material). It measures the deposited energy by reading out, from photomultiplier tubes, the light generated by the scintillator tiles.

Calorimeters absorb the energy of most particles except muons and neutrinos. The Muon Spectrometer [3] is installed in the outermost layer of ATLAS to track muons that pass through the detector. Muons are detected by measuring a series of hits left in muon chambers. A chamber consists of thousands of metal tubes equipped with a central wire within a gas volume. When a muon or any charged particle passes through the volume, it knocks electrons off the atoms of the gas. By measuring the time it takes for these electrons to drift from the starting point, it is possible to determine the position of the muon as it passes through.

1.3 The trigger and data acquisition system

An overview of the Trigger and Data Acquisition (TDAQ) system of ATLAS for Run 3 (2021-2023) is shown in Figure 1.2. The data generated by the detectors are flowing from top to bottom. At the bottom, a sub-set of the detector data are recorded into a large mass storage system. The online trigger is shown on the left side, and the DAQ is shown on the right side of the picture.

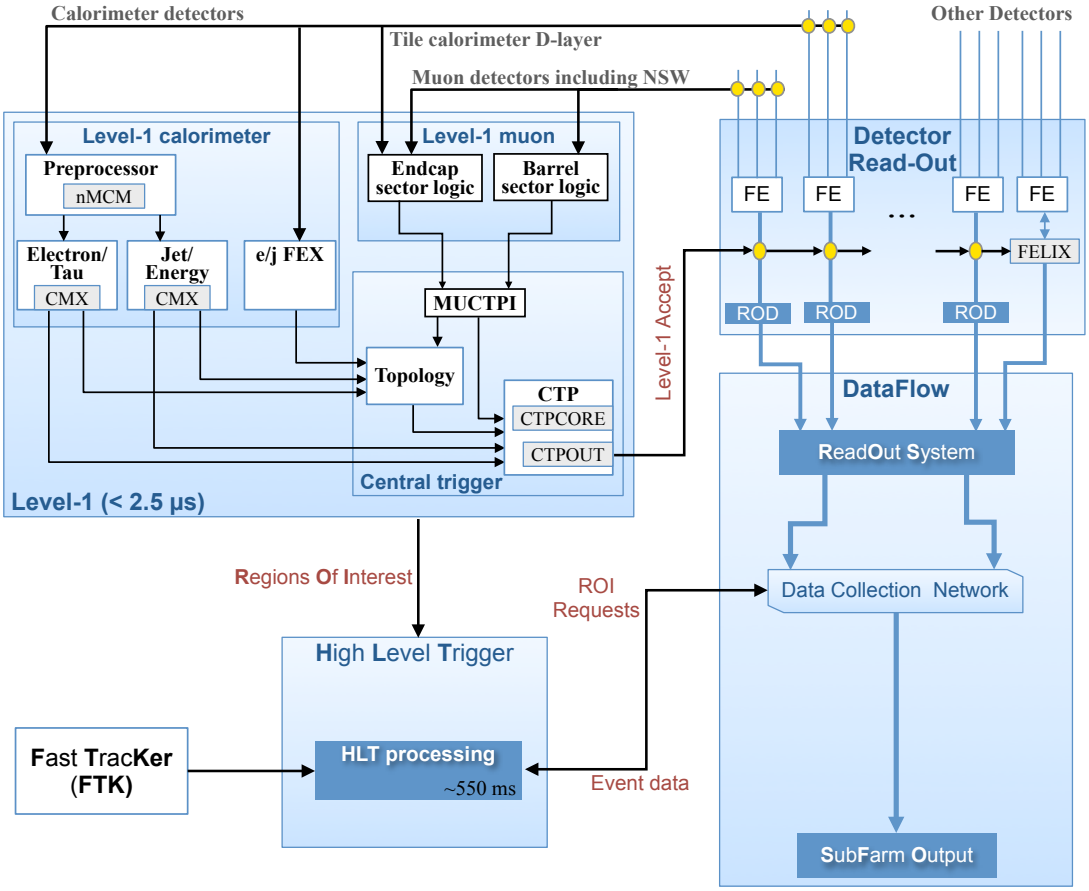


Figure 1.2 – Overview of the ATLAS TDAQ system for Run 3 (from [4])

The online trigger system of ATLAS is structured in a 2-level architecture in order to reduce the event rate from an interaction rate of 1 GHz [2] down to 1 kHz written to permanent storage. The first level, also known as Level-1 (L1), is implemented with custom electronics, while the High-Level Trigger (HLT) [5], is built from commercial computers, network switches, and custom software. Their functions are summarized as follows:

- The L1 trigger combines information from the calorimeter and muon trigger processors to generate the final Level-1-Accept (L1A) decision. It reduces the 1 GHz event rate down to less than 100 kHz with a latency of only 2.5 us. The time between the collision and the arrival of the L1A at the sub-detectors is referred to as the L1 latency [6]. Due to the experiment dimensions and the distance to the underground counting room, the cable propagation delays from the detector front-end to the underground counting

[2] With an average of 25 collisions per crossing, the LHC delivers events at 40 MHz × 25 = 1 GHz rate.

room and back to the readout system take about 1 µs of this time. Therefore, a latency budget of only 1.5 us remains to be shared between the subsystems of the L1 trigger for processing and data transfer [7]. The total latency value has to be kept below 2.5 µs in order to not loose event data due to the limited storage availability in the front-end pipelined memories. The first level of trigger also supplies information on the region where the object that passed the trigger threshold was located in the detector, Region-of-Interest (RoI), to start the HLT trigger.

- The HLT trigger reduces the rate of 100 kHz to 1 kHz applying additional selection criteria, based on the L1 RoI information and full-granularity data, by using software algorithms with an emphasis on early rejection.

The Data Acquisition (DAQ) system, collectively represented in in Figure 1.2 by the *Detector Read-Out* and *Dataflow* boxes, channels the event data from the detectors to storage as follows:

- First, the DAQ receives and buffers the event data from the detector using detector specific front-end pipeline memories, which receive data at the bunch crossing rate (40MHz). The event data are kept until the L1A arrives, and readout at the L1 trigger accept rate (100kHz).To ensure that the event data can be read out when the L1A arrives, each sub-detector has to store the event data for a fixed time. This time depends on the L1 latency and arrival time of the event data to the readout electronics.
- Second, if the L1 trigger accepts an event, all the data associated with the event is read out for all components of the detector. The so-called Readout Driver (ROD) receives event information from the pipeline memories, performs data compression and zero suppression, and makes the data available to be read out by the Readout Subsystem (ROS) via optical fiber Readout Links (ROLs) using the S-LINK protocol [8]. Then, the ROS provides RoI fragments to the HLT trigger and holds the event data in a custom memory buffer [9] for the entire HLT latency time of 550 ms.

Then the full events enter the event filter farm, where they are processed using offline-type algorithms with access to full calibration and alignment information. The events selected by the event filter are moved to the permanent storage at the computer center at a rate of 1 kHz. The event size is approximately 2.4 MB, resulting in a final data storage rate of approximately 2.4 Gb/s.

1.4 The L1 trigger system

The L1 trigger is based on identifying high-transverse energy or missing transverse energy objects. Transverse energy is the energy of an object transverse to the beam. Missing transverse

energy measures the energy that is not detected in ATLAS, but it was expected due to the laws of conservation of energy and conservation of momentum. The energy imbalance is caused by particles escaping detection, in particular neutrinos. But also by unaccounted physics processes and detector characteristics such as the noise and dead or hot cells.

The L1 trigger system [7] is a real-time low-latency high-throughput system that performs fast event selection based on information from the calorimeters and dedicated muon detectors. Figure 1.3 shows an overview of the L1 trigger system for the Run 3 operation. The data transfer and processing are based on the system-synchronous clocking technique that requires fixed latency.

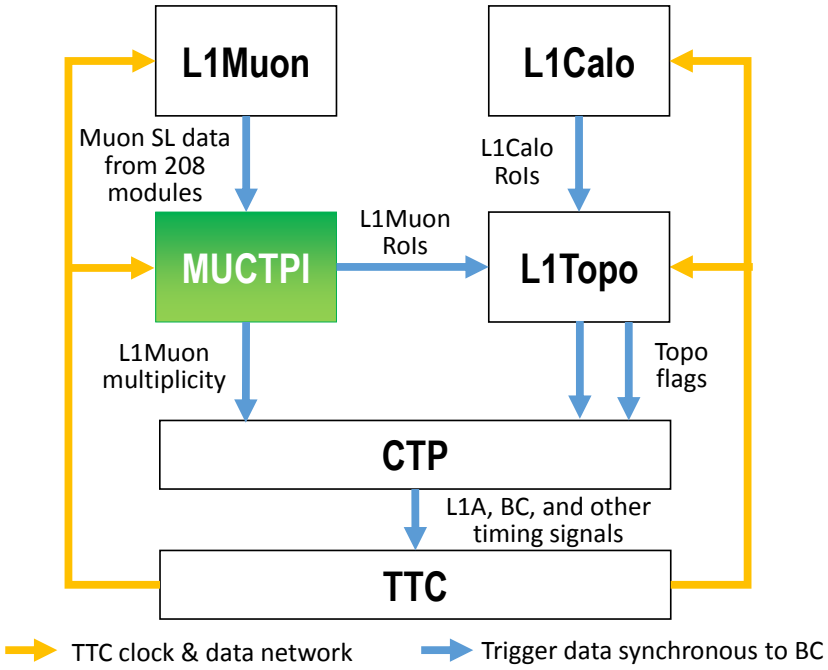


Figure 1.3 – Overview of the ATLAS L1 trigger system for Run 3

The calorimeter selection is based on information from the electromagnetic and hadronic calorimeters grouped into a single subsystem, the Level-1 Calorimeter Trigger (L1Calo). The L1Calo trigger system identifies high transverse energy objects, such as electrons and photons, jets, and τ-leptons decaying into hadrons, as well as events with large missing transverse energy and total transverse energy. The calorimeter information used in the L1 trigger decision is the multiplicity of hits for each E_t threshold per object type and energy flags.

The Level-1 Muon Trigger (L1Muon) system searches for patterns of hits consistent with tracks of muons with high transverse momentum p_T coming from the interaction point. More details on the ATLAS Muon Trigger system is presented in Section 1.5.

The Muon-to-Central-Trigger-Processor Interface (MUCTPI) calculates and sends to the Central Trigger Processor (CTP) the total number of muon candidates, the so-called *multiplicity*, for each of six p_T thresholds. The MUCTPI also sends muon position information to the Level-1 Topological Trigger Processor (L1Topo) processor [10]. As the ideas resulting from this Ph.D. work were deployed on the MUCTPI, the latter is described in more detail in Section 1.6.

The L1Topo receives topological information from the calorimeter and muon trigger systems, process topological algorithms, and provide additional trigger inputs to the CTP. An example of an L1Topo topological algorithm is the cut on the angular distance between trigger objects.

The L1A signal is generated by the CTP [6], which combines the information from the L1Topo and MUCTPI systems and performs the event selection based on physics signatures found in the event, such as energetic jets, leptons or large missing transverse energy. The L1A signal is distributed to the detector front-ends, synchronously to the Bunch Crossing (BC) clock at a fixed time after the collision through the Timing, Trigger and Control (TTC) system [7].

The TTC system is also used to distribute and fan-out the timing signals such as the BC clock, orbit[3], the eight-bit trigger type, and some commands (Bunch Counter Reset [BCR], Event Counter Reset [ECR]) to the sub-detectors and subsystems. These signals are sent to the sub-detector systems using optical links and common electronic modules. The CTP, MUCTPI, and TTC systems are developed and maintained centrally by the Electronic Systems for Experiments group of the CERN experimental physics department.

1.5 ATLAS muon trigger

The muon detector features separate trigger and high-precision tracking chambers. The precision measurement of the tracking coordinates is provided by the Monitored Drift Tubes (MDT) and the Cathode Strip Chambers (CSC) systems while the trigger information is generated by the Resistive Plate Chamber (RPC) and Thin Gap Chamber (TGC) [7] systems.

The RPC and TGC muon trigger detectors provide track information within 15-25 ns [7] after the passage of a particle, allowing to identify the beam crossing. The momentum of the muons is estimated using a coincidence scheme that measures the bending[4] of muon tracks in the magnetic field of the large superconducting air-core toroid magnets [7]. The trigger information is provided by RPC detectors in the barrel region [7], and TGC detectors in the end-cap region [7]. The track information from the front-end electronics is then sent to the

[3] One orbit corresponds to one LHC turn, equivalently to 3564 bunch crossings.

[4] The smaller the momentum, the stronger the bending, and the higher the momentum, the stiffer the track becomes.

ATLAS computing room, where, at the first stage, it is processed by the muon trigger Sector Logic (SL) modules.

The muon trigger SL [7] reconstructs muon tracks and classifies them into one of six p_T threshold values. It selects the highest p_T muon candidates for each of the 208 muon trigger sectors from RPC and TGC systems and sends the so-called *sector data* to the MUCTPI system. The sector data contains information, such as the position RoI and the transverse momentum p_T threshold value from each candidate.

1.6 MUCTPI

The MUCTPI combines the information delivered by the trigger SL modules from the two regions of the muon trigger sub-detectors (barrel and endcap) and then calculates the multiplicity for each of six p_T thresholds. The data from the trigger sectors are received using electrical cables that transmit the data in parallel. They are synchronized using programmable length pipelines in order to compensate for different propagation delays in the detector, and then the event data are processed.

Due to the geometrical position of the muon detectors and the bending of the muons in the magnetic field, a single muon could be identified in two or even three different sectors of the muon trigger detectors. The regions where a give muon candidate can be detected multiple times are refereed to as *overlap regions*. After the data are synchronized, the overlap handling algorithm [11] avoids double counting of muon candidates in overlap regions. The MUCTPI uses programmable Look up Tables (LUTs) to indicate if a given candidate is located in one of the overlap regions between adjacent trigger sectors. Next, the total muon multiplicity is calculated, and it is forwarded to the CTP which takes the final L1 decision.

As a result of the M.Sc. thesis [12] of this same author, the MUCTPI firmware has been upgraded to also provide muon topological information to L1Topo through the existing electrical trigger outputs [12]. Concurrently, the MUCTPI stores trigger sector data, the multiplicity values, and the overlap handling results until an L1A is received. When an L1A is received from the CTP, the MUCTPI adds a header and control flags to the data and sends it to the HLT and DAQ system using the S-LINK protocol.

The MUCTPI provides information for online monitoring. More than 300 counters are implemented to measure the rate of events under certain conditions. Examples are the number of occurrences of each pT threshold for each candidate from every trigger sector, the number of veto flags of each of the candidates, and the number of single and multiple overlap occurrences. The MUCTPI also features replay and snapshot memories used for in-system verification.

1.7 Thesis motivation

This Ph.D. work focuses on fulfilling three requirements of the ATLAS L1 trigger system. These requirements are *low latency*, *fixed latency*, and *reliability*. The three requirements have implications and consequently introduce different challenges in the L1 trigger data transfer and processing. The study of these implications and the solutions implemented in the MUCTPI data transfer and data processing are presented in Parts I and II, respectively.

Figure 1.4 shows how each of the two parts are connected to the event data, TTC, SL module, detector read-out, and data collection network. The cloud named as Part I represents the data transfer from the trigger SL module to the MUCTPI. The cloud named as Part II represents the MUCTPI real time data processing. Subsystems that are upstream of the trigger SL module and downstream of the MUCTPI are omitted in this simplified diagram. The reason for each of the three requirements and the implications if they are not fulfilled are summarized as follows:

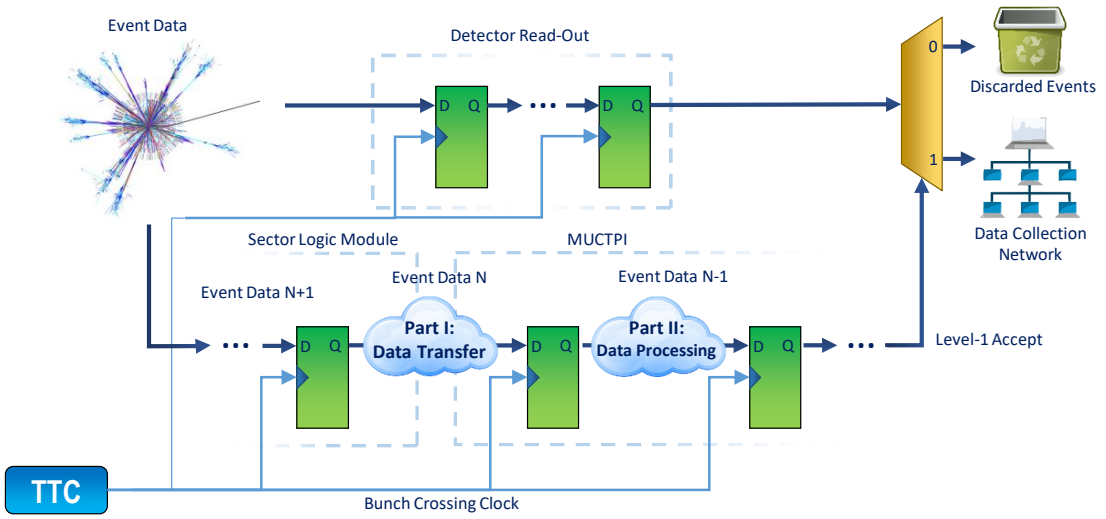


Figure 1.4 – Thesis context diagram. After each physics event is collected by the detector electronics, the trigger is responsible for deciding what should be stored. Requirements of nanoseconds latency mean that custom solutions have to be implemented both for data transfer and the data processing pipelines. Both parts are addressed by ensuring low and fixed latency, and reliability. Low latency is required due to the limited storage available in the detector front-end pipelined memories. Fixed latency is required because the L1 trigger system is a real-time system. Reliability is needed to reduce the rate of discarded rare events and accepted uninteresting events.

1. **Low latency** is required due to the limited storage available in the detector front-end pipelined memories.

If the event accepted flag arrives too late the event data are lost from the pipelined memory at the detector readout.

2. **Fixed latency** is required due to the nature of the ATLAS L1 trigger system, which is a real-time system based on system-synchronous clocking technique. System-synchronous systems require fixed latency for data transfer and processing. Otherwise, information can be corrupted.
   The first aspect is that the trigger processing is pipelined processing, for which the inputs need to be time-aligned at every processing step. Furthermore, the final L1A needs to have a fixed latency because the event-data is buffered at the front-ends and located in the buffer only by the timing of the L1A signal. If the latency varies with time, the wrong event is accepted and sent to the computer farm and the right event is lost.

3. **Reliability** is required to keep trigger efficiency high. Fake triggers could be generated if trigger information is corrupted or not reliable.
   If the trigger is not reliable, rare events can be discarded and uninteresting events sent to the Data Collection Network for further processing. This effect reduces the trigger efficiency.

1.8 Thesis organization

Chapter 2, describes the upgrade of the MUCTPI for the Phase-I upgrade, which is the practical application where the ideas resulting from this Ph.D. thesis have been deployed. Chapters 3 to 5, grouped in Part I, describe how low latency, fixed latency, and reliability have been addressed in the MUCTPI data transfer. Chapter 3 presents the characterization of the MUCTPI high-speed serial links. Chapter 4 describes the optimization studies on the FPGA transceiver configuration aiming low and fixed latency. Chapter 5 presents the development of the so-called *synchronization* Intellectual Property (IP) core, which transfers the SL input data, from the recovered clock to the system clock domain for combined data processing. The actions taken to cope with the three requirements in the MUCTPI data transfer are summarized as follows

1. **Low latency** is achieved by developing a latency-optimized configuration of the FPGA transceiver data path, see Chapter 4, and designing a low-latency synchronizer IP to transfer the SL data to the system clock domain, see Chapter 5.

2. **Fixed latency** is achieved by designing a board clock infrastructure with fixed clock-to-output timing, optimizing the transceiver clock fabric connectivity to ensure low latency variation, see Chapter 4, and designing a data synchronizer that can absorb small latency variation from the transceiver, see Chapter 5.

3. **Reliability** of the data transfer is ensured by the good performance of the high-speed serial data lines. This performance is given by a proper design of the MUCTPI hardware. Initially, a demonstrator has been developed to demonstrate that the high-speed transceiver components intended to be used at the MUCTPI are able to transfer data reliably. The demonstrator features a commercial evaluation kit and a custom mezzanine card developed by the author of this thesis. Once the MUCTPI prototype was available, the data transfer reliability has been measured using different metrics, which are described in more detail in Chapter 3.

Chapters 6 to 8, grouped in Part II, starts with an introduction on the MUCTPI data processing presented in Chapter 6. Chapter 7 presents bibliography research on sorting networks and the design of the MUCTPI sorting unit. Chapter 8 presents the implementation of the MUCTPI sorting unit using RTL, and HLS approaches, and a comparative study in terms of design effort, and performance. The actions taken to cope with low latency, fixed latency, and reliability requirements in the MUCTPI data processing are the following:

1. **Low-latency** is achieved by researching low-latency algorithms, see Chapter 7, and optimizing their implementation in view of low-latency, see Chapter 8.
2. **Fixed-latency** is achieved by researching data-oblivious algorithms that can compute the result with a fixed timing regardless of the characteristics of the input data, see Chapter 7.
3. **Reliability** of the data processing is ensured by careful design of the MUCTPI sorting unit firmware, simulation, and static timing analysis to ensure that the data output is reliable, see Chapter 8.

Chapter 9 presents the conclusions from Parts I and II and an outlook on future work.

2 MUCTPI Upgrade

This chapter presents the Phase-I upgrade of the MUCTPI. Section 2.1 describes the motivation to upgrade the ATLAS detector. Section 2.2 presents the MUCTPI architecture. Section 2.3 provides a summary of this chapter.

2.1 Motivation

The luminosity of the LHC has been increased and will further be increased over time, in order to increase the chances of rare events. Figure 2.1 [13] shows the LHC plan for the next 20 years. The LHC reached nominal luminosity of $10^{34}\text{cm}^{-2}\text{s}^{-1}$ at the beginning of Run 2 (2015-2018), it is expected to reach twice its nominal luminosity in Run 3 (2021-2024) after the Long-Shutdown 2 (LS2) (2019-2021), and 5 to 7.5 times its nominal luminosity in Run 4 and 5 (2027-2040) after the High Luminosity LHC (HL-LHC) upgrade [14] during the Long-Shutdown 3 (LS3) (2025-2027).

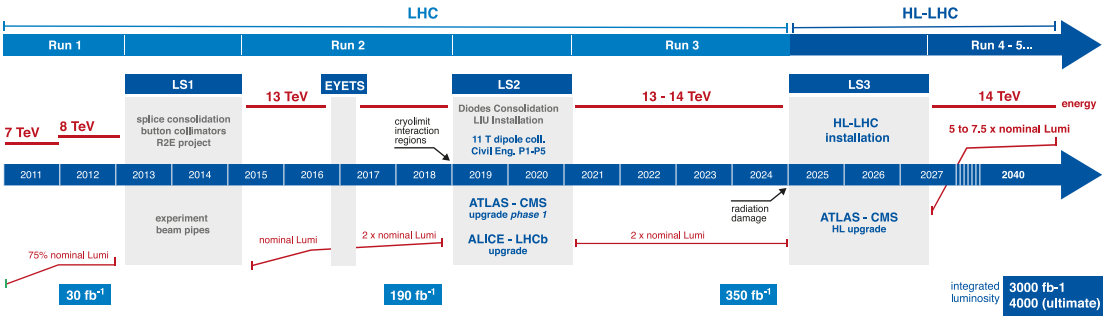


Figure 2.1 – LHC plan

As the luminosity increases, the trigger system has to become more selective to keep output rates manageable. In order to cope with the increasing luminosity of the LHC, ATLAS is

preparing two upgrades, so-called *Phase-I* and *Phase-II* upgrades. The first is being installed and commissioned during LS2, and the second will be installed during LS3. This Ph.D. work focuses on the MUCTPI upgrade for Run 3.

For example, trigger selectivity can be improved by routing more information from the detector to the trigger and processing larger parts of this information together. More information from the detector is obtained by adding new sensor channels, increasing their resolution, and/or routing existing data to the trigger processing rather than using it only when the full event data is readout. For the Phase-I upgrade, no sensors are added, but the number of muon candidates routed to the MUCTPI, and the respective p_T resolution are increased.

Processing larger parts of the detector together is achieved by increasing the integration level of the processing units. Examples are supporting overlap handling in any detector region, and sorting muon candidates from larger parts of the detector. For the legacy MUCTPI, both overlap handling and sorting have been limited to only regions within one-sixteenth of the detector. The new MUCTPI can handle overlap and sort muon candidates from regions within one half of the detector.

In terms of physical space, the higher integration thanks to the smaller physical dimensions of the optical modules combined with the higher FPGA densities available today enable the implementation of all the required MUCTPI functionality on a single Advanced Telecommunications Computing Architecture (ATCA) [15] blade. In comparison, the system used during Runs 1 and 2, so-called *legacy MUCTPI*, requires a full 9U Versa Module Europa (VME) [16] shelf with 18 boards. Figure 2.2 shows the legacy MUCTPI crate, which hosts 16 Muon Interface Octant Module (MIOCT) modules, one Muon Central Trigger Processor Interface Module (MICTP) module, one Muon Interface Readout Driver Module (MIROD) module and one custom Muon Interface Backplane (MIBAK) backplane (not seen in the picture). The sector data inputs and trigger outputs are indicated in red and black, respectively. More details on the legacy MUCTPI are available in [17].

The MUCTPI upgrade is taking into account the higher bandwidth and integration level in the interest of improved trigger selectivity. More details about these changes are described as follow:

- **Higher bandwidth:** The interface from the muon trigger SL modules to the MUCTPI system will be implemented using high-speed serial optical connections, instead of the previously used parallel electrical cables. High-speed serial optical connections provide higher bandwidth and will enable the construction of a highly integrated system. Thanks to the higher bandwidth, the SL modules can send additional event data, such as information on more muon candidates, better transverse momentum (p_T) resolution, and position information with higher granularity to the MUCTPI. On the one hand,

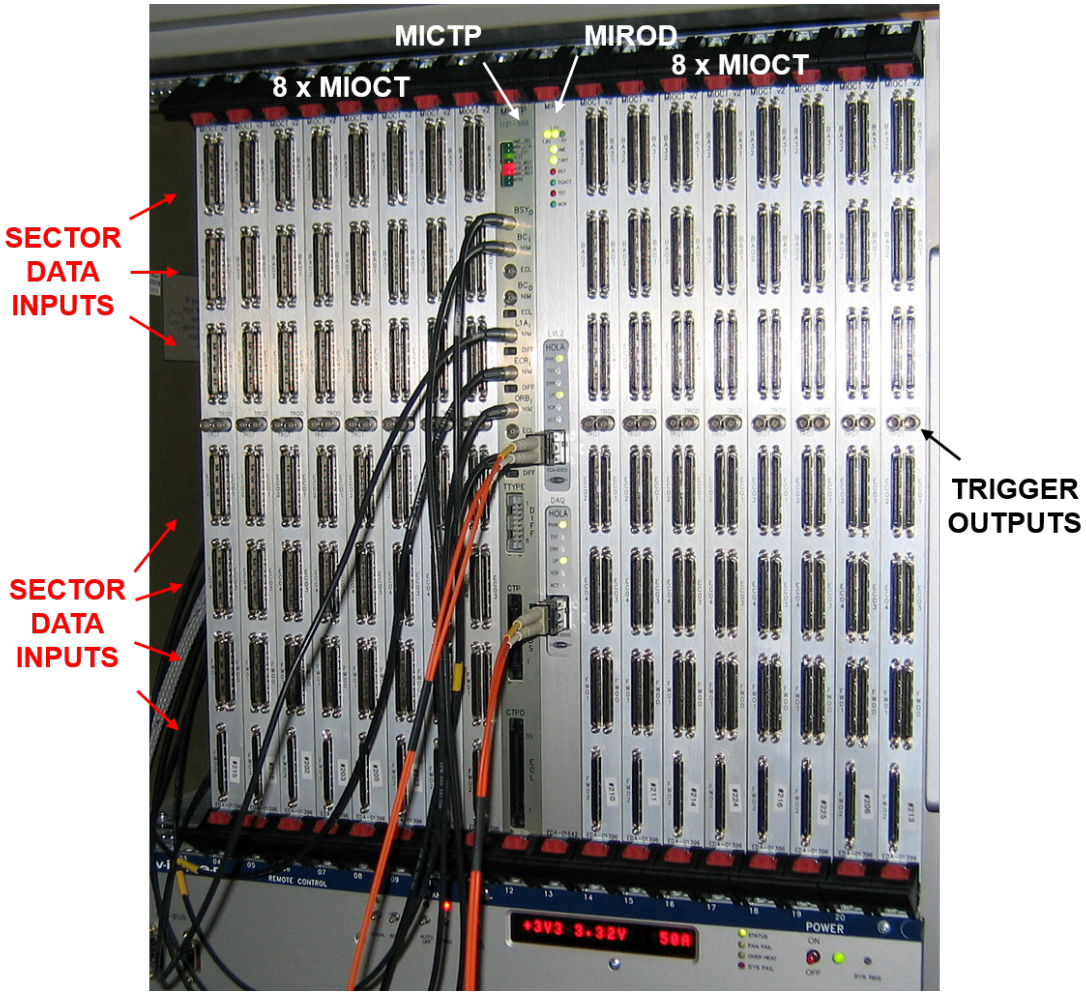


Figure 2.2 – Legacy MUCTPI system

the higher integration level reduces the number of processing components and the distance between them, hence reducing interconnection and signal propagation delays. On the other hand, the latency in the data transfer is increased due to serialization and de-serialization when compared to the currently used electrical cables, which transmits the data in parallel. Thanks to the high-bandwidth from the high-speed serial optical connections, the upgrade MUCTPI will send full detector-granularity muon position information to L1Topo at the bunch crossing rate, which will allow combined calorimeter/muon full granularity topological trigger algorithms.

- **Higher integration level:** The increased integration level will add flexibility to the MUCTPI system by enabling the processing of all MUCTPI data in a single module with low-latency. For instance, the overlap handling algorithm will be able to handle

candidates in any overlap region. In addition, the upgraded MUCTPI will sort muon candidates, according to their p_T value, from one half of the detector. Both functions have been so far limited to only regions within one-sixteenth of the detector. The higher integration level will also enable the implementation of new functionalities, such as low-latency muon-only topological processing. The term low-latency is used because muon-only topological algorithms could be processed already at the MUCTPI. This can reduce the overall latency by outputting the results directly to the CTP system, instead of reaching the CTP through the L1Topo.

2.2 MUCTPI architecture

Figure 2.3 shows the upgraded MUCTPI architecture block diagram. The new MUCTPI system [18] is based on 16/20 nm FPGA devices (Xilinx 16 nm Ultrascale+ and 20 nm Ultrascale) [19], featuring a large number of on-chip Multi-Gigabit Transceivers (MGTs). It uses 12-channel ribbon fiber optics receiver and transmitter modules (Broadcom MiniPOD) [20] for the data transfer. The higher bandwidth from the high-speed serial optical connections from the muon trigger SL modules to the MUCTPI system enables to double the number of muon candidates, up to 4 candidates per trigger sector instead of 2 can be received.

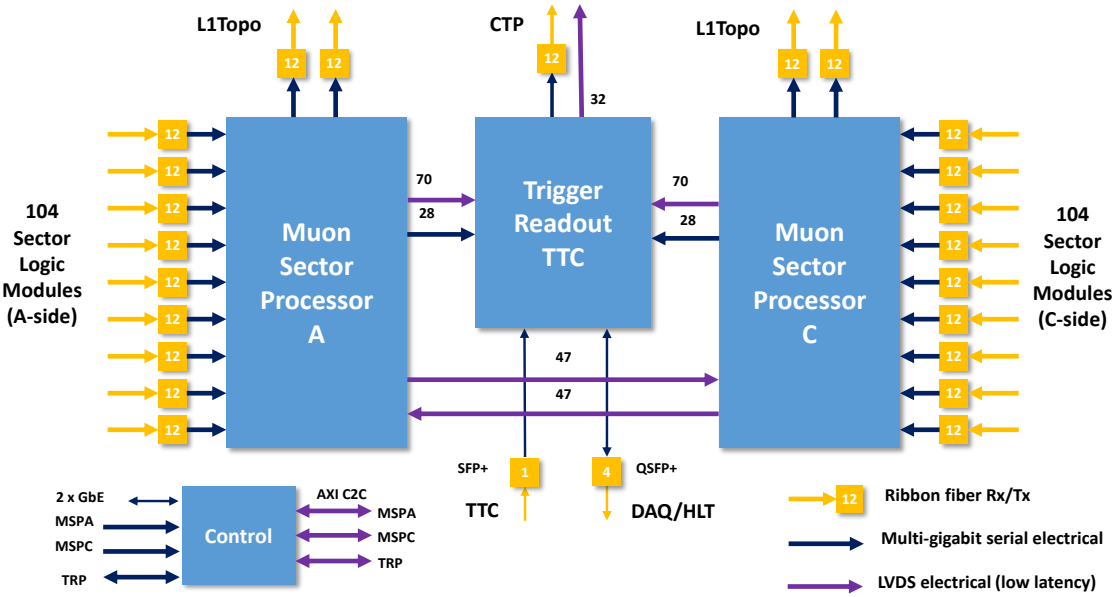


Figure 2.3 – MUCTPI architecture

The data from the SL is received and processed by Muon Sector Processor (MSP) FPGAs, which then sends trigger information to L1Topo and to the Trigger, Readout, and TTC processor (TRP) FPGA that merges the information from the two MSP FPGAs and sends trigger results to

CTP and readout information to DAQ and HLT. The control FPGA implements the control, configuration, and monitoring of the board and run the required software to interface the MUCTPI to the ATLAS run control system. More details on the functionality implemented on these FPGAs are discussed in Sections 2.2.1 to 2.2.3. The on-board and off-board connectivity is described in Sections 2.2.4 and 2.2.5.

Three prototype versions have been designed to evaluate the use of different FPGAs on the MUCTPI. Table 2.1 shows which FPGA has been used for each of the three prototype versions. Version 1 is the first prototype version. Version 2 introduces a 16 nm Ultrascale+ FPGA instead of the previously used 20 nm Ultrascale FPGA for the MSP functionality. Version 3 replaces a 32-bit dual-core System-On-Chip (SoC) by a 64-bit quad-core Multi-Processor SoC (MPSoC). The third version is preferred because it features higher performance MSP FPGAs and a 64-bit processor that will be easier to support in the future. Moreover, the third version is also the most developed, i.e. it concentrates all the knowledge acquired during the testing of the previous prototypes. Versions 1, 2, and 3 of the MUCTPI are referred as *MUCTPI-V1*, *MUCTPI-V2*, and *MUCTPI-V3*, respectively. Not all the requirements are known for Run 4 but the baseline plan is to use two MUCTPI cards to increase by a factor of two the I/O channels and processing capacity.

Table 2.1 – FPGA used in each of the three prototype versions

FPGA	**Version 1**	**Version 2**	**Version 3**
MSP	Ultrascale VU160	Ultrascale+ VU9P	
TRP	Ultrascale KU095		
SoC	Zynq-7000 7Z030 SoC		Zynq Ultrascale+ ZU3EG MPSoC

Figure 2.4 shows a photo of the first MUCTPI prototype board. The two large FPGA with a blue heat-sink on the top of the picture are the MSP FPGAs, the large FPGA in the center of the picture is the TRP FPGA. Below the TRP FPGA is the Control SoC FPGA. The Broadcom MiniPODs are identified with a dark yellow box. Several other components of the board are highlighted according to the legend at the bottom of the picture.

2.2.1 Muon Sector Processor

One large FPGA, the MSP, is in charge of the trigger processing of the data of one half of the detector. The two FPGAs together receive and process muon trigger data from 208 SL modules connected through high-speed serial optical links using MiniPOD receiver modules. The MSP FPGAs also copy information on selected muon trigger objects to several L1Topo modules using MiniPOD transmitter modules.

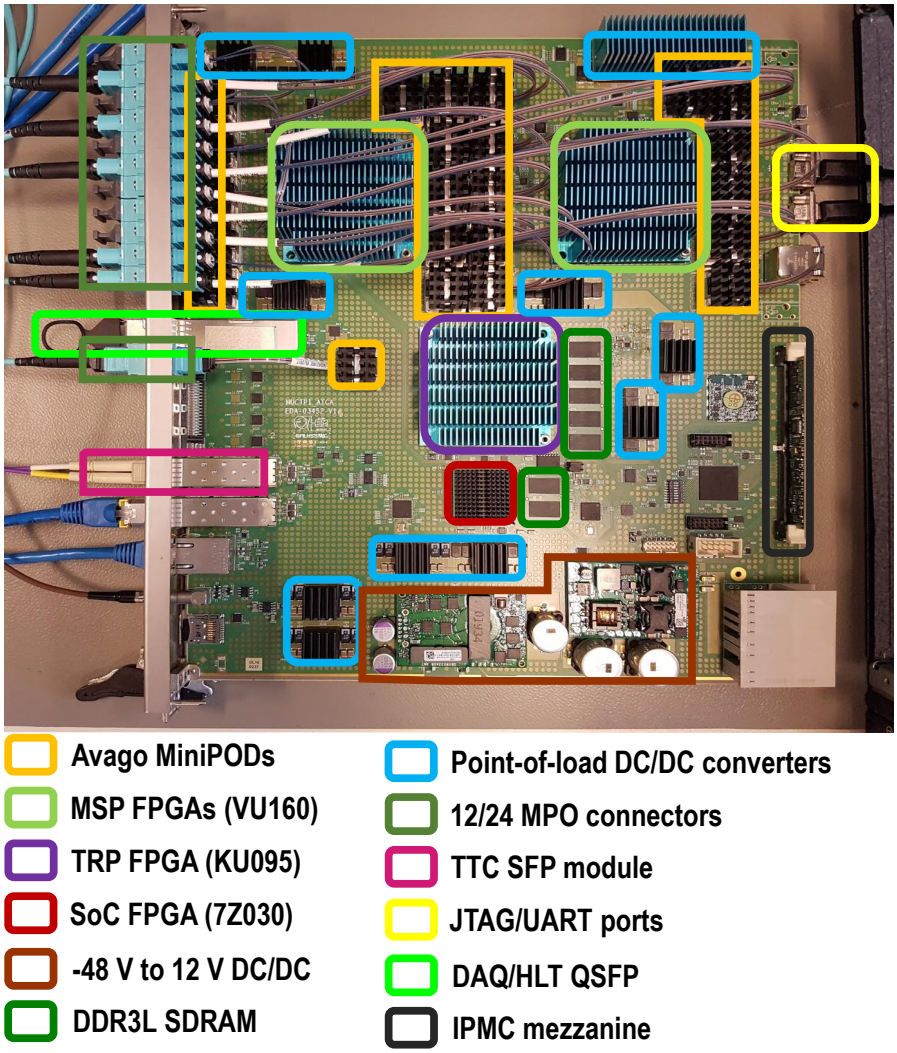


Figure 2.4 – MUCTPI prototype version 1

2.2.2 Trigger, Readout, and TTC processor

The Trigger and Readout Processor (TRP) FPGA is a Kintex UltraScale device (KU095) that merges the information received from the two MSP FPGAs through LVDS and MGT links and sends the results to the CTP. Besides, it will be used to implement muon topological trigger algorithms. This is possible because all the trigger information is available in a single module with low latency. The same FPGA also receives, decodes, and distributes the TTC information. Finally, it sends the muon trigger information to the DAQ and HLT systems when it receives a L1A decision.

2.2.3 System on chip

A Xilinx SoC/MPSoC is used for configuration, control and monitoring of the module through a Gigabit Ethernet (GbE) [21] interface. The device integrates a programmable logic part with an ARM processor subsystem. The processor subsystem will act as a control processor and run the required software to interface the MUCTPI to the ATLAS run control system. It will also be used for hardware monitoring of modules via Inter-Integrated Circuit (I2C) [22], such as the power supply, optical modules, and FPGAs. The values read include voltages, currents, temperatures, optical input power, and clock status. The SoC is also used to load the configuration bitstreams into the MSP and TRP FPGAs.

2.2.4 On-board connectivity

Connections using both general-purpose I/O pins and dedicated MGT pins are used to exchange information between the FPGAs on the board. Each MSP FPGA can share data with the other MSP FPGA using 47 LVDS pairs. Operating each LVDS pair at a bit rate of 640 Mb/s results in a total bandwidth of ≈ 30 Gb/s each way. This would be sufficient to share ≈ 12.5% of the SL trigger information between the two MSP FPGAs. In addition, 70 LVDS pairs are connected from each MSP FPGA to the TRP FPGA, resulting in a total bandwidth of ≈ 45 Gb/s. This connection will be used to send trigger results from each MSP FPGA to the TRP FPGA with low-latency.

In addition, 28 MGT links are connected from each MSP FPGA to the TRP FPGA. Operating each of these MGT links at a bit rate of 10.24 Gb/s with 8b10b encoding results in a total bandwidth of ≈ 460 Gb/s. Up to 4 links will be required for the transfer of the readout data. The remaining links could be used to transfer a subset of the muon candidate information for muon topological trigger processing.

2.2.5 Off-board connectivity

Each MSP FPGA receives the muon candidate information from 104 SL modules through 9 MiniPODs. It also transmits muon candidate information to L1Topo through up to 24 MGT links using 2 MiniPODs. The TRP FPGA receives TTC clock and data through a Small Form-factor Pluggable (SFP) module and sends information to the DAQ and HLT systems using a Quad SFP Plus (QSFP+) module. Also, one MiniPOD is available for sending trigger bits (muon multiplicities, trigger flags, etc.) to the CTP using one or more MGT links. For backward compatibility and in order to be able to minimize latency, a parallel electrical LVDS signal connection through a 68-pin SCSI VHDCI connector is also foreseen.

2.3 Summary

This chapter presented the Phase-I upgrade of the MUCTPI, the practical application where the ideas resulting from this Ph.D. thesis have been deployed. The ATLAS L1 trigger system has to become more selective to keep output rates manageable with the increased LHC luminosity. Selectivity can be achieved by extracting more information from the detector and processing larger parts of this information together.

On the implementation side, it is required to provide higher bandwidth and processing capacity, which is achieved by using high-speed serial optical connections, and highly-integrated FPGAs. The use of such components in the upgraded MUCTPI architecture has been presented, with emphasis on the processing FPGAs, SoC, and on-board and off-board connectivity.

Data transfer Part I

3 High-speed serial link testing

This chapter presents the characterization of the MUCTPI high-speed serial links. Section 3.1 presents the procedure to measure the data transfer reliability using the Bit Error Rate (BER) value, and diagnose high-speed serial links using conventional and statistical eye-diagrams. Section 3.2 presents the MUCTPI demonstrator. Section 3.3 describes the BER test firmware using an commercial IP core. Section 3.4 presents the developed software environment to configure the transceivers and measure the BER and statistical eye-diagrams. Section 3.5 describes the measurement results using only the MUCTPI. Section 3.6 presents the integration test results with RPC and TGC sector modules, and with L1Topo. Section 3.7 provides a summary of this chapter.

3.1 Introduction

As mentioned in Section 1.7, reliability is one of the requirements for the MUCTPI to receive and send trigger data successfully. The MUCTPI uses high-speed serial links to transfer data, and their reliability is ultimately judged on their BER performance. However, the BER test does not provide any qualitative information on why a given performance has been achieved or how it can be improved. Over the years, many engineers have used oscilloscopes to measure eye-diagrams of communication links. Eye diagrams provide an intuitive way of viewing how the performance is being limited.

More frequently than not, communication links are required to run with very low bit-rates, such as 10^{-12}, 10^{-15}, or even lower. On the other hand, sampling oscilloscopes have a very sparse sampling of the received data stream, which makes it extremely unlikely that a sampling oscilloscope will catch the one mistake in 10^{-15} bits that are being received. Sampling oscilloscopes are only able to sample a small part of the received data at a time. It takes much time to read the sampled data from memory and processing it until it is ready to sample data

again [23]. For this reason, many engineers are increasingly using BER contours or statistical eye diagrams that can catch low-probability errors.

The BER test result is very objective and provides a clear pass or fail result. However, judging if a sampling oscilloscope eye-diagram or a statistical eye-diagram looks good or not can be very subjective to the person who is looking at it. One could still measure the jitter, the rising and falling times, and the voltage amplitude separation, but multiple measurements would be required. Performance standards for the eye pattern diagnostic have been developed by professional associations, such as the IEEE [24], to ensure that a customer can verify with a single test the performance of standardized components, such as optical converters. These guideline measurements, known as *eye masks*, represent the keep-out regions of the eye where traces or bit errors should not exist at all, or in some cases, they are tolerated if occurring with a very low rate.

3.1.1 Bit Error Rate

BER is the ratio between the number of bit errors and the number of received bits. Higher the BER value is, lower is the reliability. Even in a system without any design issues, errors can still happen due to random noise from external sources, such as disturbances from Single Event Upset (SEU) and Electromagnetic interference (EMI). In these cases, the BER performance is limited by random noise and/or random jitter. It means that bit errors occur at random (unpredictable) times that can be bunched together or spread apart. For this reason, the number of errors that will occur over the lifetime of the system is a random variable. The computation of the probability of errors requires measurements with an infinite number of events, as indicated in Equation (3.1) [25].

$$P^{'}(\varepsilon) = \frac{\varepsilon}{n} \underset{n=\infty}{\rightarrow} P(\varepsilon), \tag{3.1}$$

where $P^{'}(\varepsilon)$ and $P(\varepsilon)$ represents the estimated and the actual values of the probability of error respectively. The parameter ε represents the number of errors detected in a given measurement, and n represents the number of received bits.

Hence the measurement of the error probability is not possible [25]. It is usually satisfactory to estimate the probability of the BER value to be lower than an upper limit with quantifiable confidence.

Usually, it is enough to say that the BER is at least as good as a required value defined for some design constraint or standard. For example telecommunications protocols, such as Synchronous Optical Networking (SONET) [26], require a BER of 10^{-10} using long Pseudo

Random Bit Sequence (PRBS) [27, p. 819], such as the PRBS-15 or PRBS-23, depending on the data transmission rate [28]. Data communications protocols, such as the Fiber Channel and Ethernet, commonly specify a BER performance of 10^{-12} using shorter bit sequences.

For example, for the MUCTPI SL inputs, it is acceptable to have a single bit error in 24 h. This link could potentially cause one fake trigger or lost event per day. Equation (3.2) presents the estimated bit error probability upper limit γ for a bit error ε_u in n_u received bits, where $\varepsilon_u = 1$, and $n_u = \Delta_{tu} F_{bit}$, with $\Delta_{tu} = 24$ h and $F_{bit} = 6.4$ Gb/s.

$$\gamma = P^{`}(\varepsilon_u) = \frac{\varepsilon_u}{n_u} = \frac{\varepsilon_u}{\Delta_{tu} F_{bit}} = \frac{1}{24 \times 60 \times 60 \times 6.4 \times 10^9} \approx 1.8 \times 10^{-15} \tag{3.2}$$

It is possible to measure that the probability of error $P(\varepsilon)$ is lower than 1.8×10^{-15} with a quantifiable Confidence Level (CL). Equation (3.3) [25] shows the definition of CL.

$$CL = P\left[P(\varepsilon) < \gamma_u | (\varepsilon_u, n_u)\right] \tag{3.3}$$

Based on this definition, the value of the confidence level for a given test without errors, i.e. $\varepsilon = 0$, is given by Equation (3.4) [25],

$$CL = 1 - e^{-\gamma n_r}, \tag{3.4}$$

where n_r represents the number of received bits in a given measurement.

Notice that CL depends only on the product γn_r that can be expressed in terms of the measurement time Δ_{tm} and Δ_{tu} used in Equation (3.2). Equation (3.5) describes this relationship.

$$\gamma n_r = \frac{1}{\Delta_{tu} F_{bit}} F_{bit} \Delta_{tm} = \frac{\Delta_{tm}}{\Delta_{tu}} \tag{3.5}$$

Finally, Equation (3.6) describes Δ_{tm} in terms of Δ_{tu} and CL.

$$\Delta_{tm} = -\Delta_{tu} \ln(1 - CL) \tag{3.6}$$

Therefore, for a $CL = 95\%$, one needs to measure no bit error in a time interval $\Delta_{tm} \approx 3 \times \Delta_{tu}$ to ensure that the bit error probability is lower than one bit error per time interval Δ_{tu}. In other words, to demonstrate that the error probability is lower than one error per day with $CL = 95\%$, one needs to measure no bit error during three days.

3.1.2 Eye-diagram

An eye-diagram is normally generated by an oscilloscope configured in infinite persistence display mode, which superimposes multiple oscilloscope acquisitions. After the accumulation of thousands of waveforms, the overlay of the samples triggered by the transmitter clock generates a so-called eye diagram, named so because the resulting image looks like the opening of an eye. If the eye diagram looks closed, it means that the edge timing is slow, and data-dependent or other jitter is significant. Figure 3.1 shows the connectivity block diagram for measuring the eye-diagram of two trigger outputs of the legacy MUCTPI using a 1 GHz analog bandwidth oscilloscope [29]. The eye diagram of both trigger outputs is measured from the accumulation of waveforms triggered by the rising edge of the MUCTPI transmitter clock.

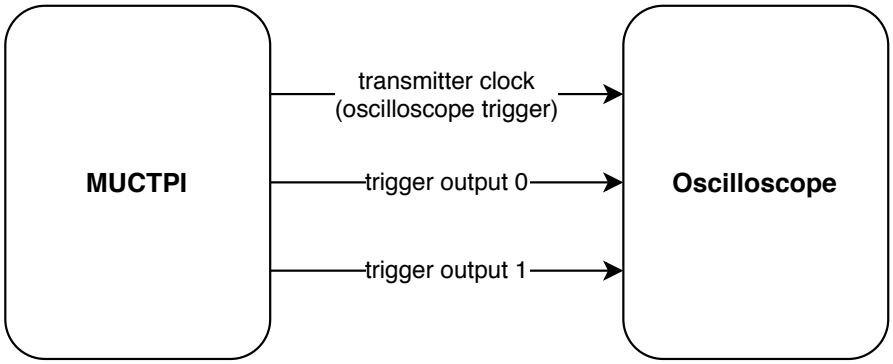


Figure 3.1 – Block diagram of the eye diagram measurement

Figure 3.2 shows the eye diagram of both trigger outputs running at 320 Mb/s. Voltage and time divisions are 150 mV and 500 ps, respectively. The eye is very wide for both trigger outputs.

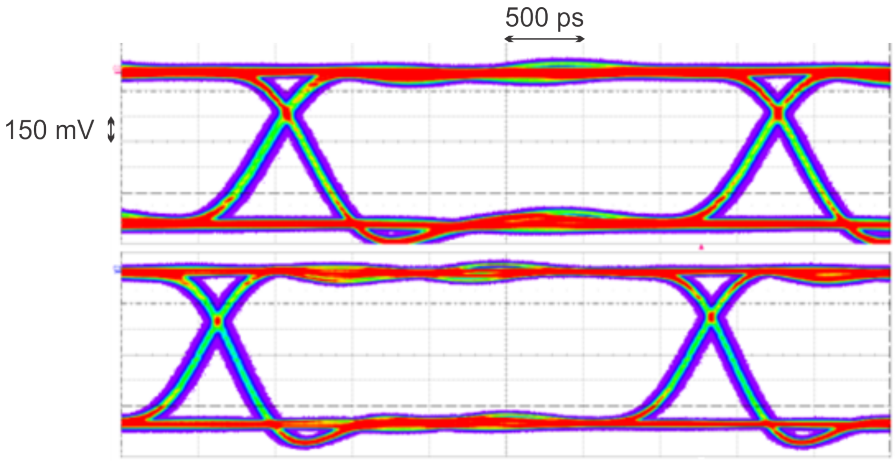


Figure 3.2 – Eye-diagram of two MIOCT outputs operating at 320 Mbps

3.1.3 Statistical eye-diagram

The statistical eye-diagram is generated by measuring the BER repeated times after applying different time and voltage offsets to the receiver sampler circuit. Figure 3.3 shows a statistical eye-diagram example. For all the statistical eye-diagrams in this thesis, the time and voltage offsets are represented in the x and y-axis, respectively. The x-axis is defined from -0.5 to 0.5 Unit Interval (UI), which corresponds to the time for transmitting one bit. The y-axis is represented in millivolts (mV) ranging from -190.5 mV to 190.5mV for Xilinx UltraScale GTH, and from -203.2mV to 203.2mV for Xilinx UltraScale/Ultrascale+ GTY FPGA transceivers.

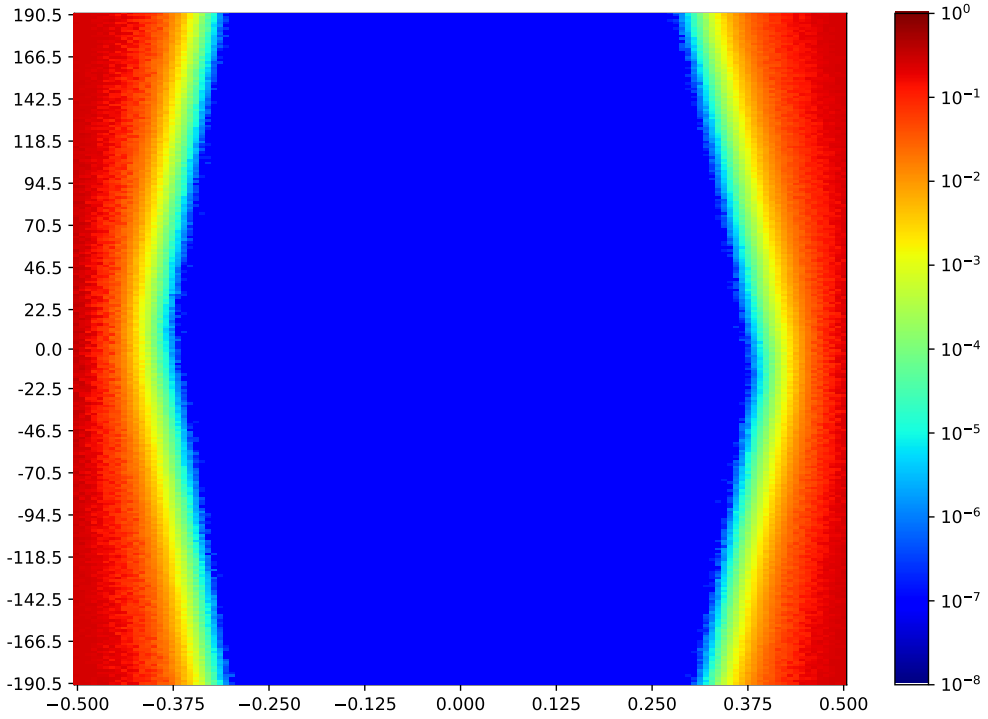


Figure 3.3 – Statistical eye diagram example

Xilinx MGT can generate statistical eye-diagrams non-disruptively, i.e., without disturbing the data transfer, and without requiring any external instrument. MGT is described in Chapter 4. To address the eye-scan functionality, an additional sampler with programmable time and voltage offsets is implemented in the receiver part of the MGT, after the Physical Medium Attachment (PMA) equalizer [30].

The error counter increments every time the data sample from the additional sampler with configurable time and voltage offsets disagrees with the data sampled by the main sampler circuit, with fixed offsets. Then, the BER is computed from the ratio between the number of bit errors and the total number of received bits.

The number of received bits for each time and voltage offset is defined in terms of the bit error probability upper limit γ, introduced in Section 3.1.1, also known as *target BER*. Notice that Xilinx considers $n_u = n_r$, i.e., the number of received bits in a measurement is the inverse of the target BER. In all eye-diagrams presented in this thesis, the target BER is set to 10^{-7}. This means that for each time and voltage offset, the received bit counter increments at least until 10^7. Hence, the time taken to measure each eye-diagram is inverse proportional to the target BER, i.e., as lower the target BER is, longer is the eye measurement. Reading an eye-diagram with all time and voltage offsets, e.g., 32895 measurement points, and with a target BER of 10^{-7} takes ≈ 1 min. A similar eye-diagram but with a target BER of 10^-15 is estimated to take 10^8 longer, i.e., 190 years. If one reduces the number of time and voltage offsets to only 81 points, it would still take 170 days to complete the measurement.

Equation (3.7) presents the computation of CL for the BER values without errors when $n_u = n_r$.

$$CL = 1 - e^{-\gamma n_r} = 1 - e^{-\frac{1}{n_u} n_r} = 1 - e^{-1} \approx 63\% \tag{3.7}$$

3.1.4 Eye mask compliance

A high-speed serial link with good amplitude separation and low jitter has an eye-diagram with a very wide opening in both time and amplitude axis. However, instead of performing several measurements to detect failed links, one can do it in a single test. The openness of an eye-diagram can be verified by performing an eye mask compliance test. An eye mask defines a region at which the eye-diagram should not exist [27, p. 362]. The IEC 61280-2-2 standard [p. 23][31] define two techniques to test eye-diagrams. In the first, known as *no-hits* technique, no traces (oscilloscope) or bit errors (statistical eye-diagram) should exist within the mask region. In the second, known as *hit-ratio* technique, a very small ratio of hit to samples (oscilloscope) or a very low BER (statistical eye-diagram) is allowed within the mask region. To improve the testing reproduce-ability, standards such as the IEEE Standard for Ethernet (IEEE Std 802.3-2015) [21] uses the hit-ratio technique.

Figure 3.4 [20] shows the reference mask provided by the MiniPOD manufacturer. It defines the eye mask coordinates of the MiniPOD receiver module. This mask is specified at a test point located at the host circuit board after the electrical connector. The variables $\{X1, X2, Y1, Y2\}$ are defined to $\{0.29\text{ UI}, 0.5\text{ UI}, 150\text{ mV}, 425\text{ mV}\}$ [20]. This mask is a scaled version of the mask in [21] and allows the same hit-ratio of 5×10^{-5}. Figure 3.5 shows a statistical eye diagram with the same mask. The diamond in the center represents the eye mask, and it is color-coded in green to indicate success and in red to indicate a failure in the eye mask compliance test using the hit-ratio technique. The link of this example passes the test by a large margin. The

mask top and bottom areas, defined by $Y2$ value, are not used for the statistical eye-diagram measurement.

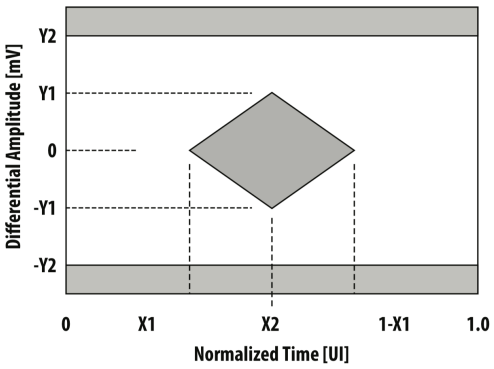


Figure 3.4 – MiniPOD eye-diagram mask

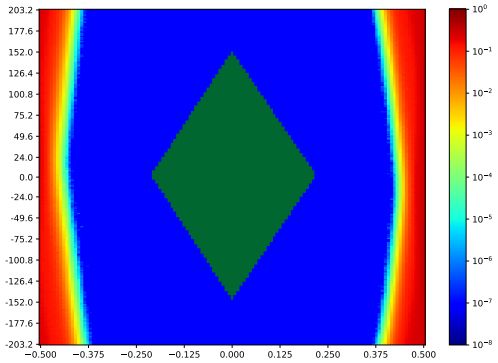


Figure 3.5 – Eye diagram with mask

3.2 MUCTPI demonstrator

The MUCTPI demonstrator has been developed to demonstrate the feasibility of using Xilinx UltraScale transceivers [19, 32, 33], and the 14 Gb/s Broadcom MiniPODs [20] in the MUCTPI application. The MUCTPI demonstrator hardware consists of a Xilinx VCU-108 evaluation board [34] and a custom double-width FPGA Mezzanine Card (FMC), so-called MPOD FMC [35]. The MPOD FMC, respective FPGA firmware and low-level software have been developed by the author of this thesis to receive TTC information, to transmit and receive data using Broadcom MiniPODs, measure online statistical eye diagrams, and to synchronize the SL input from the recovered clock to the system clock domain for combined data processing.

Figure 3.6 shows the MUCTPI demonstrator system where the FPGA evaluation board is on the left side and the MPOD FMC, the breakout optical cable [36], and the 8 columns LC adaptor are on the right side. The custom FMC card includes:

- Two jitter cleaners [37, 38] used to clean the TTC clock and then generate the MGT reference clock.
- Electrical LEMO [39] interface and optical SFP module [40] to receive TTC information.
- Transmitter and receiver 14 Gb/s Broadcom MiniPODs used to demonstrate error-free data transfer from SL modules to MUCTPI.
- 40 Gb/s QSFP+ module [41] used to measure eye-diagrams of the interface to HLT and DAQ systems.

- SubMiniature version A (SMA) [42] outputs used to measure clock jitter.
- Serial Peripheral Interface (SPI) [43] and I2C [44] interfaces for configuring several components of the board.

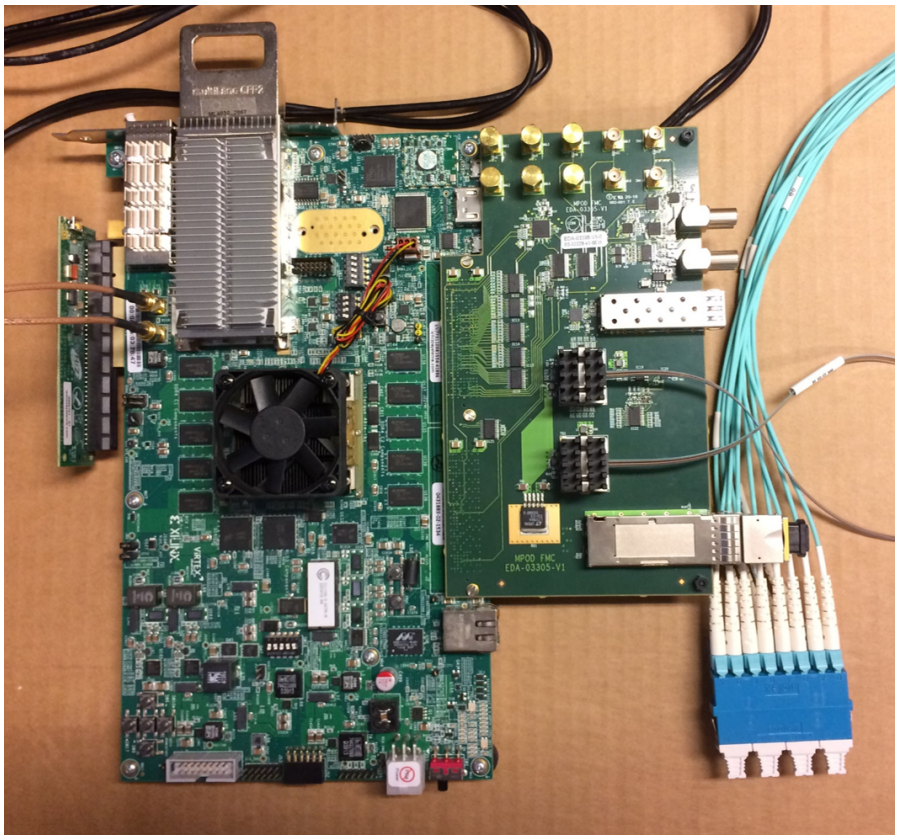


Figure 3.6 – MUCTPI system demonstrator

3.3 Bit-error-rate test firmware

Figure 3.7 shows the IBERT firmware for the MUCTPI-V2 and MUCTPI-V3 and the front-panel connectivity block diagram. The firmware is based on the Xilinx Integrated Bit Error Ratio Tester (IBERT) core [45, 46] that provides a broad-based PMA evaluation and demonstration platform for MGTs. It is parameterizable to be used with different line rates, reference clock rates, clock topologies, and data width. The IP core implements the transceiver configuration, pattern generator and checker, bit error, and total bit counters to measure the BER, access to Dynamic Reconfiguration Port (DRP) of the transceiver, and communication logic that allows the design to be controlled through the Joint Test Action Group (JTAG) interface. The pattern generator and checker supports several PRBS sequences and clock patterns. In addition, each

of the three FPGAs implements a SYSMON block for measuring the voltage in the transceiver power lines and also the FPGA temperature.

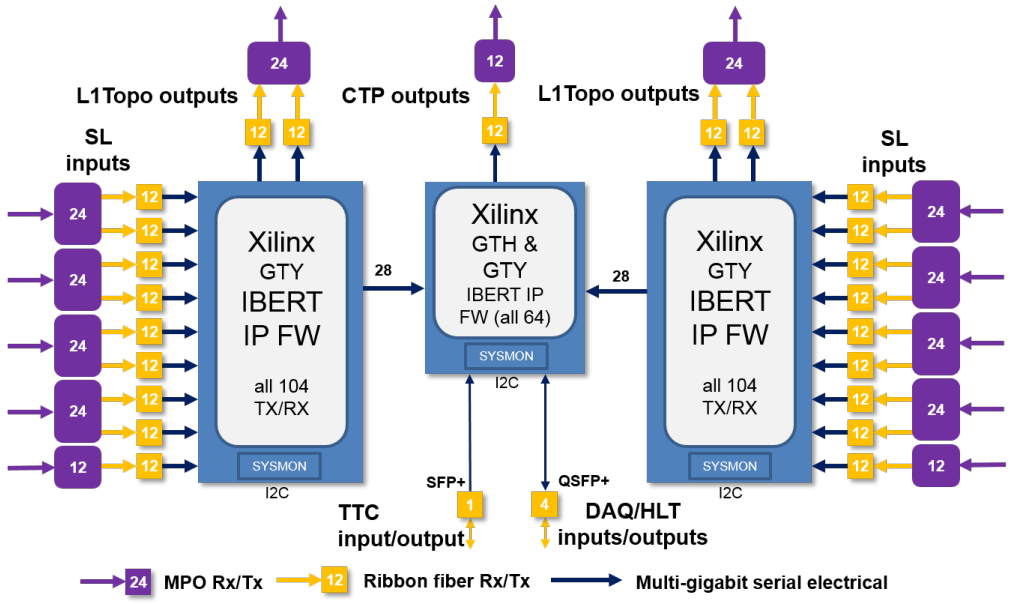


Figure 3.7 – IBERT firmware and connectivity block diagram

First, the user configures transmitter and receiver PMA settings such as emphasis, differential swing, and equalization. Second, the user ensures that the same data pattern is configured in the transmitter and receiver sides. Finally, the BER is computed from the ratio between the bit error and the total bit counter. The same IP core also implements the measurement of the statistical eye-diagram described in section 3.1.3.

For most of the cases, two 12-channel ribbon fibers coming from two MiniPODs, shown with a yellow line and yellow box respectively, are grouped together in groups of 24 channels. The exceptions are one MiniPOD at each MSP FPGA and one MiniPOD at the TRP FPGA, which are cabled to the front-panel individually. All the MiniPOD connections are accessed from the front-panel using 12/24 Multi-fiber Push On (MPO) connectors [36] indicated by a purple box. In addition, the 28 on-board high-speed serial link connections from each MSP FPGAs to the TRP FPGA are shown with dark blue lines. Finally, the SFP+ TTC input interface and the QSFP+ DAQ/HLT I/O are shown at the bottom of the picture with yellow boxes.

3.4 Bit-error-rate test software

In order to simplify the MUCTPI Printed Circuit Board (PCB) layout, swapping between the high-speed serial link channels and polarity inversions have been allowed. Due to the very high

number of high-speed serial links in the MUCTPI (334), reading the schematics thoroughly in order to extract the inter-connectivity, the pin assignments, and the link polarities is very difficult, time-consuming, and susceptible to human errors. To avoid these problems, the author of this thesis has developed the following two software packages:

- PCBpy: Python tool to extract connectivity from the back-annotated PCB net-list in order to generate VHDL wrappers, placements & polarity constraints and net-list verification reports [47]. The automatic generation of VHDL wrapper and constraints accelerates the design flow, in particular, when large FPGAs are used.
- IBERTpy: Python tool to manage Vivado IBERT tests by generating TCL scripts to automate the mapping between links in Vivado, configuring their respective polarities, running the BER tests and eye-scan measurements, plotting eye-diagrams, running eye-mask checks, generating horizontal, vertical, and area opening histograms, and compile all the results into a report (PDF) [48]

Figure 3.8 shows the serial link test automation flow diagram. First, the PCBpy tool reads the board design netlist and FPGA package pin files provided by the user. Second, IBERTpy generates TCL control scripts to configure the high-speed serial links connectivity and polarity. Third, the automatically generated TCL scripts control the Xilinx Vivado tool to run BER tests and to measure statistical eye diagrams. Step 4 illustrates the Xilinx Vivado connectivity to the MUCTPI through Ethernet using a hardware server (“virtual cable") running on the SoC, which is connected to the FPGAs via the JTAG chain. In step 5, Xilinx Vivado writes the BER results and the statistical eye diagrams into Comma-Separated Values (CSV) files. In step 6, IBERTpy reads the CSV files generated by Xilinx Vivado. In step 7, IBERTpy generates the statistical eye-diagram plots from the CSV files, generates histograms with the area, horizontal, and vertical opening, and run mask compliance tests. Finally, IBERTpy compiles all the test results in a PDF report file.

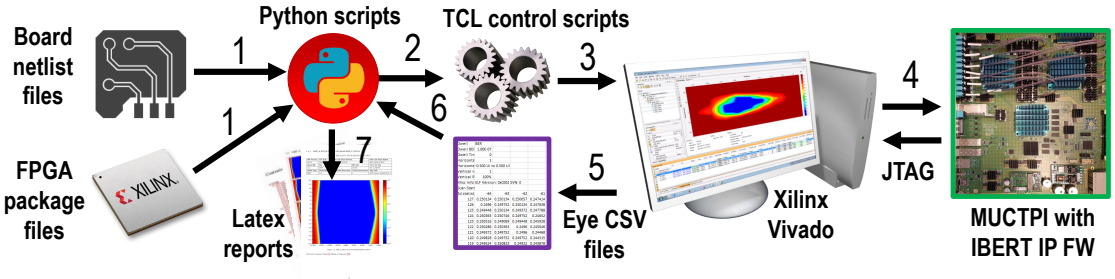


Figure 3.8 – Serial link test automation

Figure 3.9 shows a collage with three pages of the PDF report generated by IBERTpy. In the left at the back, one page of the table of contents is shown giving cross-reference hyperlinks

to summary and detailed view of the statistical eye diagrams. In the left at the front, shows one page of the summary view with all the 12 statistical eye-diagrams from one MiniPOD external loopback connection. Potential eye-opening differences between links from the same MiniPOD interface are easily detected in the summary view. On the right side of the picture, a detailed view of one eye-diagram is shown, including information such as the transceiver type, time-stamps, vertical and horizontal opening, measurement settings, and software version. The complete report is available at the IBERTpy GIT repository [48].

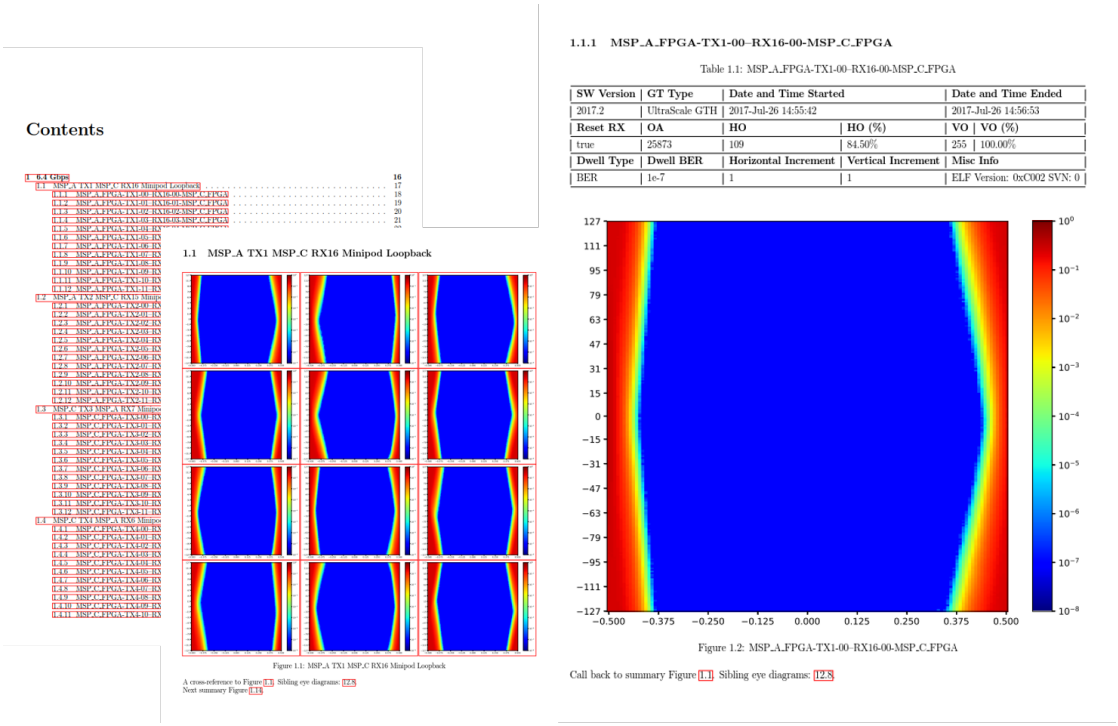


Figure 3.9 – IBERTpy generated report

The PCBpy has also been used to detect accidental polarity inversion of differential lines in the MUCTPI schematics. These errors have been detected and fixed before the first PCB had been produced. In addition, the VHDL wrappers and placing constraints generated by PCBpy have been used for other firmware developments in all the MUCTPI FPGAs.

Within CERN, the IBERTpy has also been used to generate eye-diagrams from the high-speed serial links of the Barrel Calorimeter Processor board [49], part of the Compact Muon Solenoid (CMS) Phase-II upgrade.

3.5 Test laboratory results

This section presents all the tests performed with the three MUCTPI prototype versions before the integration tests with SL and L1Topo systems. The Section 3.5.1 covers the BER tests. Section 3.5.2 describes the measurement of the eye diagram of one high-speed output of the MUCTPI to L1Topo. Section 3.5.3 presents the statistical eye diagrams from a randomly selected SL input driven by one of the L1Topo outputs connected through an external loopback. Section 3.5.4 covers the eye-opening area study for the 208 SL inputs. The study in Section 3.5.4 also covers the MUCTPI high-speed transmitter outputs because for the tests in the laboratory, the SL receivers are connected to one of the L1Topo or CTP MGT outputs through an external optical loopback. Section 3.5.5 presents the eye-diagram mask compliance test.

3.5.1 BER test

All of the MUCTPI serial connections, including on-board and off-board MGT links, for prototype versions 1, 2, and 3 have been checked for errors by transmitting and receiving PRBS-31 pattern data. Besides, two long-term BER run measurements have been performed for MUCTPI V2 and V3.

First, it has been measured the BER of 112 MGTs of MUCTPI-V2 running concurrently at 12.8 Gb/s during 10 days. 56 MGTs are connected using an external optical loopback from MSP and TRP MGT transmitters to MSP MGT receivers, and 56 are on-board MGTs from MSP to TRP FPGA.

Second, it has been measured the BER of 264 MGTs running at 12.8 Gb/s, where all the 208 MUCTPI V3 MGT SL inputs are driven by MGTs transmitters from MUCTPI V2 and V3 using an external optical loopback, and all the 56 MUCTPI V3 on-board MGT are connected using an internal electrical loopback. In order to test all the SL MGT inputs, the testing has been segmented in two parts with 3 days each. Notice that two MUCTPI prototypes features 120 off-board MGT transmitters, which is lower than the number of MGT inputs from one MUCTPI. In each of the test parts, 104 off-board MGT transmitters from both MUCTPI prototypes are connected to 104 MUCTPI V3 SL inputs. The on board links are tested in both test parts.

No errors have been detected in both long-term tests. For the first test, the BER is measured to be lower than 9×10^{-16} with $CL = 99.99\%$ for the 112 links. 9×10^{-16} corresponds to a single bit error per day in a link running in 12.8 Gb/s.

For the second test, i.e. including all the MUCTPI V3 MGT SL inputs and on-board MGT links, the BER is measured to be lower than 9×10^{-16} with a $CL = 95\%$ for the 208 SL inputs and $CL = 99.75\%$ for the 56 on-board links.

3.5.2 High-speed oscilloscope eye diagram

Figure 3.10 shows the eye diagram measured from one of the MSP MGT outputs running at 11.2 Gb/s using a high-speed oscilloscope [50] equipped with an optical-to-electrical converter [51]. 11.2 Gb/s is the bit rate used in the MSP MGT outputs connected to the L1Topo. The eye diagram shows a very wide horizontal opening of 76% at the transmitter output. Different FPGA transceiver pre-emphasis and Minipod TX input equalization control settings have been used, but no significant performance gain has been achieved. This was expected because the PCB tracks from the FPGA MGT to the Minipod TX are short, and the attenuation is negligible. The attenuation from loss with connectors and ribbon fibers at the MUCTPI high-speed outputs is lower than 3 dB. In general, for low loss channels, it is advised not to use any TX emphasis and let the RX adaptation handle all the equalization of the link [30]. The FPGA vendor considers low loss channels the ones with less than 14 dB attenuation at Nyquist.



Figure 3.10 – Oscilloscope eye diagram of one MSP MGT output running at 11.2 Gb/s

3.5.3 Statistical eye-diagram

This section presents the diagnostic of the MSP SL MGT inputs together with the MSP MGT L1Topo and CTP MGT outputs. These links have been tested at 6.4 Gb/s, for Run 3, and

12.8 Gb/s, used as a stress test. This stress test is meant to check how large is the operating margin and also to understand if these inputs can operate at higher bit rates in the future.

Figures 3.11 to 3.13 show the eye diagrams from a randomly selected SL input driven by one of the L1Topo outputs connected through an external loopback for MUCTPI version 1, 2 and 3 respectively. The eye diagrams of all MUCTPI versions show an excellent area opening of ≈ 75%. For the third prototype, the MiniPOD TX high-frequency equalization gain has been increased to equalize skin-effect losses across the circuit board. The setting value being used is 0x33 [20]. A study of the opening area for all the 208 inputs is presented in Section 3.5.4.

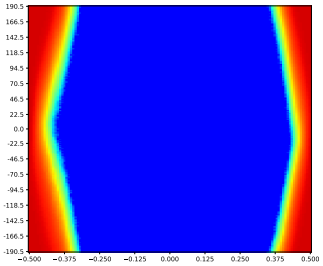


Figure 3.11 – V1 at 6.4 Gb/s

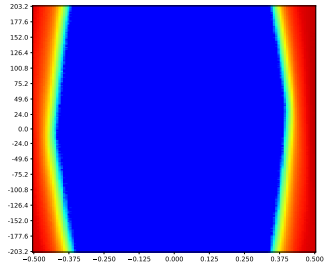


Figure 3.12 – V2 at 6.4 Gb/s

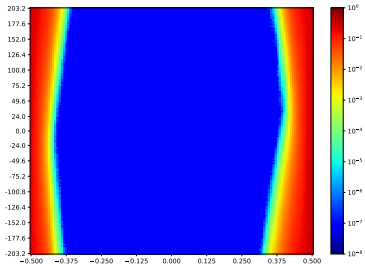


Figure 3.13 – V3 at 6.4 Gb/s

Figures 3.14 to 3.16 show the eye-diagram for MUCTPI V1, V2, and V3, respectively, from the same SL input running at 12.8 Gb/s. The MUCTPI V1 presents a lower opening area of ≈ 47% because this link uses an Ultrascale GTH transceiver that is tuned for lower bit rates. The Ultrascale GTH can run at up to 16.375 Gb/s. MUCTPI V2 and V3 have a higher opening area of ≈ 57% because this MUCTPI version features only Ultrascale+ GTY transceivers that are tuned for higher bit rates. The Ultrascale+ GTY transceiver can run at up to 30.5 Gb/s.

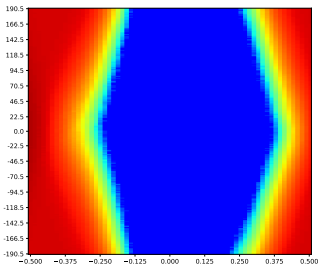


Figure 3.14 – V1 at 12.8 Gb/s

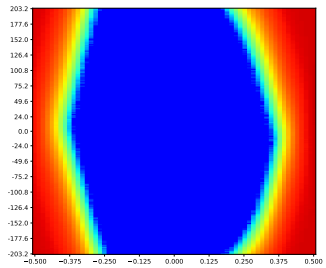


Figure 3.15 – V2 at 12.8 Gb/s

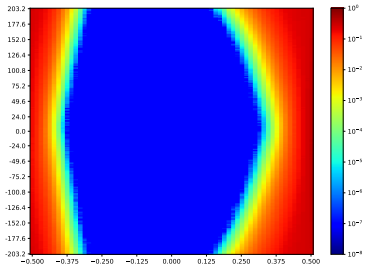


Figure 3.16 – V3 at 12.8 Gb/s

3.5.4 Eye opening area study

Figure 3.17 shows the Opening Area Percentage Histogram (OAPH) for all the 208 MUCTPI-V1 SL inputs running at 6.4 Gb/s. The opening area ranges from 55% up to 80%, with an average opening area of 67%. Two groups or sets have been found. They correspond to the different

performances given by the Ultrascale GTH and GTY transceivers of the MSP FPGA in the MUCTPI V1. The set with lower opening area corresponds to receivers with GTY transceivers, which can run up to 30.5 Gb/s, and the set with higher opening area corresponds to receivers with GTH transceivers which can run up to 16.375 Gb/s.

Figure 3.18 shows the OAPH for the MUCTPI-V2 SL inputs running at 6.4 Gb/s. The opening area ranges from 66 % up to 78%, with an average of 74%. Only one set is found as all the receivers are implemented using the Ultrascale+ GTY transceivers. The Ultrascale+ GTY performs almost as well as the Ultrascale GTH transceivers when running at 6.4 Gb/s, moving up the overall opening area worst case by more than 10%, i.e. from 55%, for the MUCTPI-V1, to 66%, for the MUCTPI-V2.

Figure 3.19 shows the OAPH for the MUCTPI-V3 SL inputs running at 6.4 Gb/s. The opening area ranges from 70 % up to 78%, with an average of 75%. The slight improvement in the opening area, compared to MUCTPI V2, is thanks to the equalization setting being used in MUCTPI V3, see Section 3.5.3. There are no schematics or layout differences between MUCTPI V2 and V3 with regard to the high-speed serial links.

Figure 3.20 shows the OAPH for the MUCTPI V1 SL inputs running at 12.8 Gb/s. This bit-rate is used as a stress-test. The opening area ranges from 40% up to 62%, with an average value of 50%. The two different sets that have been found for the MUCTPI V1 running at 6.4 Gb/s are more close together when running at 12.8 Gb/s. The closer distance between the two sets indicates that the difference in performance between Ultrascale GTH and GTY is more significant for lower rates. This histogram also shows the significant degradation of performance when running the links in 12.8 Gb/s. The opening area worst case is moved down by more than 15% compared to the link running at 6.4 Gb/s for the same version of the MUCTPI.

Figure 3.21 shows the OAPH for the MUCTPI V2 SL inputs running at 12.8 Gb/s. The opening area ranges from 44% up to 62%, with an average of 54%. The opening area is moved up by 4% both in worst case and average values when compared to MUCTPI-V1. Figure 3.22 shows the OAPH for the MUCTPI V3 SL inputs running at 12.8 Gb/s. The opening area ranges from 39% up to 63% with an average of 55%. The worst-case opening area is decreased by 5% compared to MUCTPI V2.

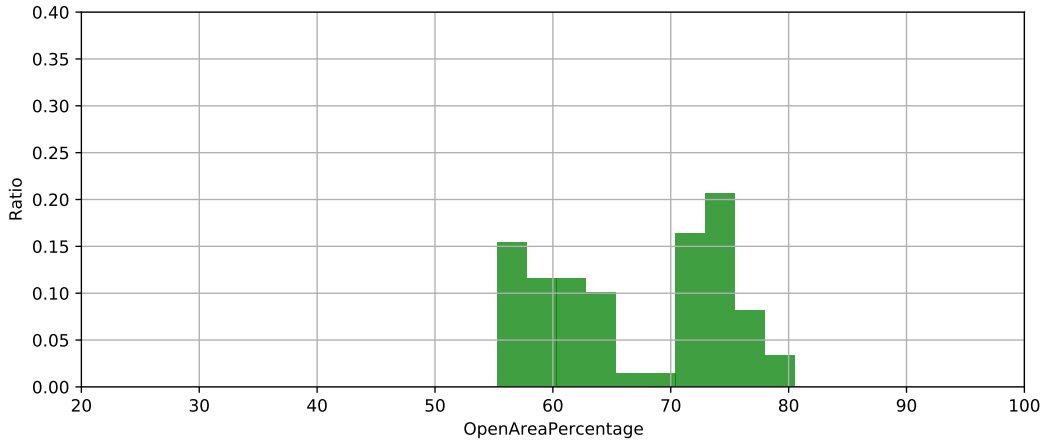


Figure 3.17 – OAPH MUCTPI-V1 SL 6.4 Gb/s

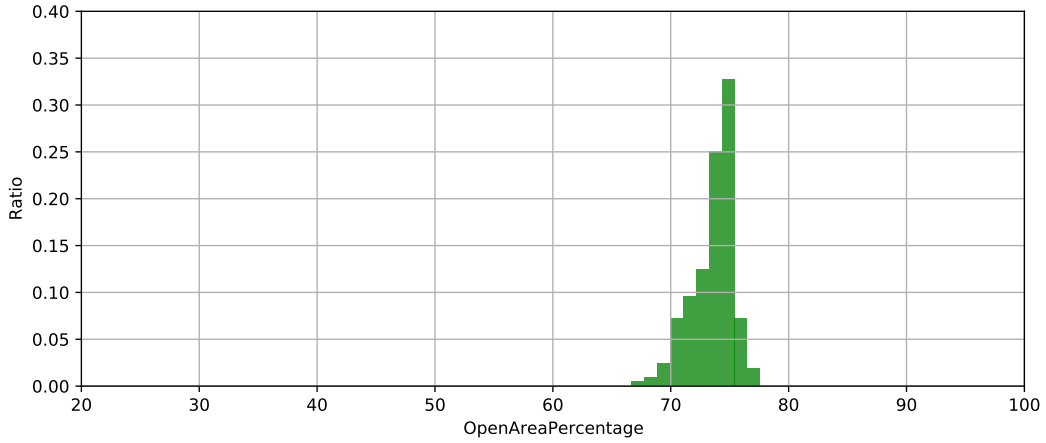


Figure 3.18 – OAPH MUCTPI-V2 SL 6.4 Gb/s

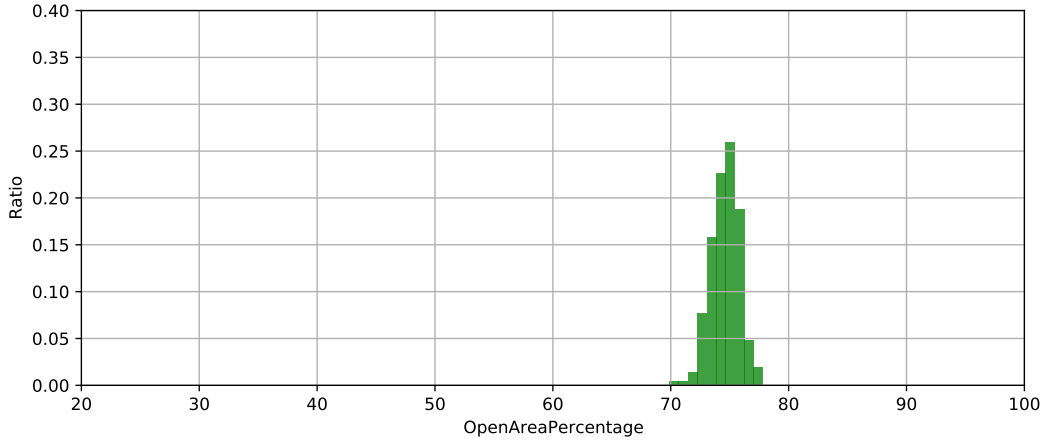


Figure 3.19 – OAPH MUCTPI-V3 SL 6.4 Gb/s

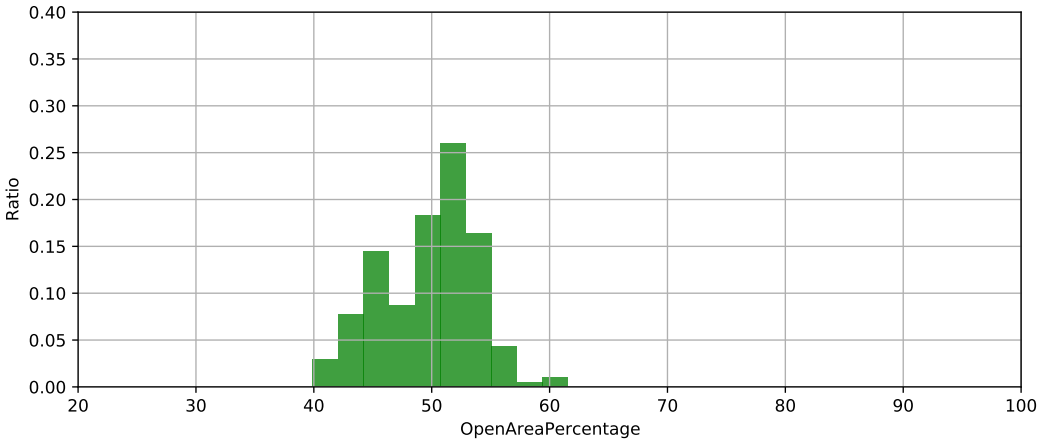


Figure 3.20 – OAPH MUCTPI-V1 SL 12.8 Gb/s

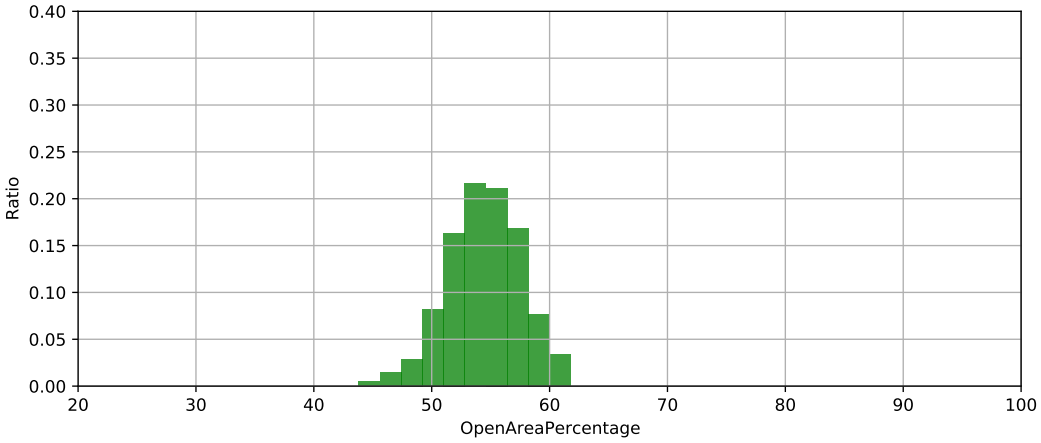


Figure 3.21 – OAPH MUCTPI-V2 SL 12.8 Gb/s

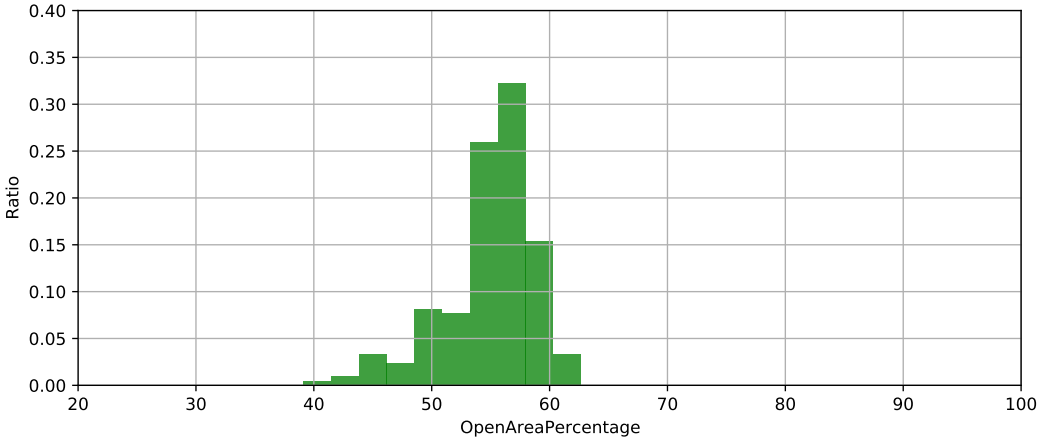


Figure 3.22 – OAPH MUCTPI-V3E SL 12.8 Gb/s

3.5.5 Eye-diagram mask compliance test

The eye-diagram mask check test, presented in Section 3.1.4, has been performed for all the on-board and off-board high-speed serial links in MUCTPI V1, V2, and V3 running at 6.4 Gb/s and 12.8 Gb/s. All the links passed the test.

Figures 3.23 and 3.24 show the eye-diagrams with the mask check of the worst-case and best-case opening area links running at 6.4 Gb/s, respectively. Both have 70% and 78% opening area, respectively, and they pass the eye-diagram mask compliance test with a large margin.

Figures 3.25 and 3.26 show the eye-diagrams with the mask check of the worst-case and best-case opening area links running at 12.8 Gb/s, respectively. This bit-rate is used as a stress-test. Both have 40% and 63% opening area, respectively. The worst-case eye-diagram opening area passes the test with a very low margin. There are bit errors within the right corner of the mask, but the BER at this region is lower than the acceptable hit-ratio of 5×10^{-5}. The best-case eye-diagram opening area passes the test with a good margin.

The results presented here are consistent with the BER test presented in Section 3.5.1. In both cases, no errors have been detected, and all links have passed the test.

3.6 Integration test results

Integration tests have been performed with the RPC and TGC sector logic modules transmitting data to MUCTPI, and the MUCTPI transmitting data to L1Topo. The goal of this test is to verify if all the systems are able to transfer data without errors. The sector logic module links are running in 6.4 Gb/s and the L1Topo links in 11.2 Gb/s. In both cases, the test data pattern has been set to PRBS-31. This section covers the data transfer reliability measurements done during the integration tests. Synchronization test and latency measurements are covered in Chapter 5.

3.6.1 RPC and TGC sector logic modules

The integration tests started in November 2016 with the TGC sector logic module prototype and the MUCTPI demonstrator. Later in November 2017, a new integration test has been performed with the TGC sector logic module prototype and the MUCTPI prototype version 1. Finally, in November 2018, integration tests have been performed with RPC sector logic module interface card.

The BER test using the IBERT firmware described in Section 3.3 worked smoothly in the three integration tests, no errors have been found after an overnight test.

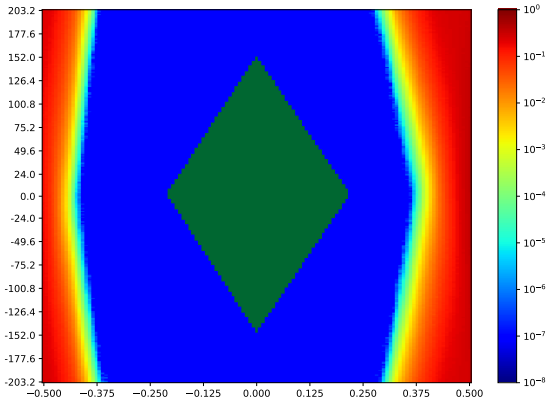


Figure 3.23 – Worst V3 at 6.4 Gb/s

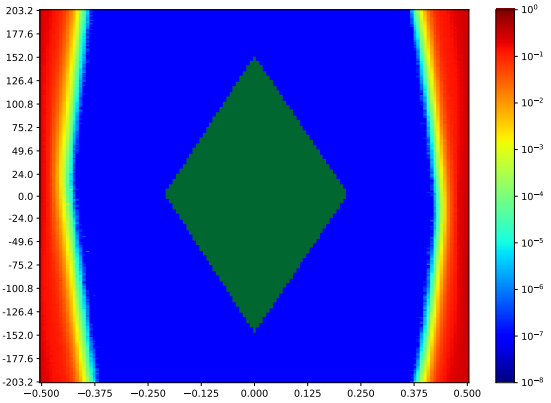


Figure 3.24 – Best V3 at 6.4 Gb/s

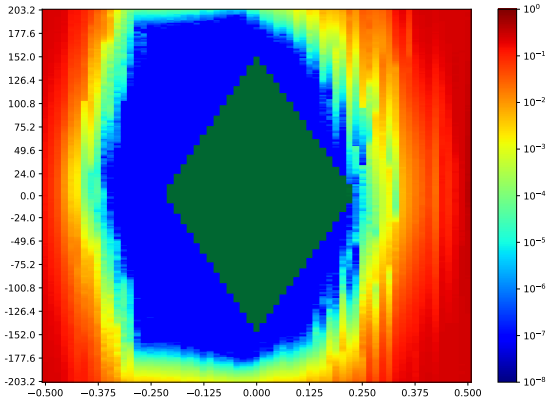


Figure 3.25 – Worst V3 at 12.8 Gb/s

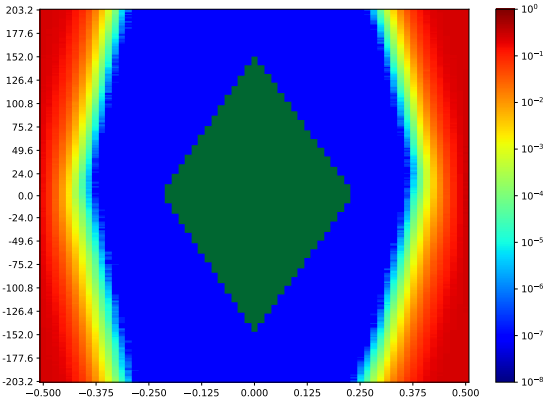


Figure 3.26 – Best V3 at 12.8 Gb/s

Figure 3.27 shows the block diagram of the TGC SL and MUCTPI integration test. A common clock is distributed to TGC SL and MUCTPI using the TTC system. All the 12 SL outputs of one TGC SL module are connected to the MUCTPI through a passive optical breakout cassette [52]. The cassette interconnects 24 individual optical fibers to a single MPO-24 trunk cable. For this test, only 12 out of the 24 optical fiber inputs of the cassette are used. Notice that the clock is not transmitted along with each SL output. Instead, the MUCTPI recovers the clock from the data.

Figures 3.28 to 3.39 shows the eye diagram of each of the 12 outputs of the TGC sector logic module prototype connected to the MUCTPI prototype version 1. UltraScale GTH and GTY transceivers have been used at the MUCTPI and 7-series GTX transceivers have been used at the TGC sector logic module card.

The eye-opening is very good, with area opening ranging from 58% to 74%. The eye diagrams of TGC SL channels 0,1,2,3, 5 and 9 connected to MUCTPI GTH channels are wider than the SL

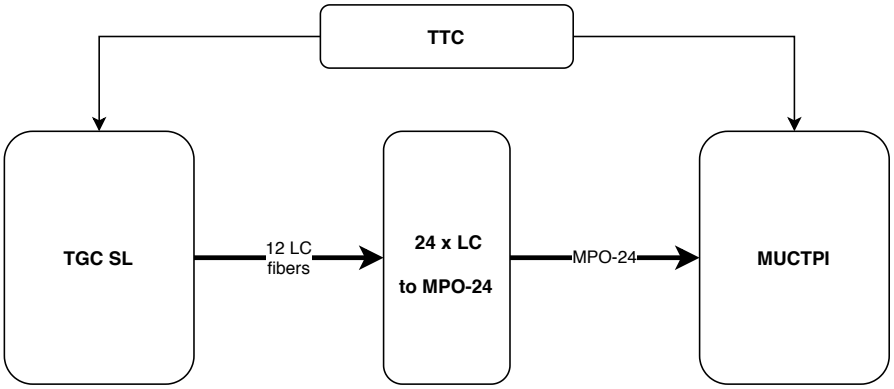


Figure 3.27 – TGC integration test block diagram

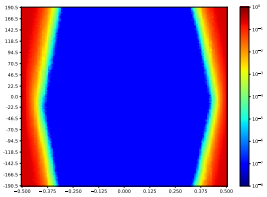


Figure 3.28 – Ch. 0

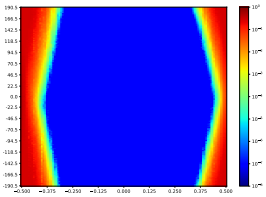


Figure 3.29 – Ch. 1

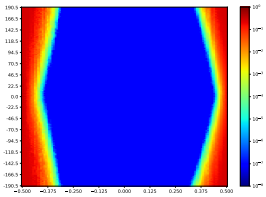


Figure 3.30 – Ch. 2

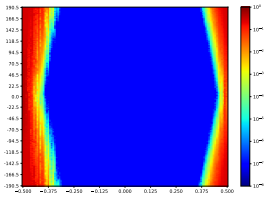


Figure 3.31 – Ch. 3

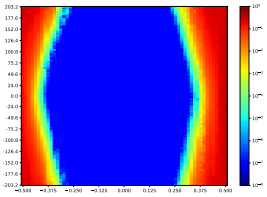


Figure 3.32 – Ch. 4

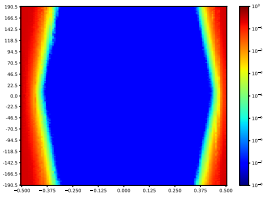


Figure 3.33 – Ch. 5

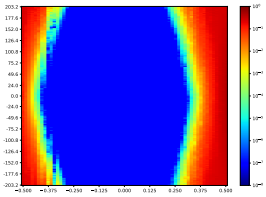


Figure 3.34 – Ch. 6

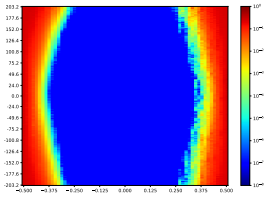


Figure 3.35 – Ch. 7

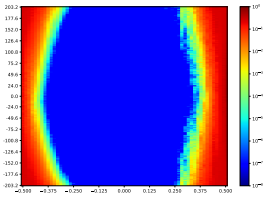


Figure 3.36 – Ch. 8

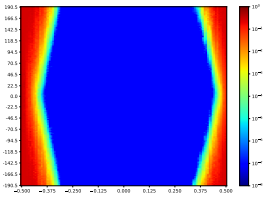


Figure 3.37 – Ch. 9

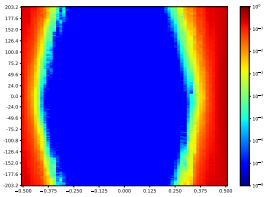


Figure 3.38 – Ch. 10

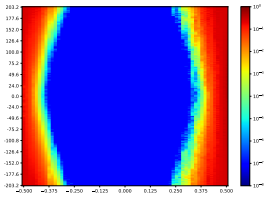


Figure 3.39 – Ch. 11

outputs 4, 6, 7, 8, 10 and 11 connected to MUCTPI GTY channels. This performance difference has also been observed in the laboratory tests presented in Section 3.5.4 in the Figure 3.17, which UltraScale GTH outperforms UltraScale GTY transceiver for low bit rates.

Figures 3.40 and 3.41 show two eye diagrams of the RPC SL interface card optical output connected to the MUCTPI prototype version 1. The connectivity is similar to Figure 3.27 except that the RPC SL features only one output. An UltraScale GTH and a 7-series GTP transceivers have been used at the MUCTPI and RPC interface card, respectively. In the first figure, no optical attenuator is used, and in the second figure, a passive 7dB optical attenuator is inserted in the path. This is to measure the closing of the eye diagram for channels with higher attenuation. In both cases, it has been measured a very wide opening area of 65%. No significant difference has been observed between the two eye diagrams because the FPGA transceiver linear equalizer at the receiver can compensate very well low loss channels.

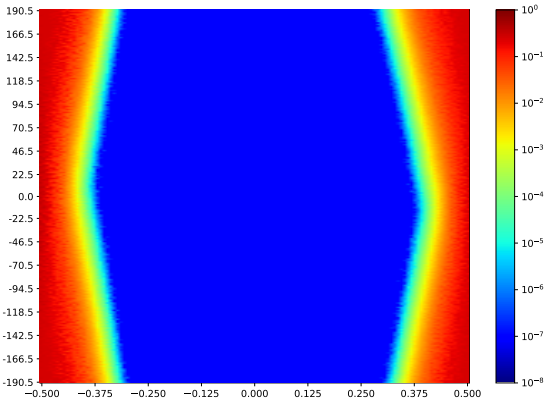


Figure 3.40 – RPC eye-diagram

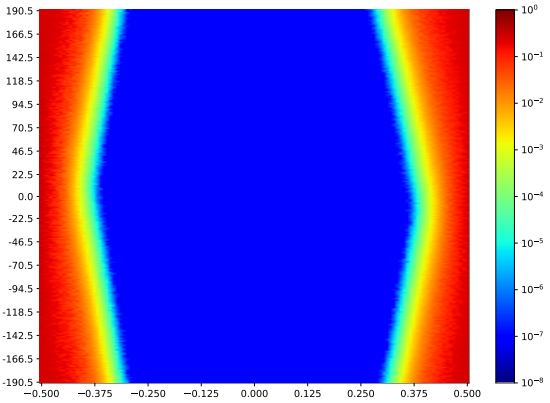


Figure 3.41 – RPC eye-diagram 7 dB

3.6.2 L1Topo

The integration test between the MUCTPI and the L1Topo processor took place in May 2019. For this integration test, the MUCTPI version 2 has been used. UltraScale+ GTY transceivers have been used on both ends. All the 48 optical outputs of the MUCTPI MSP FPGAs running at 11.2 Gb/s have been connected to L1Topo. No errors have been found in 45 out of 48 links after BER measurement test of 39 h. This corresponds to a BER of $\approx 1.9 \times 10^{-15}$ with a confidence level of 95%. Unfortunately, this measurement could not run longer because both MUCTPI and L1Topo were needed in their laboratories for development work. Three channels are known to be failing at the L1Topo side. The problem is understood and should be fixed for their next prototype.

Figures 3.42 and 3.43 show the best and worst measured eye diagrams. The vertical opening at the center of the eye is 100% in both cases, and the horizontal opening is 68% and 69% for worst and best cases, respectively.

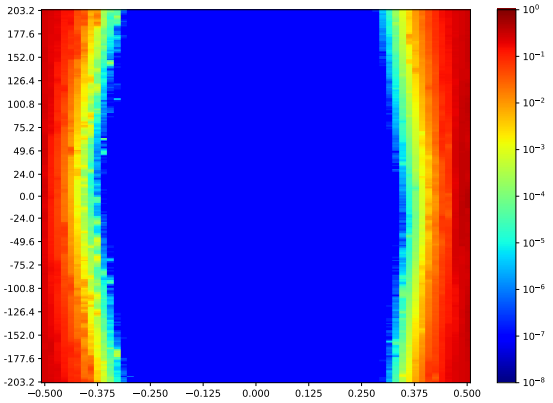


Figure 3.42 – Best L1Topo eye-diagram

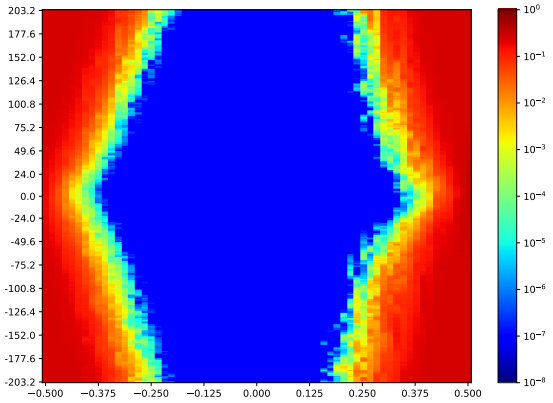


Figure 3.43 – Worst L1Topo eye-diagram

A configurable optical attenuator [53] has been used to measure the closing of the eye diagram after gradually increasing the channel attenuation. The links with the eye diagram shown in Figures 3.42 and 3.43 have been selected. After gradually increasing the attenuation, both links started having errors with an attenuation of 7.75 dB and 8.25 dB, respectively. The BER measurement time for each attenuation value is 10 s, which corresponds to a BER $\approx 10^{-11}$. Note that the higher power margin has been measured in the link that initially had the worst eye-opening.

Figures 3.44 to 3.49 shows the eye-diagram of the link with lower power margin connected to the MUCTPI through the configurable optical attenuator set to 1.25 dB (minimum insertion loss), 5.25 dB, 7.25 dB, 7.75 dB, 8.25 dB and 9.25 dB respectively. The attenuation level has been also verified in real-time using an optical power meter [54] connected to the monitoring output of the configurable optical attenuator.

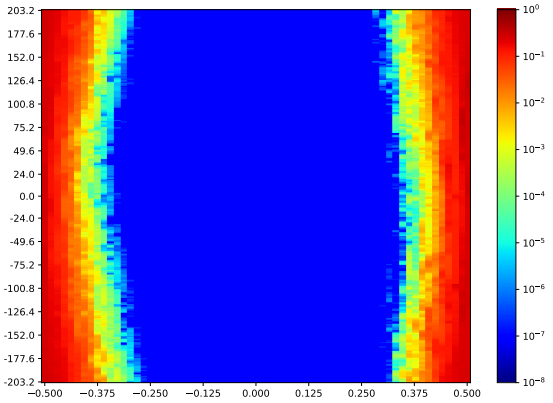


Figure 3.44 – Best eye-diagram 1.25 dB

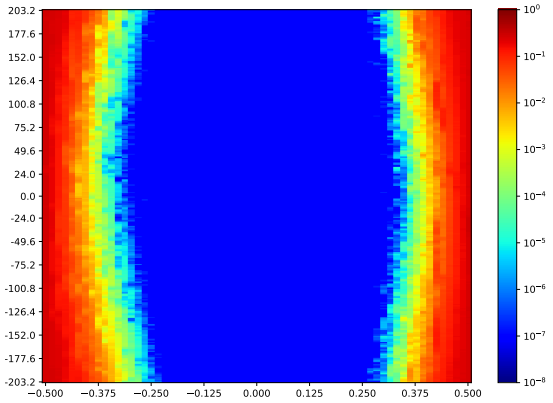


Figure 3.45 – Best eye-diagram 5.25 dB

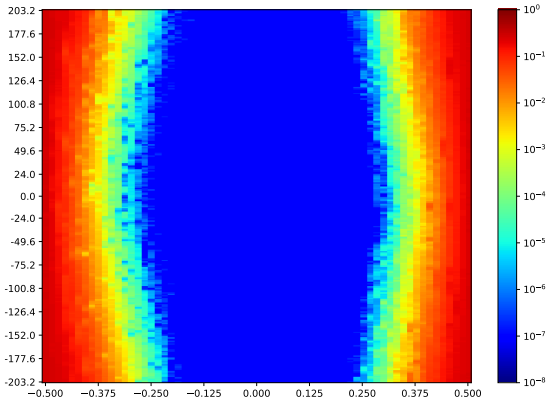


Figure 3.46 – Best eye-diagram 7.25 dB

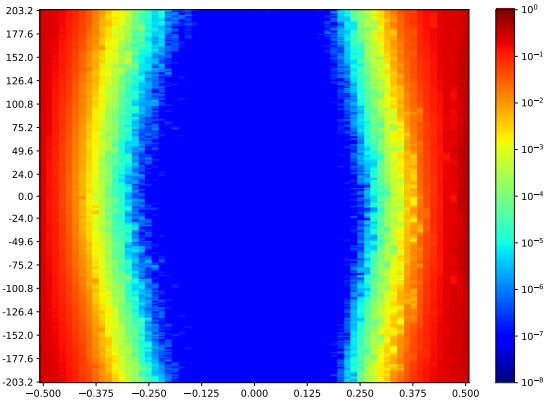


Figure 3.47 – Best eye-diagram 7.75 dB

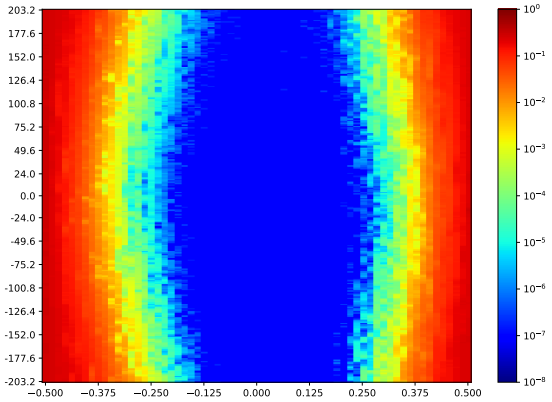


Figure 3.48 – Best eye-diagram 8.25 dB

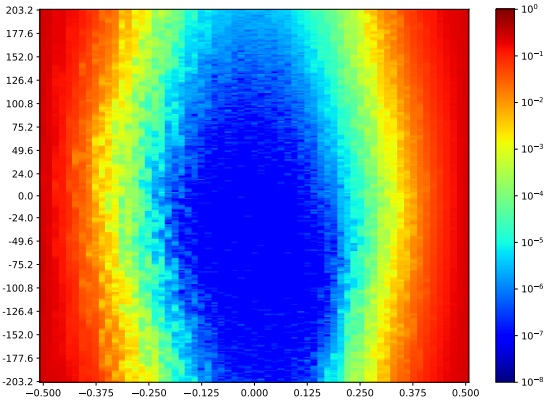


Figure 3.49 – Best eye-diagram 9.25 dB

Note that no significant closing of the eye is seen for 1.25 dB and 5.25 dB attenuation. At 7.25 dB, the eye starts closing quicker, and already in 7.75 dB, the link started having errors. The eye closes even more for 8.25 dB and 9.25 dB attenuation values. At the last point, the eye is already very closed, and the BER is high.

Figures 3.50 to 3.53 show the eye-diagram of the link with the higher power margin connected to the MUCTPI through the configurable optical attenuator set to 5.25 dB, 7.25 dB, 8.25 dB and 9.25 dB respectively.

No significant closing of the eye has been seen for 5.25 dB attenuation. At 7.25 dB, the eye is already more closed, but no errors have been detected. For 8.25 dB and 9.25 dB, errors have been detected, and the vertical opening at the center of the eye is significantly reduced to 60.78% and 20%, respectively. Both tests indicate that the power margin for the MUCTPI links to L1Topo is in the order of 7 dB. This optical power margin is very good because no major

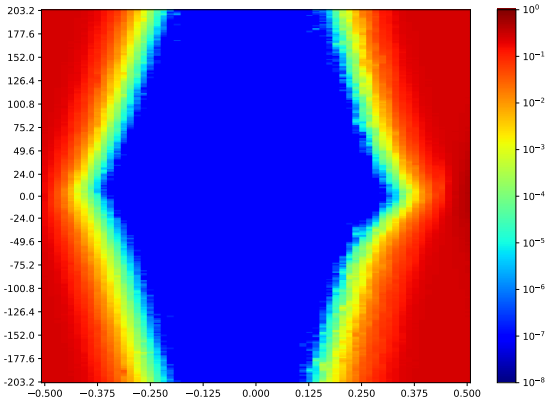


Figure 3.50 – Worst eye-diagram 5.25 dB

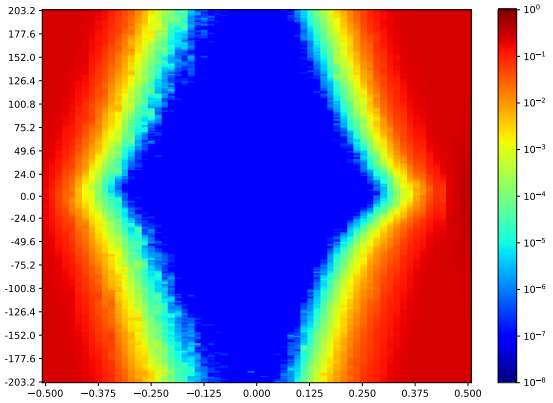


Figure 3.51 – Worst eye-diagram 7.25 dB

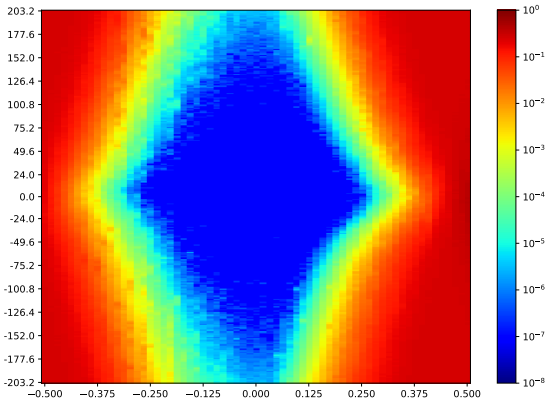


Figure 3.52 – Worst eye-diagram 8.25 dB

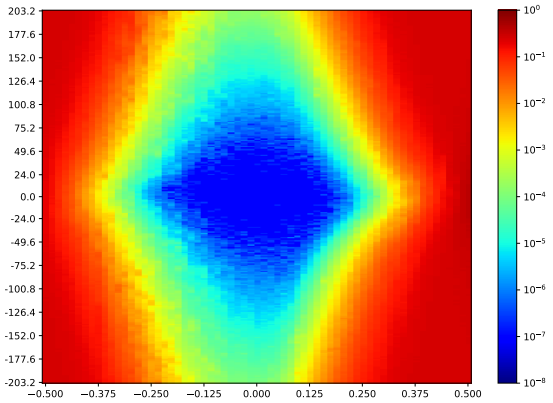


Figure 3.53 – Worst eye-diagram 9.25 dB

changes are expected in the optical installation after MUCTPI and L1Topo are deployed. Also, even if more connectors are used, the insertion loss from each standard MPO connector is limited to 0.75 dB [55].

3.7 Summary

A BER test firmware and software have been developed to measure the BER and the eye diagram of the MUCTPI high-speed serial links. The firmware development is greatly simplified with the usage of IPs provided by the vendor. With respect to the software, two Python packages have been developed. The first extract inter-connectivity from the back-annotated PCB netlist in order to generate VHDL wrappers, placement & polarity constraints, and netlist verification reports. The second manages Vivado IBERT tests by generating TCL scripts to automate the interconnection between links in Vivado, configures their respective polarities, measures

the BER value and the statistical eye-diagrams, runs eye-mask checks, generates horizontal, vertical, and area opening histograms and compiles all the results into a PDF report.

Two long-term BER run measurements have been performed for MUCTPI V2 and V3. First, it has been measured the BER of 112 MGTs of MUCTPI-V2 running at 12.8 Gb/s during 10 days. Second, it has been measured the BER of 264 MGTs running at 12.8 Gb/s, where all the 208 MUCTPI V3 MGT SL inputs are driven by MGTs transmitters from MUCTPI V2 and V3 using an external optical loopback, and all the 56 MUCTPI V3 on-board MGT are connected using an internal electrical loopback. The second long-term test has been segmented in two parts with 3 days each. No errors have been detected in both long-term tests. For the first test, the BER is measured to be lower than 9×10^{-16} with $CL = 99.99\%$ for the 112 links. For the second test, i.e. including all the MUCTPI V3 MGT SL inputs and on-board MGT links, the BER is measured to be lower than 9×10^{-16} with a $CL = 95\%$ for the SL inputs and $CL = 99.75\%$ for the on-board links.

In addition to the long-term BER test, metrics extracted from eye-scans, such as horizontal, vertical and area opening, and eye mask compliance checks have been used as a way to detect failing links and to compare the measured performance to other links in the same board or to a different prototype version of the MUCTPI. A high-speed oscilloscope has been used to measure the optical eye-diagram of one of the MUCTPI outputs to L1Topo operating at 11.2 Gb/s. The eye diagram demonstrated a very wide horizontal opening of 76% at the transmitter output.

A comparative study of the eye-diagram opening area and an eye-mask compliance check for MUCTPI prototype versions 1, 2, and 3 demonstrated that all versions perform very well in the SL bit rate of 6.4 Gb/s, used for the Phase-I upgrade. It has been measured that the opening of the eye-diagram decreases when operating at 12.8 Gb/s, used as a stress test. However, even with the smaller eye-diagram opening, all the links for all prototype versions pass the eye-diagram mask compliance test, and the BER is lower than one error per day with a confidence level of 95%.

Integration tests have been performed from RPC and TGC sector logic modules to the MUCTPI, and from the MUCTPI to L1Topo. No errors have been found in any of these tests. The first integration tests with TGC sector logic modules took place even before the first MUCTPI prototype being available thanks to the MUCTPI demonstrator.

The eye-opening from the sector logic modules to the MUCTPI is very good, with an opening area ranging from 58% to 74%. No performance degradation has been measured after a passive 7dB optical attenuator is inserted in the path. Tests using an optical attenuator module demonstrated that the power margin for the MUCTPI links to L1Topo is in the order of 7 dB. The power margin in both cases is very good because no major changes are expected in

the optical installation after RPC, and TGC sector logic modules, MUCTPI, and L1Topo are deployed.

4 FPGA transceiver latency optimization

This chapter presents the optimization studies on the FPGA transceiver configuration in the interest of low and fixed latency. Section 4.1 introduces the importance of carefully controlling and measuring FPGA transceiver latency for low-latency system-synchronous applications. Section 4.2 provides a brief introduction to FPGA transceivers. Section 4.3 presents the optimization work in the FPGA clock fabric and data path transceiver configuration. Section 4.4 provides a summary of this chapter.

4.1 Introduction

As the high-speed transceiver circuits are heavily pipelined, they contribute a significant part to the system latency. Therefore, their latency has to be carefully controlled and measured. As deterministic latency is not a requirement for most of the applications, transceiver design usually simplifies its synchronization circuit at the cost of increased and non-deterministic latency.

This chapter describes the investigation work performed as part of this thesis in the configuration of FPGA transceivers to ensure that the low latency and fixed latency requirements previously mentioned in Section 1.7 are fulfilled.

4.2 FPGA transceivers

Transceivers are composed of a transmitter and a receiver unit to serialize and deserialize data, respectively. Multi-Gigabit Transceivers (MGTs) operates in serial bit rates above 1 Gb/s and often support many different use modes with configurable serial bit rates and parallel interface widths. Nowadays, most of the FPGA devices feature MGT blocks as part of their I/O resources given that FPGAs are suitable for parallel data processing and are highly configurable. They

are often used in data communication because they enable serial data transmission in high bit rates keeping the processing running parallel. Thus, in a lower clock frequency.

Figure 4.1 shows a simplified block diagram of the FPGA-based high-speed data transfer scheme between the SL module to the MUCTPI. A similar scheme applies to different subsystems of the ATLAS L1 trigger system. The SL module implements the transmitter side of the transceiver while the MUCTPI implements the receiver. The electrical-to-optical converters are not shown in this diagram. The transmitter uses a reference clock derived from the bunch crossing clock to ensure the transfer is synchronous to the other elements of the L1 trigger system. At the SL module, the transmitter part of the transceiver provides a user interface clock to the FPGA user logic, where the trigger functionality is implemented. The trigger information of the SL module is connected to the data input of the transceiver, which is then serialized and transmitted to the MUCTPI synchronous to a multiplied clock derived from the transmitter reference clock.

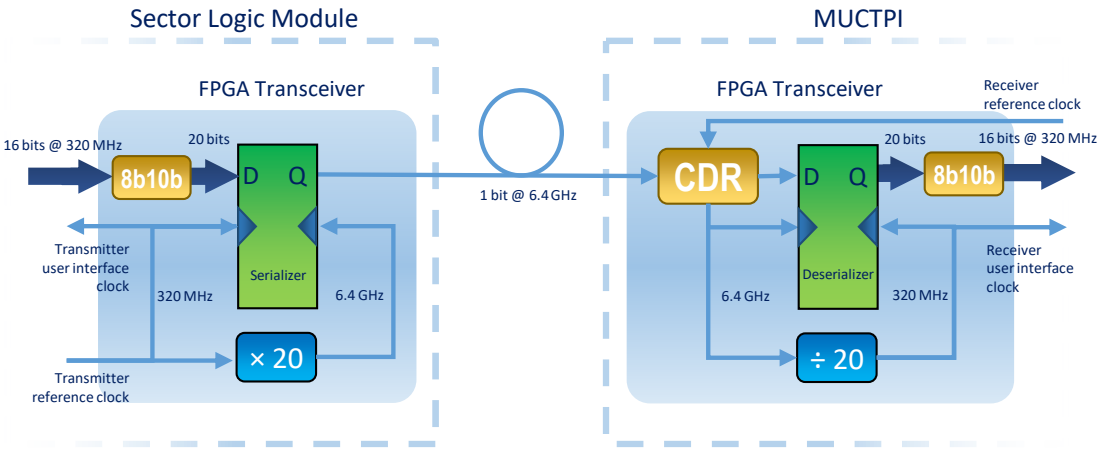


Figure 4.1 – Simplified block diagram of a FPGA-based high-speed data transfer scheme

The user interface data port and the serializer have different input widths because 8b10b encoding [30] is used in the data transfer from the SL to MUCTPI and MUCTPI to L1Topo. For every rising edge of the transmitter user interface clock, the user drives 16 data bits, but a total of 20 bits are serialized after 8b10b encoding.

The 8b10b encoding scheme is used to ensure the data stream contains enough transitions that are important to guarantee that the CDR can recover the clock at the receiver. 8b10b encoding also ensures the data are DC-balanced, which allows the use of a capacitive coupling. Capacitive coupling brings a lot of benefits such as exempting from the need of using level shifting converters, removing common-mode errors, and protecting against input-voltage fault

conditions. Finally, 8b10b encoding also offers easy discrimination at the receiver between control commands and data symbols, and the detection of single-bit errors.

The clock is not transmitted along with the data. Therefore the MUCTPI has to recover the clock from the received data using a Clock Data Recovery (CDR) block embedded in the receiver side of the FPGA transceiver. This CDR uses as reference a multiplied clock derived from the receiver reference clock, which is also derived from the bunch crossing clock. The recovered clock and the received data are connected to a deserializer block that outputs the received data in parallel to the FPGA user logic. The recovered clock is divided in the same ratio of the deserialization and is connected to the FPGA user logic in order to drive the clock used by the MUCTPI synchronization IP.

4.3 Latency optimization

This section describes the optimization work in the FPGA transceiver configuration in order to minimize the data transfer latency and also its variation, i.e., latency uncertainty.

4.3.1 Latency evaluation test system

Figure 4.2 shows the block diagram of the test system developed to measure the Xilinx GTX to Xilinx GTH and GTY transceiver latency and its uncertainty for different configurations. The GTX transceiver has been implemented using the Xilinx KC-705 FPGA evaluation kit [56], and the GTH and GTY transceivers have been implemented using the Xilinx VCU-108 evaluation kit [57].

In this test system, both transmitter and receiver use the same reference clock, which is generated by the Silicon Labs Si5338 jitter cleaner and clock generator evaluation board [58]. Tx and RX operate in the bit rate of 6.4 Gb/s with their user clock interfaces running in 320 MHz to minimize the latency in the transceiver Physical Coding Sublayer (PCS) [30].

For the latency measurement, the transmitter sends a periodic sequence to the receiver, and a pulse, so-called *TriggerPulse*, is asserted in both ends every time this sequence is repeated. The TriggerPulse output from TX and RX have been connected to a 1 GHz analog bandwidth oscilloscope [29] using cables of the same length. The transceiver-to-transceiver data transfer latency is given by measuring the time offset between the pulses at transmitter and receiver sides. The time for asserting the TriggerPulse and the delay from the cables are not relevant because they are canceled in the latency computation.

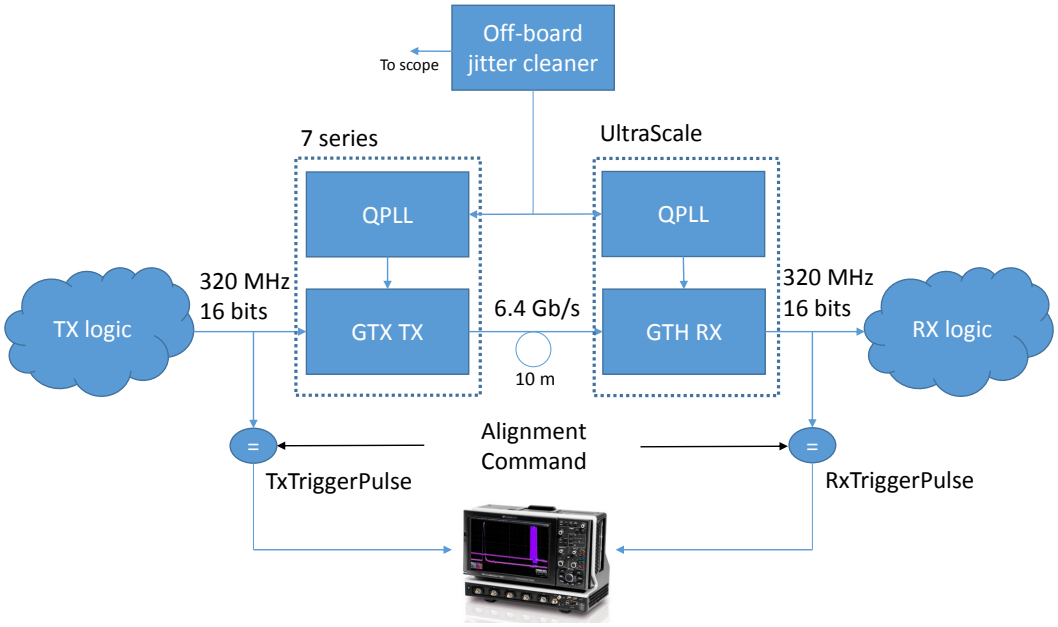


Figure 4.2 – Latency measurement test system block diagram

4.3.2 Data path latency test results

Several transceiver settings have been investigated in view of minimizing the latency. Bypassing the TX Phase Adjust First In First Out (FIFO) and the RX Elastic Buffer in the transceiver PCS reduced the data-path latency from ≈ 67 ns down to ≈ 50 ns. The configuration with FIFO and the buffer is used in most of the transceiver applications because it eases the crossing of PMA parallel clock domain to PCS user interface clock and vice-versa. But this simplification in the clock domain crossing is achieved at the price of increased latency, which is not desired for the MUCTPI application.

Figure 4.3, which has been adapted from [30], illustrates the optimized data path interconnection configuration for the transmitter part of the transceiver. The data flow from right to left, starting from the TX user interface. Then the path through the 8b10 is selected, and next, the Phase Adjust FIFO is bypassed as it contributes significantly to the latency. Then, the polarity is inverted in case the differential pair polarity in the PCB has been inverted. A single path is available in the PMA, the data are serialized in the Parallel In Serial Out (PISO) block, pre/post emphasis is applied if required, and the data are connected to the output pins through the TX driver. Blocks in Figures 4.3 to 4.5 and 4.9 not described in the text are out of scope of this Ph.D. thesis. Detailed information of the transceiver primitives is available in the transceiver user guide [30].

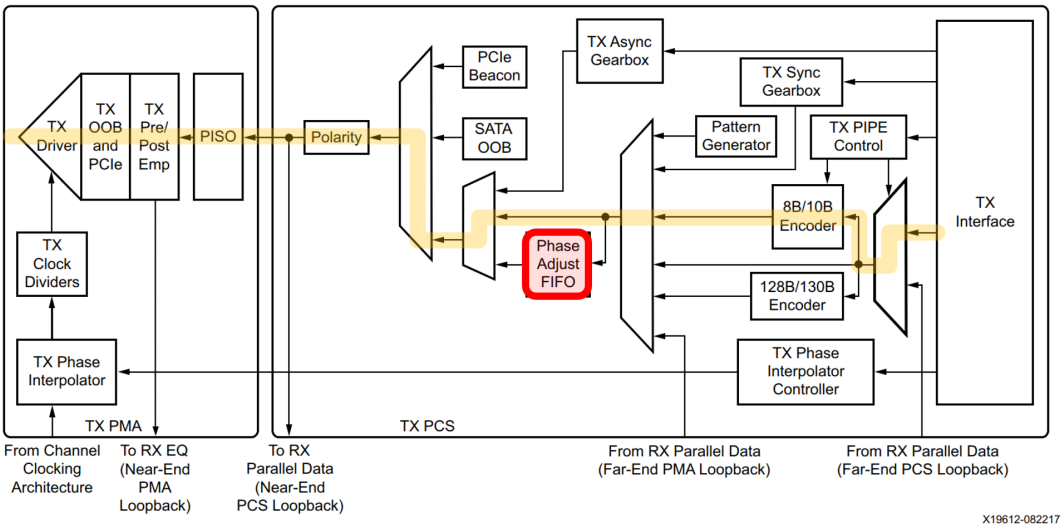


Figure 4.3 – GTY TX latency-optimized data path

The latency-optimized transmitter data path configuration found by the author of this thesis has also been used by RPC and TGC trigger colleagues to design their respective sector logic interfaces.

Figure 4.4, which has been adapted from [30], illustrates the optimized data path interconnection configuration for the receiver part of the transceiver. The data flows from the left to the right, starting in the input driver where the channel equalization is performed. The data are then deserialized in the Serial In Parallel Out (SIPO) block. In the PCS, the data are connected to the Comma Detect and Align block that detects the 8b10 alignment command in order to align the input data to an 8b10b 20-bit word boundary. After the data are aligned, the word is decoded to a word of 16 bits, and the data are connected to the user interface bypassing the RX Elastic Buffer in view of minimizing the latency.

4.3.3 Clock fabric latency uncertainty test results

The test system described in Section 4.3.1 has been used to measure the latency uncertainty. The latency uncertainty has been quantified by measuring the receiver TriggerPulse skew when triggering the scope with the transmitter TriggerPulse. Figure 4.5 shows the receiver flag skew for the default transceiver configuration at which the TX Phase Adjust FIFO and the RX Elastic Buffer in the transceiver PCS are not bypassed. The transceiver reset is asserted every 3s, and thousands of waveforms are captured. The region indicated by the green arrows corresponds to the actual latency uncertainty, while the region in red corresponds to the clock

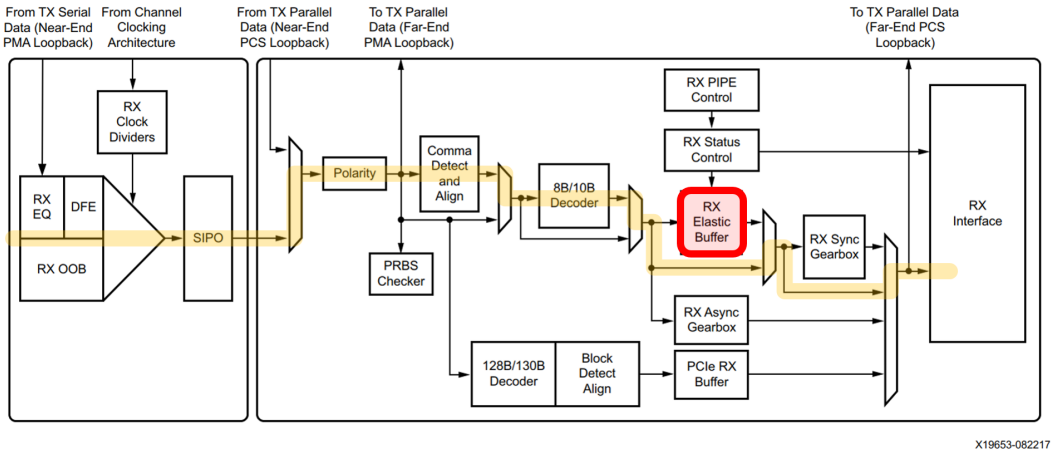


Figure 4.4 – GTY RX latency-optimized data path

period. Therefore the latency uncertainty here is equivalent to two user interface clock periods, i.e., 6.25 ns.

Figures 4.6 and 4.7 shows the latency variation measurement of the receiver flag to the transmitter flag after bypassing the TX Phase Adjust FIFO and the RX Elastic Buffer. The curve C1 corresponds to the transceiver reference clock (320MHz), C3, and C4 to the transmitter and receiver TriggerPulse flags, respectively. The scope is triggering at the transmitter TriggerPulse (C3). The first waveform is measured when the transmitter interface clock is generated by a programmable clock divider in the TX PMA and the second using a direct connection from the reference clock in the clock fabric.

The latency variation has been reduced from 6.250 ns to 3.125 ns in both plots. This corresponds to half of the latency uncertainty measured for the default transceiver configuration. However, only in the second plot, the reference clock has a constant phase to the transmitter TriggerPulse that is triggering the scope. This means that only in the second case the transmitter TriggerPulse, and therefore the transmitter user interface clock, has a fixed phase relationship to the reference clock, which in the detector is derived from the bunch crossing clock. Therefore the transmitter latency is fixed if the transmitter reference clock frequency is set to the same frequency of the transmitter user interface clock. This allows to drive the transmitter user interface clock directly from the reference clock avoiding the TX PMA programmable divider. This divider inserts latency uncertainty because its reset pulse is not synchronized to the transmitter reference clock.

Figure 4.8 shows the transmitter clock fabric configuration that ensures fixed latency on the transmitter side. The clock flows from bottom-left to center-right of the picture. It starts from

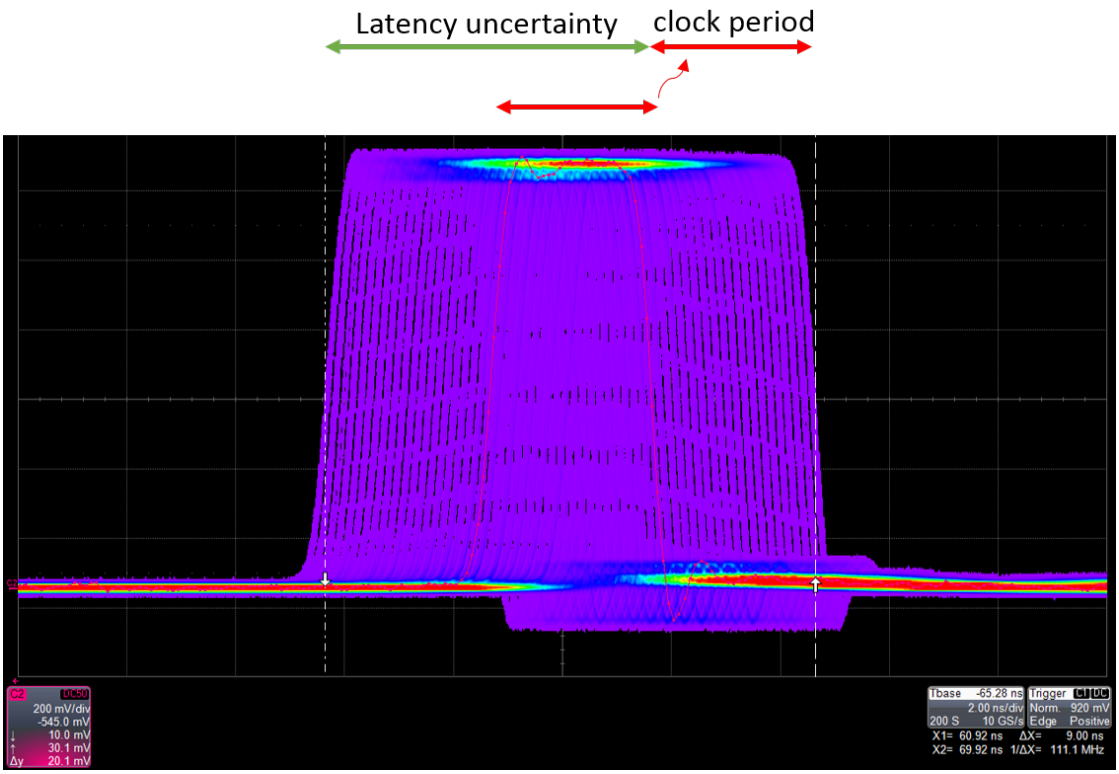


Figure 4.5 – Latency uncertainty measurement before optimization in the clock fabric

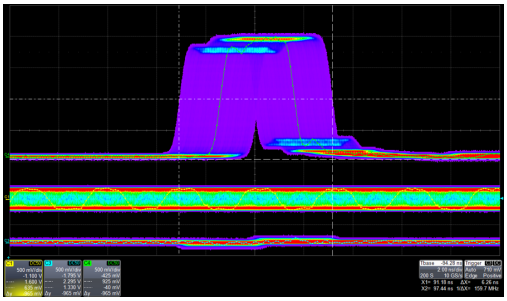


Figure 4.6 – Latency variation when TX-OUTCLK = TXPROGDIVCLK

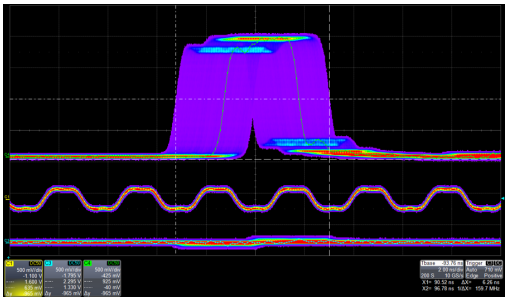


Figure 4.7 – Latency variation when TX-OUTCLK = TXPLLREFCLK_DIV1

the reference clock input buffer and connects to the Delay Aligner through the reference clock distribution block and two multiplexers. All the dividers are avoided using this path. The Delay Aligner block adjusts the phase difference between the PMA parallel clock domain and the transmitter user interface clock domain when the TX buffer is bypassed. After the clock phase is adjusted, it is connected to the transmitter user interface through the last multiplexer.

The latency-optimized transmitter clock fabric configuration found by the author of this thesis has also been used by RPC and TGC trigger colleagues to design their respective sector logic interfaces.

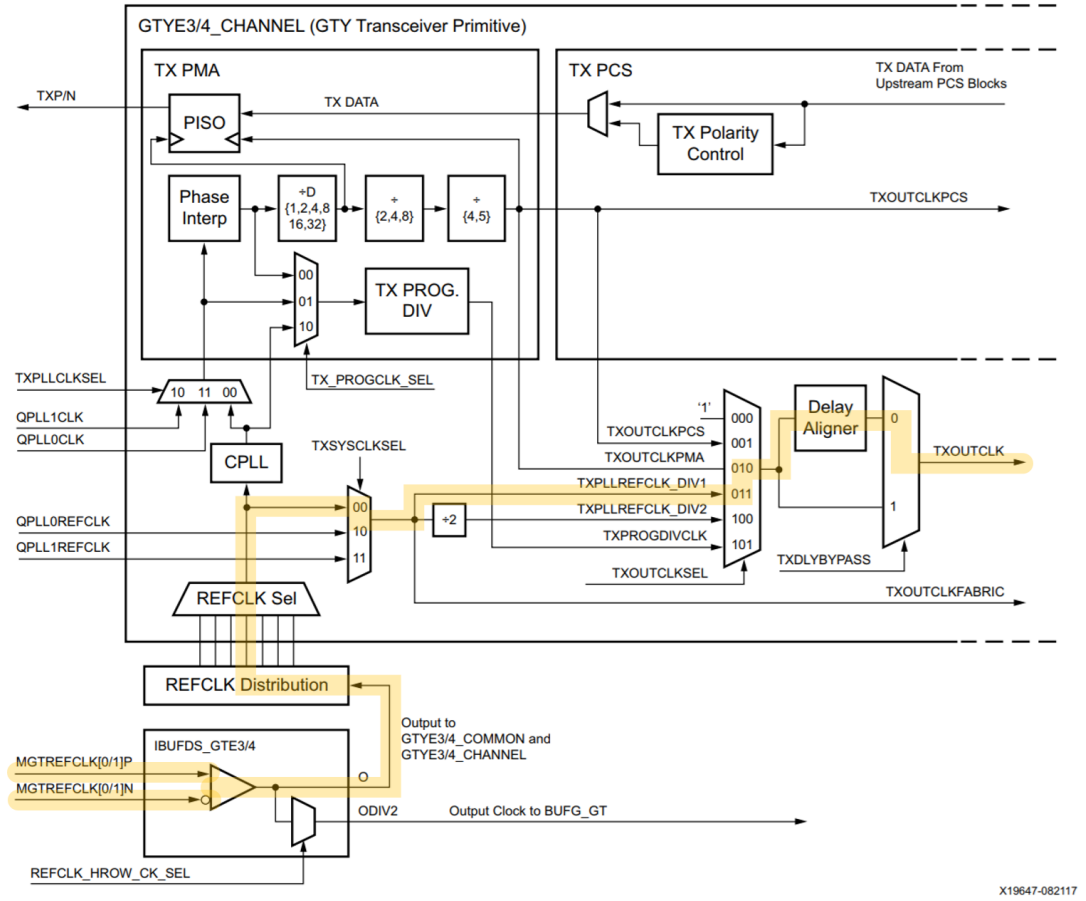


Figure 4.8 – Latency-fixed transmitter clock fabric configuration

Figure 4.9 shows the receiver clock fabric configuration in order to have the latency uncertainty reduced to one user interface clock period, i.e., 3.125 ns in 6.4 Gb/s. The clock flows from top-left to center-right of the picture. The clock is recovered from the data and is divided down with the same ratio while the data are parallelized. The latency uncertainty comes from the clock dividers in the RX PMA block, in which the reset assertion time has no fixed phase relationship to the received data word. Therefore, every time the transceiver is initialized, the clock dividers start in a different state or phase with respect to the input data word. The PMA clock dividers can not be avoided because the reference clock can not be used to drive the user clock interface at the receiver. This is not possible because the phase relationship between the recovered and the reference clocks is unknown as the transmitter clock is not sent along with

the data. After passing through the PMA clock dividers, the clock reaches the Delay Aligner block. This block adjusts the phase difference between the PMA parallel clock domain and the receiver user interface clock domain when the Rx Elastic Buffer is bypassed. Finally, the clock is connected to the receiver user interface through a multiplexer.

The latency uncertainty in the receiver can also be eliminated by performing the word alignment outside the core. As the CDR outputs the recovered clock with the non-deterministic phase after each initialization, the received data are then shifted following the current recovered clock phase. Therefore, for having fixed latency, one needs to align the received data by phase-shifting the recovered clock until a given known aligned data are received. The Xilinx GTH/GTY receiver has non-deterministic latency because the automatic word alignment provided by the transceiver shifts the data instead of phase-shifting the recovered clock. More details on this technique can be found at [59, 60]. Implementing the ideas in [59, 60] require to configure the transceiver comma alignment in the PMA manual mode. This option is not supported by the vendor, and it can only be used after hacking the vendor transceiver IP. As the latency uncertainty of 3.125 ns represents only 12.5% of the bunch clock period, we have decided not to implement this RX latency uncertainty mitigation technique for the MUCTPI SL synchronization. Such small latency uncertainty can be absorbed by the SL synchronization IP presented in Chapter 5.

4.4 Summary

This chapter described the basic concepts of an FPGA transceiver, the work for minimizing the data path latency, and mitigating the latency variation. After optimizing the transceiver configuration, the transceiver-to-transceiver latency has been reduced to $\approx$ 50 ns, and the latency uncertainty has been reduced to 3.125 ns.

The latency-optimized transmitter data path and clock fabric configurations found by the author of this thesis have also been adopted in the RPC and TGC sector logic interfaces.

Results in the total data-path latency are given in Chapter 5, which features tests that take into account the latency for transferring data from the receiver user interface clock domain to the bunch crossing clock domain, at which the data are processed. This additional clock domain crossing is imposed by the fact that the phase relationship between the recovered and system clocks is unknown.

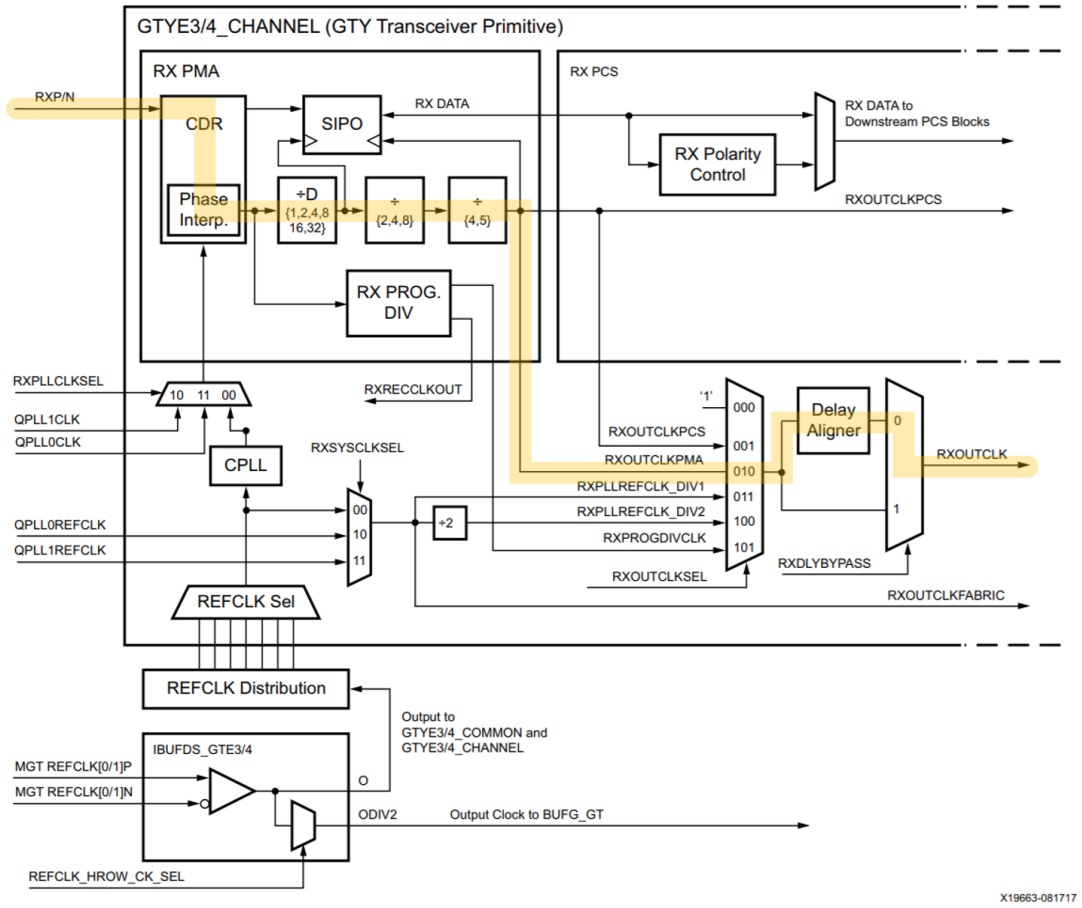


Figure 4.9 – Optimized receiver clock fabric configuration. Latency uncertainty reduced to one user interface clock period

5 Synchronization and Alignment

This chapter presents the development and testing of the synchronization IP. Section 5.1 introduces the concept of frame synchronization. Section 5.2 describes the RPC and TGC sector logic modules data frame formats. Section 5.3 presents the requirements of the synchronization IP. Section 5.4 presents the firmware development. Section 5.5 covers the functional simulation used to check the SL synchronization against errors, and to measure the maximum latency-uncertainty limits for error-free operation. Section 5.6 presents the integration test results with the RPC and TGC sector logic modules. Section 5.7 provides a summary of this chapter.

5.1 Introduction

Figure 5.1 shows the block diagram of the FPGA-based system-synchronous high-speed data transfer scheme from the SL module to the MUCTPI. The SL module sends, and the MUCTPI receives data synchronously to the TTC system clock, which is distributed separated from the data.

The data frame containing trigger information of a given event is generated in the SL using the TTC system clock with period $T_{BC} \approx 25$ ns. One hundred sixty bits are sent every bunch crossing. However, the number of bits allocated to the data frame is reduced to 128 bits, i.e., eight 16-bit words because 8b10b encoding is used.

The transmitter reference clock is generated from the system clock after a multiplication factor of 8, resulting in a 320 MHz clock. The clock multiplication is performed using an on-board jitter cleaner. The transceiver input data interface forwards a copy of the transmitter reference clock with a different phase, so-called transmitter user clock, to be used by the logic driving the transceiver data input in the FPGA user logic. Therefore, a synchronizer is needed to transfer the trigger data frame from the system clock to the transmitter user clock domain.

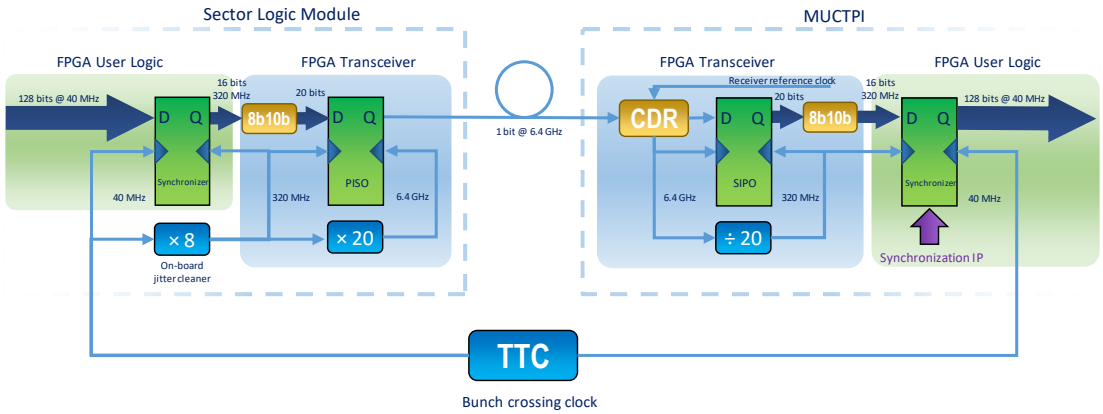


Figure 5.1 – Block diagram of a FPGA-based high-speed data transfer scheme

This synchronizer in the SL module is out of the scope of this Ph.D. work and, therefore, is not described here.

After the transmitter data are synchronized, the data are serialized and transmitted by the FPGA transceiver in the SL module and received and deserialized by the transceiver in the MUCTPI. The transceiver at the MUCTPI outputs 16-bit words synchronously to the clock recovered from the input data, so-called recovered clock with period T_{rec}, where $T_{rec} = 20\times \text{ UI} = 20 \times \frac{1}{6.4\text{ GHz}} = 3.125$ ns. Although transmitters and receivers use the same reference clock, the phase offset of each of the SL inputs is unknown and therefore has to be extracted from the clock embedded in the received data. Finally, the synchronization IP, which is the focus of this chapter, transfers the data, for each input, from the recovered clock to the system clock domain for combined data processing.

5.2 Data frame format

Tables 5.1 and 5.2 show the format of the 128-bit data frame sent from the RPC and TGC sub-detectors, respectively. The RPC and TGC sub-detectors send information from up to 2 and 4 muon candidates with the highest p_T threshold, respectively, per bunch crossing. The RPC candidate information consists of the RoI position number represented in 5 bits, the p_T threshold in 3 bits, and candidate flags in 4 bits. The TGC candidate information consists of the RoI position number represented in 8 bits, the p_T threshold in 4 bits, and candidate flags also represented in 4 bits. If there is no valid candidate to be sent, a predefined p_T threshold value is used to indicate that no valid candidate has been detected.

Next, global flags and Bunch Crossing Identifier (BCID) represented in 4 and 12 bits, respectively, are sent. The BCID is used to identify from which bunch crossing the current frame

Table 5.1 – RPC SL Data Format

Word	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Muon Candidate 1 (CAD1)															
1	Muon Candidate 2 (CAD2)															
2	Global flags								BCID							
3	CRC-8															
4	0xC5 (D5.6)															
5	0xC5 (D5.6)															
6	0xC5 (D5.6)															
7	0xC5 (D5.6)															

0xFD (K29.7)
0xBC (K28.5)
0xC5 (D5.6)
0xBC (K28.5)
0xC5 (D5.6)

Muon Candidate Format

Flags	0	pT	0	RoI

Observations:

1) 8b10b encoding is enabled
2) 16-bit word 0 is sent first
3) LSB bit is sent first (default for Xilinx)

	K character enabled
	K character disabled

Table 5.2 – TGC SL Data Format

Word	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Muon Candidate 1 (CAD1)															
1	Muon Candidate 2 (CAD2)															
2	Muon Candidate 3 (CAD3)															
3	Muon Candidate 4 (CAD4)															
4	Global flags								BCID							
5	CRC-8															
6	0xC5 (D5.6)															
7	0xC5 (D5.6)															

0xFD (K29.7)
0xBC (K28.5)
0xC5 (D5.6)

Muon Candidate Format

Flags	pT	RoI

Observations:

1) 8b10b encoding is enabled
2) 16-bit word 0 is sent first
3) LSB bit is sent first (default for Xilinx)

	K character enabled
	K character disabled

has been generated. Later, a CRC-8 code is computed from the muon candidate information, global flags, and BCID. Finally, the CRC-8 code is sent together with the K29.7 control symbol to indicate that the portion of the data frame containing trigger information is finished.

The data format is then padded with D5.6 data symbol and with K28.5 comma symbol. The latter one is used by the transceiver to align the input serial bitstream to a 20-bit boundary containing two 8b10b symbols. The K28.5 comma symbol is selected because it contains a bit sequence that can not be found elsewhere in the data stream. In order to enable the transceiver to operate with a 32-bit interface[1], if needed in the future, the K28.5 symbol is not repeated within a window of 40 bits (32 bits in the data format). If the K28.5 symbol is repeated within a window of 40 bits, the transceiver will align the input 32-bit word in different positions. Therefore, a data shifting mechanism with knowledge of the data format would be required.

5.3 Requirements

The synchronization IP should address the two following issues with a low and fixed latency.

- **Unknown phase offset:** The phase offset for each of the 208 MUCTPI sector data inputs is different due to the length mismatch among the clock and data optical fibers, as well as the part-to-part skew of each of the sector logic module components. As the sector logic modules connect to two types of muon detectors, which are also located in different parts of the ATLAS detector, data from a given collision will be propagated through the front-end and back-end electronics with different delays. Therefore, the phase-offset from the system clock with period $T_{sys} = T_{BC}$ to the recovered clock, defined here as Φ_{rec} is composed of the two following components. Figure 5.2 shows a timing diagram with their definition.
  - Φ^{s}_{rec} represents the phase offset from the first system clock rising edge to the beginning of the first complete frame. As each frame lasts T_{BC}, the lower and upper bounds for Φ^{s}_{rec} are $0 \leq \Phi^{s}_{rec} < T_{BC}$. This phase offset compensation is defined here as synchronization.
  - Φ^{a}_{rec} represents the phase offset from the beginning of the first complete frame to the beginning of the frame of interest, i.e., the frame corresponding to the bunch crossing of interest for combined data processing. In Figure 5.2, the first complete frame contains data from BCID N-1, and the frame of interest contains data from BCID N. $\Phi^{a}_{rec} = k \times T_{BC}$, where $k \in \mathbb{Z}^{\geq}$, i.e., k is a non-negative integer. This phase offset compensation is defined here as alignment.

[1] 16-bit interface is used today

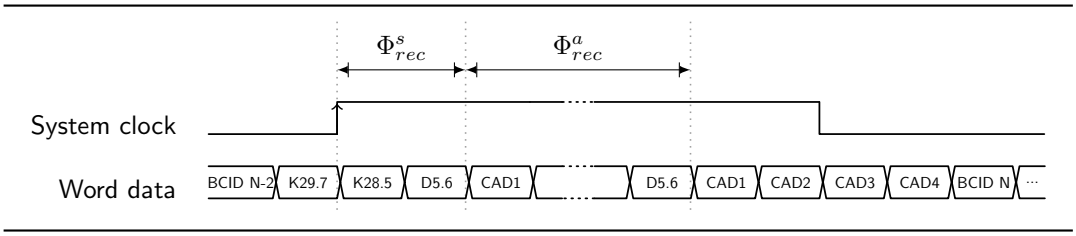


Figure 5.2 – Timing diagram with Φ^s_{rec} and Φ^a_{rec} definition

- **Latency Uncertainty:** The data transfer from the transceiver to transceiver after resetting both transmitter and receiver has a latency uncertainty of T_{rec}. This latency uncertainty comes only from the MUCTPI receiver that recovers the clock from the data with a latency uncertainty of T_{rec}. The SL transmitter uses the data-path and clock fabric configuration, presented in Chapter 4, that enables latency-deterministic operation. As the receiver latency is unknown for a given initialization, it is not possible to know if the phase offset variation after resetting the receiver is positive or negative compared to the latency before the reset. In addition, it is also not possible to know the absolute value of the phase variation. However, it is possible to define the following lower and upper bounds $V_L T_{rec} \leq \Delta\Phi_{mgt} \leq V_R T_{rec}$, where $\Delta\Phi_{mgt}$ represents the variation of the phase offset after resetting the receiver, and V_L and V_R are the number of T_{rec} periods which the latency can variate from Φ^{ref}_{rec} to the left and right, respectively. As the link from the sector logic module to the MUCTPI has latency variation of T_{rec}, the upper and lower limits of $\Delta\Phi_{mgt}$ are defined with $V_L = -1$ and $V_R = 1$.

Equations (5.1) to (5.5) summarizes all the effects on the phase offset of the recovered clock that has to be addressed by the synchronization IP.

$$\Phi_{rec} = \Phi^s_{rec} + \Phi^a_{rec} + \Delta\Phi_{mgt}, \tag{5.1}$$

where

$$0 \leq \Phi^s_{rec} < T_{BC}, \tag{5.2}$$

$$\Phi^a_{rec} = k \times T_{BC}, \tag{5.3}$$

$$k \in \mathbb{Z}^{\geq}, \tag{5.4}$$

$$V_L T_{rec} \leq \Delta\Phi_{mgt} \leq V_R T_{rec}. \tag{5.5}$$

Note that Φ^{s}_{rec} and Φ^{a}_{rec} are constant and will not change unless the cabling is altered. $\Delta\Phi_{mgt}$ changes only when the receiver recovers from a reset but will remain constant until the receiver is reinitialized again. Note that the phase variation from clock jitter is ignored here because the jitter is very low compared to $\Delta\Phi_{mgt}$. The clock jitter is very low thanks to the use of jitter cleaners in the MGT reference clocks in the SL and MUCTPI.

5.4 Firmware

Figure 5.3 shows the block diagram of the MUCTPI synchronization IP, which transfers the SL data, for each input, from the recovered clock to the system clock domain for combined data processing. The firmware is designed to absorb the latency uncertainty from the receiver and output the SL data with a fixed latency. It only requires a one-time calibration to accommodate the different delays from the SL optical fibers, as well as the part-to-part skew of each of the sector logic module components. The design is based on the use of dual-port memories that enable writing the input data using the recovered clock and reading the output data using the system clock. Two and four dual-port memories with a length of 32 16-bit words store muon candidate data for RPC and TGC inputs, respectively. Two other memories store the global flags, BCID, and the CRC-8 code.

The Write control block drives a common write address pointer to all the memories, and a dedicated write enable flag for each of them. These signals are generated based on the transceiver data and control character input, the alignment pulse, and the alignment pulse delay select. The alignment pulse is used by both write and read control blocks to make sure that write and read pointer are synchronized. This pulse acts as an active-low reset.

For the write side of the memory, the alignment pulse has to be transferred from the system clock domain to the recovered clock domain. This clock domain transfer is implemented using a two-stage bit synchronizer represented with a black box in the top left side of the firmware block diagram. This bit synchronizer is implemented using two registers in a chain clocked by the recovered clock, as described in [61]. A placing constraint is used to ensure that both registers are placed in the same FPGA slice in view of maximizing the Mean Time Between Failure (MTBF) [62]. The propagation delay uncertainty from the first register of the bit synchronizer operating in a metastable state is described in Section 5.5.5.

After the alignment pulse is asserted, the write pointer increment enable is asserted at the last word of the frame, i.e., 3 or 5 clock cycles after the end-of-frame (K29.7) control character is detected, for RPC and TGC inputs, respectively. The write address pointer starts to increment always from 0. The memory write enable is asserted according to the data format described in Section 5.2. The alignment pulse delay select adjusts the delay added to the alignment pulse for the write control block only. This is needed to ensure no latency variation in writing

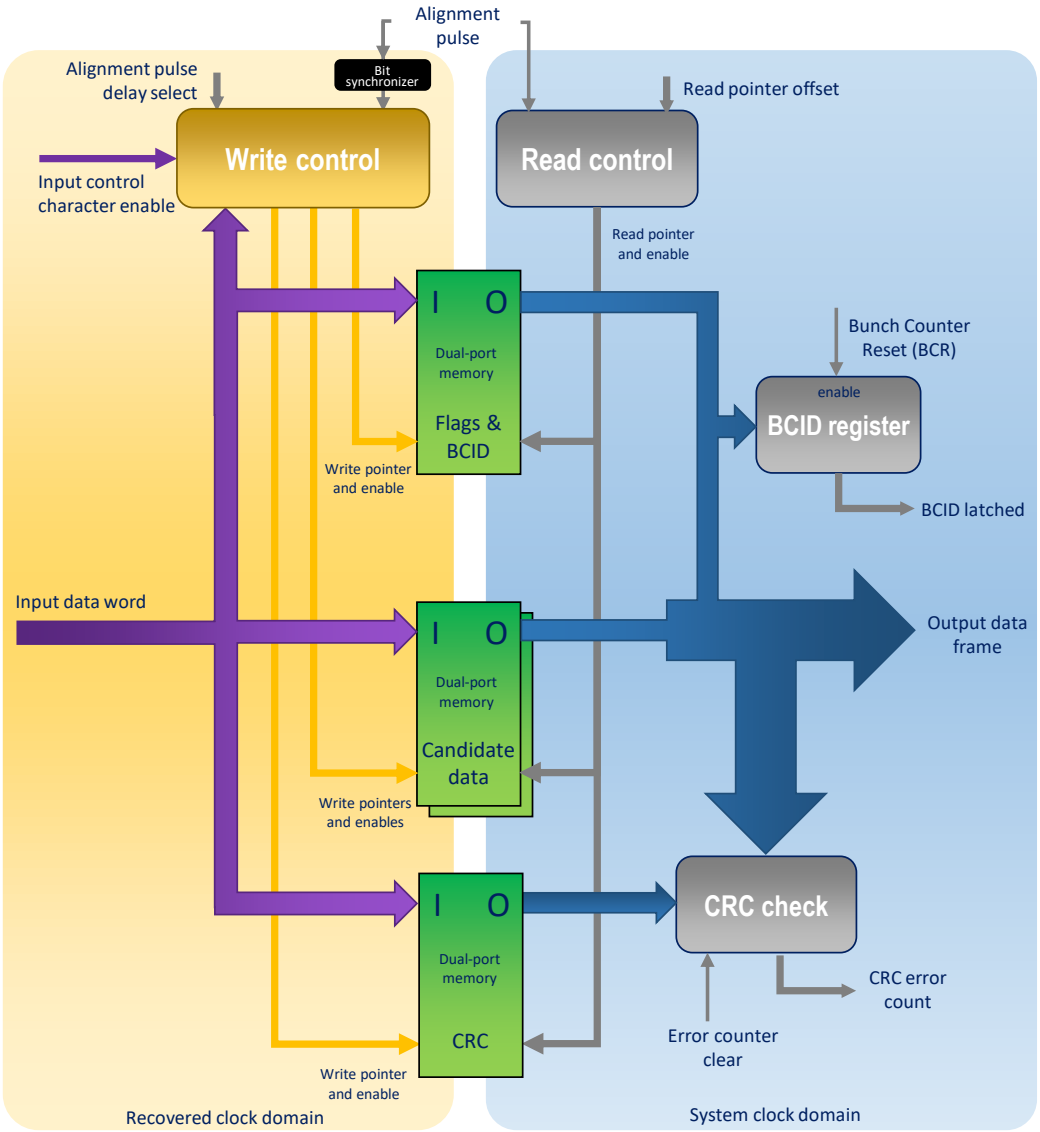


Figure 5.3 – MUCTPI synchronization block diagram

data to the dual-port memory. The working principle is to prevent the condition at which the alignment pulse is asserted at the same time, the last word of the frame is detected. If, for a given transceiver initialization, the alignment pulse is asserted just before the border between two frames, the write pointer is incremented immediately. However, if the latency is increased in a second initialization, the last word from the same frame is missed, and the write pointer is incremented only after receiving the last word of the next frame. Shifting the alignment pulse away from the border between two frames absorbs the latency uncertainty from the

receiver and bit synchronizer. More information on this function is described in Sections 5.5.4 and 5.5.6.

The Read control block drives a shared read address pointer to all memories based on the alignment pulse, and a configurable read pointer offset. After the alignment pulse is asserted, the read pointer offset is loaded to the read pointer counter, which is incremented at every rising edge of the system clock.

The BCID register and CRC check blocks are used to check if a given frame of data is corrupted and/or misaligned. The BCID register loads the BCID value from the output frame every time the Bunch Counter Reset (BCR) is received, i.e., at the beginning of every orbit. The CRC unit computes the CRC-8 code from the event data and compares it against the received CRC-8 code in order to detect CRC errors.

Data corruption can happen even if the input stream is error-free. This effect is seen if the write and read pointer values are overlapping in time in such a way that the dual-port memories output portions of two different frames. In other words, the memory outputs part of the data coming from an earlier bunch crossing while the other portion of the output frame comes from a later bunch-crossing.

A misaligned frame can happen if the received frame corresponds to a different bunch crossing of the one that is expected. For example, both blocks are used to find the read address pointer offset, which outputs the latest error-free frame written to the memory. This procedure is described in Section 5.5.7.

Optional input and output registers, not shown in the block diagram, are instantiated to ease placing and routing. They increment the synchronization latency by one T_{rec} and one T_{sys}, respectively. Section 5.5.9 describes the synchronization latency in more detail.

In order to minimize processing time, the system clock domain can run in an integer multiple frequencies of 40MHz, having a enable flag asserted every 25 ns. As a matter of the fact, the synchronization IP at the MUCTPI runs at 160MHz (T_{sys} = 6.25 ns) with an enable flag asserted every 4 clock cycles, i.e., every 25 ns.

5.5 Functional simulation

This section describes the design of a comprehensive functional simulation of the synchronization IP. This functional simulation has been designed to check the synchronization block for design errors, to elaborate the synchronization calibration procedure, to find the minimum and maximum latency read pointer offsets, and to obtain the simulated values for the synchronization latency. Some of the simulations in this section account for an increased phase

variation space, i.e., beyond the latency variation measured for the SL data transfer, in order to quantify the latency-uncertainty margin that the MUCTPI can tolerate ensuring error-free operation. The following sections describe how the simulations have been implemented and the achieved results.

5.5.1 Work environment

All the simulations presented in this section use the Mentor Modelsim [63] simulator. The functional testbench is written in Python 3.7 [64] using Cocotb [65] to apply stimulus and read simulation results from Modelsim. Scientific libraries such as Pandas [66], NumPy [67] and Matplotlib [68] are used to manipulate data and plot results.

5.5.2 Unit test

Figure 5.4 shows the unit test block diagram. The TTC, SL, and Control and Data Analysis python coroutines are shown on the left and right sides of the picture. The synchronization IP, i.e., the Device Under Test (DUT), is placed in the center. The TTC coroutine drives the system clock, clock enable flag, and BCR. The SL coroutine drives the transceiver recovered clock, data, and control character symbols according to one of the data formats described in Section 5.2. The TTC and SL coroutines are started together in order to run in parallel. The TTC coroutine starts immediately, and the SL coroutine start after a configurable phase offset Φ_{rec}. The recovered clock phase offset is defined according to Equation (5.1) in increment steps of UI = $\frac{1}{6.4\ \text{GHz}}$ = 0.15625 ns. For example, if and only if $\Phi_{rec} = 0$, the two following conditions are true:

1. Rising edges of the transceiver recovered clock, and the system clock are aligned
2. First SL data word is sent at the same time as the system clock enable flag

The Control and Data Analysis coroutine drives the alignment pulse, alignment delay, read pointer offset, and error counters clear while reading out the BCID latched and CRC error count values. Procedure 5.5.1 describes, in more detail, the steps executed in the unit test. Steps from 2 to 8 are executed by the Control and Data Analysis coroutine.

5.5.3 Reference and running phase offset test

Section 5.3 describes the unknown phase offset and latency uncertainty issues that have to be addressed by the synchronization IP. It means that for an unknown phase offset, the synchronization block has to safely transfer the data to the system clock domain being able

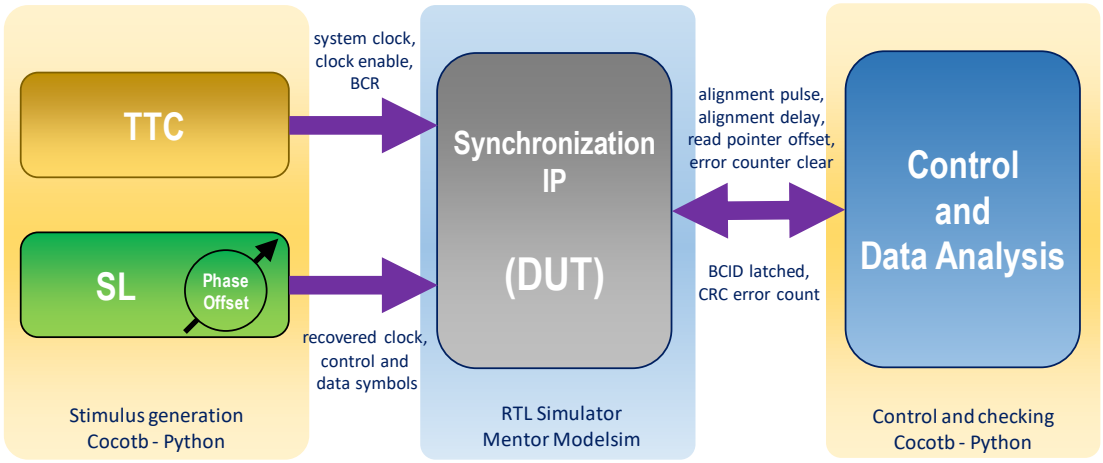


Figure 5.4 – Unit test block diagram

to tolerate small latency variations. In other words, for each of the different values of the recovered clock phase offset of reference, Φ_{rec}^{ref}, i.e., the phase offset at the moment the system is calibrated, the synchronization IP has to synchronize and align the input data for each of the values of the recovered clock phase offset of running, Φ_{rec}^{run}, i.e., all the phase offsets after the system has been calibrated.

The first set, Φ_{rec}^{ref}, is defined from Equation (5.1) with $\Phi_{rec}^{a} = 0$ because the testing of the alignment functionality is not addressed yet, and $\Delta\Phi_{mgt} = 0$ because the calibration procedure is executed once and therefore has no latency variation. The alignment functionality, i.e. having $\Phi_{rec}^{a} \neq 0$, is addressed in Section 5.5.7. The second set, Φ_{rec}^{run}, is created for each Φ_{rec}^{ref} and is defined from Equation (5.1) with $\Phi_{rec}^{a} = 0$ and a given upper and lower bounds for $\Delta\Phi_{mgt}$ depending on each test. For some tests, a larger phase variation space is used to investigate how the align delay and read pointer offset parameters can be optimized in view of increased margin of operation. For this cases, the latency uncertainty interval is defined using $V_L = -8$ and $V_R = 8$. Hence, Φ_{rec}^{ref} and Φ_{rec}^{run} are given by Equations (5.6) and (5.7).

$$\Phi_{rec}^{ref} \in \mathbb{R} \mid 0 \leq \Phi_{rec}^{ref} < T_{BC}. \tag{5.6}$$

$$\Phi_{rec}^{run} \in \mathbb{R} \mid \Phi_{rec}^{ref} - 8T_{rec} \leq \Phi_{rec}^{run} \leq \Phi_{rec}^{ref} + 8T_{rec}. \tag{5.7}$$

Figure 5.5 shows the two-dimensional color-coded visualization of the data set defined in Equations (5.6) and (5.7). The left and right y-axis represent the reference or calibration recovered clock phase offset in ns and UI, respectively. The x bottom and upper axis represent

Procedure 5.5.1 – Unit test coroutine steps

1. Start TTC and SL coroutines with the specified recovered clock phase offset Φ_{rec}, and sub-detector type, i.e., RPC or TGC.
2. Reset the synchronization IP. The transceiver reset stays untouched.
3. Set alignment delay and read address pointer offset.
4. Assert and deassert the alignment pulse signal to synchronize the dual-port memory write and read pointers.
5. Wait for 32 clock cycles to make sure the content of all dual-port memories has been overwritten. This is needed to make sure the memory output does not correspond to a previous test for any read pointer offset set in step 2.
6. Assert and deassert the CRC error counter clear to start counting errors from the time that the configuration has been finished.
7. Wait until a new BCR arrives. This is important to make sure the BCID register has been loaded with a new BCID value derived from the alignment delay and read pointer offset values defined in step 2. This step is implemented as follows: First, clearing a register that indicates a new BCR arrived. Second, polling this same register until it is asserted again. This register lives outside the synchronization IP and is shared for all the channels.
8. Read the BCID latched and CRC error counter values.
9. Terminate the TTC and SL coroutines

the running recovered clock phase offset in ns and UI, respectively. Φ_{rec}^{ref} and Φ_{rec}^{run} are color coded in light grey and black respectively. The pair of blue lines spaced by ±20 UI ($\pm T_{rec}$) from Φ_{rec}^{ref} represent the limits of the SL to MUCTPI latency uncertainty.

5.5.4 Latency variation effect in the memory write side

The reference and running phase offset test, described in Section 5.5.3, has been executed in order to study the effect, on the memory output data, of the phase offset between the alignment pulse to Φ_{rec}^{ref} and Φ_{rec}^{run}. For this reason, the alignment delay is set to 0, and the read pointer offset to 15. The read pointer offset has been set to 15, the middle of the memory capacity, to make sure write and read pointer values are never overlapping in time.

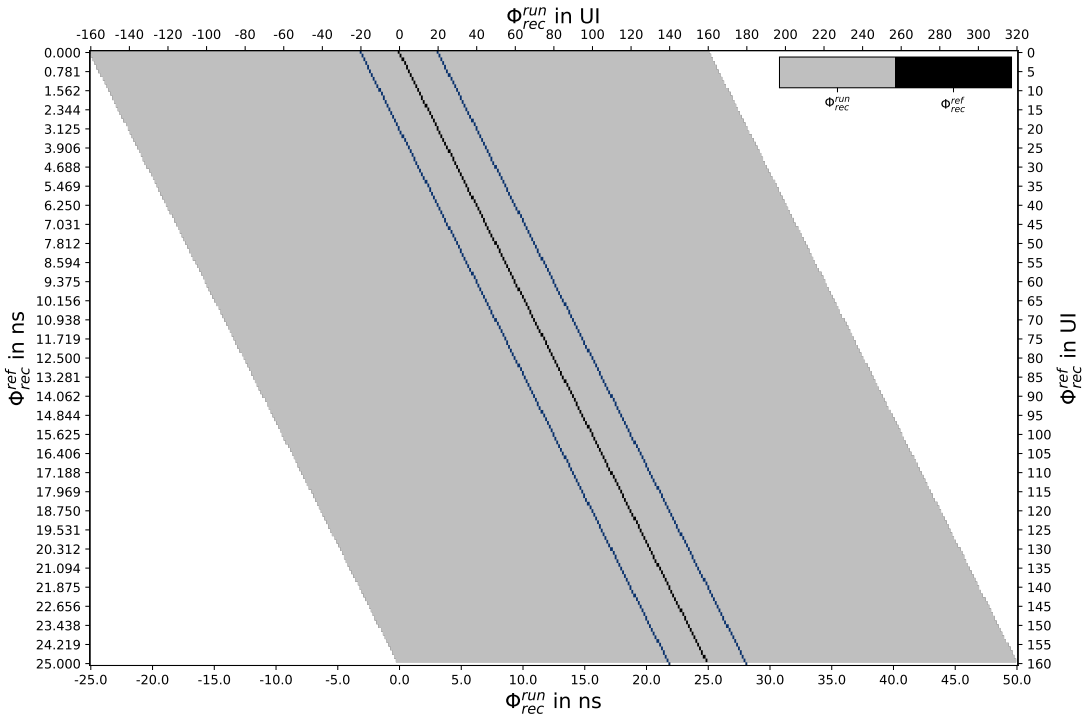


Figure 5.5 – Color-coded visualization of the reference and running phase offset dataset

Figure 5.6 shows the two-dimensional color-coded visualization of the BCID error value for each pair of Φ^{ref}_{rec} and respective Φ^{run}_{rec} values. A BCID error is detected when the BCID value read with a given Φ^{run}_{rec} is different from the one read with the reference phase offset Φ^{ref}_{rec}. The y and x-axis and the pair of blue lines are defined in the same way as in Figure 5.5. The light blue line in the center represent the Φ^{ref}_{rec} for each range of Φ^{run}_{rec} values. The error-free tests and the tests with BCID error are color-coded in grey and black, respectively.

The most important result to be extracted from Figure 5.6 is the presence of BCID errors for $\Phi^{run}_{rec} \in \mathbb{R} \mid \Phi^{ref}_{rec} - T_{rec} \leq \Phi^{run}_{rec} \leq \Phi^{ref}_{rec} + T_{rec}$, the transceiver latency uncertainty region, when $\Phi^{ref}_{rec} \in \mathbb{R} \mid 0 \leq \Phi^{ref}_{rec} < 40$ UI. It means that even a latency variation of -1 UI will cause error if $\Phi^{ref}_{rec} = 20$ UI. In a similar way, even a latency variation of 1 UI will cause error if $\Phi^{ref}_{rec} = 19$ UI.

In fact, taking into account the SL to MUCTPI latency variation, any input with $\Phi^{ref}_{rec} \in \mathbb{R} \mid 0 \leq \Phi^{ref}_{rec} < 40$ UI will present BCID error after resetting the transceiver multiple times. For $\Phi^{ref}_{rec} \in \mathbb{R} \mid 0 \leq \Phi^{ref}_{rec} < 20$ UI, BCID errors will occur in a subset of the clock phases later than Φ^{ref}_{rec}. For $\Phi^{ref}_{rec} \in \mathbb{R} \mid 20 \leq \Phi^{ref}_{rec} < 40$ UI, the opposite happens, BCID errors will occur in a subset of the clock phases earlier than Φ^{ref}_{rec}.

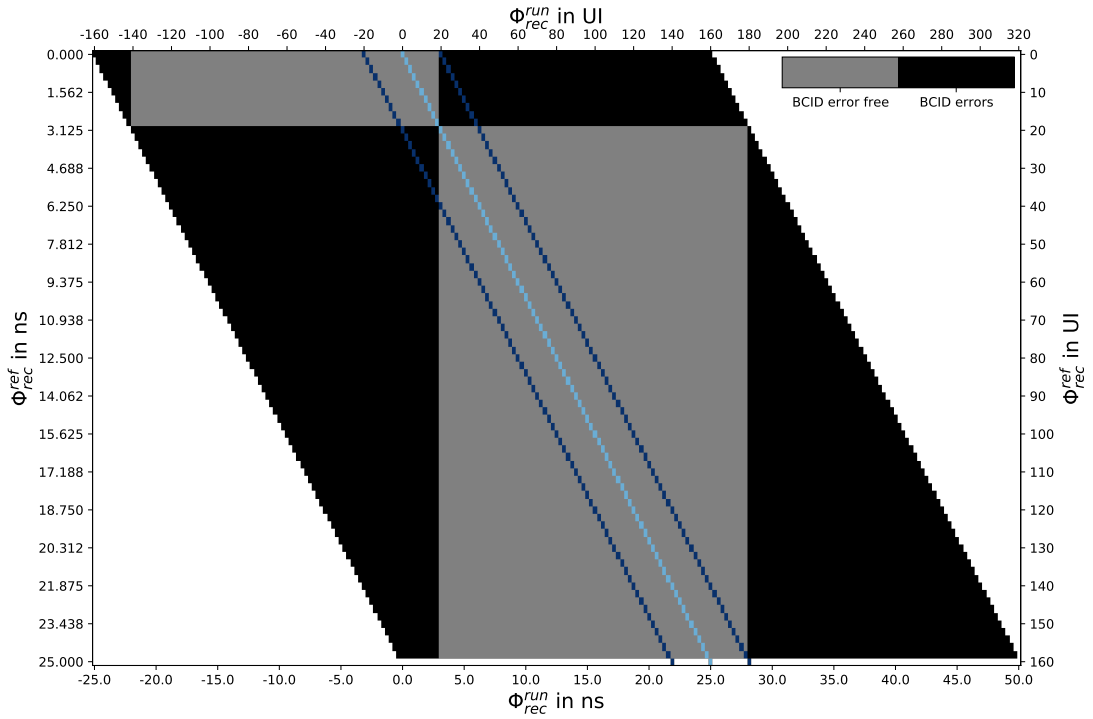


Figure 5.6 – Color-coded visualization of the BCID error for align delay set to 0

Figures 5.7 and 5.8 shows simplified waveforms showing the BCID error resulted from latency variation if the alignment pulse is asserted close to frame boundaries ($\Phi_{rec}^{ref} \in \mathbb{R} \mid 0 \leq \Phi_{rec}^{ref} <$ 40 UI). For both pictures, the alignment pulse is asserted in a valid system clock cycle, which corresponds to $\Phi_{rec} = 0$.

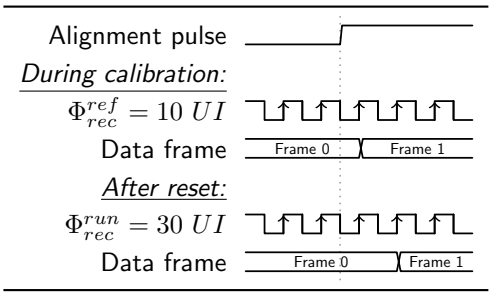


Figure 5.7 – Late Φ_{rec}^{run} waveform

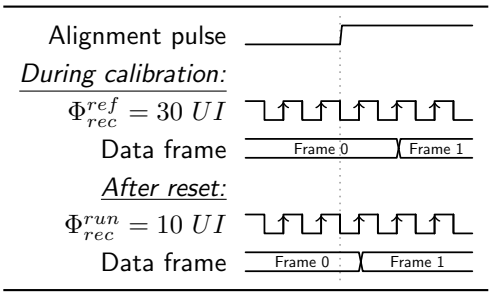


Figure 5.8 – Early Φ_{rec}^{run} waveform

Figure 5.7 illustrates the BCID error effect seen with Φ_{rec}^{ref} = 10 UI and Φ_{rec}^{run} = 30 UI, i.e., when the data frame arrives later than at the time of calibration. During calibration, i.e., Φ_{rec}^{ref} = 10 UI, the alignment pulse is asserted after the arrival of the last word of the frame, which sets the pointer increment enable, more detail in Section 5.4. Therefore, the write

pointer is incremented only at the end of frame 1. However, after reset, i.e., $\Phi_{rec}^{run} = 30$ UI, the alignment pulse is asserted earlier than the last word of frame 0, causing the increment of the write pointer already in frame 0.

Similarly, Figure 5.8 illustrates the BCID error effect seen with $\Phi_{rec}^{ref} = 30$ UI and $\Phi_{rec}^{run} = 10$ UI, i.e., when, after calibration, the data frame arrives earlier than the time it has been received during calibration. During calibration, the write pointer is already incremented in frame 0. However, after reset, the write pointer is incremented only at the end of frame 1.

Note that this issue comes exclusively from the latency variation effect in the write side of the memory. The read pointer offset is set to 15, middle of the memory capacity, which gives a very large slack to make sure the data are not read earlier or later than it should.

This latency variation of only 20 UI, in both cases, causes the BCID to be shifted by one bunch-crossing, which corresponds to a latency variation of $T_{BC} = 160$ UI in the read side of the memory. For the MUCTPI, this is an unacceptable effect that should be mitigated by the synchronization IP.

5.5.5 Metastability effect on the memory write side

Section 5.5.4 describes the effect of the transceiver data latency variation with respect to the write alignment pulse on the memory output data. In Section 5.5.4, the bit synchronizer connected to the alignment pulse is considered to have a fixed propagation delay.

Figure 5.9 shows the resulting data latency variation with respect to the write alignment pulse with and without metastability being taken into account in the bit synchronizer unit. The vertical grey dashed lines are separated by T_{rec}.

The top part of Figure 5.9 shows the alignment pulse with fixed propagation delay and the phase variation of the transceiver data defined with $\Delta\Phi_{mgt} \in \mathbb{R} \mid -T_{rec} \leq \Delta\Phi_{mgt} \leq T_{rec}$, the same way as in Section 5.5.4. The data represented with a dashed line corresponds to the length of the transceiver latency variation.

If the metastability effect in the bit synchronizer unit is taken into account, the respective propagation delay with respect to the functional simulation propagation delay can be shifted by $-T_{rec}$ and T_{rec}, if hold and setup timing is violated respectively [69].

Being more accurate, if hold violation occurs and the pulse metastability resolves to high, the bit synchronizer propagation delay is shifted by $-T_{rec}$ compared to the functional simulation propagation delay. However, if the pulse metastability resolves to low, the alignment pulse rising edge is sampled only in the next clock cycle, in the same way as in the functional simulation.

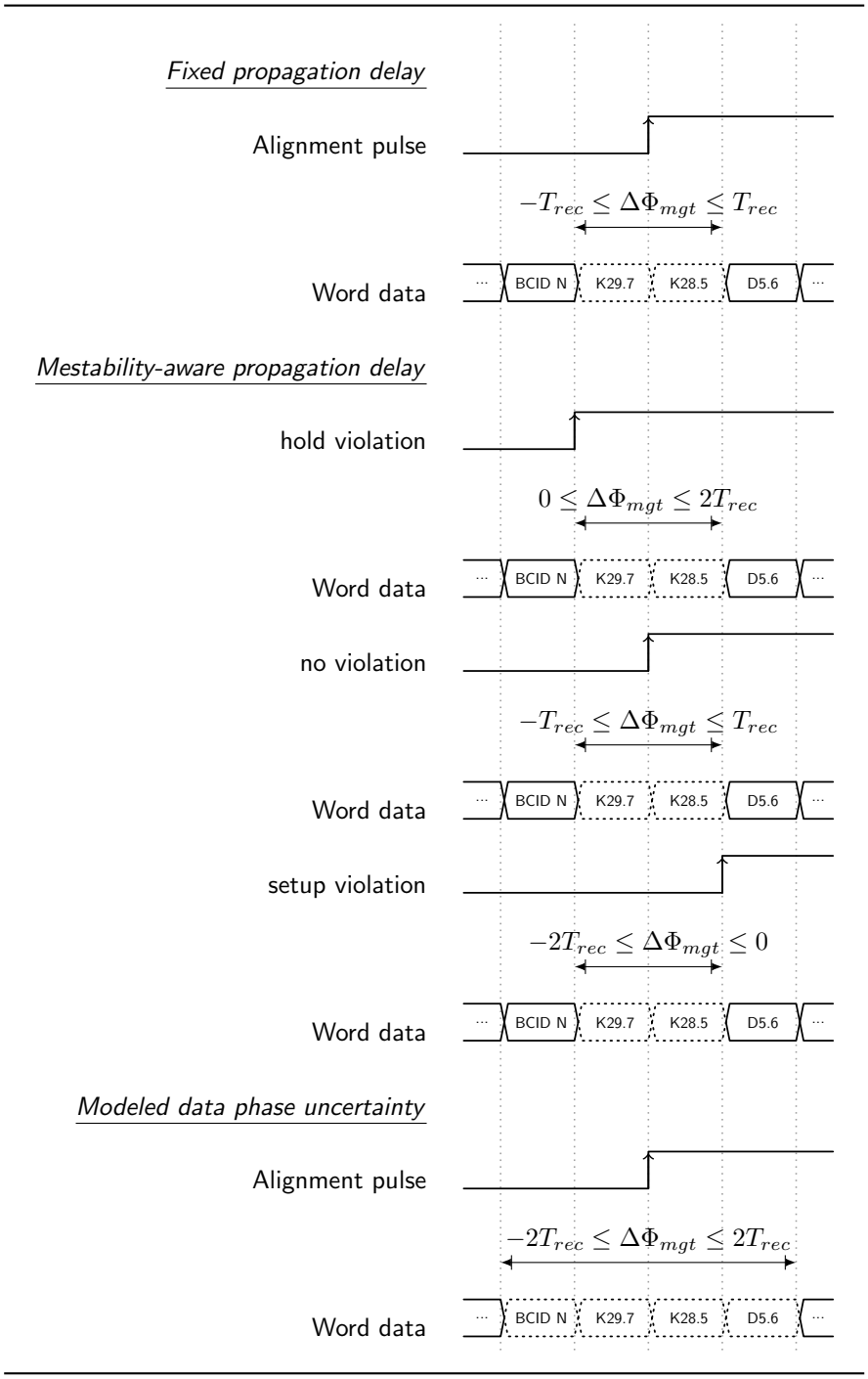


Figure 5.9 – Metastability effect on the write alignment pulse propagation delay

Similarly, if setup violation occurs and the pulse metastability resolves to low, the bit synchronizer propagation delay is shifted by T_{rec} with respect to the functional simulation propagation delay. Still, if the pulse metastability resolves to high, the pulse is successfully sampled with the same propagation delay as in the functional simulation.

For this study, it is interesting to look only for the cases that the propagation delay is altered with respect to the functional simulation because these cases are the ones that can affect the behavior of the synchronization IP. Therefore, in this section, hold, and setup timing violations abbreviate to the cases that the pulse metastability resolves to high and low, respectively, i.e., the cases in which the phase offset between the data and the alignment pulse is shifted by $-T_{rec}$ and T_{rec}.

The center of Figure 5.9 shows the three different propagation delays that are obtained for the alignment pulse, in the case hold or setup timing is violated compared to the so-called *no violation delay*. The no violation delay corresponds to the same propagation delay obtained when metastability does not occur or when the metastability effect is not taken into account in the functional simulation. In all of the three cases, the word data propagation delay is the same. Note that the bit synchronizer is connected only to the alignment pulse and not to the data. Although the data phase offset remains unaltered, the phase offset from the data to the alignment pulse is changed when hold or setup timing is violated. For each of the three cases, the phase offset interval with respect to the alignment pulse is shown. If hold timing is violated and the pulse metastability resolves to high, the data phase offset to the resulting alignment pulse is defined with $\Delta\Phi_{mgt} \in \mathbb{R} \mid 0 \leq \Delta\Phi_{mgt} \leq 2T_{rec}$. Similarly, if setup timing is violated and the pulse metastability resolves to low, the data phase offset is defined with $\Delta\Phi_{mgt} \in \mathbb{R} \mid -2T_{rec} \leq \Delta\Phi_{mgt} \leq 0$.

Limiting to the write side of the memory, the variation on the propagation delay of the bit synchronizer can be modeled as an additional phase variation on the received data. This is achieved by superimposing the phase offset from the data to the alignment pulse for the three different bit synchronizer propagation delays shown in the center of the figure. Equations (5.8) to (5.12) show the union of the three intervals.

$$\Delta\Phi^{M}_{mgt} = \Delta\Phi^{H}_{mgt} \cup \Delta\Phi^{N}_{mgt} \cup \Delta\Phi^{S}_{mgt}, \tag{5.8}$$

where

$$\Delta\Phi^{H}_{mgt} \in \mathbb{R} \mid 0 \leq \Delta\Phi^{H}_{mgt} \leq 2T_{rec}, \tag{5.9}$$

$$\Delta\Phi^{N}_{mgt} \in \mathbb{R} \mid -T_{rec} \leq \Delta\Phi^{N}_{mgt} \leq T_{rec}, \tag{5.10}$$

$$\Delta\Phi^{S}_{mgt} \in \mathbb{R} \mid -2T_{rec} \leq \Delta\Phi^{S}_{mgt} \leq 0, \tag{5.11}$$

resulting in

$$\Delta\Phi^{M}_{mgt} \in \mathbb{R} \mid -2T_{rec} \leq \Delta\Phi^{M}_{mgt} \leq 2T_{rec}. \tag{5.12}$$

$\Delta\Phi^{M}_{mgt}$ represents the modeled data phase offset interval. $\Delta\Phi^{H}_{mgt}$ and $\Delta\Phi^{S}_{mgt}$ represent the data phase offset interval when hold or setup timing is violated, respectively. $\Delta\Phi^{N}_{mgt}$ represents the data phase offset interval without metastability.

This approach eliminates the need to perform a metastability-aware functional simulation [69] to cover the different propagation delays of the bit synchronizer unit. The bottom part of Figure 5.9 shows the modeled data phase offset with respect to the no violation alignment pulse if hold or setup timing is violated. The data represented with a dashed line corresponds to the length of the latency variation being considered when operating the write side of the memory. Therefore, only for the write side of the memory, the Φ^{run}_{rec} interval that BCID errors are not accepted, is increased to $\Phi^{run}_{rec} \in \mathbb{R} \mid \Phi^{ref}_{rec} - 2T_{rec} \leq \Phi^{run}_{rec} \leq \Phi^{ref}_{rec} + 2T_{rec}$.

Note that the bit synchronizer propagation delay uncertainty affects only the write side of the memory because there is no bit synchronizer in the read alignment pulse. There is no need to have a bit synchronizer for the read alignment pulse because no data are transferred through different clock domains. Therefore, the required latency variation tolerance in the read operation of the memory remains $\Phi^{run}_{rec} \in \mathbb{R} \mid \Phi^{ref}_{rec} - T_{rec} \leq \Phi^{run}_{rec} \leq \Phi^{ref}_{rec} + T_{rec}$.

5.5.6 Addressing latency variation in the memory write side

The BCID error effect described in Section 5.5.4 is avoided by adjusting the phase offset from the frame boundary to the alignment pulse. Note that the tolerance for latency variation, shown in Figure 5.6, is long (T_{BC}), but it is not centered at Φ^{ref}_{rec}. For example, when $\Phi^{ref}_{rec} =$ 19 UI, Φ^{run}_{rec} can be shifted by −159 UI without causing any BCID error. However, it causes BCID error if shifted by 1 UI. Similarly, when $\Phi^{ref}_{rec} = 20$ UI, Φ^{run}_{rec} can be shifted by 159 UI but not by −1 UI.

The total latency tolerance cannot be changed because it is limited by the frame period T_{BC}. However, the latency tolerance can be centered at Φ^{ref}_{rec}, increasing the so-called *latency variation tolerance symmetry*. The latency variation tolerance remains the same but now the symmetry is increased.

The highest latency variation tolerance symmetry is achieved by moving the alignment pulse to the center of the received frame. The phase offset from the alignment pulse to the frame boundary is measured by reading out the BCID latch value while gradually moving the align-

Procedure 5.5.2 – Write calibration procedure

BCID_old = **None**
For *i* **in** 0 to 7 **then**

1. Read *BCID* value by executing Procedure 5.5.1, steps from 2 to 8 using read pointer offset set to 15, and alignment delay select set to *i*
2. **if** *BCID_old* ≠ **None then**
   If *BCID* ≠ *BCID_old* **then**
   return *i*
   End If
   End If
3. *BCID_old* = *BCID*

End For
return 0

ment pulse in 8 steps of T_{rec}. Procedure 5.5.2 shows the alignment delay calibration steps to find the so-called BCID change value, i.e., the alignment delay from 1 to 7 that causes the BCID to be different from the BCID of reference, i.e., the BCID read with alignment delay set to 0. Note that Procedure 5.5.1, steps from 2 to 8, have to be repeated each time Procedure 5.5.2 step 1 is executed.

If a small or large delay value (1,6,7) causes the BCID to be different from the reference, it means that the original alignment pulse edge is already close to the frame boundary, which should be avoided. However, if the BCID changes only when a centered alignment delay value (2,3,4,5) is set, the original alignment pulse phase was already close to the center of the frame. Note if the BCID does not change for any value of alignment pulse delay from 1 to 7, it means that the edges of all the delayed alignment pulses are in the same frame. This case corresponds to a BCID change from delay value 7 to 0. In this case, the BCID change value is assigned to 0.

Figure 5.10 shows a simplified waveform of the iteration through the eight alignment delay values. The alignment pulse in the system domain is transferred to the recovered clock domain and is delayed in 8 steps of T_{rec}. For the alignment pulse select values that are asserted before or at the same time as the last word of the frame N, i.e., from 0 to 4, the write control engine is started and begin to increment the write address pointer in frame N+1. In other words, frame N and N+1 are written at address offsets 0 and 1, respectively. However, for the alignment pulse delay values that are asserted after the last word of frame N, i.e., from 5 to 7, the write control engine is started only in frame N+1 and begins to increment the write pointer only in

frame N+2. A different BCID is read starting from alignment pulse delay set to 5, and therefore, the BCID change value is set to 5.

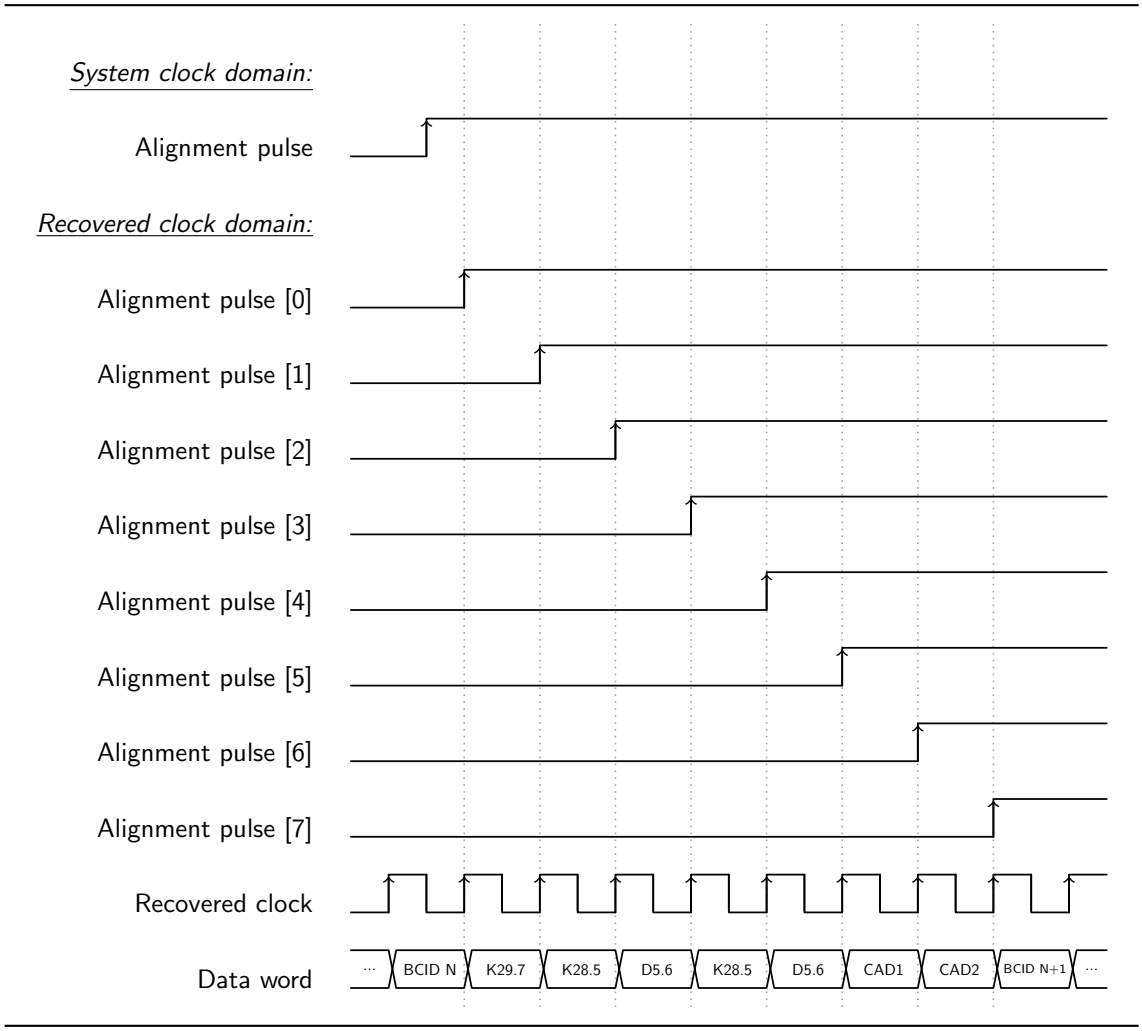


Figure 5.10 – Alignment delay iteration example for a RPC input

The unit test, described in Procedure 5.5.1, has been executed multiple times with $\Phi_{rec}^{ref} \in \mathbb{R} \mid 0 \leq \Phi_{rec}^{ref} \leq T_{BC}$ in steps of T_{rec} for all the alignment pulse delay values. Note that only Φ_{rec}^{ref} is considered for the moment and therefore no latency variation is taken into account in this test.

Figure 5.11 shows the color-coded visualization of the BCID change and frame-center values, for each Φ_{rec}^{ref} value. The BCID change value, shown in grey, corresponds to the delay, which caused the BCID value to change when compared to the value read with alignment delay set to 0. The frame-center value corresponds to the diametrically-opposed alignment delay value

that moves the alignment pulse rising edge to the center of the received frame, giving the highest latency variation tolerance symmetry.

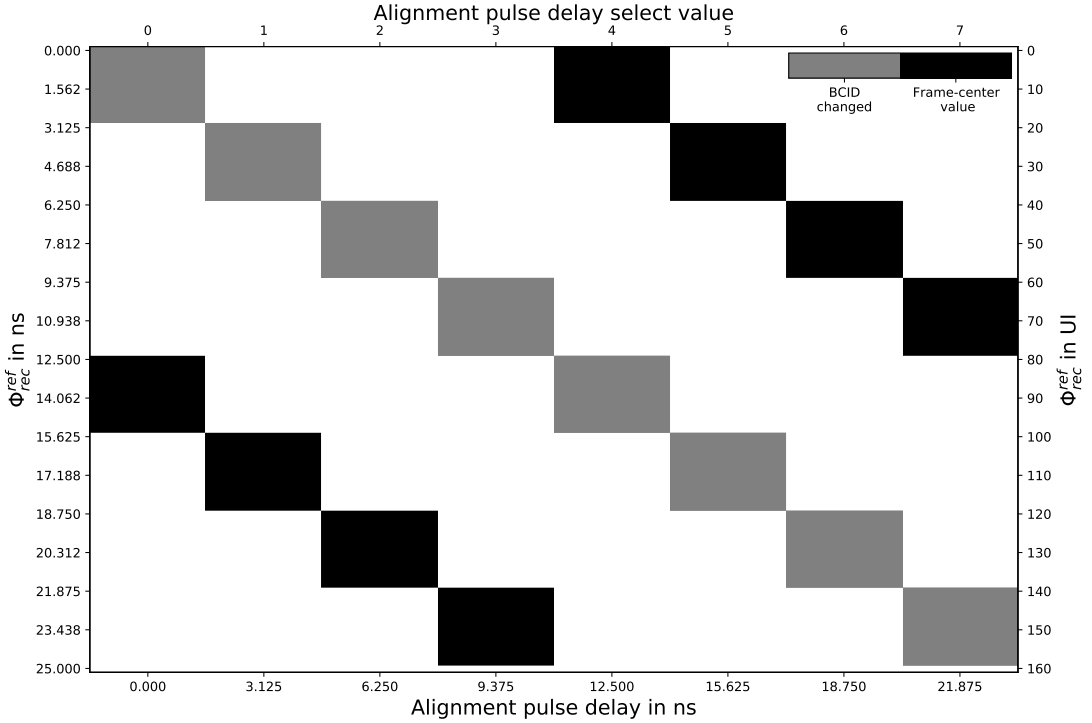


Figure 5.11 – Color-coded visualization of BCID change and frame-center values

In addition, Figure 5.11 can be used as reference to measure the Φ_{rec}^{ref} value for a given SL input. For example, reading out the BCID latch values for the SL input in Figure 5.10, the BCID change would be detected with the alignment delay set to 5. Based on Figure 5.11, this SL input would have $\Phi_{rec}^{ref} \in \mathbb{R} \mid 100\ \text{UI} \leq \Phi_{rec}^{ref} < 120\ \text{UI}$.

Note that the alignment delay value does not delay the data themselves. Therefore it has no influence in the synchronization latency. In fact, the transceiver data are directly connected to the memory data write input. Thus, the synchronization latency is controlled by the read pointer offset but also depends on Φ_{rec}^{run}, and the constant T_{rec}. More detail in the simulation of the latency is described in Section 5.5.9.

Note that the write pointer can be incremented, without any influence in the latency, in any position after the end-of-frame word. It does not need to wait until the last word of the frame because, in fact, no data are written after the end–of-frame word. In this work, the write pointer is incremented in the last word of the frame, to compensate the different position of

the end-of-frame word in RPC and TGC data formats, ensuring that Φ_{rec}^{ref} is measured, using Figure 5.11, in the same way for RPC and TGC inputs.

Figure 5.12 shows the color-coded visualization of the BCID error value with the alignment delay value set to the frame-center value shown in Figure 5.11. The y and x axis, and the light blue line in the center are defined in the same way as in Figure 5.6. The pair of dark blue lines are placed $-T_{rec}$ and T_{rec} from Φ_{rec}^{ref}. The pair of light grey lines are placed $-3T_{rec}$ and $4T_{rec}$ from Φ_{rec}^{ref}. Examining Figure 5.12, the following two conclusions are extracted:

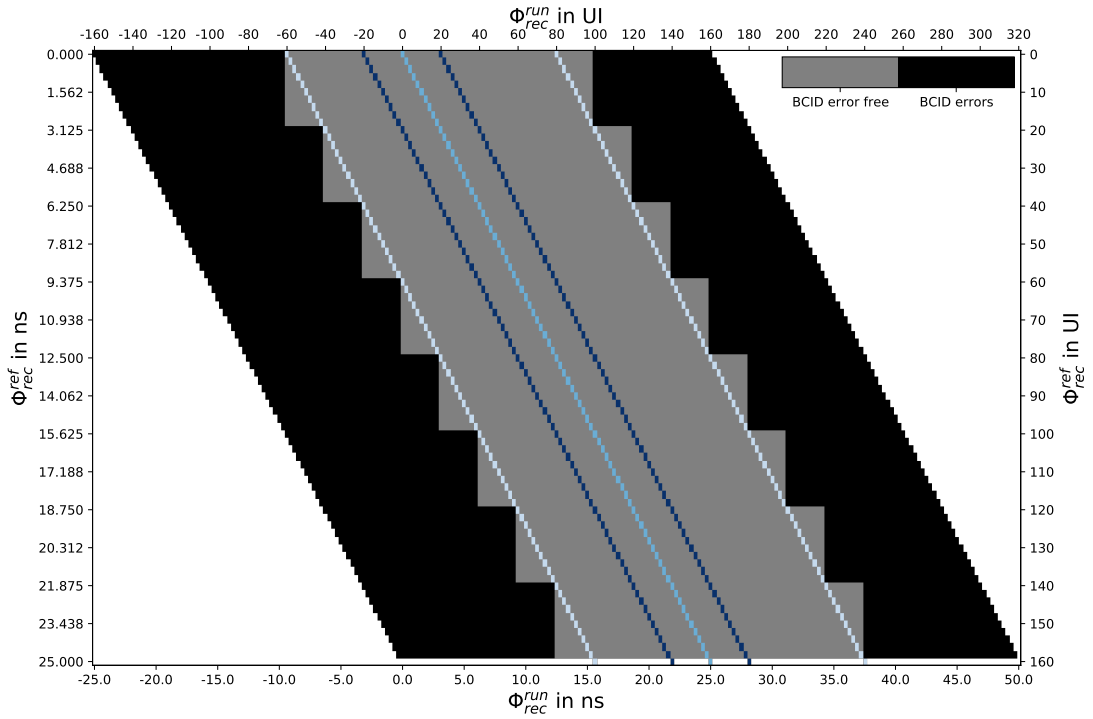


Figure 5.12 – Color-coded visualization of the BCID error for align delay set to frame-center value

1. The total latency variation tolerance remains T_{BC}. However, the latency variation tolerance symmetry has been changed.

2. For any value of Φ_{rec}^{ref}, error-free operation is guaranteed if $\Phi_{rec}^{run} \in \Phi_{rec}^{ef}$, where Φ_{rec}^{ef} represents the error-free phase offset interval defined as $\Phi_{rec}^{ef} \in \mathbb{R} \mid \Phi_{rec}^{ref} - 3T_{rec} \leq \Phi_{rec}^{ef} \leq \Phi_{rec}^{ref} + 4T_{rec}$[2].

[2] If the alignment pulse delay is set to the value previous to the frame-center value, the latency variation tolerance is guaranteed with $\Phi_{rec}^{ef} \in \mathbb{R} \mid \Phi_{rec}^{ref} - 4T_{rec} \leq \Phi_{rec}^{ef} \leq \Phi_{rec}^{ref} + 3T_{rec}$

Therefore, using the frame-center alignment pulse delay ensures error-free operation, in the memory write side, even taking into account the transceiver clock and data latency uncertainty, and the metastable propagation delay of the bit synchronizer. For the upcoming analysis, the frame-center alignment delay is always selected. In addition, as it is already known that BCID errors can exist for $\Phi_{rec}^{run} \notin \Phi_{rec}^{ef}$, Φ_{rec}^{run} is limited to $\Phi_{rec}^{run} \in \Phi_{rec}^{ef}$ in the upcoming tests.

5.5.7 Finding the error-free read pointer offsets

The test described in this section is designed to find the read pointer offsets that gives the latest and earliest BCID ensuring no CRC error, i.e., making sure that no data are attempted to be read:

1. Before it is is actually written to the dual-port memory. This is relevant to the latest BCID given by the so-called minimum latency read pointer offset.

2. After it is overwritten to the dual-port memory by the next memory cycle. This is relevant to the earliest BCID given by the so-called maximum latency read pointer offset. The memory cycle term is used here to represent the time from address position 0 to 31. In other words, a new memory cycle begins every time the write address pointer restarts.

The read pointer offset values from the minimum to the maximum-latency output are used to cover the data frame alignment functionality by compensating the phase offset Φ_{rec}^{a}, introduced in Equation (5.1) and described in Equations (5.3) and (5.4).

The test executes the unit test, described in Procedure 5.5.1, multiple times with $\Phi_{rec}^{ref} \in \mathbb{R} \mid 0 \leq \Phi_{rec}^{ref} \leq T_{BC}$ in steps of T_{rec} for all the read pointer offset values. Note that only Φ_{rec}^{ref} is considered and therefore no latency variation is taken into account in this test.

Figures 5.13 and 5.14 show the color-coded visualization of the BCID offset and CRC error values for RPC and TGC inputs respectively. The y and x axis represent Φ_{rec}^{ref} and the read pointer offset values respectively. Each point in this figure represents the BCID offset and CRC error values read in each of the times Procedure 5.5.1 has been executed. The values written in red and grey corresponds to the BCID values with and without CRC errors, respectively. Points with background in yellow and purple are originated from bunch crossings of different orbits. The following three conclusions are extracted from Figures 5.13 and 5.14:

First, the plots are different because, before reading the memory, one has to wait to receive only the data format words that are actually written to the memory, instead of all the eight words of the data format. Only the first 4 and 6 words of the data format are actually written to the memory for RPC and TGC inputs, respectively.

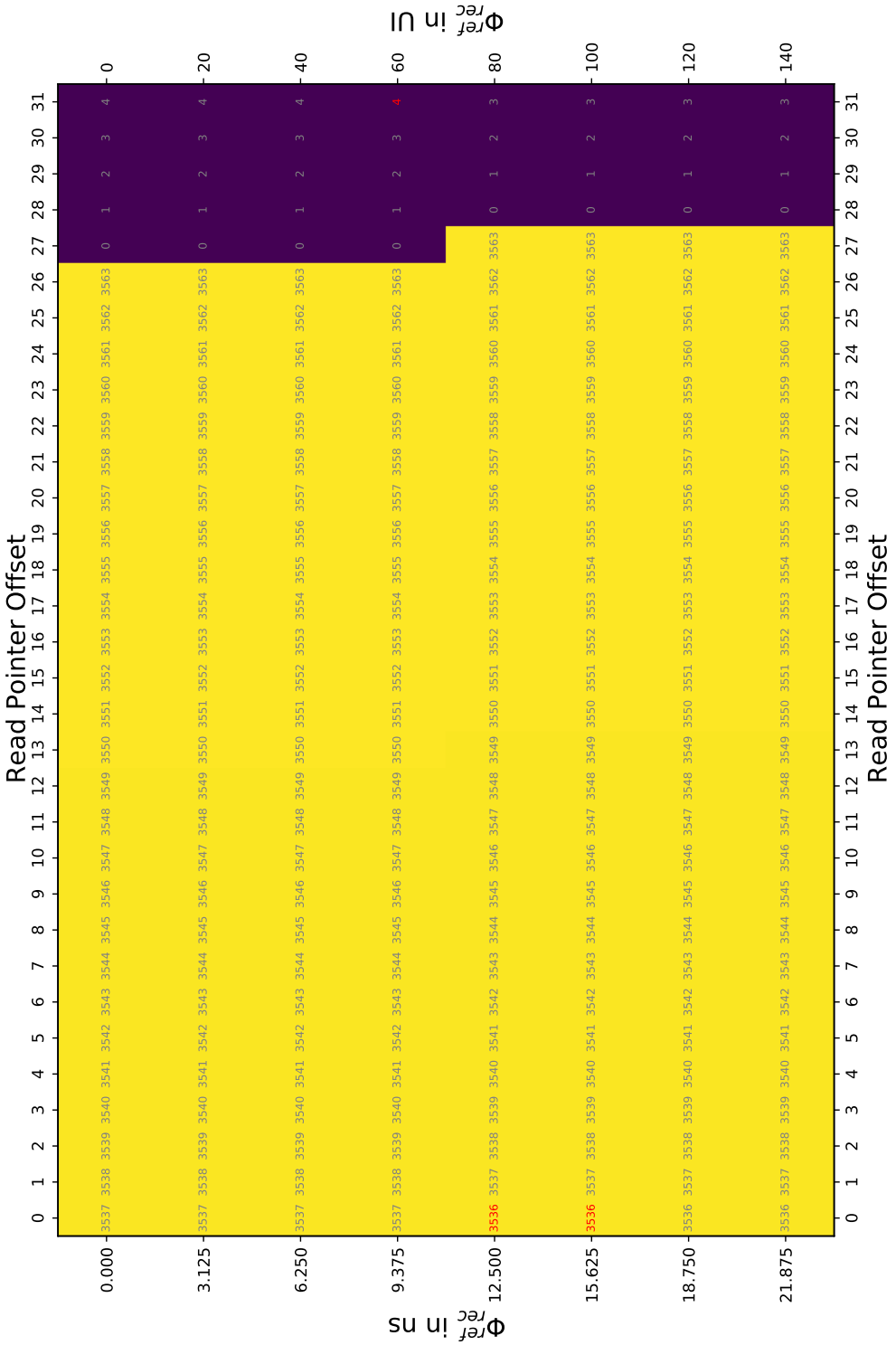


Figure 5.13 – Color-coded visualization of the RPC BCID offset and CRC error values

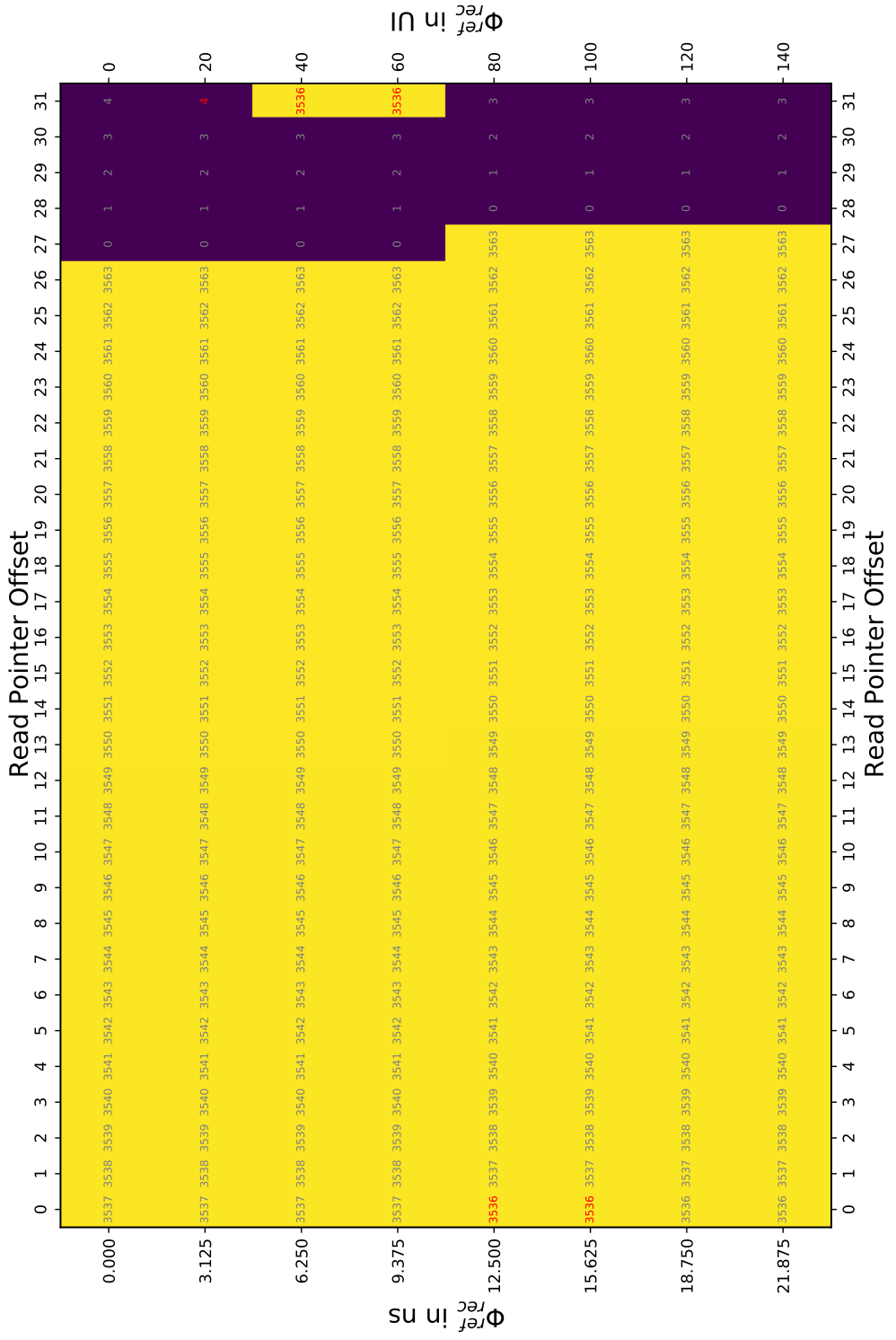


Figure 5.14 – Color-coded visualization of the TGC BCID offset and CRC error values

Second, the number of read pointer offsets containing CRC errors seen in Figures 5.13 and 5.14, i.e., 3 and 5 for RPC and TGC inputs respectively, depends also on the number of words written to the memory. Tables 5.3 and 5.4 illustrates the RPC and TGC data frame reading combinations. There are eight combinations corresponding to each of the possible combinations of data found in the memory at the moment that the output is read when read and write pointers have the same value. Cells in green and red correspond to complete and incomplete data frames, respectively. White cells correspond to data words that are not written to the memory. When the read and write pointers are the same, every incomplete frame, shown in red, causes a CRC error, which is also seen in Figures 5.13 and 5.14.

Table 5.3 – RPC data frame combinations

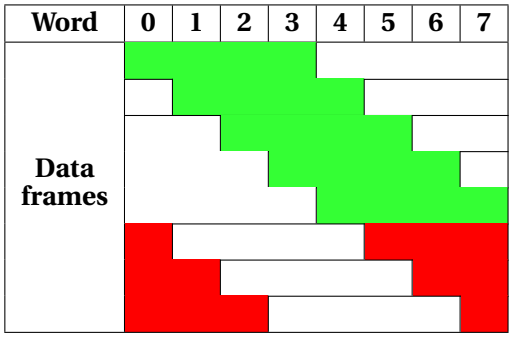
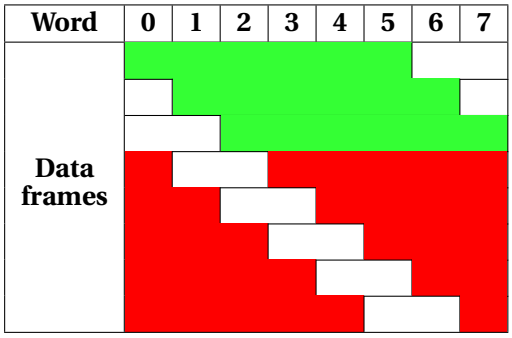


Table 5.4 – TGC data frame combinations



Third, the minimum latency read pointer offset is extracted from Figures 5.13 and 5.14 by looking to the read pointer offset, per Φ_{rec}^{ref}, that gives the latest BCID without CRC errors. Similarly, the maximum latency read pointer offset is given by the earliest BCID instead.

Table 5.5 shows the minimum, and maximum latency read pointer offsets for RPC and TGC inputs. The Φ_{rec}^{ref} range is expressed in terms of the BCID change position. The mapping between Φ_{rec}^{ref} and BCID change position is given by Figure 5.11. For example, if minimum latency output is needed for a TGC input with BCID change position 2 ($\Phi_{rec}^{ref} \in \mathbb{R} \mid 40 \text{ UI} \leq \Phi_{rec}^{ref} \leq 60 \text{ UI}$), the read pointer offset should be set to 30.

Table 5.5 – Read pointer offset values

Configuration		**BCID change position**							
Type	**Detector**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Minimum latency	RPC	31	31	31	30	31	31	31	31
	TGC	31	30	30	30	31	31	31	31
Maximum latency	RPC or TGC	0	0	0	0	1	1	0	0

These values are valid if the input data has no latency variation. Section 5.5.8 describes the process of finding the read pointer offset for different intervals of latency variation tolerance.

5.5.8 Addressing latency variation in the memory read side

In order to study the effect of the latency variation in the memory read side, the reference and running phase offset test, described in Section 5.5.3, has been executed for minimum and maximum latency read pointer offsets, for RPC and TGC inputs using the following configuration:

- Alignment delay set according to Figure 5.11
- Read pointer offset set according to Table 5.5
- All possible reference phase offset values are simulated, i.e., $\Phi_{rec}^{ref} \in \mathbb{R} \mid 0 \leq \Phi_{rec}^{ref} < T_{BC}$
- The memory write side error-free phase offset interval is used, i.e., $\Phi_{rec}^{run} \in \mathbb{R} \mid \Phi_{rec}^{ref} - 3T_{rec} \leq \Phi_{rec}^{run} \leq \Phi_{rec}^{ref} + 4T_{rec}$
- Steps of 2 UI are used for minimizing simulation time

Figures 5.15 to 5.18 show the color-coded visualization of the BCID or CRC error using the configuration mentioned above. The BCID and CRC error are combined together in order to check if a given frame data are corrupted and/or misaligned, more details in Section 5.4. Examining these plots, the following conclusions are extracted:

First, for the minimum latency read pointer offsets, only positive variation in the phase offset causes errors. It is because, for reading data in a given memory position, it is not a problem if the input data arrive earlier than expected. But if it arrives later, data from the previous memory cycle is read instead. Similarly, for the maximum latency read pointer offset, only negative phase offset variation causes errors. It is because only data arriving before the memory is read could overwrite the earliest data found in the memory.

Second, the minimum latency error plots for RPC and TGC inputs are different because the part of the data that will first cause errors, in case it read too early because the incoming data arriving too late, is the ending of the frame. As the end-of-frame word is placed in different positions for RPC and TGC inputs, it will cause errors in different values of Φ_{rec}^{ref}. The end-of-frame word is placed in different positions because the portion of the data frame containing trigger information has different lengths for RPC and TGC inputs. On the other side, the maximum latency BCID or CRC error plots are the same for RPC and TGC inputs because the part of the data that will first cause errors, in case it is overwritten by incoming data arriving

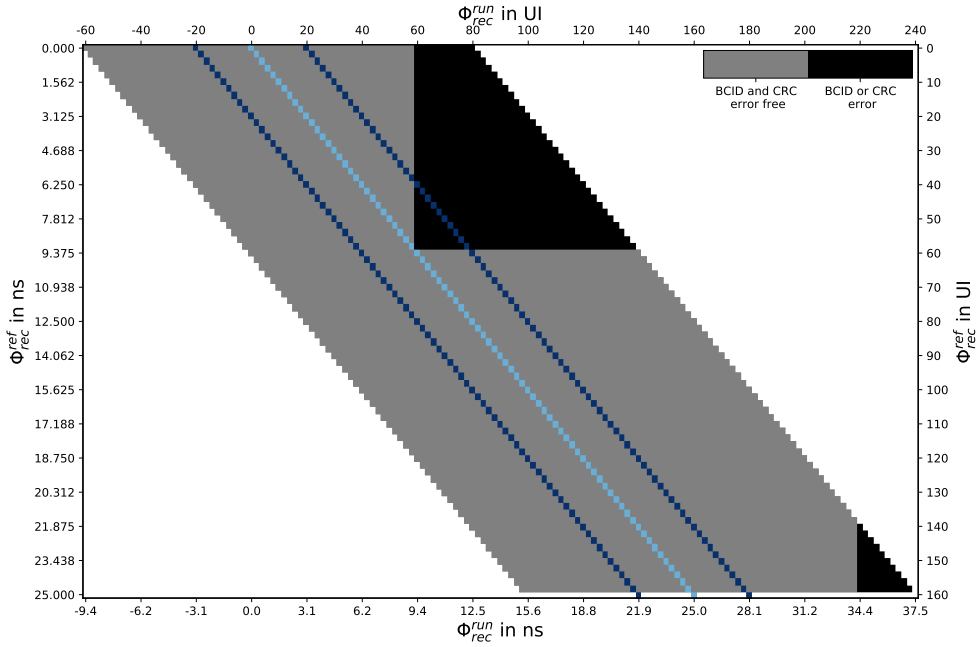


Figure 5.15 – Color-coded visualization of the minimum-latency RPC BCID or CRC error value

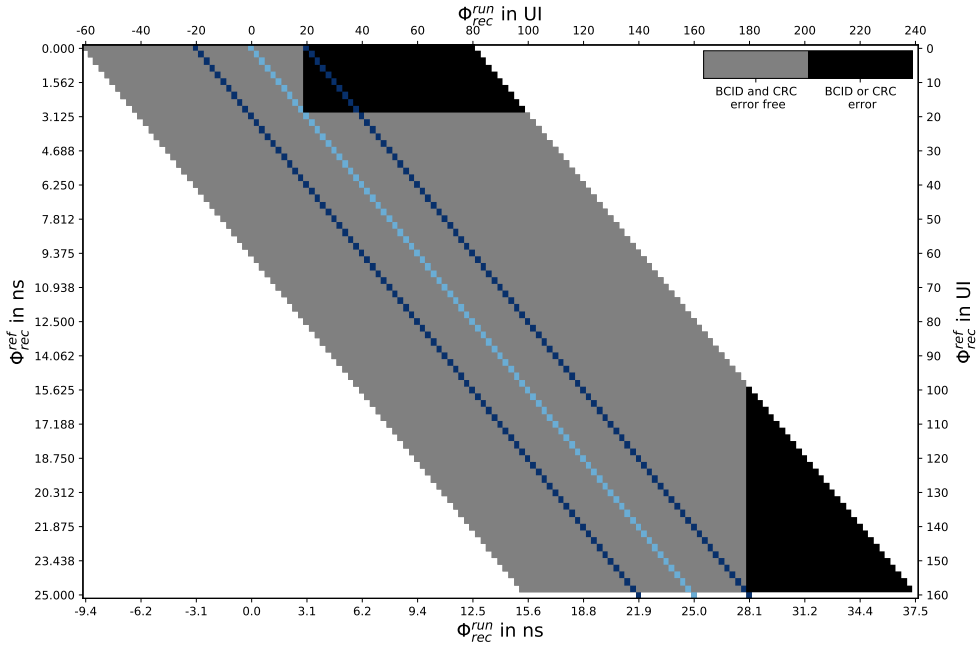


Figure 5.16 – Color-coded visualization of the minimum-latency TGC BCID or CRC error value

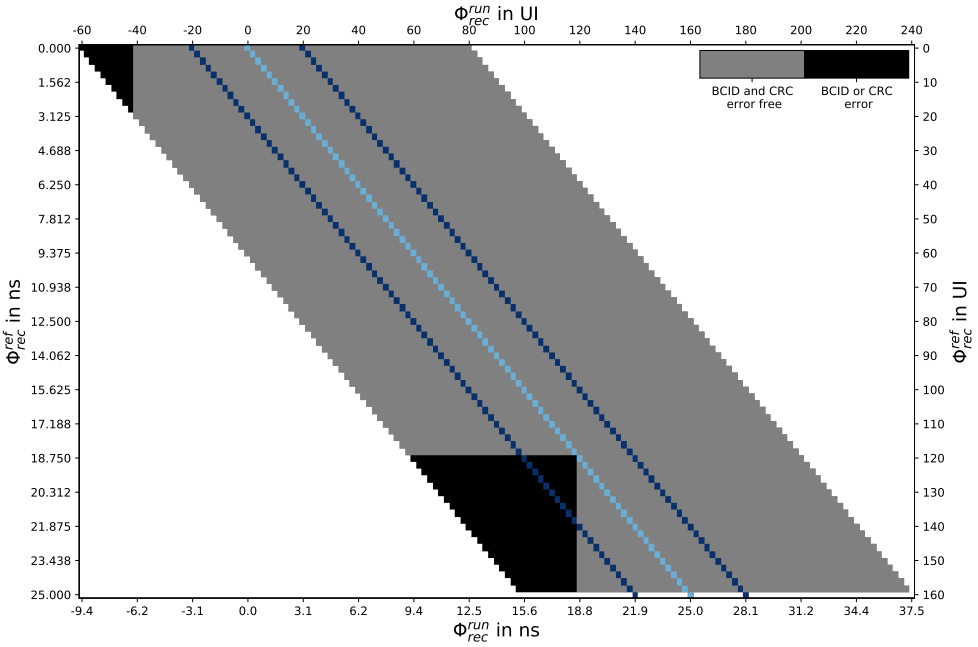


Figure 5.17 – Color-coded visualization of the maximum-latency RPC BCID or CRC error value

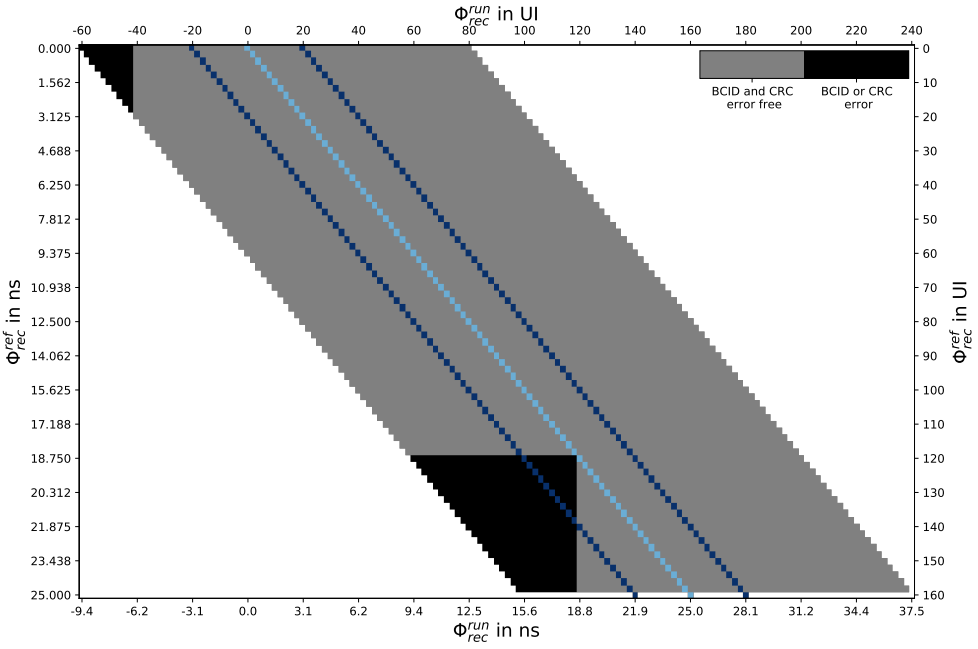


Figure 5.18 – Color-coded visualization of the maximum-latency TGC BCID or CRC error value

too early, is the beginning of the data frame. As the beginning of the frame is placed in the same position for RPC and TGC inputs, the plots are the same.

Third, in both cases, even a latency variation of only ± 1 UI will cause errors. This happens with RPC and TGC inputs set to the minimum latency read pointer offset when $\Phi_{rec}^{ref} = 59$ UI and $\Phi_{rec}^{ref} = 19$ UI, respectively. Also when the maximum latency read pointer offset is selected and $\Phi_{rec}^{ref} = 120$ UI for both RPC and TGC inputs.

BCID or CRC errors are not tolerated in the MUCTPI and should be mitigated. The latency variation can be addressed by decrementing and incrementing the minimum and maximum latency read pointer offsets, respectively, for the Φ_{rec}^{ref} that are needed. For example, if no errors are tolerated in $\Phi_{rec}^{run} \in \mathbb{R} \mid \Phi_{rec}^{ref} - T_{rec} \leq \Phi_{rec}^{run} \leq \Phi_{rec}^{ref} + T_{rec}$, the RPC and TGC minimum latency read pointer offsets corresponding to $\Phi_{rec}^{ref} \in \mathbb{R} \mid 40 \text{ UI} \leq \Phi_{rec}^{run} < 60 \text{ UI}$ and $\Phi_{rec}^{ref} \in \mathbb{R} \mid 0 \leq \Phi_{rec}^{run} < 20$ UI, respectively should be decremented. Also the maximum latency read pointer offset corresponding to $\Phi_{rec}^{ref} \in \mathbb{R} \mid 120 \text{ UI} \leq \Phi_{rec}^{run} < 140$ UI should be incremented.

Table 5.6 shows all the read pointer offsets that should be used to ensure latency variation tolerance for $\Phi_{rec}^{run} \in \mathbb{R} \mid \Phi_{rec}^{ref} + V_L T_{rec} \leq \Phi_{rec}^{run} \leq \Phi_{rec}^{ref} + V_R T_{rec}$. V_L and V_L are described in Section 5.3. The read pointers offsets highlighted in blue are the offsets that gives the minimum possible latency and maximum possible alignment delay for the MUCTPI, with the latency variation limited to $\Phi_{rec}^{run} \in \mathbb{R} \mid \Phi_{rec}^{ref} - T_{rec} \leq \Phi_{rec}^{run} \leq \Phi_{rec}^{ref} + T_{rec}$, i.e. $V_L = -1$ and $V_R = 1$. If additional latency variation tolerance is needed, read pointer offsets values with the respective V_L and V_R should be selected.

Table 5.6 – Latency-variation-tolerant read pointer offset values

Configuration				**BCID change position**							
Type	**Detector**	V_L	V_R	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Minimum latency	RPC	-3	0	31	31	31	30	31	31	31	31
			1	31	31	30	30	31	31	31	31
			2	31	30	30	30	31	31	31	31
			3	30	30	30	30	31	31	31	31
			4	30	30	30	30	31	31	31	30
	TGC		0	31	30	30	30	31	31	31	31
			1	30	30	30	30	31	31	31	31
			2	30	30	30	30	31	31	31	30
			3	30	30	30	30	31	31	30	30
			4	30	30	30	30	31	30	30	30
Maximum latency	RPC or TGC	0	4	0	0	0	0	1	1	0	0
		-1		0	0	0	0	1	1	1	0
		-2		0	0	0	0	1	1	1	1
		-3		1	0	0	0	1	1	1	1

In order to check the new read pointer offset values, the reference and running phase offset test has been repeated with the read pointer offsets with $V_L = -1$ and $V_R = 1$ found in Table 5.6. Figures 5.19 and 5.20 show the latency minimum BCID or CRC error plots and Figure 5.21 shows the maximum latency error plot for both RPC and TGC inputs. No errors exist in $\Phi_{rec}^{run} \in \mathbb{R} \mid \Phi_{rec}^{ref} - T_{rec} \leq \Phi_{rec}^{run} \leq \Phi_{rec}^{ref} + T_{rec}$ for any of the 3 plots. Next, the test has been repeated using read pointer offsets with $V_L = -3$ and $V_R = 4$. Figure 5.22 shows the respective color-coded visualization of the minimum and maximum-latency RPC and TGC BCID or CRC error values. No errors are found for $\Phi_{rec}^{run} \in \Phi_{rec}^{ef}$, i.e. $\Phi_{rec}^{run} \in \mathbb{R} \mid \Phi_{rec}^{ref} - 3T_{rec} \leq \Phi_{rec}^{run} \leq \Phi_{rec}^{ref} + 4T_{rec}$.

5.5.9 Latency simulation

In order to simulate the synchronization latency, the reference and running phase offset test, described in Section 5.5.3, has been repeated using the following configuration:

- Alignment delay set according to Figure 5.11
- Read pointer offset set according to Table 5.6 with $V_L = -1$ and $V_R = 1$
- All possible reference phase offset values, $\Phi_{rec}^{ref} \in \mathbb{R} \mid 0 \leq \Phi_{rec}^{ref} < T_{BC}$
- Latency variation found in SL to MUCTPI link, i.e., $\Phi_{rec}^{run} \in \mathbb{R} \mid \Phi_{rec}^{ref} - T_{rec} \leq \Phi_{rec}^{run} \leq \Phi_{rec}^{ref} + T_{rec}$
- Steps of 1 UI

Figures 5.23 and 5.24 shows the color-coded synchronization IP latency for the minimum latency read pointer offset with $V_L = -1$ and $V_R = 1$ for RPC and TGC inputs respectively. Figure 5.25 shows the definition of the synchronization latency Δ_t. Δ_t has been defined as the time from receiving the end-of-frame word, at the input of the dual-port memory, to outputting the complete data frame. A forked coroutine monitors both sides of the dual-port memory and extracts the simulation time of both vertical markers shown in Figure 5.25. The delay from the additional registers before and after the dual-port memory is not accounted here. The following conclusions are extracted from Figures 5.23 and 5.24.

First, as the synchronization latency is defined from the end-of-the frame and not from the beginning of the frame, the frame length does not affect the latency value. For this reason, the minimum and maximum latency values for RPC and TGC are the same. However, minimum and maximum values occur in different Φ_{rec}^{ref} values for RPC and TGC inputs.

Second, the minimum latency value of 3.125 ns, corresponding to 20 UI, is found with $\Phi_{rec}^{ref} =$ 40 UI and $\Phi_{rec}^{run} = 60$ UI, for RPC inputs, and $\Phi_{rec}^{ref} = 0$ UI and $\Phi_{rec}^{run} = 20$ UI, for TGC inputs.

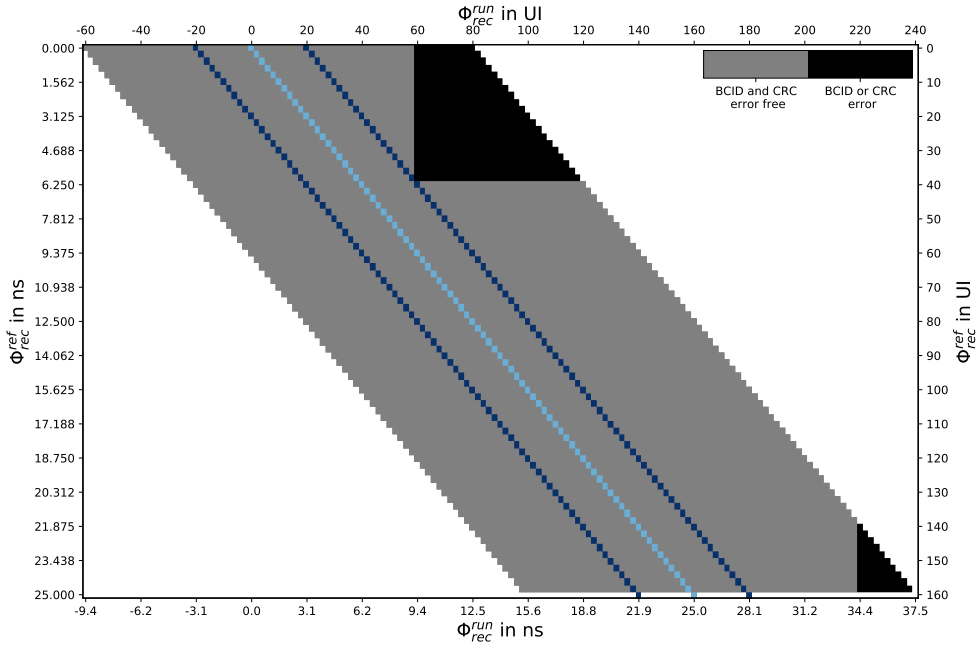


Figure 5.19 – Minimum-latency RPC BCID or CRC error values with latency variation tolerance with $V_L = -1$ and $V_R = 1$

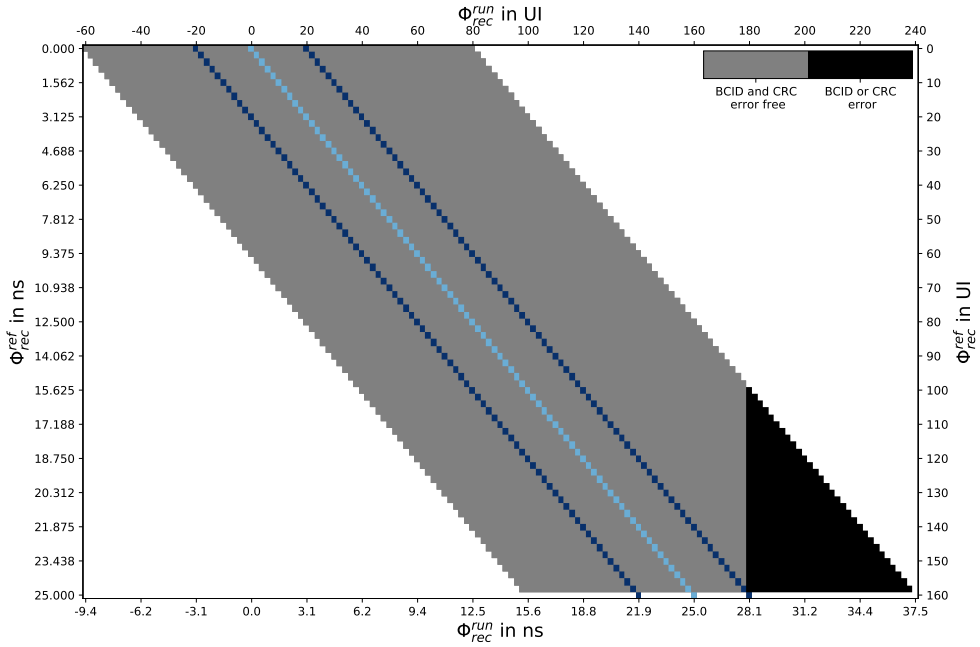


Figure 5.20 – Color-coded visualization of the minimum-latency TGC BCID or CRC error values with latency variation tolerance with $V_L = -1$ and $V_R = 1$

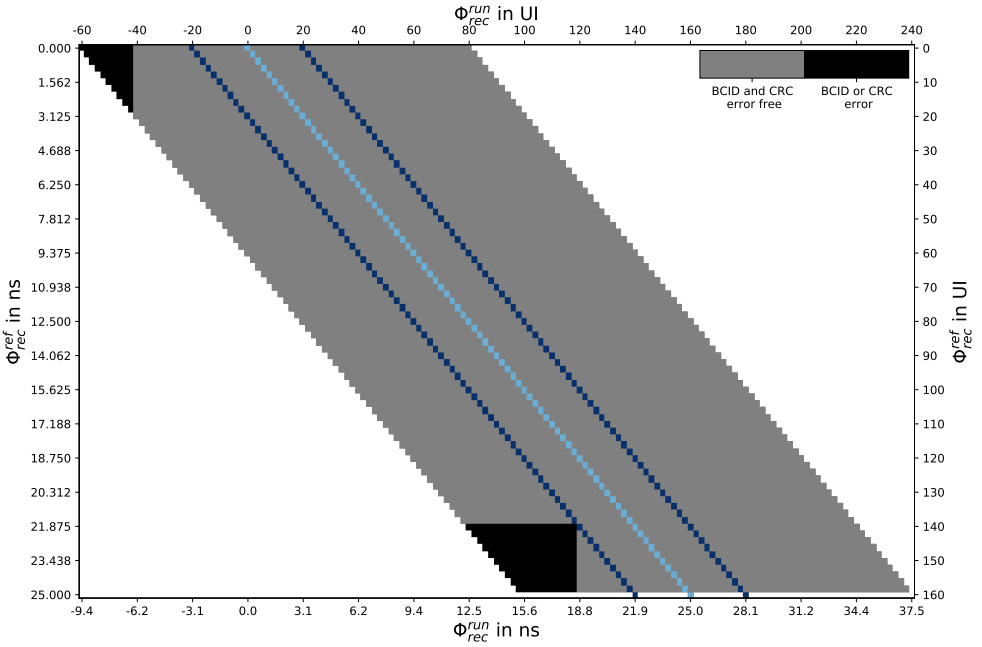


Figure 5.21 – Color-coded visualization of the maximum-latency RPC and TGC BCID or CRC error values with latency variation tolerance with $V_L = -1$ and $V_R = 1$

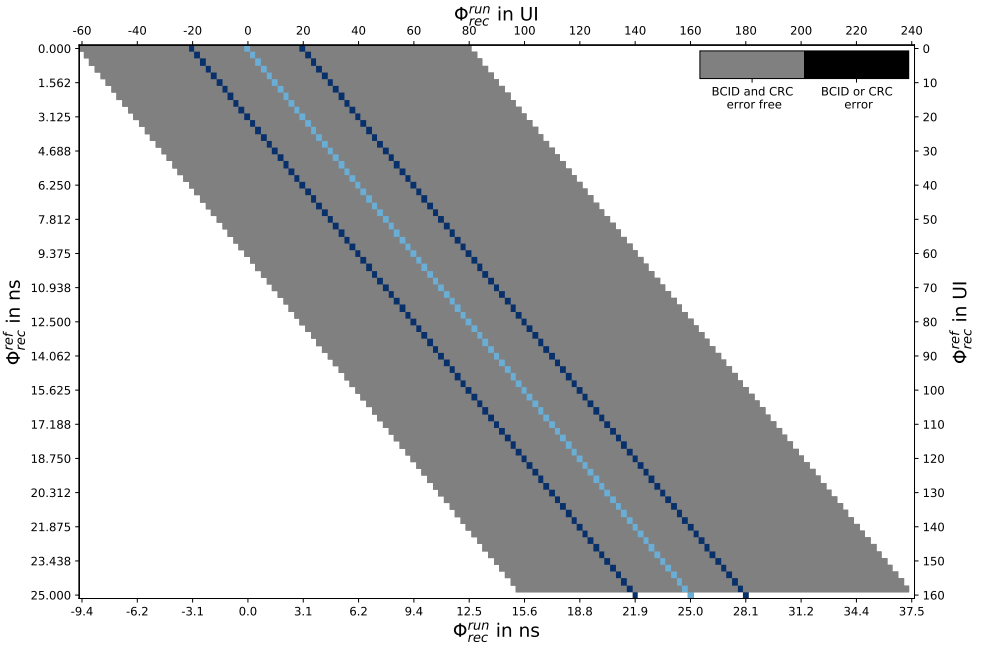


Figure 5.22 – Color-coded visualization of the minimum and maximum-latency RPC and TGC BCID or CRC error values with latency variation tolerance with $V_L = -3$ and $V_R = 4$

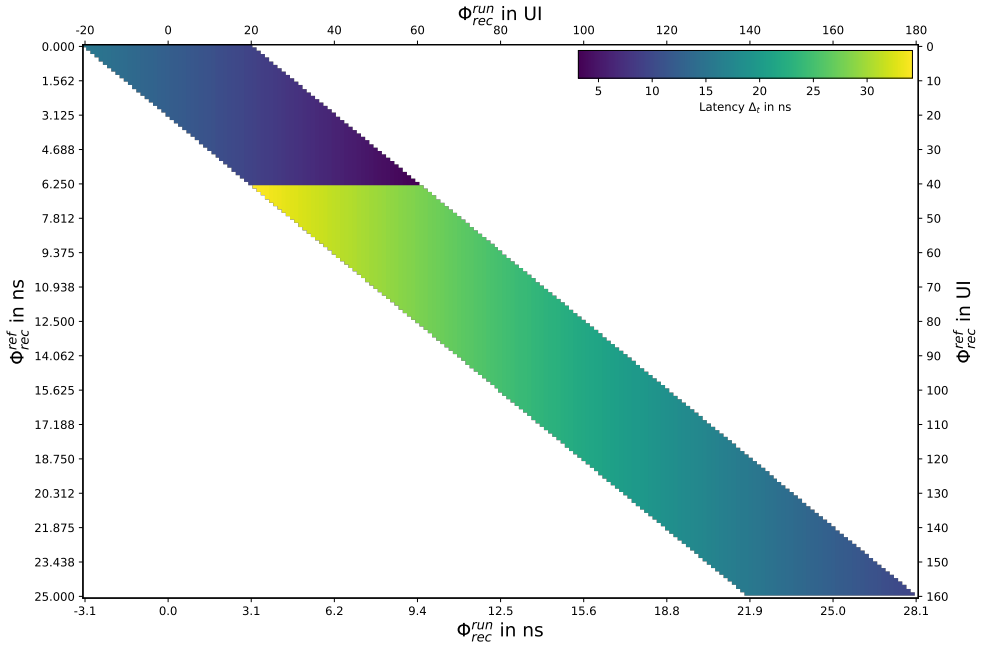


Figure 5.23 – RPC synchronization latency for the minimum latency read pointer offset with $V_L = -1$ and $V_R = 1$

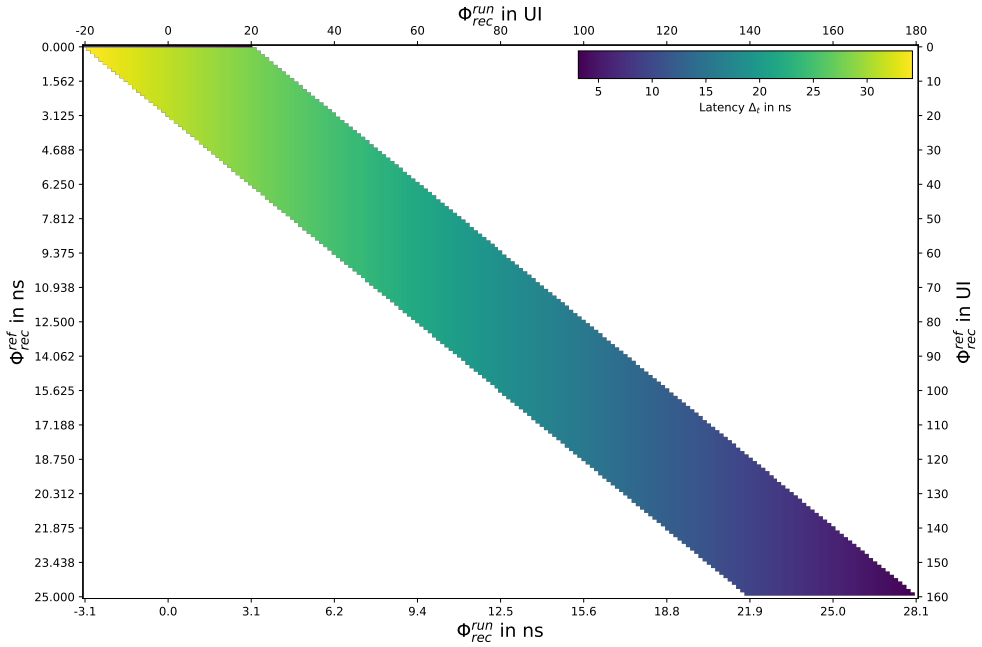


Figure 5.24 – TGC synchronization latency for the minimum latency read pointer offset with $V_L = -1$ and $V_R = 1$

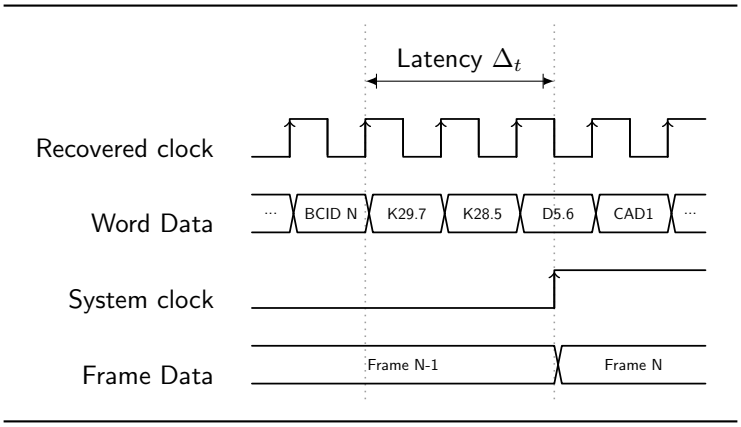


Figure 5.25 – Synchronization unit latency Δ_t

This is the case when the end-of-frame word is at the same time this memory position is read in the output. Note that it is only possible when during calibration the end-of-frame arrived T_{rec} earlier than the system clock edge, but after resetting the transceiver, the latency variation moved the data by T_{rec}. Therefore in this case the latency is given only by the time taken to write the data to the memory given by T_{rec}, the write clock period.

Third, the maximum latency value of ≈ 34.22 ns, corresponding to 219 UI, is with $\Phi_{rec}^{ref} = 41$ UI and $\Phi_{rec}^{run} = 21$ UI, for RPC inputs, and $\Phi_{rec}^{ref} = 1$ UI and $\Phi_{rec}^{run} = -19$ UI, for TGC inputs. The worst-case latency occurs when, during calibration, the data are received 1 UI after the time received at the best case latency. This forces the read address pointer to be set in such a way that a delay of 159 UI, for waiting the next system clock period, and more 20 UI, for compensating the latency variation to the left, are added in order to read the memory data safely. In addition to that, after the calibration, the input is moved by 20.UI to the right due to latency variation. This results in a worst-case latency of 20 + 159 + 20 + 20 = 219 UI.

Equations (5.13) and (5.14) describe the synchronization latency computation.

$$\Delta_t = T_{rec} + V_R T_{rec} + \Phi_{rd}^{rec} - \Delta\Phi_{mgt}, \tag{5.13}$$

where

$$\Phi_{rd}^{rec} \in \mathbb{R} \mid 0 \leq \Phi_{rd}^{rec} < T_{BC}, \tag{5.14}$$

and Φ_{rd}^{rec} represents the phase offset from the end-of-frame word to the reading time limit Φ_{rd}^{sys}. Φ_{rd}^{sys} is defined $V_R T_{rec}$ earlier than the system clock edge that reads the output data.

The minimum latency is given when $\Phi_{rd}^{rec} = 0$ and $\Delta\Phi_{mgt}$ is maximum, but not higher than $V_R T_{rec}$, otherwise it will cause errors. The maximum latency is given when $\Phi_{rd}^{rec} \rightarrow T_{BC}$ and $\Delta\Phi_{mgt}$ is minimum.

Table 5.7 shows the simulated synchronization latency values, in ns, for the MUCTPI, i.e. $T_{rec} = 3.125$ ns, $T_{sys} = 6.25$ ns, and $-T_{rec} \leq \Delta\Phi_{mgt} \leq T_{rec}$. Using $V_R = 1$ is the option that gives the minimum latency but no slack when $\Delta\Phi_{mgt} = T_{rec}$. An higher V_R can be selected in view of increased slack.

Table 5.7 – Latency values for the MUCTPI given in ns

V_R	Dual-port memory only		Additional input register		Additional output register		Additional input and output register	
	Min	Max	Min	Max	Min	Max	Min	Max
1	3.125	34.219	6.250	37.344	9.375	40.469	12.500	43.594
2	6.250	37.344	9.375	40.469	12.500	43.594	15.625	46.719
3	9.375	40.469	12.500	43.594	15.625	46.719	18.750	49.844
4	12.500	43.594	15.625	46.719	18.750	49.844	21.875	52.969

The latency values are listed according to Figure 5.25, i.e., from the end of the frame instead of the beginning. If the latency from the beginning of the frame is required, one should add 9.375 ns and 15.625 ns for RPC and TGC inputs, respectively.

5.5.10 Output phase variation

The output phase variation test complements the tests described in Sections 5.5.6 and 5.5.8, by checking if the phase of the output data is constant. This test is complementary because the output phase variation can also be detected by looking for BCID errors.

The output phase for each Φ_{rec}^{run} is compared against the Φ_{rec}^{ref} output phase. This test uses the simulation time of the output data for Φ_{rec}^{ref} and Φ_{rec}^{run}. The difference between these two values gives the output phase variation. Figure 5.26 shows the color-coded value of the output phase variation using the data from the test executed in Section 5.5.9. No output phase variation exists for any value of Φ_{rec}^{ref} and Φ_{rec}^{run}.

5.5.11 Summary

The functional simulation, described in this section, addressed the following tasks:

1. Check the synchronization IP for design errors
2. Elaborate the alignment delay calibration procedure

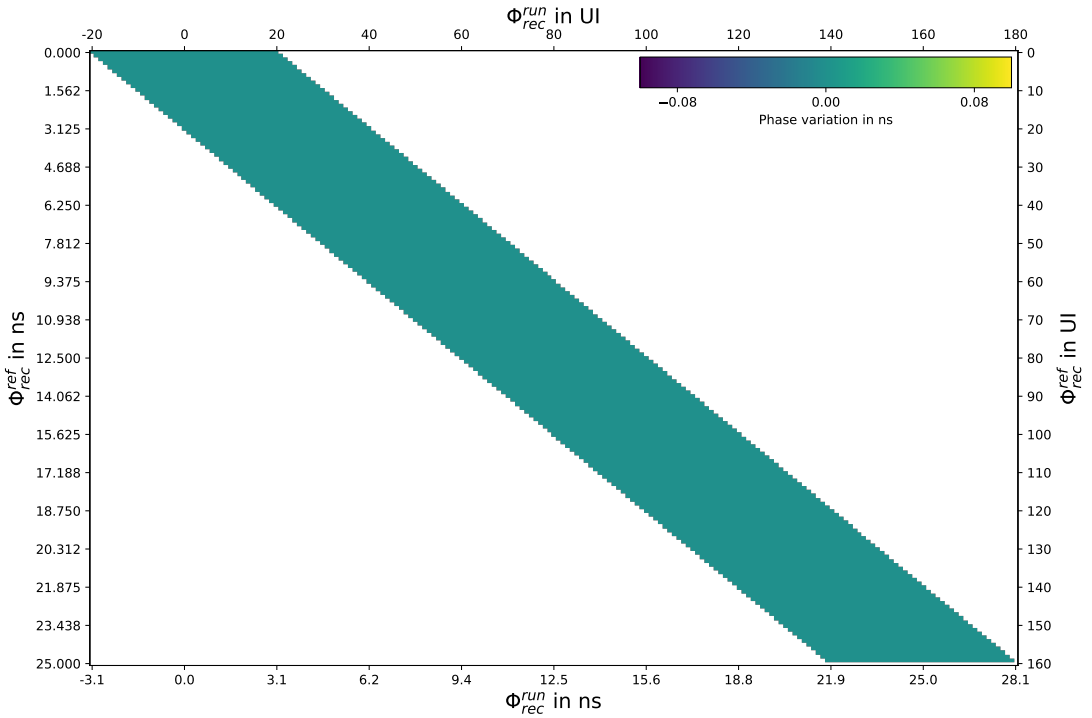


Figure 5.26 – RPC and TGC output phase variation using the minimum latency read pointer offset with $V_L = -1$ and $V_R = 1$

3. Find the minimum and maximum latency read address pointer offsets
4. Define the error-free latency variation limits
5. Define the synchronization latency values

After addressing, the latency variation, in the write and read sides of the memory, the synchronization IP is guaranteed to be error free for any $\Phi^{run}_{rec} \in \mathbb{R} \mid \Phi^{ref}_{rec} + V_L T_{rec} \leq \Phi^{run}_{rec} \leq \Phi^{ref}_{rec} + V_R T_{rec}$, with V_L and V_R depending from the read pointer selected in Table 5.6.

One can use Figures 5.23 and 5.24, as reference, to measure the synchronization latency for a given SL input in the MUCTPI. The Φ^{ref}_{rec} is obtained from the BCID change value, which is measured using Procedure 5.5.2.

5.6 Integration tests

Integration tests have been performed with the RPC and TGC sector logic modules. The goal is to verify if data could be received, synchronous to the BC clock, without errors. Figure 5.27

shows the block diagram of the RPC and TGC integration tests. The TTC system distributes the same 40 MHz clock to the sector logic module and MUCTPI. The RPC or TGC sector logic module sends a periodic pattern following the data format described in Tables 5.1 and 5.2, respectively. The MUCTPI receives the data and synchronizes them to the 40 MHz clock domain. Comparators are used in both endings to indicate that data from the BCID 0 is transmitted or received. The compactor outputs, so-called, transmitter, and receiver flags are connected to the oscilloscope.

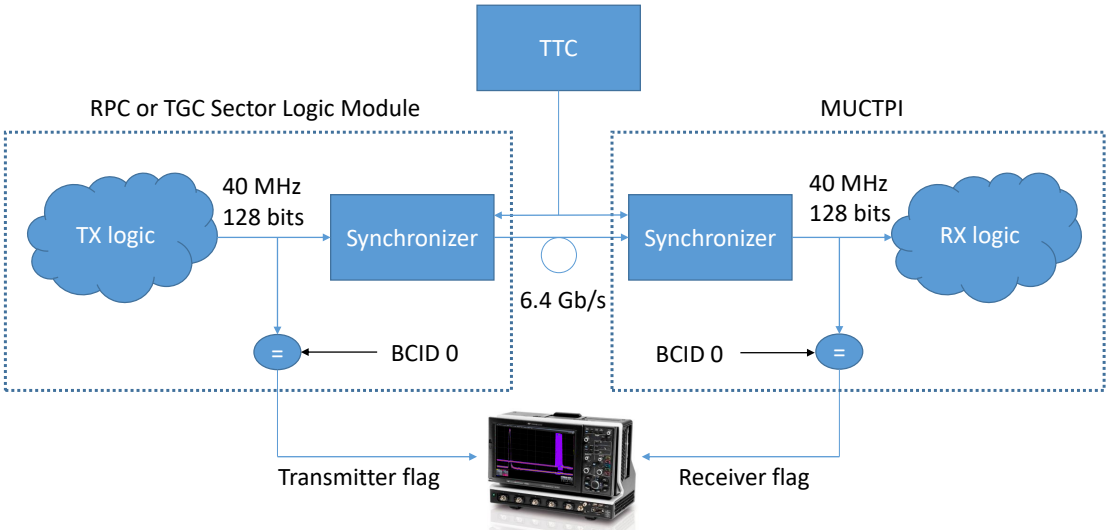


Figure 5.27 – RPC and TGC integration test block diagram

The alignment delay has been set to the frame-center value, and the read pointer offset has been set to the minimum latency read pointer offset with $V_R = 1$. The value of the CRC error count has been checked after multiple resets and during an overnight test, without errors.

A snapshot memory connected to the MUCTPI synchronization output has been used to record data from the SL input during 4096 bunch crossings. The data has been checked for errors in software. No errors have been detected.

The synchronization IP firmware has been implemented, at the time of this test, without additional input or output registers. This means that the values measured here are compared to the *dual-port memory only* option listed in Table 5.7. Equation (5.15) describes the expected total latency Δ_T^{exp}.

$$\Delta_T^{exp} = \Delta_{TTS}^{exp} + \Delta_T^{MGT} + \Delta_{TRS}^{exp}, \tag{5.15}$$

where Δ_{TTS}^{exp} and Δ_{TRS}^{exp} represent the expected latency value for transmitter and receiver synchronization respectively. Note that the value from the beginning of the frame to the system clock is used. Δ_{T}^{MGT} represent the total latency in the transmitter and receiver transceivers.

Equations (5.16) to (5.19) describes the expected synchronization latency Δ_{TSR}^{exp} and Δ_{TST}^{exp}, and total latency Δ_{TR}^{exp} and Δ_{TT}^{exp}, for RPC and TGC inputs respectively, assuming:

1. Transmitter takes the same time to synchronize the data as the receiver, $\Delta_{TTS}^{exp} = \Delta_{TRS}^{exp}$
2. Minimum and maximum synchronization latency values are extracted from Table 5.7 using the option *dual-port memory only* and $V_R = 1$.
3. In order to compute the latency from the beginning of the frame to the system clock edge, 9.375 ns and 15.625 ns is added to the value extracted from Table 5.7 for RPC and TGC inputs, respectively
4. From Section 4.3.2, $\Delta_{T}^{MGT} \approx 50$ ns

$$\Delta_{TSR}^{exp} \in \mathbb{R} \mid 13 \text{ ns} \leq \Delta_{TSR}^{exp} \leq 40 \text{ ns} \tag{5.16}$$
$$\Delta_{TST}^{exp} \in \mathbb{R} \mid 19 \text{ ns} \leq \Delta_{TST}^{exp} \leq 46 \text{ ns} \tag{5.17}$$

$$\Delta_{TR}^{exp} \in \mathbb{R} \mid 76 \text{ ns} \leq \Delta_{TR}^{exp} \leq 130 \text{ ns} \tag{5.18}$$
$$\Delta_{TT}^{exp} \in \mathbb{R} \mid 88 \text{ ns} \leq \Delta_{TT}^{exp} \leq 142 \text{ ns} \tag{5.19}$$

Figure 5.28 shows the oscilloscope acquisition waveform used to measure the latency between the RPC sector logic module prototype to the MUCTPI. The sector logic module asserts a flag (oscilloscope channel 3) when the 128-bit word associated with BCID 0 is sent in the 40 MHz clock domain logic. When the same 128-bit word is received, the MUCTPI asserts a second flag (oscilloscope channel 2). Approximately 5 ns has to be deducted from the measured value to compensate for the combinatorial delay in the TRP FPGA. In addition, 15 $m \times 5\ \frac{ns}{m} = 75$ ns should be deducted from the measured value, for the latency in the optical fibres. Sixteen ns has to be deducted from the longer electrical cable, which is used to connect the MUCTPI flag to the scope compared to the cable used to the RPC flag. Three-quarters of BC should be added to compensate for the fact that the flag is generated with a 40MHz clock in the RPC and 160MHz at the MUCTPI. Therefore the latency is ≈ 109 ns, which corresponds to $\approx 4.3\ T_{BC}$ clock period.

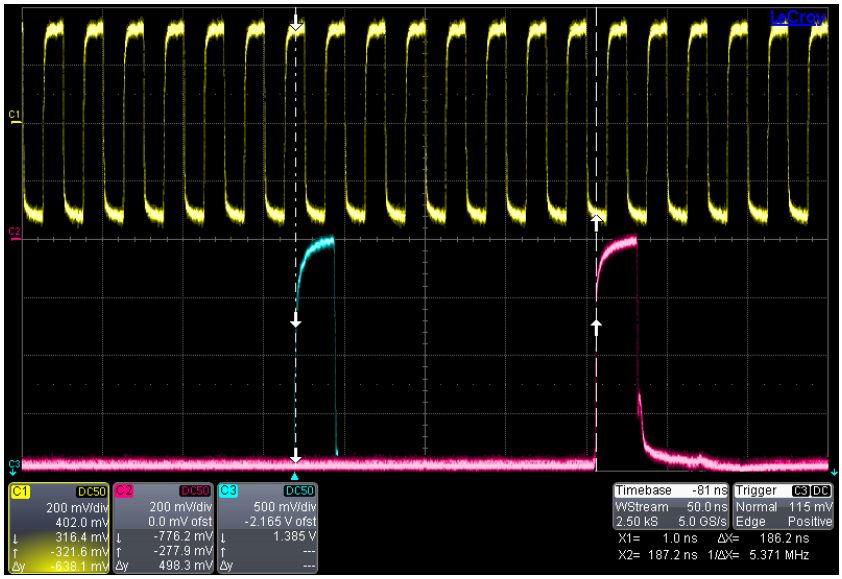


Figure 5.28 – RPC to MUCTPI latency measurement waveform

Figure 5.29 shows the oscilloscope acquisition waveform used to measure the latency between the TGC sector logic module to the MUCTPI. The sector logic module asserts a flag (oscilloscope channel 2) when the 128-bit word associated with BCID 0 is sent in the 40MHz clock domain logic. When the same 128-bit word is received, the MUCTPI asserts a second flag (oscilloscope channel 4). Approximately 5 ns have to be deducted from the measured value to compensate for the combinatorial delay in the TRP FPGA. In addition, 5 $m \times 5\ \frac{ns}{m} = 25$ ns should be deducted from the measured value, for the latency in the optical fibres. Therefore the latency is ≈ 112 ns, which corresponds to $\approx 4.5\ T_{BC}$ clock period.

In both cases, the phase of transmitter and receiver flags kept unchanged after resetting and power cycling both systems. The measured latency values are within the expected latency intervals defined in Equations (5.18) and (5.19).

5.7 Summary

This chapter described the synchronization and alignment unit of the MUCTPI. The requirements have been presented in Sections 5.2 and 5.3, followed by the description on the implemented FPGA firmware in Section 5.4.

The synchronization IP has been investigated in detail using a comprehensive functional simulation, presented in Section 5.5. This functional simulation has been implemented to: First, check the design for errors. Second, elaborate on the alignment delay procedure. Third,

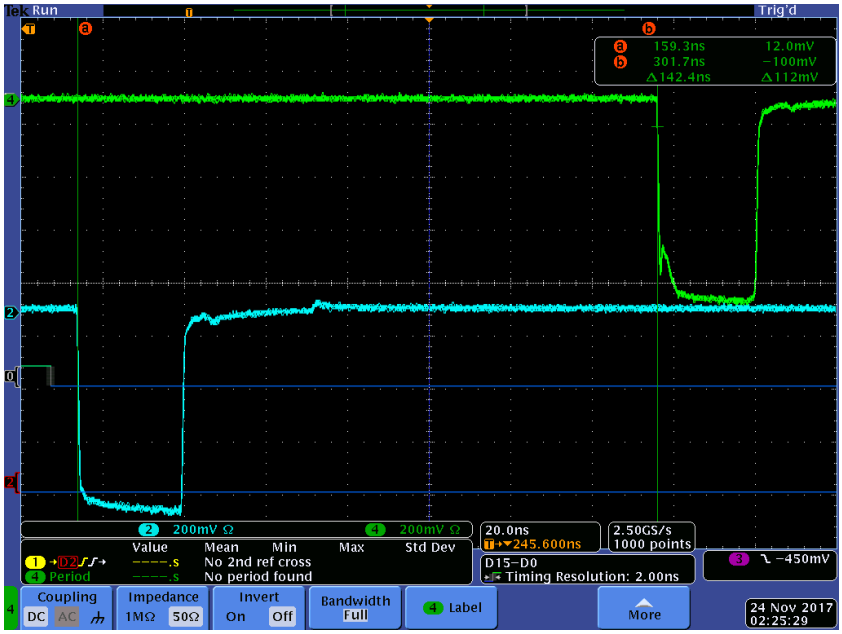


Figure 5.29 – TGC to MUCTPI latency measurement waveform

measure the minimum, and maximum latency read pointer offsets. Fourth, measure the error-free latency variation limits. Finally, measure the minimum and maximum synchronization latency values.

In the integration tests with RPC and TGC sector logic modules, presented in Section 5.6, it has been demonstrated that the synchronization IP kept error-free operation after resetting and power cycling both systems. The CRC error count has been checked during an overnight test, without errors. The measured latency is compatible with the simulated values and fits in the allocated latency budget for the data transfer.

Chapters 3 to 5 demonstrated that the MUCTPI can receive data with a low and fixed latency. Furthermore, it has been proved that the MUCTPI can receive and send trigger information reliably, i.e. with very low BER.

Data processing Part II

6 Data processing issues and challenges

The second part of this Ph.D. work, Data Processing, describes the muon candidate sorting firmware, one of the latency-critical algorithms of the MUCTPI. This algorithm is part of the trigger functionality of the MUCTPI, and it is implemented in the MSP FPGA. This chapter describes the context of the muon candidate sorting algorithm starting by describing the MSP firmware in Section 6.1 and the trigger unit in Section 6.2. Next, the sorting algorithm used in the MUCTPI for Run 2 is described in Section 6.3. The implementation is extended to a higher number of elements, and the implementation results are shown in Section 6.4. Finally, the summary of this chapter is given in Section 6.5.

6.1 Introduction

Figure 6.1 shows the block diagram of the MSP firmware with the trigger unit highlighted in yellow. The same firmware is used in both MSP FPGAs. In the top left, the reference clock and the 104 high-speed inputs, from the SL modules, are connected to the so-called Sector Logic Receiver. This unit implements the transceiver IP, described in Chapter 4, for all the 104 inputs. Next, the recovered clocks, data, and control symbols are connected to the so-called Sector Logic Interface, as well as the timing signals such as the system clock and the BCR. The Sector Logic Interface implements the synchronization IP also for all the 104 inputs. The synchronization IP transfers the input data from the recovered clock to the system clock domain for combined processing. For more details, please see Chapter 5.

Next, the output from the Sector Logic Interface, containing the data from all the 104 SL inputs, is connected to the trigger unit. There are 32 and 72 RPC and TGC inputs, respectively. In addition to the BCID, and global flags, each RPC and TGC input holds information from 2 and 4 muon candidates, respectively, see Chapter 5. Therefore, data from a total of 352 candidates, so-called *SL data*, flow from the sector logic interface to the trigger unit. The trigger unit

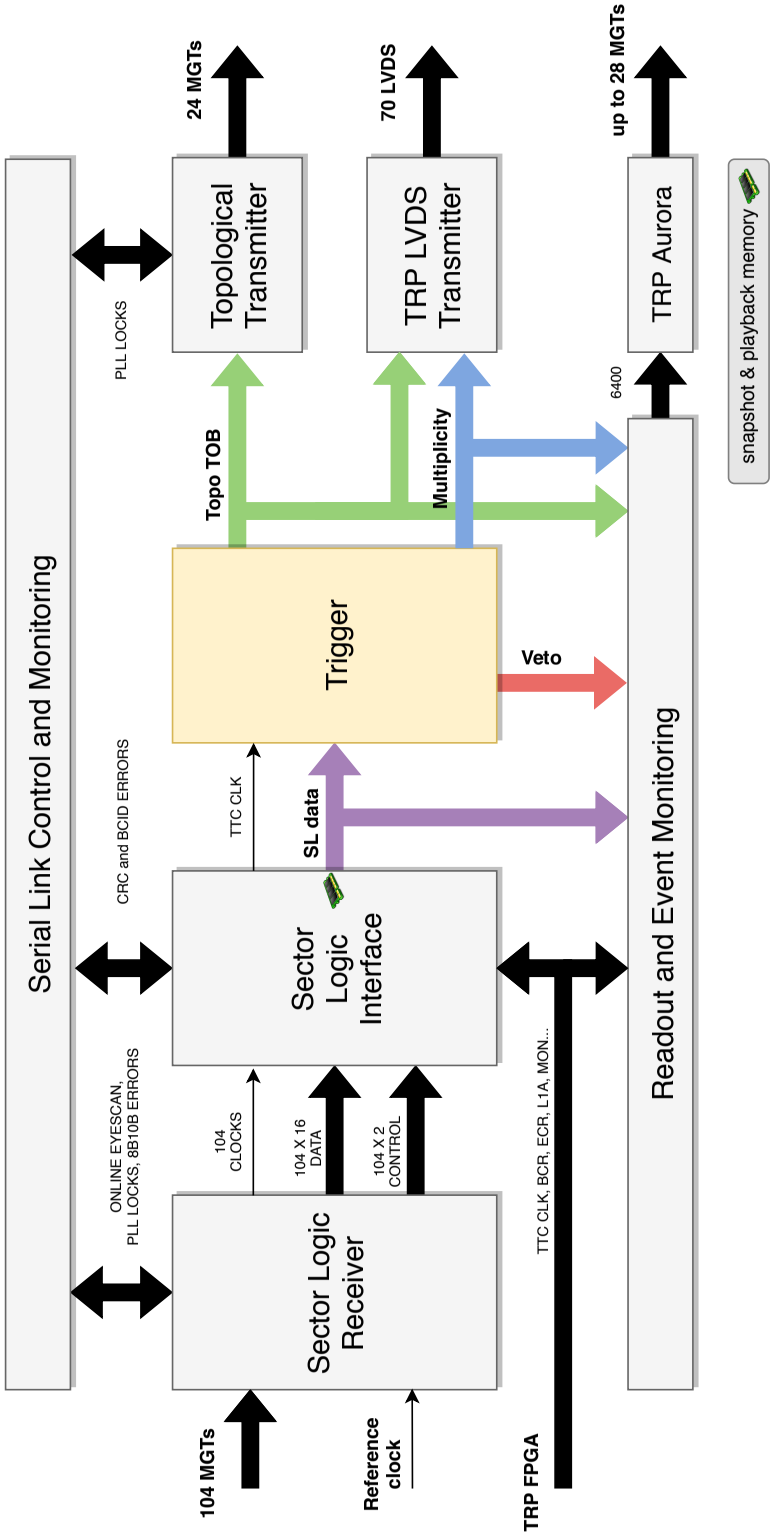


Figure 6.1 – MSP block diagram

computes the topological Trigger Object (TOB), so-called *Topo TOB*, the *Multiplicity* and the *Veto* flags. For more details on each of these signals and the trigger unit, please refer to Section 6.2.

The Sector Logic Interface features snapshot and playback on-chip memories for the *SL data* output, shown in purple. These memories enable in-system verification by storing snapshots of the data to memory and playing data back from memory to the data line. The same memory is used for both functions.

Data from the Sector Logic Interface and Trigger units are connected to the so-called Readout and Event Monitoring, Topological Transmitter, and TRP LVDS Transmitter. The Readout and Event Monitoring implements two readout interfaces. Each readout interface holds the *SL data*, the *Topo TOB*, the *Multiplicity* and the *Veto* flags until the L1A or the so-called MON signal arrives. MON stands for monitoring, and this readout path is used to capture data using a configurable trigger mechanism. This trigger mechanism can be configured to generate random triggers, for monitoring purposes, or to take snapshots of the captured data, for in-system verification.

The TRP Aurora unit sends data from both readout interfaces to the TRP FPGA. The data from the trigger readout interface of the two MSP FPGAs are combined in the TRP FPGA and sent to the HLT and DAQ systems. The data from the monitoring readout instance are written to external memory. The TRP Aurora unit implements a multi-lane high-speed interface to the TRP using the Xilinx Aurora 64B/66B IP [70]. The high-speed transmitter interface to L1Topo is implemented in the Topological Transmitter block. The TRP LVDS transmitter implements the low-latency LVDS interface to the TRP FPGA.

The Serial Link monitoring unit implements counters and registers containing monitoring information from the Sector Logic Receiver, Sector Logic Interface, and Topological Transmitter. Examples are PLL lock signals, 8b10b, CRC, and BCID errors. In addition, it also implements non-disruptively online serial link monitoring, which enables to measure the statistical eye-diagram from all the 104 inputs simultaneously, without disturbing the data transfer, i.e., during system operation.

Figure 6.2 shows one example of a statistical eye-diagram measured while the *SL data* are received. For more information on statistical eye-diagram, please refer to Chapter 3. The x-axis represents the time offset in UI, and the y-axis represents the voltage amplitude offset in mV. This eye-diagram has an excellent vertical and horizontal opening of 100% and 80%, respectively. The eye-diagram shown here is wider than the ones shown in Chapter 3 thanks to lower data-dependent jitter. The link characterization performed in Chapter 3 used PRBS-31 data, which contains longer sequences of 0s or 1s, compared to 8b10b encoded data.

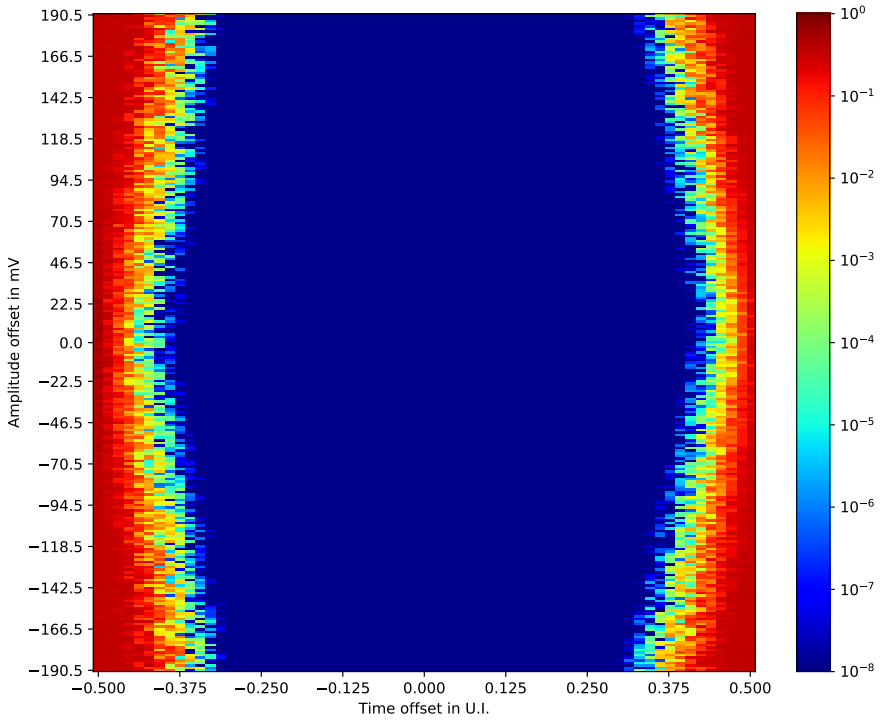


Figure 6.2 – Online serial link eye-diagram

6.2 Trigger unit

Figure 6.3 shows the block diagram of the trigger unit. The Overlap Handling and Masking units receive information from up to 352 muon candidates, at the bunch crossing rate. Notice that for every bunch crossing, most of the 352 inputs will actually be empty. Both units avoid double counting of muon candidate tracks that transverse more than one detector region. The Overlap Handling logic is implemented using pre-calculated results to indicate if a given pair of muon candidates are within an overlap region. If both candidates are within one overlap region, the candidate with the lowest p_T value is suppressed. The suppression is indicated by asserting the *Veto* flag of the suppressed candidate. The Overlap Handling unit supports every combination of overlapping trigger sectors from the same or adjacent regions within one half of the detector [71]. The front-end electronics handles overlap within the same trigger sector.

The masking unit replaces the p_T value to 0, at the bunch crossing rate when the *Veto* flag associated with a given muon candidate is asserted. Thus, only non-suppressed candidates have a valid p_T value, i.e. $p_T > 0$. After masking, all the resulting 352 muon candidates are sent to the sorting and multiplicity units.

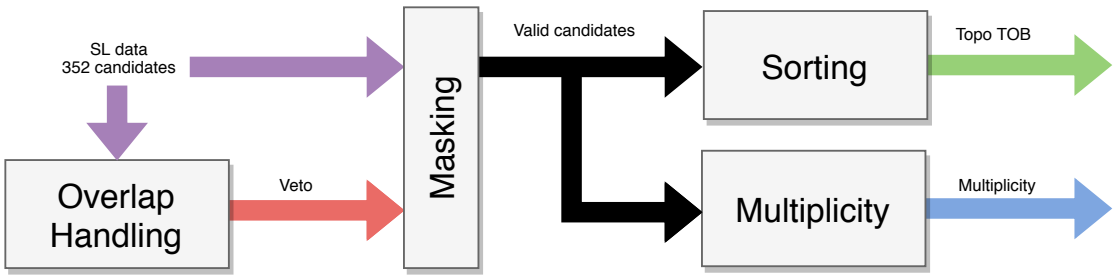


Figure 6.3 – MSP trigger block diagram

The sorting unit receives information from up to 352 muon candidates from the masking unit, at the bunch crossing rate. Again, the firmware always works on all possible candidates, but in most cases, most of them are empty. First, it sorts the candidate sector number according to the p_T value. Next, it outputs a sorted list containing complete information from up to 16 candidates with the highest p_T value, also at the bunch crossing rate. The sorted output list, *Topo TOB*, contains the sector number, flags, RoI, and the p_T value for each of the 16 selected muon candidates. The algorithm used for sorting the muon candidates in Run 2 is described in more detail in Section 6.3 because it represents the starting point for the research work described in Part II. The multiplicity summing unit counts up to 7 muon candidates for up to 32 p_T threshold values.

6.3 Sorting unit used in Run 2

As a starting point for the Run 3, the same sorting algorithm implemented in the MUCTPI for Run 2, as part of the author's master's thesis [12], has been investigated. The algorithm is divided into comparison, selecting, and multiplexing stages. The three stages are processed one after the other. They are described in Sections 6.3.1 to 6.3.3.

6.3.1 Comparison

The comparison implements, in parallel, every needed comparison to find the highest p_T candidate. It outputs a matrix with dimension $n \times n$, where n corresponds to the number of elements to be sorted. An example of five elements is shown in Table 6.1.

The cells in the diagonal compare a given element against itself, therefore it always evaluates to True. All the cells below the diagonal are derived from the associated cell above the diagonal. For example, $\overline{(p_{Ta} \geq p_{Tb})}$ is equivalent to a logic NOT of $(p_{Ta} \geq p_{Tb})$. Thus, comparators are required only to compute the elements above the matrix diagonal.

Table 6.1 – Comparison matrix for sorting five elements using a parallel processing approach

	a	**b**	**c**	**d**	**e**
a	$p_{Ta} \geq p_{Ta}$	$p_{Ta} \geq p_{Tb}$	$p_{Ta} \geq p_{Tc}$	$p_{Ta} \geq p_{Td}$	$p_{Ta} \geq p_{Te}$
b	$\overline{p_{Ta} \geq p_{Tb}}$	$p_{Tb} \geq p_{Tb}$	$p_{Tb} \geq p_{Tc}$	$p_{Tb} \geq p_{Td}$	$p_{Tb} \geq p_{Te}$
c	$\overline{p_{Ta} \geq p_{Tc}}$	$\overline{p_{Tb} \geq p_{Tc}}$	$p_{Tc} \geq p_{Tc}$	$p_{Tc} \geq p_{Td}$	$p_{Tc} \geq p_{Te}$
d	$\overline{p_{Ta} \geq p_{Td}}$	$\overline{p_{Tb} \geq p_{Td}}$	$\overline{p_{Tc} \geq p_{Td}}$	$p_{Td} \geq p_{Td}$	$p_{Td} \geq p_{Te}$
e	$\overline{p_{Ta} \geq p_{Te}}$	$\overline{p_{Tb} \geq p_{Te}}$	$\overline{p_{Tc} \geq p_{Te}}$	$\overline{p_{Td} \geq p_{Te}}$	$p_{Te} \geq p_{Te}$

The number of required comparators is given by the number of different combinations of 2 elements chosen from a set of n elements, as described in Equation (6.1).

$$c = \binom{n}{2} = \frac{n(n-1)}{2}, \tag{6.1}$$

where c corresponds to the number of comparators. For the example of five elements shown above, only ten comparators are required. For Run 2, 26 candidates were sorted, resulting in 325 comparators only. However, for the sorting unit of the upgraded MUCTPI, 352 muon candidates have to be sorted, so 61,776 comparators are needed. This represents an increase of almost a factor of 200 of the number of needed comparators.

6.3.2 Selection

The highest p_T value is found by checking every row of the resulting matrix from the comparison stage. The element with the highest p_T value is given by the row where every comparison result is True. The logical AND of the comparison results of each row are computed in parallel and assigned to an output vector with dimension n. This output vector flags the position of the candidate with the highest p_T value in one-hot encoding. One-hot encoding means that only one element of the vector is high, i.e., only the position associated with the highest p_T candidate.

A copy of the matrix is created, and all the results involving the highest p_T candidate are inverted. The process described above is repeated in order to find the second highest p_T candidate. It results in a second one-hot encoding vector representing the position of the second highest p_T candidate. This process is repeated until the sixteenth highest p_T candidate is found, resulting in 16 one-hot encoding vectors indicating the respective muon candidate position. From Run 2 to Run 3, the number of output candidates has increased from 2 to

16 muon candidates. This represents an increase factor of 8 times in the number of output vectors. The two following points should be noted:

1. A given output vector requires information from the previous vector to be computed. So they are computed one after the other and can not be implemented in parallel.
2. The size of the comparison matrix has been increased from 26×26 to 352×352. This represents an increase of almost a factor of 200 of the number of needed comparators.

6.3.3 Multiplexing

The selection output vector flags the position of the candidate with the highest p_T value in a one-hot encoding scheme. Sixteen one-hot multiplexors have been implemented to output the complete muon candidate information for each of the selected 16 candidates. Figure 6.4 shows, as an example, the implementation of a one-hot multiplexor. This multiplexor consists of the output connected to a logical OR between all the inputs, and an enabling flag.

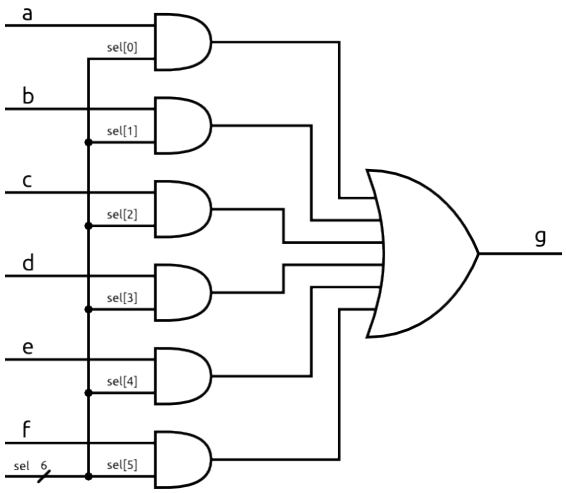


Figure 6.4 – Logic diagram for a 6-input one-hot multiplexor

6.4 Implementation results

The algorithm described in Section 6.3 has been synthesized for $n \in \mathbb{Z}^+ \mid 16 \leq n \leq 104$. For $n > 104$, synthesis did not finish after one week. For $n > 88$, routing was unsuccessful due to insufficient routing resources. This can happen with high routing congestion circuits.

Figure 6.5 shows the number of comparators and LUTs needed to implement the sorting unit described in Section 6.3. The Xilinx Vivado tool [62] has been used for synthesis and

implementation. The x-axis represents the number of elements n with $n \in \mathbb{Z}^+ \mid 16 \leq n \leq 104$. The left y-axis and the blue curve represent the number of comparators c for each value of n according to Equation (6.1). The right y-axis and the points in red represent the number of LUTs to synthesize the sorting unit. The design unit has been synthesized for the values of $n \in \mathbb{Z}^+ \mid 16 \leq n \leq 104$, incremented in steps of 8. The number of registers is not shown because it represents less than 1% of the available registers in the device for any value of n.

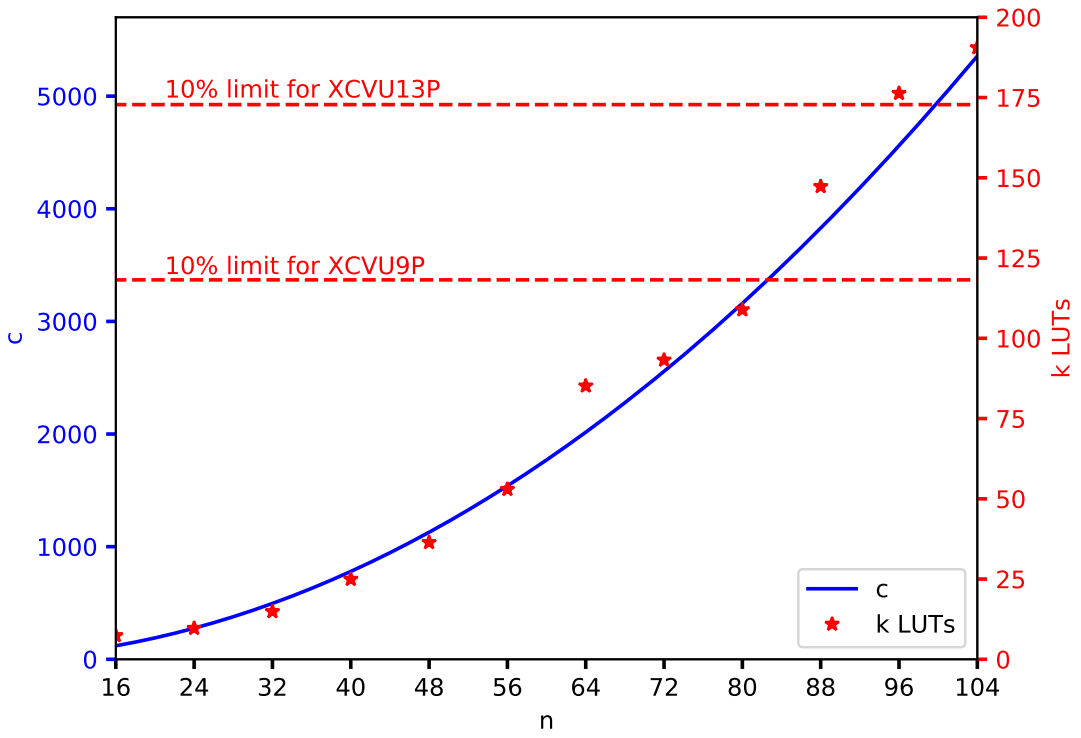


Figure 6.5 – Number of comparators and LUTs for up to 104 muon candidates

In order to guarantee that the remaining MSP FPGA functionality can be implemented, the logic resources available to the sorting unit has been limited to 10% of the device. The lower and upper horizontal dashed lines indicate the limit of 10% of the LUTs available in the Xilinx Ultrascale+ VU9P and VU13P FPGAs, respectively. The first device is the selected device for the MSP FPGA, and the second, for comparison only, is the largest pin-compatible device available in the UltraScale Architecture Migration Table [72].

Note that the number of LUTs is proportional to the number of comparators. The number of comparators increases proportionally to n^2. Despite synthesis has been unsuccessful for $n > 104$, prohibitive LUT usage values are already demonstrated for $n > 80$.

The value of the total combinatorial delay obtained after placing and routing, not shown in the plot, is also large. For instance, the smallest sorting unit, $n = 16$, takes 20 ns. The largest sorting unit that has been implemented with success, $n = 88$, takes 120 ns, which already represents 60 % of the total latency budget of the MUCTPI.

Finally, synthesis required a long time to be completed. Several days are needed for the sorting units with $n > 80$. Some of the synthesis stages required up to 100 GB of RAM memory. This usually is available only in high-performance computers. The values of LUTs, latency, and compilation time from the sorting algorithm described in Section 6.3 are very high. Therefore, not acceptable for the MUCTPI application.

6.5 Summary

This chapter described the data processing issues and challenges in the MUCTPI. Section 6.1 presented the MSP firmware, including the connectivity between the transceiver interface and synchronization IP, the result of the data transfer part of the thesis, to the trigger unit, where the muon candidate sorting unit is implemented. Section 6.2 covered the functionality of the sorting, overlap handling, and multiplicity units.

Section 6.3 described the sorting algorithm used for the MUCTPI in Run 2. It has been shown that the comparison and multiplexing stages process all the output paths in parallel. However, the selection stage processes each of the output selection vectors one after the other, i.e., in series instead of parallel. The selection stage has to run sequentially because, to find the N highest p_T muon candidate, all the comparison results involving the N-1 highest p_T muon candidate have to be inverted. As it can not be parallelized, this stage contributes predominantly to the total combinatorial delay.

Section 6.4 presented the implementation of the extension of the Run 2 muon candidate sorting algorithm to the required input and output values in Run 3. Although the implementation was unsuccessful for $n > 104$, the results from $n \leq 104$ are already sufficient to demonstrate that this sorting algorithm is unacceptable for the MUCTPI application in Run 3.

The next chapter describes sorting networks, the fastest practical method to sort data in hardware. The state-of-the-art is reviewed, and optimizations for the MUCTPI application are presented.

7 Sorting Networks

This chapter describes the state-of-art in sorting networks and the development of the MUCTPI sorting unit. Section 7.1 offers a briefly history of sorting networks. Section 7.2 describes how sorting networks are built, represented, and validated. Section 7.3 describes the Batcher merge-exchange sort algorithm, which later enabled the creation of the merging and mergesort algorithms from the same author, described in Sections 7.4 and 7.5, respectively. Section 7.6 presents special sorting networks for particular values of n that are faster than the respective Batcher sorting network. Section 7.7 describes different optimization techniques for sorting and merging networks. Section 7.8 presents a comparative study on the Batcher sorting methods concerning the delay and the number of comparisons. Section 7.9 describes the implementation of faster sorting networks using the divide-and-conquer principle. The selected sorting network for the MUCTPI is presented in Section 7.10. Section 7.11 describes the validation of the selected networks using the zero-one principle. Finally, Section 7.12 presents a summary of this chapter.

7.1 Introduction

In 1964, Batcher discovered the merge-exchange sorting algorithm, which is the first systematic exchange sorting algorithm based on simultaneous disjoint comparisons, i.e., several non-overlapping comparisons that can run in parallel [73, p. 111]. Four years later, he published the first systematic method to generate merging networks, which enabled the generation of new types of sorting networks [74]. The merging term means generating a sorted sequence from two-sorted sub-sequences. The Batcher methods are said to be systematic because they generate networks for any number of elements n.

Though Batcher sorting networks are not asymptotically optimal, they are the fastest practical methods to sort data in hardware [75]. An algorithm is asymptotically optimal when, for large

inputs, it performs at worst a constant factor worse than the best possible algorithm. Batcher merge-sorting networks require at most $O((\log n)^2)$ steps to sort n keys, while sorting networks must use at least $O(\log n)$ steps [73]. This means that either faster networks are possible or the lower-bound should be raised. In 1983, a paper described the so-called AKS networks that require $C \cdot \log n$ steps to sort n keys, where C is a large constant [76].

The exact value of C is unknown, but considering $C = 87$, AKS network outperforms Batcher merge-sorting network only when $n = 1.2 \times 10^{52}$ [77]. Knowing that there are about 3.6×10^{51} protons in the Earth planet, even if computer technology would ever advance to the point where each key can be stored in a single proton, a Batcher merge-sorting network is still faster than an AKS network for any system that can be built on Earth [75]. For this reason, it is said that the Batcher merge-sorting networks are faster than AKS networks in practice. Sorting networks are suitable to be implemented in hardware because non-overlapping comparisons belonging to the same step can be computed in parallel. Therefore the delay is proportional to the number of steps and not to the number of comparisons, while in single-threaded software implementation, the delay is proportional to the number of comparisons.

7.2 Introduction to merging and sorting networks

Merging and sorting networks belong to a broader class of networks known as *permutation networks*. Permutation networks consist of several instances of comparison-exchange modules having two inputs and two outputs. Figures 7.1 and 7.2 shows two types of comparison-exchange modules for ascending and descending order, respectively. These modules exchange the input values $\langle x_1, x_2 \rangle$ according to their comparison result. As sorting algorithms often sort data in ascending order, normally, the upper right port outputs the element with the minimum value, and the lower right port outputs the maximum element, as shown in Figure 7.1. However, in this thesis, the block shown in Figure 7.2 is preferred because the MUCTPI has to sort the input data in descending order. Therefore, this thesis adopts the convention that the upper and lower right ports output the element with the maximum and minimum values, respectively. If one needs to change the output order of the permutation network, only the comparison-exchange block needs to be replaced, i.e., the permutation network connectivity is kept unchanged.

The block diagram shown in Figures 7.1 and 7.2 are not suitable to describe networks for large values of n. Therefore, Knuth created a more concise way of describing permutation networks, so-called Knuth diagram [73]. All the Knuth diagrams and respective permutation networks presented in this thesis have been generated using the SNpy package [78]. SNpy is a Sorting Network Python package, created by the author of this thesis, for generating, optimizing, combining, plotting, and writing HDL and C description of permutation networks.

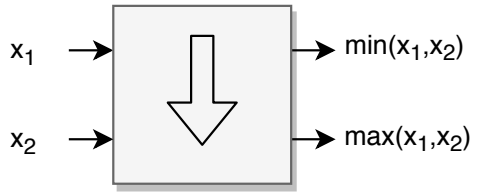


Figure 7.1 – Comparison-exchange module for ascending order output

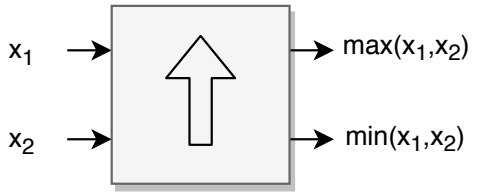


Figure 7.2 – Comparison-exchange module for descending order output

Figure 7.3 shows the Knuth diagram for a single comparator-exchange module. It consists of a horizontal line for each input and output. Each line is numbered according to the respective key index $\langle x_1, x_2, ..., x_n \rangle$. If a comparator-exchange module is required, two dots are placed at each of the respective input and output lines, and they are connected together using a vertical line. The two dots and the vertical line represents the comparison-exchange module shown in Figure 7.2. On the left side of the line, the pair of inputs enter the module. At the right side of the line, the two inputs are exchanged if the element 1 is lower than element 2, for the notation used in the thesis. The dashed vertical lines separate different steps. All the steps are identified with the respective step number at the top and bottom of the Knuth diagram between the two dashed lines. A step consists of all non-overlapping comparisons that can be computed simultaneously. When one comparison overlaps with another, they have to be implemented in different steps. Two comparisons are in overlap when at least one of the outputs of the first comparison is connected to the input of the second comparison.

Figure 7.4 shows the Knuth diagram for a 4-key sorting network. The working principle is the following: At stage 1, the pairs $\langle x_1, x_2 \rangle$ and $\langle x_3, x_4 \rangle$ are compared simultaneously. At stage 2, the highest element given by the pair $\langle x_1, x_2 \rangle$ at the key position x_1 is compared to the highest element of the pair $\langle x_3, x_4 \rangle$ at key position x_3. At this point, the highest element is already known and is placed at the key position x_1. Still in stage 2, the lowest element is obtained by comparing the lowest outputs from pairs $\langle x_1, x_2 \rangle$ and $\langle x_3, x_4 \rangle$ placed at key positions $\langle x_2, x_4 \rangle$. The lowest element is placed at the position of x_4. At stage 3, it only remains to compare the pair $\langle x_2, x_3 \rangle$ to find the second and third-highest elements. Notice that at least three stages are needed to sort four inputs without having overlapping comparisons.

7.2.1 Zero-one principle

The order of the stages of a given network matters and can not be changed. For instance, if in the 4-key sorting network, described in Figure 7.4, the stage order is changed from (1,2,3) to (3,1,2), and the input vector (0,1,0,1) is connected to the inputs $\langle x_1, x_2, x_3, x_4 \rangle$. The altered network outputs the vector (1,0,1,0) instead of (1,1,0,0).

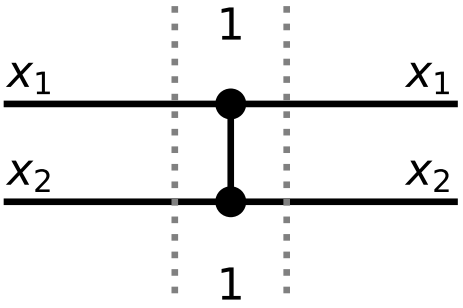


Figure 7.3 – Single comparison

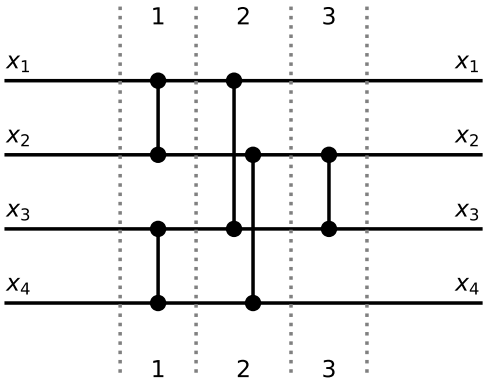


Figure 7.4 – 4-key sorting network

Notice that if the altered 4-key sorting network is used to sort large integers, one can already demonstrate that the network does not always sort using only a sequence of 0s and 1s. It means that it is not necessary to test all the combinations of large integers to demonstrate that this altered 4-key network is not a sorting network. In fact, the zero-one principle states that if a network with n elements sorts all 2^n sequences of 0s and 1s, it will sort any arbitrary sequence of n numbers [73, p. 223]. Depending on the data width, the zero-one principle reduces by a huge factor the number of combinations that have to be tested before validating a permutation network.

The zero-one principle is very important for constructing and validating sorting networks. It is often used to validate the entire sorting network. In addition, while constructing a new network, it is also used to demonstrate that a range of elements in a given stage is sorted before being connected to the next stage.

7.3 Batcher merge-exchange sorting algorithm

Batcher understood that in order to have an exchanging algorithm able to run faster than order $O(n^2)$, comparison-exchanges between nonadjacent pairs of keys need to be selected [73, p. 110]. In 1964, he discovered the merge-exchange sorting algorithm [74, p. 111], which is described in Procedure 7.3.1.

Figure 7.5 shows the Knuth diagram of the 8-key sorting network built from the comparison-exchange operations given by Procedure 7.3.1. Table 7.1 shows the values of p, q, r, and d

Procedure 7.3.1 – Batcher merge-exchange sorting algorithm

Let $\langle x_1, ..., x_n \rangle$ be the keys of the vector to be sorted, and $\langle k_1, ..., k_n \rangle$ be their values. Assuming $n \geq 2$.

1. **Initialize p.** Set $p \leftarrow 2^{t-1}$, where $t = \lceil \log_2 n \rceil$. Notice that steps 2 through 5 is performed for $p = 2^{t-1}, 2^{t-2}, ..., 1$.
2. **Initialize q, r, and d.** Set $q \leftarrow 2^{t-1}, r \leftarrow 0, d \leftarrow p$.
3. **Loop on i.** For i in $0 \leq i < n - d$ and $i \wedge p = r$, do step 4. $i \wedge p$ represents the bit-wise and of the binary representations of i and p.
4. **Compare and exchange.** If $k_{i+1} > k_{i+d+1}$, interchange $x_{i+1} \leftrightarrow x_{i+d+1}$.
5. **Loop on q.** If $q \neq p$, set $d \leftarrow q - p$, $r \leftarrow p$, $q \leftarrow q/2$, and return to step 3.
6. **Loop on p.** Notice that at this point $\langle k_1, ..., k_n \rangle$ is p-ordered. This means that it consists of p sorted vectors. Set $p \leftarrow \lfloor p/2 \rfloor$. If $p > 0$, go back to step 2.

during the execution of Procedure 7.3.1. Columns from 1 to 6 corresponds to each iteration of Procedure 7.3.1 and to each stage of Figure 7.5.

Table 7.1 – Values of p, q, r, and d for each stage or iteration of the mergesort algorithm for N=8

	1	**2**	**3**	**4**	**5**	**6**
p	4	2	2	1	1	1
q	4	4	2	4	2	1
r	0	0	2	0	1	1
d	4	2	2	1	3	1

The Batcher merge-exchange sorting algorithm sorts n elements essentially by sorting $\langle x_1, x_3, x_5, ... \rangle$ and $\langle x_2, x_4, x_6, ... \rangle$ independently, with values of $p > 1$. Then, steps 2 through 5 are executed with $p = 1$ in order to merge the two sorted sub-sequences together.

The stages 1, 2 and 3, in Figure 7.5, sort the odd and even sequences by implementing Procedure 7.3.1 with $p = 4, 2, 2$, respectively. Stages 4, 5, and 6 merge the two sub-sequences into a sorted sequence by implementing Procedure 7.3.1 with $p = 1, 1, 1$.

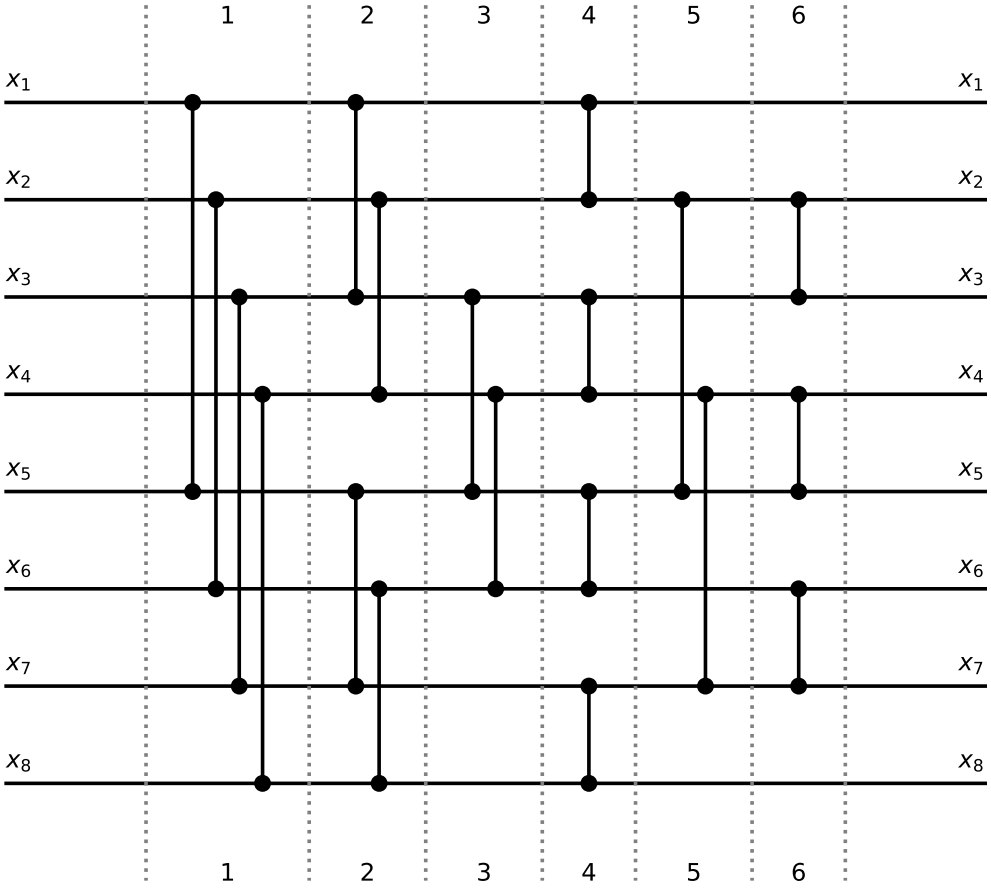


Figure 7.5 – Knuth diagram of the Batcher merge-exchange sorting network with $n = 8$

Equation (7.1) describes the number of stages d^s, also known as the network delay, of the merge-exchange sorting network for power-of-two values of n [74, p. 231], where $n = 2^p$.

$$d^s(2^p) = \begin{cases} \frac{1}{2}p(p+1), & \text{if } p \geq 1 \\ 0, & \text{otherwise} \end{cases} \tag{7.1}$$

The recursive Equation (7.2) [73, p. 226] describes the number of comparison-exchanges modules $c^S(n)$ to implement the merge-exchange sorting algorithm for n elements. Opposed to Equation (7.1), Equation (7.2) is also valid for non-power-of-two values of n.

$$c^S(n) = \begin{cases} c^S(\lceil n/2 \rceil) + c^S(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil, \lfloor n/2 \rfloor), & \text{if } n \geq 2 \\ 0, & \text{otherwise,} \end{cases} \tag{7.2}$$

where $C(m,n)$ represents the number of comparators needed to merge the sub-sequences with lengths m and n, which is given by the recursive Equation (7.3) [73, p. 224].

$$C(m,n) = \begin{cases} C(\lceil m/2 \rceil, \lceil n/2 \rceil) + C(\lfloor m/2 \rfloor, \lfloor n/2 \rfloor) + \lfloor (m+n-1)/2 \rfloor, & \text{if } m \cdot n > 1 \\ mn, & \text{otherwise} \end{cases} \tag{7.3}$$

7.4 Batcher odd-even and bitonic merging networks

Procedure 7.4.1 describes the so-called Batcher (m,n) odd-even merging network [73, p. 122] and [74]. This network is a generalisation of the final merging step, i.e. with $p = 1$, of the Procedure 7.3.1. Opposed to Procedure 7.3.1 that merges two sorted sub-sequences of the same length, when $p = 1$, Procedure 7.4.1 merges two sorted sub-sequences with any length values m and n. Procedure 7.4.1 is executed recursively, i.e., it is invoked again with a new value of m, and n every time the merge instruction, written in bold, is invoked in step 1.

Figure 7.6 shows the Knuth diagram of the Batcher ($m = 4$, $n = 4$) odd-even merging network. Notice that this network is equivalent to the network stages 4, 5, and 6 shown in Figure 7.5 except on the order of the inputs. In the merge-exchange algorithm with $n = 8$, the two sorted sub-sequences at the end of stage 3 are presented interleaved one to another. The first sorted sub-sequence is $\langle x_1, x_3, x_5, x_7 \rangle$, and the second is $\langle x_2, x_4, x_6, x_8 \rangle$. However, in the Batcher ($m = 4$, $n = 4$) odd-even merging network, the two sorted sub-sequences are presented one after the other. The first sorted sub-sequences is $\langle x_1, x_2, x_3, x_4 \rangle$, and the second is $\langle y_1, y_2, y_3, y_4 \rangle$.

Equation (7.1) describes the delay $d^M(n)$ for the Batcher (m, n) odd-even merging network with $m \leq n$. The recursive Equation (7.3) describes the number of compararison-exchange modules $C(m,n)$ to merge the sub-sequences with lengths m and n.

$$d^M(m,n) = \begin{cases} 1 + \lceil \log_2 \max(m,n) \rceil, & \text{if } m \cdot n \geq 1 \\ 0, & \text{otherwise} \end{cases} \tag{7.4}$$

Procedure 7.4.1 – Batcher (m,n) odd-even merging network

- If $m = 0$ or $n = 0$, the network is empty. If $m = n = 1$, the network is a single comparison-exchange module.
- If $m \cdot n > 1$, let the sequence to be merged be $\langle x_1, ..., x_m \rangle$ and $\langle y_1, ..., y_n \rangle$.
  1. **Merge** the odd sequences $\langle x_1, x_3, ..., x_{2\lceil m/2 \rceil - 1} \rangle$ and $\langle y_1, y_3, ..., y_{2\lceil n/2 \rceil - 1} \rangle$, obtaining the sorted result $\langle v_1, v_2, ..., v_{\lceil m/2 \rceil + \lceil n/2 \rceil} \rangle$; and **merge** the even sequences $\langle x_2, x_4, ..., x_{2\lfloor m/2 \rfloor - 1} \rangle$ and $\langle y_1, y_3, ..., y_{2\lfloor n/2 \rfloor - 1} \rangle$, obtaining the sorted result $\langle w_1, w_2, ..., w_{\lfloor m/2 \rfloor + \lfloor n/2 \rfloor} \rangle$.
  2. Apply the comparison-exchange operations

     $$w_1 : v_2, w_2 : v_3, w_3 : v_4, ..., w_{\lfloor m/2 \rfloor + \lfloor n/2 \rfloor} : v^*$$

     to the sequence

     $$\langle v_1, w_1, v_2, w_2, v_3, w_3, v_{\lfloor m/2 \rfloor + \lfloor n/2 \rfloor}, w_{\lfloor m/2 \rfloor + \lfloor n/2 \rfloor}, v^*, v^{**} \rangle$$

     Here $v^* = v_{\lfloor m/2 \rfloor + \lfloor n/2 \rfloor + 1}$ does not exist if both m and n are even, and $v^{**} = v_{\lfloor m/2 \rfloor + \lfloor n/2 \rfloor + 2}$ does not exist unless if both m and n are odd.

Batcher has devised another type of merging network, so-called bitonic merging [73, p. 230], which lowers the delay time $d^B(m,n)$ at the price of more comparators. Equation (7.5) [73, p. 228] describes the $d^B(m,n)$ for a (m,n) bitonic merging network, and Equation (7.6) [73, p. 231] describes the number of comparison-exchange operations. The procedure for building bitonic merging network is available in [73, p. 230].

$$d^B(m,n) = \begin{cases} \lceil \log_2(m+n) \rceil, & \text{if } m \cdot n \geq 1 \\ 0, & \text{otherwise} \end{cases} \tag{7.5}$$

$$c^B(n) = \begin{cases} c^B(\lceil n/2 \rceil) + c^B(\lfloor n/2 \rfloor) + \lceil n/2 \rceil, & \text{if } n \geq 2 \\ 0, & \text{otherwise} \end{cases} \tag{7.6}$$

Bitonic merging is optimum, in the sense that no parallel merging method based on simultaneous disjoint comparisons can sort in fewer than $\lceil \log_2(m+n) \rceil$ steps [73, p. 231] and [74].

Notice that when $m = n$ and n is power of two, the delay for bitonic and odd-even merging is the same. Therefore, odd-even merging is also optimum in this condition, i.e., merging power-

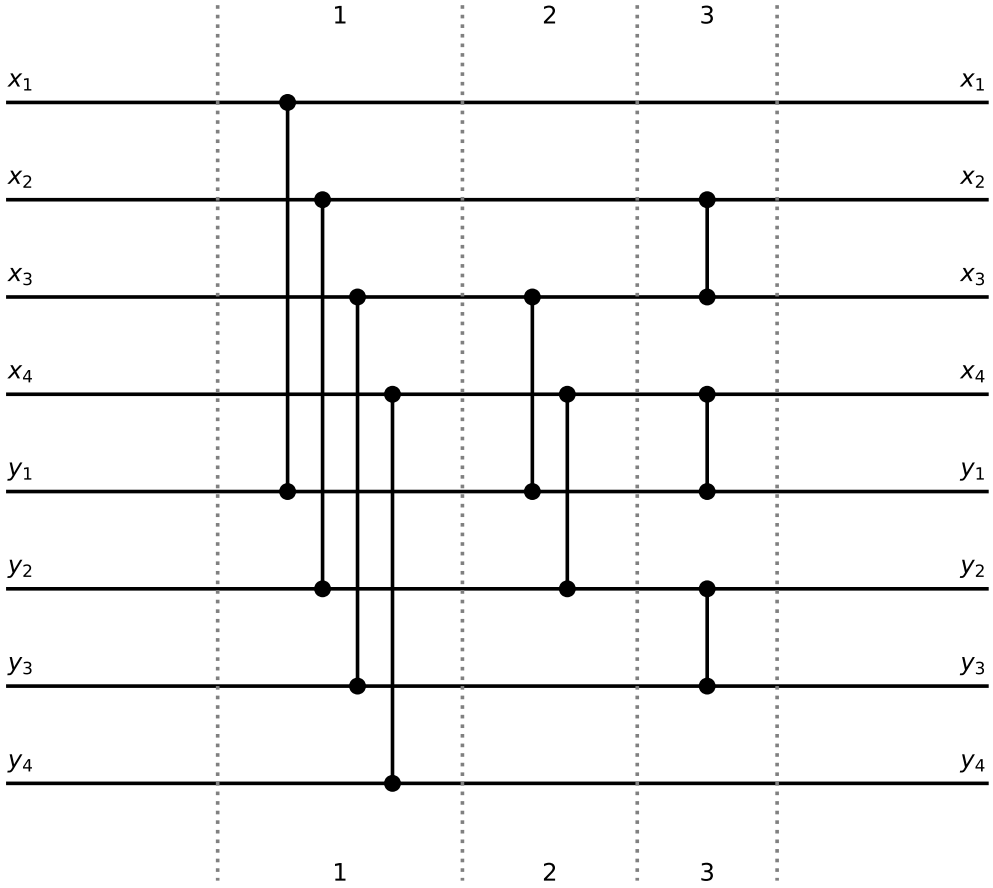


Figure 7.6 – Knuth diagram of the Batcher ($m = 4$, $n = 4$) odd-even merging network

of-two subsequences of the same length. As in the MUCTPI there is no need for merging subsequences of different lengths, the odd-even merging networks are preferred because they have an optimum delay and require less comparison-exchanges modules, compared to the bitonic merging. The lower number of comparison operations results in reduced inter-connectivity, which translates to reduced routing congestion that enables lower latency.

7.5 Odd-even and bitonic mergesort networks

Both odd-even and bitonic merging networks can be used recursively to generate sorting networks. This technique is known as *sorting-by-merging scheme* [74]. The only condition is making sure that the two input sub-sequences of each merging network is sorted. This is achieved by merging recursively until reaching a ($m = 1$, $n = 1$) merging network featuring a single comparison-exchange module. The ($m = 1$, $n = 1$) merging network is special because it is also a sorting network, due to the fact that a sub-sequence of one element is always sorted.

For instance, attacking the issue backward and using $n = 8$: First, a ($m = 4$, $n = 4$) merging network merges two sub-sequences of 4 inputs, which are not known yet to be sorted. Then, each of these two subsequences of 4 elements is merged using two ($m = 2$, $n = 2$) merging networks. Finally, each sub-sequence of two elements are merged using two ($m = 1$, $n = 1$) merging networks. In view that the input sub-sequences of the ($m = 1$, $n = 1$) merging networks are sorted, the output of all ($m = 1$, $n = 1$) merging networks are also sorted. The same repeats for the outputs of ($m = 2$, $n = 2$) and ($m = 4$, $n = 4$) merging networks. Therefore, the recursive use of merging networks generate sorting networks. The sorting networks generated from merging networks are often refereed to as mergesort networks.

Figures 7.7 and 7.8 show the 8-key odd-even and bitonic mergesort networks, respectively. Notice that the network delay d^s is the same as the one obtained with the merge-exchange network described in Equation (7.1). However, the number of comparison exchange operations is increased for the bitonic mergesort.

Equations (7.7) and (7.8)[73, p. 226 and 231] describes the number of comparison-exchange operations for odd-even and bitonic mergesort networks, respectively, for n being power of two, i.e. $n = 2^p$.

$$c^M(2^p) = \begin{cases} (p^2 - p + 4)2^{p-2} - 1, & \text{if } p \geq 1 \\ 0, & \text{otherwise} \end{cases} \tag{7.7}$$

$$c^B(2^p) = \begin{cases} \frac{1}{4}p(p+1)2^p, & \text{if } p \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{7.8}$$

Notice that for $n = 8$, all the networks described here require six stages. The merge-exchange and odd-even mergesort networks require 19 comparison-exchange operations, given by Equations (7.2), (7.3) and (7.7). However, the bitonic mergesort network requires 24 comparison-exchange operations, given by Equation (7.8).

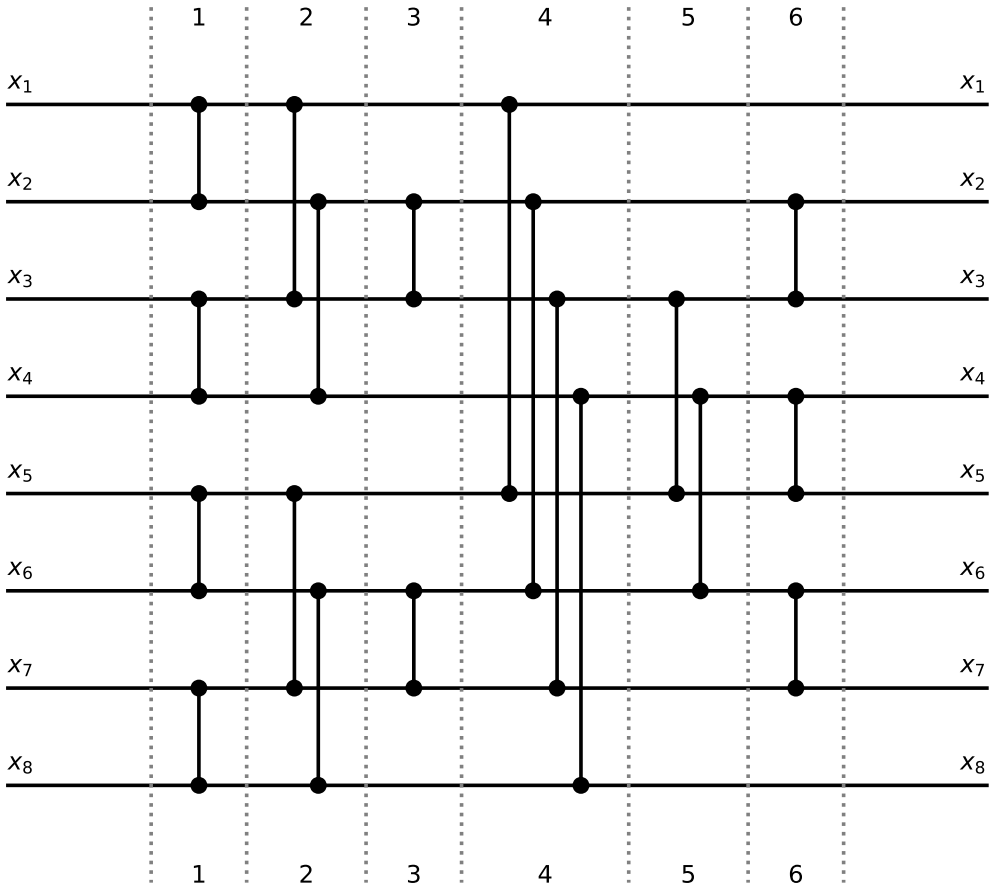


Figure 7.7 – Knuth diagram of the Batcher odd-even mergesort network with $n = 8$

Although bitonic mergesort requires more comparators than odd-even mergesort networks, it features modularity, i.e., a large network can be split up into several identical modules. For example, a 16-key bitonic mergesort network can be constructed from 8 4-key bitonic mergesort networks [74]. This is the reason why it is often used. However, the modularity property is not useful for the MUCTPI application because:

1. The required network can be implemented into a single FPGA module, i.e., it is not needed to split up the network into several FPGA devices.

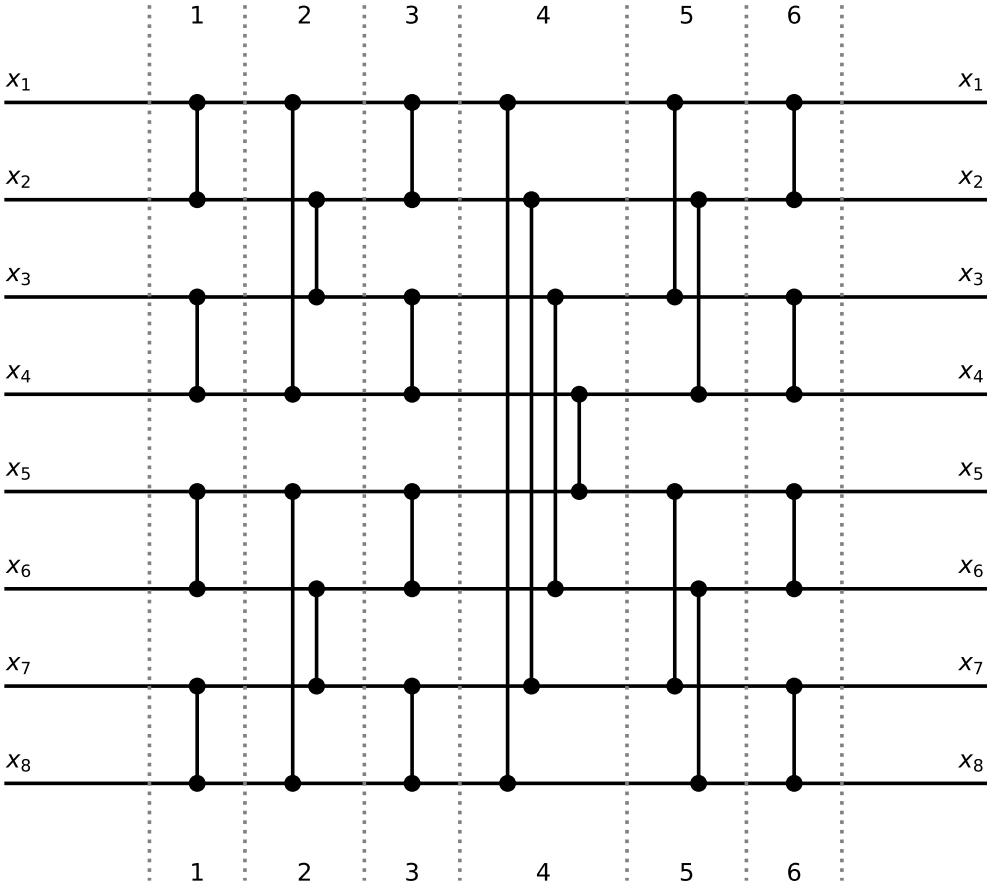


Figure 7.8 – Knuth diagram of the Batcher bitonic mergesort network with $n = 8$

2. As the number of elements n is fixed for the MUCTPI, the modularity property is not useful to reconfigure the sorting network for different values of n.

7.6 Special sorting networks

Since the 1950s, many researchers have been interested in designing either optimally fast or optimally efficient sorting networks. These researchers found optimally efficient sorting networks for up to 8 elements and optimally fast for up to 10 elements. Nobody seemed to

know how to design either optimally efficient or optimally fast sorting networks for larger networks. When Batcher discovered the odd-even and bitonic mergesort networks, it remained the question if they were optimally fast, optimally efficient or neither. The question remained open until David C. Van Voorhis discovered a 16-key network with nine steps, i.e., one step shorter than Batcher sorting networks.

Until today many authors investigate either optimally fast or optimally efficient sorting networks. In some cases, it took decades to prove the optimality of some sorting networks. This section describes two of these networks that have been discovered by David C. Van Voorhis in 1972, and Sherenaz W. Al-Haj Baddar in 2009.

7.6.1 David C. Van Voorhis 16-key sorting network

Figure 7.9 shows the Knuth diagram of the David C. Van Voorhis 16-key sorting network [73, p. 229]. It requires nine stages, one stage less than Batcher networks, and 61 comparison-exchange modules, two fewer than the Batcher merge-exchange or odd-even mergesort networks.

The delay optimality of this network remained unclear until 2014, when a group of authors published a paper [79] proving delay optimality for the networks with $11 \leq n \leq 16$ listed in [73, p. 229].

7.6.2 Sherenaz W. Al-Haj Baddar 22-key sorting network

Figure 7.10 shows the Knuth diagram of the Sherenaz W. Al-Haj Baddar 22-key sorting network [77]. It requires 12 stages, one fewer stage than the previously fastest 22-key sorting network known, and 116 comparison-exchange modules, only two more than Batcher merge-exchange network.

Although this is the fastest 22-key sorting network known, the delay optimality of this network remains unclear until the time this thesis has been written. The current lower bound for the delay of a 22-key network is set to 7 steps [75].

7.7 Network optimisations

Depending on each application, some of the comparison-exchange modules can be optimized away. Some of the fastest sorting networks known today have been discovered after optimizing away comparison-exchange modules from a larger network. This is the case, for example, of the 21-key sorting network generated from the Baddar 22-key sorting network.

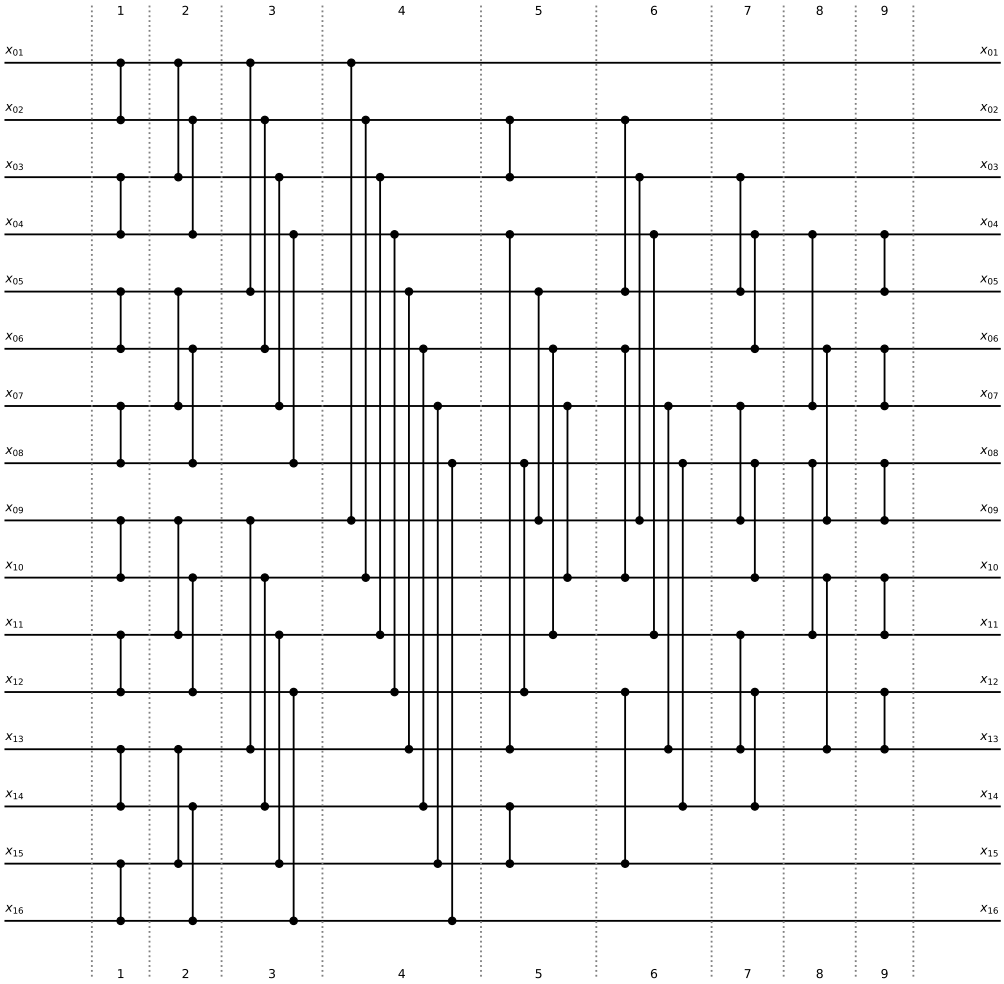


Figure 7.9 – Knuth diagram of the Voorhis 16-Key sorting network

This section describes two different types of sorting network optimizations that have been implemented in the SNpy package [78]. The first one optimizes away comparison-exchange modules from unused inputs and outputs. The second one optimizes away comparison-exchange modules with priory knowledge that a given set of inputs are already sorted, or a given set of outputs does not need to be sorted.

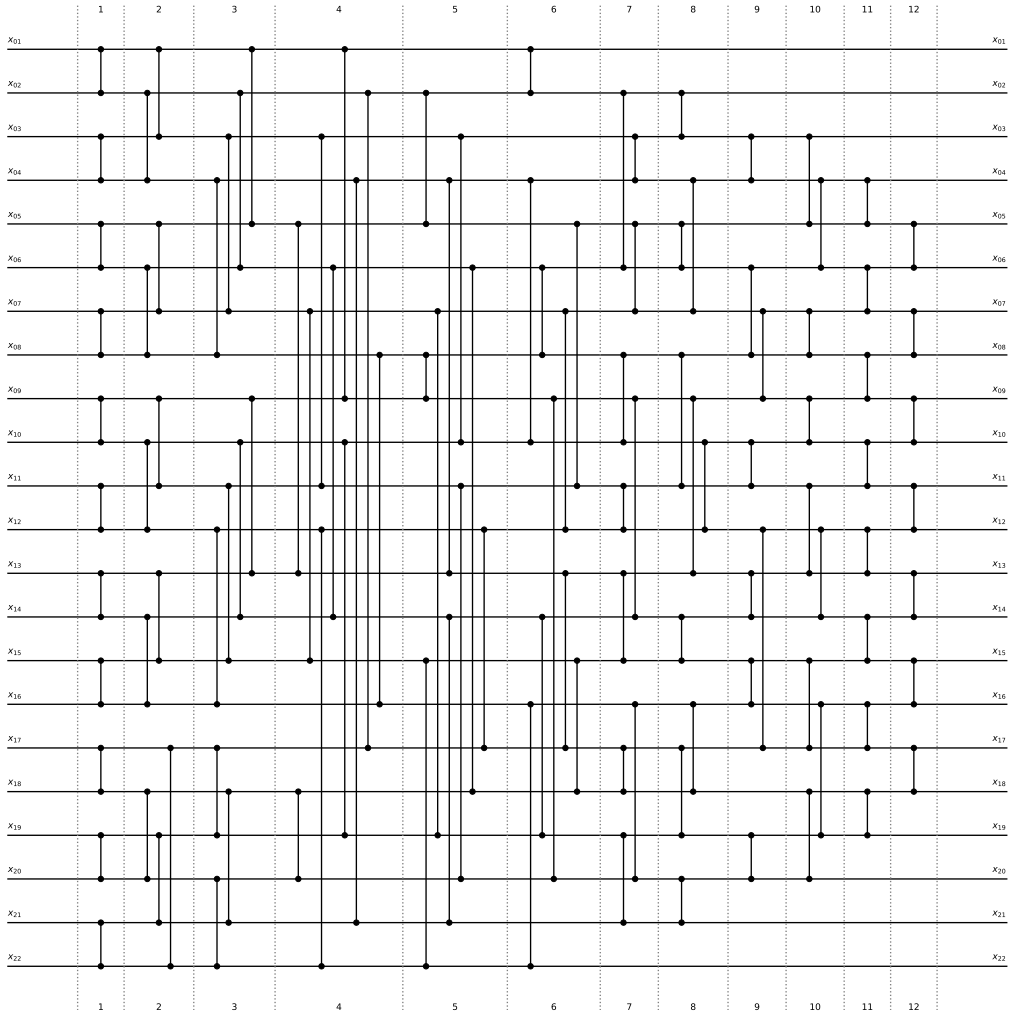


Figure 7.10 – Knuth diagram of the Baddar 22-Key sorting network

7.7.1 Input and output optimisation

Figure 7.11 shows the Knuth diagram of a 6-key sorting network generated from an 8-key Batcher odd-even mergesort network. The input optimization, i.e., the optimization in the number of elements, is performed by removing all the comparison exchange modules for which at least one of the inputs is driven by an unused element, shown in red. Every comparison-exchange module driven by one of the elements in $\langle x_7, x_8 \rangle$, i.e., 7 of them, also shown in red, have been optimized away. Although 7 out of 19 comparisons have been

removed, the delay remains unchanged because the remaining comparisons can not be reorganized into a reduced number of stages without causing overlapping comparisons.

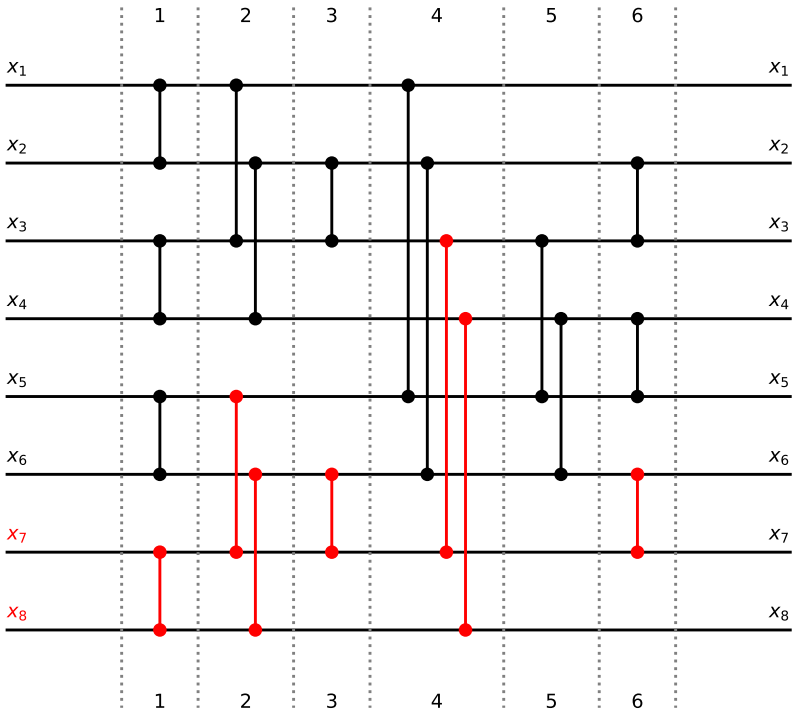


Figure 7.11 – Knuth diagram of a 6-key sorting network generated from a 8-key sorting network. Unused input elements and comparisons are shown in red

Figure 7.12 shows the Knuth diagram of an 8-key input 2-key output sorting network generated from an 8-key Batcher odd-even mergesort network. The output optimization is performed by removing every comparison-exchange module that is not needed to find the required elements at the output. In Figure 7.12, all the eight inputs are connected, but only the two highest elements are required. The other six outputs are shown in magenta. After optimization, four comparison-exchange modules, also shown in magenta, are removed.

7.7.2 Pre-sorted input and unsorted output optimisation

Figure 7.13 shows the Knuth diagram of a particular 8-key permutation network, derived from the 8-key Batcher odd-even mergesort network, having the following characteristics.

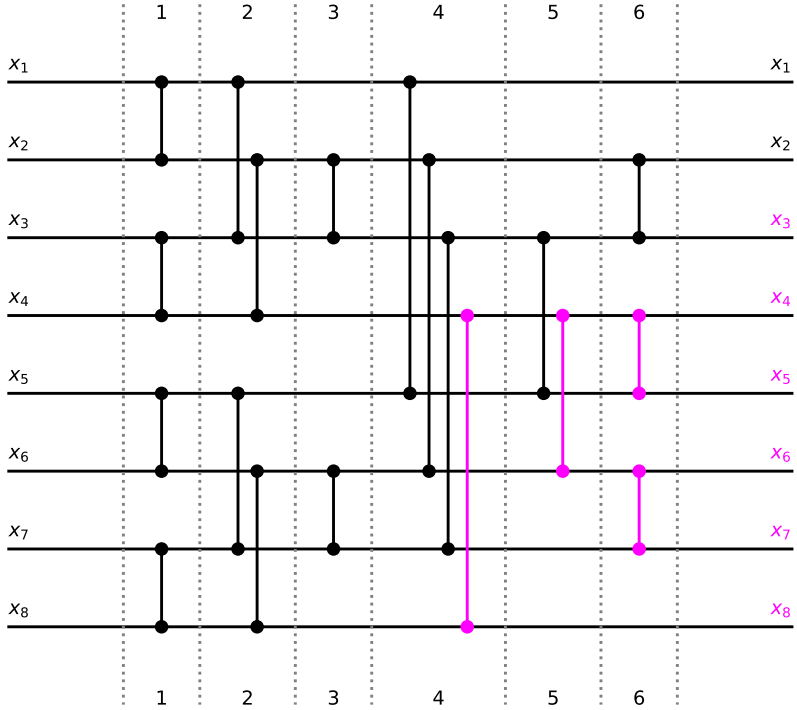


Figure 7.12 – Knuth diagram of 8-key input 2-key output sorting network. Unused output elements and comparisons are shown in magenta

- Pre-sorted inputs: the sub-sequences $\langle x_1, x_2, x_3, x_4 \rangle$ and $\langle x_5, x_6 \rangle$, shown in blue, are known to be already sorted.

- Unsorted outputs: the outputs $\langle x_2, x_3, x_4, x_5, x_6, x_7 \rangle$, shown in green, are not required to be sorted, i.e., only the highest and lowest element are read from the output.

The pre-sorted input optimization is performed by removing every redundant comparison in view that a given set of elements is known to be already sorted. This optimization is implemented by removing comparison-exchange modules belonging exclusively to the set of inputs, which are already sorted, starting from the first stage. If, in some point, one of the input elements interacts with another element that is not part of the pre-sorted input set, the first element is no longer considered part of the pre-sorted set. Consequently, comparison-exchange modules involving this element are no longer removed.

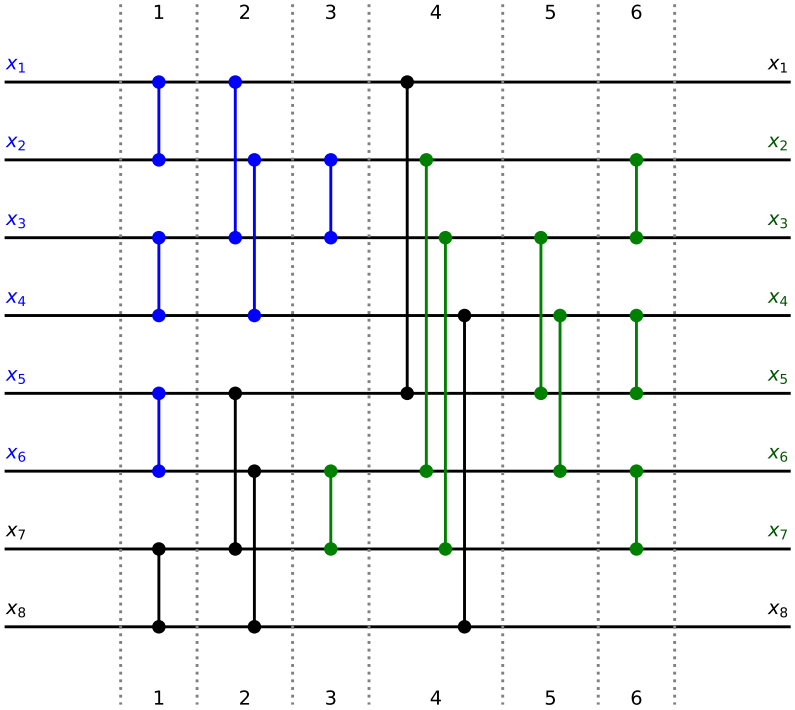


Figure 7.13 – Knuth diagram of a particular 8-key permutation network. Pre-sorted input elements and the respective removed comparisons are shown in blue. Output elements that do not need to be sorted and the respective removed comparisons are shown in green.

The non-sorted output optimization removes every redundant comparison-exchange module in view that a given set of outputs are not required to be sorted. This is implemented similarly to the input optimization, but progressing backward, starting from the last stage. Every comparison-exchange module belonging exclusively to the set of unsorted outputs are removed. If one of the unsorted outputs interact with another element that is required to be sorted at the output, the first element is no longer considered part of the non-sorted set, and consequently, comparison-exchange modules involving this element are no longer removed.

Notice that after pre-sorted input and unsorted output optimization, the resulting network shown in Figure 7.13 requires only three from the initial six stages. Only stages 1, 2, and 4 are needed, and stages 3, 5, and 6 can be completely removed.

7.8 Batcher sorting methods comparison

This section compares the delay and number of comparators for three different Batcher sorting methods. They are the merge-exchange method described in Procedure 7.3.1, and the odd-even and bitonic mergesort algorithms described in Section 7.5. All of the sorting networks and the comparative plots have been generated using the SNpy package.

7.8.1 Delay

Figure 7.14 shows the delay given by Batcher sorting networks. The x-axis represents the number of element n in a logarithmic scale of base 2. The number of elements is incremented in unitary steps with $n \in \mathbb{Z}^+ \mid 2^1 \leq n \leq 2^9$. The y-axis represents the delay d, in stages, extracted from the generated network. Notice that Equation (7.1) can not be used instead because it is valid only for power-of-two values of n.

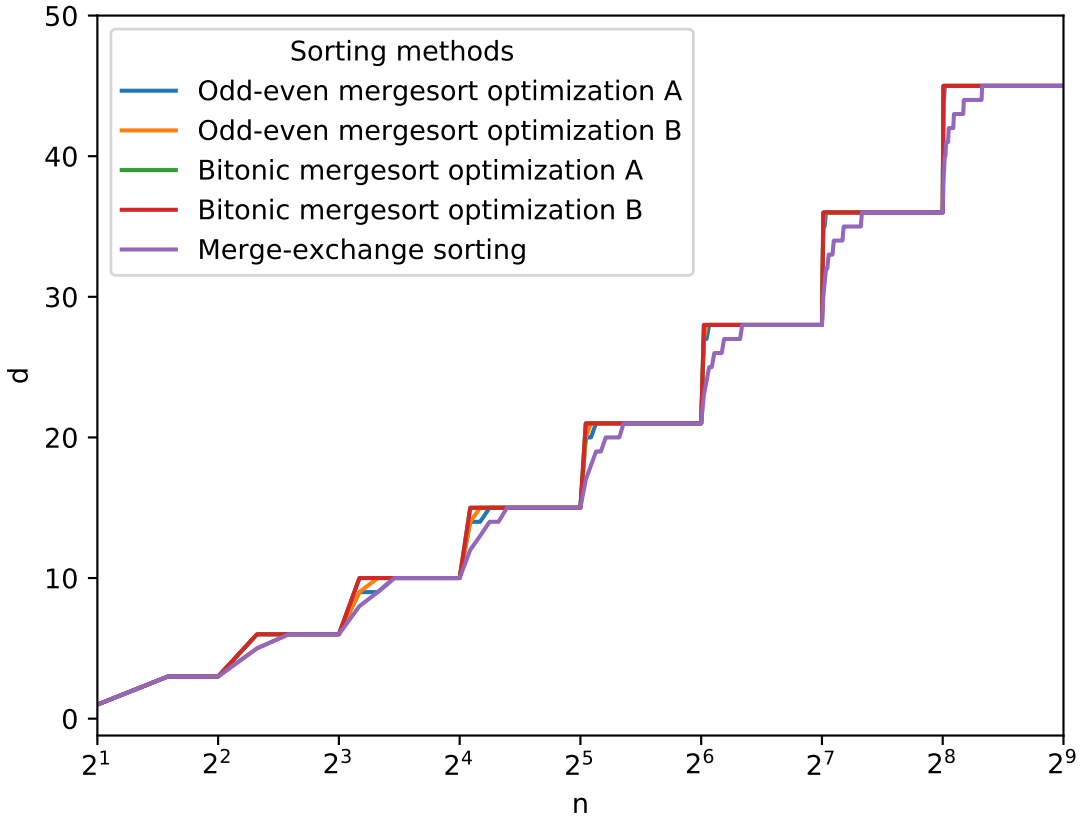


Figure 7.14 – Delay for Batcher sorting networks

The odd-even and bitonic mergesort networks have been generated using the sorting-by-merging scheme described in Section 7.5. The odd-even and bitonic mergesort networks with non-power-of-two input values of n have been derived from the respective larger network with the power-of-two value of n. The number of input elements has been reduced using the input optimization described in Section 7.7.1. Reducing the number of input elements also means reducing the number of output elements, because of $O \leq I$. For instance, the network with $n = 22$ has been generated from the network with $n = 32$ after removing ten input and output elements. The odd-even and bitonic mergesort networks with non-power-of-two input values have been reduced to the required size using the two following optimization options:

1. **Option A:** Removes top and bottom input lines. For instance, a 22-key sorting network is generated from a 32-key network removing the 5 top and bottom input lines. When the number of input lines to be removed is an odd number, an extra line is removed from the top when compared to the ones removed from the bottom.
2. **Option B:** Removes only the bottom input lines. For instance, a 22-key sorting network is generated from a 32-key network removing the ten bottom input lines.

The Batcher merge-exchange does not require any optimisation because it is also defined for non-power-of-two values of n. The following conclusions are extracted from Figure 7.14:

- **For power-of-two values of n:** The delay is the same for all the Batcher methods.
- **For non-power-of-two values of n:** The network generated from the merge-exchange algorithm provided the lowest delay value. The difference is more evident when $n \rightarrow 2^p$ from the right-hand side, i.e., for $n > 2^p$ and $n \ll 2^{p+1}$.
  - **For $n > 2^p$ and $n \ll 2^{p+1}$:** Odd-even mergesort outperformed the bitonic method regardless the optimization option.
    - **For odd-even merge sort:** Optimization option A outperforms option B.
    - **For bitonic merge sort:** Optimization options A and B have the same performance.

7.8.2 Number of comparisons

Figure 7.15 shows the number of comparisons required by Batcher sorting networks. The x-axis is represented in the same way as Figure 7.14. The y-axis represents the number of comparison c in a logarithmic scale of base 10. For all the sorting networks, the number of comparisons has been extracted from the respective generated network. The number of

comparisons for the merge-exchange sorting networks corresponds to the value defined in Equations (7.2) and (7.3) for all values of n. For odd-even and bitonic mergesort networks with power-of-two values of n, the number of comparisons corresponds to the value defined in Equations (7.7) and (7.8) respectively.

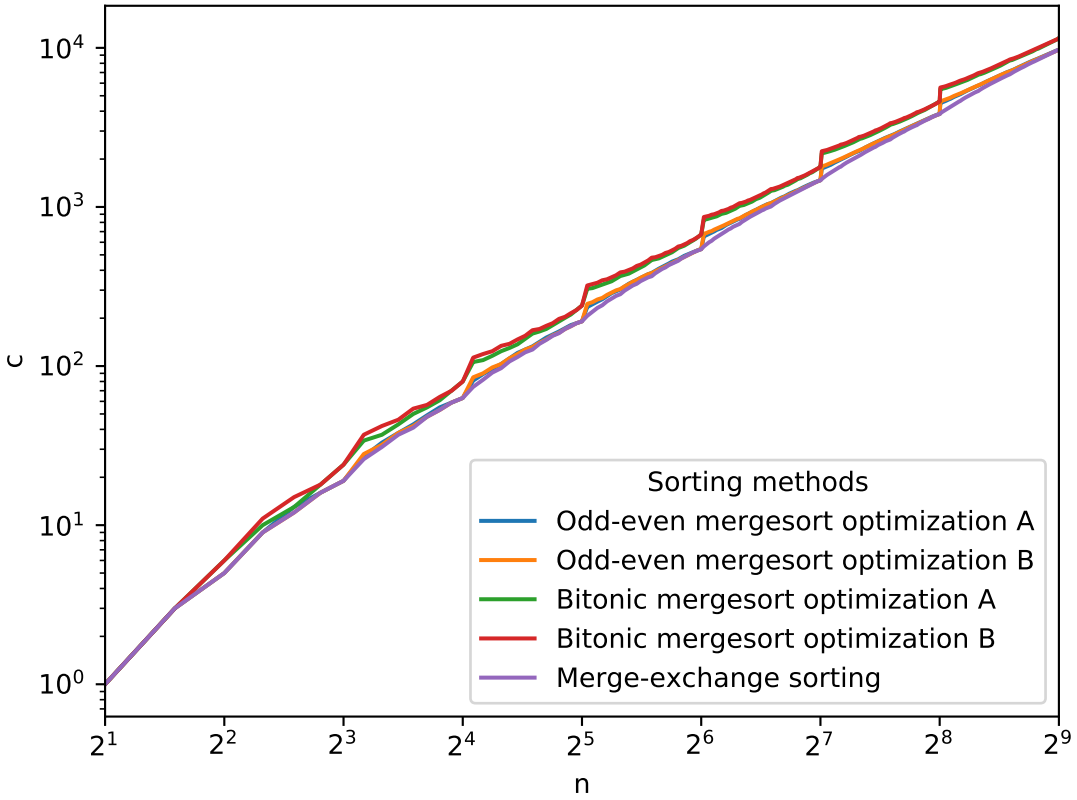


Figure 7.15 – Number of comparisons for Batcher sorting networks

The following conclusions are extracted from Figure 7.15:

- **For power-of-two values of n:**
  - **For $n = 2$:** The number of comparisons required by merge-exchange, odd-even and bitonic mergesort networks are the same.
  - **For $n > 2$:** The number of comparisons required by merge-exchange and odd-even mergesort networks are the same. Both outperforms the bitonic mergesort network.
- **For non-power-of-two values of n:** The network generated from the merge-exchange algorithm provides the lowest number of comparisons. Next, the odd-even outperforms

the bitonic mergesort network for any value of n. The performance difference between the merge-exchange and odd-even mergesort is more pronounced with $n > 2^p$ and $n \ll 2^{p+1}$.

- **For $n > 2^p$ and $n \ll 2^{p+1}$:** Optimization option A outperform option B for both odd-even and bitonic merge-sort networks.

7.8.3 Summary

The results from Figures 7.14 and 7.15 indicate that, within the Batcher sorting methods, the merge-exchange is preferred because it always gives the lowest delay value without requiring more comparators than any of the others Batcher methods. Alternatively, the bitonic mergesort method is preferred in applications that modularity is needed.

However, if special networks such as the ones described in Section 7.6 are taken into account, networks faster than the Batcher networks exist for some values of n. A summary of the fastest networks known for $2 \leq n \leq 32$ is presented in [75].

7.9 Divide-and-conquer method

Though the sorting networks presented so far are fast when the number of elements in the input I and output O are the same, the delay can be further minimized when $O \ll I$. In fact, it has been observed in this work, that splitting a large network, so-called top-level network, into smaller networks that outputs O elements, resulted in more efficient networks when $O \ll I$.

For instance, looking at the Batcher odd-even mergesort network with $n = 64$ constructed using the sort-by-merging scheme, the last merging network is of the type $(m = 32, n = 32)$. It means that before this point, the network sorted two sub-sequences of 32 elements. If one imagines that only 16 elements are required at the output, 16 elements of each of the two sub-sequences of 32 elements have been sorted without being needed to. Therefore, knowing that only 16 elements are needed at the output, the lowest 16 elements of each of these sub-sequences should be early-rejected, avoiding them to arrive in a later merging scheme.

For this same reason, if it is used alone, the technique of early rejecting inputs is preferred when compared to the output optimization, described in Section 7.7.1, when $O \ll I$. In some cases, splitting up the top-level network into smaller networks with $n = O$ is not possible or convenient. Hence, the technique presented here and in Section 7.7.1 are combined together. Several examples of the use of these two techniques combined together are presented in the current section.

As sorting networks are slower than merging networks, the sorting networks are used only to generate sub-sequences of length O. Next, in a second step, merging networks are used to generate a single sorted sequence from the sub-sequences generated by the sorting networks. An example using the number of input and output elements required by the MUCTPI is presented next.

Figure 7.16 shows the block diagram of a 352-key input 16-key output sorting network, i.e. $I = 352$, and $O = 16$, which is faster and more efficient than the Batcher merge-exchange network with $n = 352$. Four 88-key sorting networks are implemented to sort the 4 input sub-sequences $\langle x_1, x_2, ..., x_{88} \rangle$, $\langle x_{89}, x_{90}, ..., x_{176} \rangle$, $\langle x_{177}, x_{178}, ..., x_{264} \rangle$, $\langle x_{265}, x_{266}, ..., x_{352} \rangle$. Each of these networks is optimized to output 16 elements instead of the 88 input elements. Then, 3 ($m = 16, n = 16$) merging networks are connected together in a binary tree to get the 16 highest p_T elements sorted at the output. Similarly to the sorting networks, each of the merging networks is optimised to output 16 instead of the 32 input elements. Notice that the technique of early rejecting inputs and the output optimization are combined together.

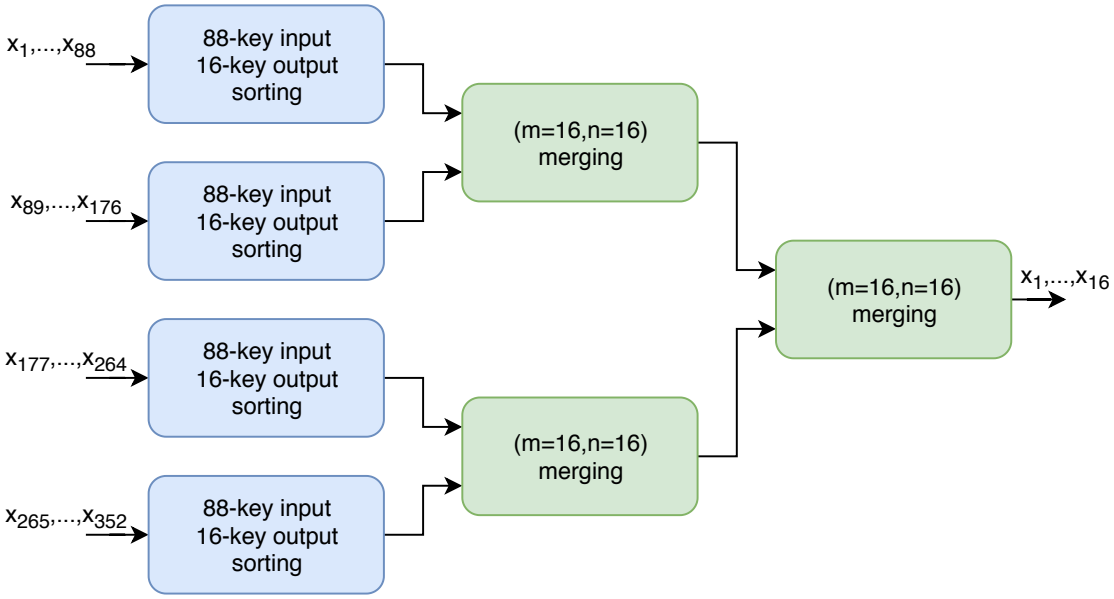


Figure 7.16 – Example of a 352-key input 16-key output sorting network block diagram

The same principle can be used to split up a sorting network into an arbitrary number of smaller sorting networks of the same size. This first processing stage, where sorting networks are used to generate sorted sub-sequences, is called the sorting part. The second processing stage, where each of the sorted sub-sequences is merged in order to obtain a single sorted sequence, is called the merging part.

The divide-and-conquer method requires generating, optimizing, and combining sorting and merging networks of different sizes for each of the several ways the networks can be combined together, defined here as implementation options. This extensive process has been implemented as part of the features of the SNpy package, in an effort to get early comparative complexity and performance results for each of the implementation options. This study accelerates the firmware development flow because, instead of implementing several options in hardware, only the selected option is implemented.

The divide-and-conquer principle study is implemented using the following three steps, for each of the implementation options:

1. The respective set of sorting and merging networks are generated using the algorithms described in Sections 7.3 to 7.5.
2. Both sorting and merging networks are optimized using the techniques described in Section 7.7.
3. The delay and the number of comparisons are extracted and combined together according to the required inter-connectivity for each implementation option.

Table 7.2 describes the 22 different options of implementing a 352-key input 16-key output sorting network dividing the input sequence into sub-sequences of the same length.

The sorting part is implemented by instantiating R instances of the Batcher merge-exchange sorting algorithm, in parallel, where R is defined in $R \in \mathbb{Z}^+ \mid 1 \leq R \leq L$, and L is described in Equation (7.9). The Batcher merge-exchange sorting algorithm has been selected because it is the Batcher sorting method with the lowest delay values, in particular for n not power-of-two, see Section 7.8.

$$L = \left\lceil \frac{I}{O} \right\rceil. \tag{7.9}$$

The upper limit L ensures that no elements are lost after dividing the top-level network into smaller sorting and merging networks. This is guaranteed by making sure that the length of each of the divided sub-sequences I_s, i.e., the number of input elements of each of the networks of the sorting part is always higher than or equal to the number of output elements O required by the top-level network. Equation (7.10) describes I_s in function of the constant I and the parameter R.

$$I_s = \left\lceil \frac{I}{R} \right\rceil, \tag{7.10}$$

Table 7.2 – 22 divide-and-conquer options for implementing a 352-key input 16-key output sorting network. R represents the number of input sub-sequences. The remaining columns are divided in sorting, merging, and total parts. I_s represents the length of each input sub-sequence. c_s and d_s represent the number of comparison and delay needed to sort each sub-sequence. C_s represent the total number of comparisons in the sorting part. l_m and i_m represent the number of levels and instances of merging networks. C_m and D_m represent the total number of comparisons and delay in the merging part. C and D represent the total number of comparisons and delay for sorting and merging parts together. The delay from the most efficient merging and sorting parts are highlighted in green and blue, respectively.

R	**Sorting part**				**Merging part**				**Total**	
	I_s	c_s	d_s	C_s	l_m	i_m	C_m	D_m	C	D
1	352	4446	45	4446	0	0	0	0	4446	45
2	176	1792	36	3584	1	1	48	5	3632	41
3	118	1014	28	3042	2	2	96	10	3138	38
4	88	726	28	2904	2	3	144	10	3048	38
5	71	534	26	2670	3	4	192	15	2862	41
6	59	407	21	2442	3	5	240	15	2682	36
7	51	348	21	2436	3	6	288	15	2724	36
8	44	288	21	2304	3	7	336	15	2640	36
9	40	250	20	2250	4	8	384	20	2634	40
10	36	216	19	2160	4	9	432	20	2592	39
11	32	174	15	1914	4	10	480	20	2394	35
12	30	164	15	1968	4	11	528	20	2496	35
13	28	150	15	1950	4	12	576	20	2526	35
14	26	138	15	1932	4	13	624	20	2556	35
15	24	122	15	1830	4	14	672	20	2502	35
16	22	111	15	1776	4	15	720	**20**	2496	35
17	21	104	15	1768	5	16	768	25	2536	40
18	20	96	14	1728	5	17	816	25	2544	39
19	19	90	14	1710	5	18	864	25	2574	39
20	18	82	13	1640	5	19	912	25	2552	38
21	17	74	12	1554	5	20	960	25	2514	37
22	16	63	**10**	1386	5	21	1008	25	2394	35

The number of comparators and delay for each sorting network are shown in the columns c_s and d_s, respectively. Both values are extracted from the generated network, after the number of output elements is reduced to O, using the output optimization described in Section 7.7.1. The number of comparisons before output optimization corresponds to the values in Equation (7.2) and Figure 7.15. The number of stages d_s remains unchanged after output optimization and corresponds to the values in Figure 7.14. The total number of comparators in the sorting part

C_s is given by the product $c_s \cdot R$. The total number of stages remains d_s because all the sorting networks are implemented in parallel.

The merging part is performed by implementing $R-1$ instances of the ($m = 16$, $n = 16$) odd-even merging network interconnected in a binary tree. The odd-even merging network is optimal when $m = n$ and n is power of two, see Section 7.4. The number of comparisons and stages for each merging network are extracted from the generated network after reducing the number of output elements from 32 to 16, using the output optimization described in Section 7.7.1. The number of comparisons and delay are given by the constants $c_m = 48$, and $d_m = 5$, respectively. The number of comparators, before output optimization, corresponds to the one given in Equation (7.3). The number of stages d_m remains unchanged after output optimization and corresponds to the one given in Equation (7.4).

The number of levels of merging networks l_m in the binary tree is given by $\lceil \log_2 R \rceil$. The number of instances i_m is given by $R-1$. The total number of stages of the merging networks of the binary tree D_m is given by $l_m \cdot d_m$, and the total number of comparators C_m is given by $i_m \cdot c_m$. Finally, summing up sorting and merging parts, the total number of comparators C is given by $C_s + C_m$, and the total number of stages D is given by $d_s + D_m$.

For $R = 1$, a single 352-key merge-exchange sorting network, after optimizing the output to 16 elements, sorts the input data alone, i.e., no merging part is required. In total, 4446 comparisons and 45 stages are needed. For $R = 2$, the top-level network is implemented using two 176-key merge-exchange sorting networks and one ($m = 16$, $n = 16$) odd-even merging network. The number of comparisons reduces to 3632 (18% lower), and the number of stages to 41 (9 % lower).

Notice that the value of l_m, and consequently D_m, increases with $\lceil \log_2 R \rceil$. For example, it takes the same 20 steps to merge 10 or 16 sorted sub-sequences, i.e. $R = 10$ and $R = 16$ respectively. Although the merging part is equally fast for both cases, the second case is more efficient because a higher number of sub-sequences are merged for the same value of D_m.

In general, increasing the value of R reduces both the number of comparisons and stages required. However, the fastest network is not necessarily the one with the highest R. The best results are given by a trade-off of the following two properties:

1. **The higher value of R, results in $I_s \rightarrow O$.** As higher, the R value is, faster is the sorting part.

2. **R is power-of-two.** The merging part is more efficient when R is power-of-two. The number of levels l_m of merging networks increases with $\lceil \log_2 R \rceil$.

The fastest sorting part is given when $R = 22$, which results in 22 instances of 16-key sorting networks with $d_s = 10$, highlighted in blue. The most efficient merging part that still has a high value of R is given by $R = 16$, which results in 4 levels of 5-step merging networks, resulting in $D_m = 20$, highlighted in green. Both configuration results in a total of 35 stages. Both implementation options, $R = \{16, 22\}$, are good candidates for the MUCTPI implementation. Luckily, there are special networks faster than the merge-exchange network for $I_s = \{22, 16\}$. In fact, the fastest sorting networks with $I_s = \{22, 16\}$ known in the literature have already been described in Section 7.6. Table 7.3 shows the two fastest divide-and-conquer options for implementing a 352-key input 16-key output sorting network. It uses the fastest sorting networks known in the literature for $R = \{16, 22\}$.

Table 7.3 – The two fastest divide-and-conquer options for implementing a 352-key input 16-key output sorting network. R represents the number of input sub-sequences. The remaining columns are divided in sorting, merging, and total parts. I_s represents the length of each input sub-sequence. Method represents the sorting method being used. c_s and d_s represent the number of comparison and delay needed to sort each sub-sequence. C_s represent the total number of comparisons in the sorting part. l_m and i_m represent the number of levels and instances of merging networks. C_m and D_m represent the total number of comparisons and delay in the merging part. C and D represent the total number of comparisons and delay for sorting and merging parts together. The fastest total delay is highlighted in green.

R	**Sorting part**					**Merging part**				**Total**	
	I_s	method	c_s	d_s	C_s	l_m	i_m	C_m	D_m	C	D
16	22	baddar22	113	12	1808	4	15	720	20	2528	**32**
22	16	voorhis16	61	9	1342	5	21	1008	25	2350	34

The 12-step Baddar 22-key sorting network used in Table 7.3 is three steps faster than the merge-exchange network used in Table 7.2, and the 9-step Voorhis 16-key sorting network is only one step faster. Using the Baddar 22-key sorting network reduces the overall delay of the implementation option $R = 16$ in 3 steps, resulting in a total delay value of $D = 32$, highlighted in green. The implementation option $R = 22$ reduces 1 step only. The number of comparators c_s is read after reducing the number of output elements from 22 to 16, using the output optimization described in Section 7.7.1. This result indicates that for a 352-key input 16-key output sorting network, the best implementation option is not given by the highest value of R, but instead, it is given by the highest power-of-two value of R, i.e., $R = 16$.

Section 7.10 describes in more detail the characteristics of the selected implementation option. The selected 32-step 352-key input 16-key output sorting network sorts the input data using 13 fewer steps than the 45-step 352-key Batcher merge-exchange, odd-even, or bitonic sorting networks. The delay reduction of 13 steps is given by the fact that the 45-step sorting network outputs 352 elements while the selected 32-step network output only the highest 16 elements required by the MUCTPI.

The total number of steps can be further reduced by optimizing away comparison-exchange modules from the first stages of the sorting networks using the pre-sorted input optimization, described in Section 7.7. This optimization is possible thanks to the fact that RPC and TGC sector logic modules send sorted sub-sequences of length 2 and 4, respectively. This optimization requires one less step for RPC inputs, and 3 fewer steps TGC inputs, meaning that only one step is reduced for the optimized network. The resulting delay is given by the worst-case path, i.e., the RPC inputs. On the other hand, this optimization constrains the way that the sorting network inputs are connected, i.e., the sorted sub-sequences have to be connected together and at the respective input lines at which the pre-sorted input optimization has been employed. Due to the added constraints in the input connection and the low delay reduction, the pre-sorted input optimization has not been implemented for the MUCTPI.

The overall latency can be further reduced by replacing the last Batcher merging network by the Alekseev selection network [73, p. 232] and [80]. Alekseev has observed that one can select the largest t elements of a sequence of length 2t by splitting the original sequence in two sub-sequences, sorting each separately, and comparing and interchanging $\langle x_1 : x_{2t}, x_2 : x_{2t-1}, ..., x_t : x_{t+1} \rangle$. The compare and interchanging step is performed in only one stage because all comparisons are implemented in parallel, i.e., no data dependencies. In the MUCTPI sorting network, the last Batcher merging network already receives two sorted sub-sequences. Therefore, only the comparing and interchanging step is needed. This reduces the overall network depth by four steps, resulting in 28 steps. This gain comes at the price of outputting an unsorted sequence with the 16 highest p_T muon candidates, instead of a sorted sequence if the Batcher merging network is implemented. Given that having a sorted output sequence is rather desired, and the latency reduction is relatively low, this optimization option has not been implemented.

Therefore, the MUCTPI sorting network remains with 32 steps, ensuring that the output sequence is sorted, the muon candidates from RPC and TGC sector logic modules are not required to be sorted, and they can be connected in any order. In the case of a future upgrade of the muon trigger detector, more muon candidates are received from the sector logic modules, the pre-sorted optimization can be used to reduce the network delay further. For example, sorted sub-sequences of length $\{2,4,8\}$ enables the reduction of up to $\{1,3,6\}$ network steps. On the other hand, in case the number of outputs increase and the highest p_T muon candidate output sequence is not required to be sorted, the Alekseev selection network can always reduce the last merging network to a single step. For example, unsorted output sequences of length $\{32,64,128\}$ enables the reduction of up to $\{5,6,7\}$ network steps, respectively.

7.10 MUCTPI sorting network

This section presents the selected MUCTPI sorting network in light of the previous discussion. Figure 7.17 shows the block diagram of the resulting 352-key input 16-key output sorting network with $R = 16$. Each of the blocks of type S, so-called S-network, corresponds to the 12-step 22-key input 16-key output sorting network investigated in Section 7.9. The Knuth diagram of the S-network is shown in Figure 7.18. Table 7.4 describes the pair of keys that are connected to comparison-exchange modules for each of the stages from 1 to 12.

The first S-network instance is connected to the sub-sequence $\langle x_1, x_2, ..., x_{22} \rangle$, the second to $\langle x_{23}, x_{24}, ..., x_{44} \rangle$, and so on until the sixteenth S-network instance that is connected to $\langle x_{331}, x_{332}, ..., x_{352} \rangle$. Each of the networks of type M shown in Figure 7.17, so-called M-network, corresponds to the 32-key input 16-key output merging network investigated in Section 7.9. The Knuth diagram of the M-network is shown in Figure 7.19. Table 7.5 describes the pair of keys that are connected to comparison-exchange modules for each of the stages from 1 to 5.

The sub-sequences $\langle x_1, x_2, ..., x_{16} \rangle$ and $\langle x_{23}, x_{24}, ..., x_{38} \rangle$ originated from the first and second S-network instances are connected to the first M-network. Similarly, the sub-sequences $\langle x_{45}, x_{46}, ..., x_{60} \rangle$ and $\langle x_{67}, x_{68}, ..., x_{82} \rangle$ originated from the third and fourth S-network instances are connected to the second M-network. This continues until the eighth M-network of the first level is connected to the sub-sequences $\langle x_{309}, x_{310}, ..., x_{324} \rangle$ and $\langle x_{331}, x_{332}, ..., x_{346} \rangle$ originated from the fifteenth and sixteenth S-network instances. The same principle is applied to the second, third, and fourth levels of the merging part until a single sorted sequence is driven by the output lines $\langle x_1, x_2, ..., x_{16} \rangle$.

Figure 7.20 shows the Knuth diagram of the MUCTPI sorting network after implementing all the S-networks and M-networks. It is not possible to distinguish the pairs in the printed version, but one can still identify the sorting part in the first third of the figure (from left to right), and the merging part in the remaining part of the plot. As the figure has been generated from vectors, one can still magnify the plot in the electronic version. The 12-step Baddar 22-key sorting network after output optimization is replicated vertically 16 times in the sorting part from stages 1 to 12. Next, the 5-step 32-key input 16-key output odd-even merging network is replicated vertically 8, 4, 2, and 1 time in stages 13 to 17, 18 to 22, 23 to 27, and 28 to 32, respectively. The merging part merges the 16 sorted sub-sequences, and at the same time, it routes the output data to the top lines, i.e., $\langle x_1, x_2, ..., x_{16} \rangle$.

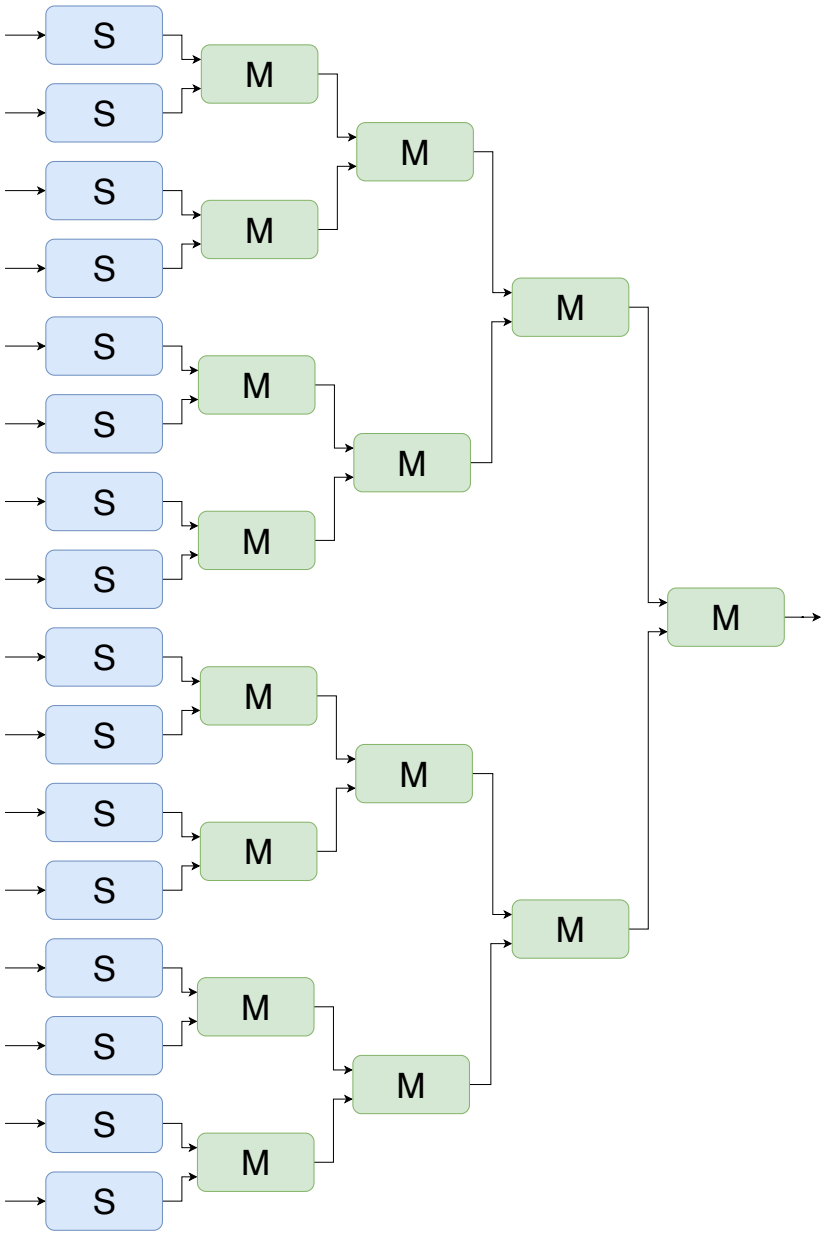


Figure 7.17 – Selected 352-key input 16-key output sorting network with $R = 16$

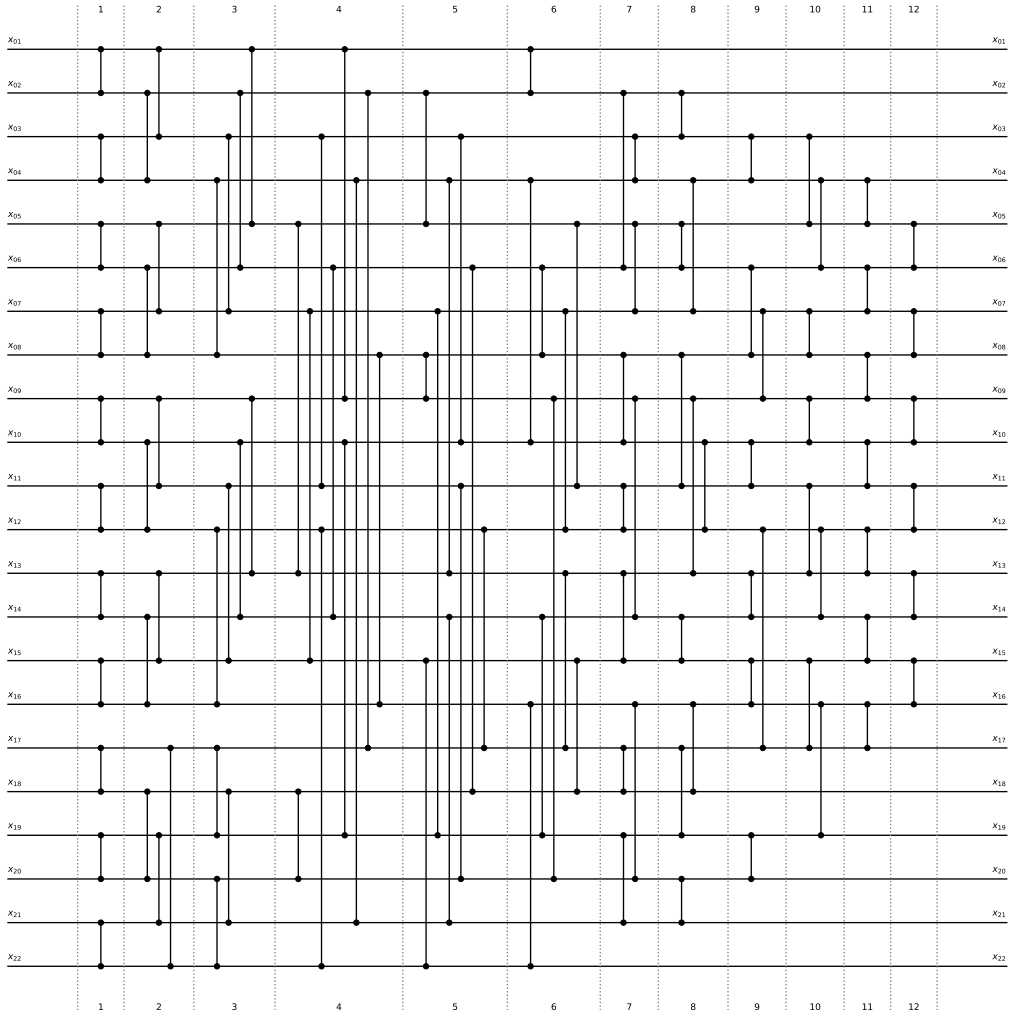


Figure 7.18 – Knuth diagram of the S-network (Baddar 22-key input 16-key output sorting network)

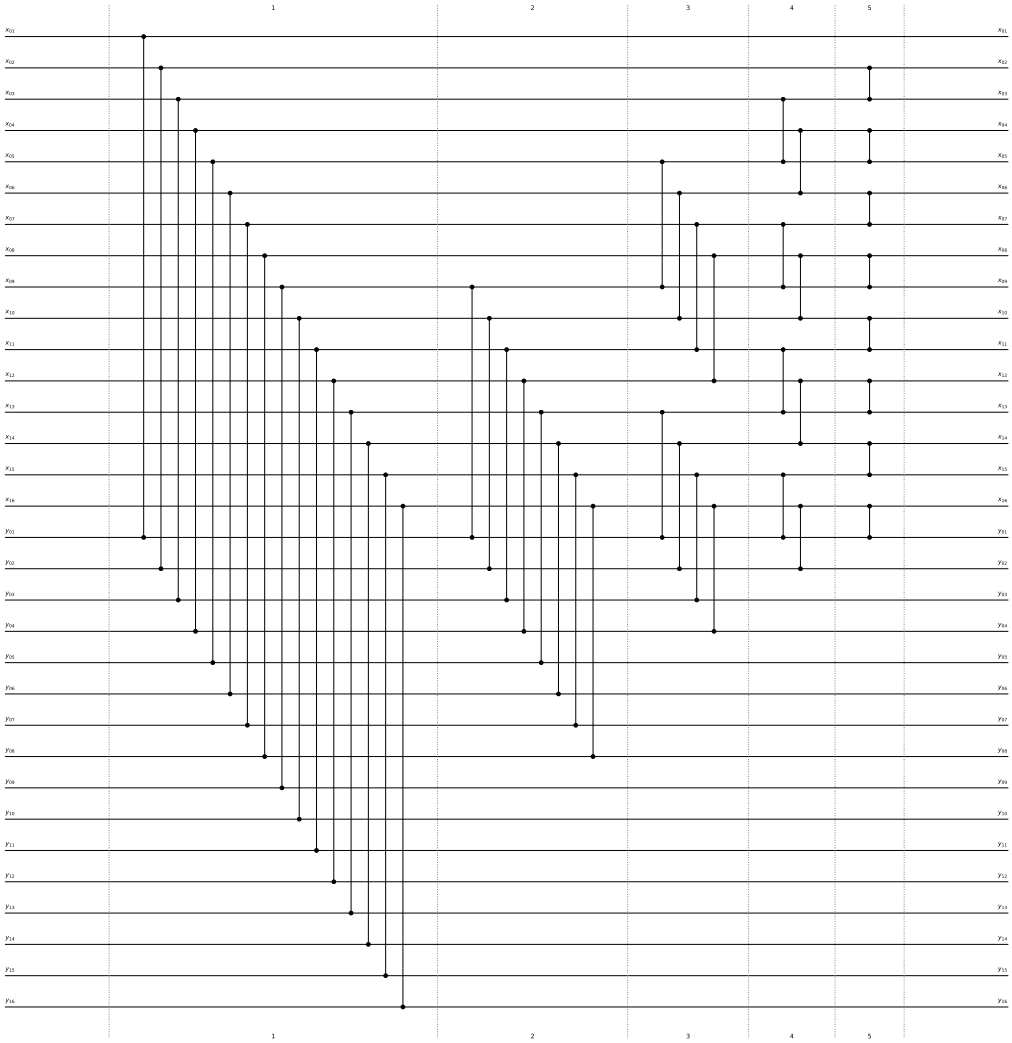


Figure 7.19 – Knuth diagram of the M-network (32-key input 16-key output odd-even merging network)

	1	2	3	4	5	6	7	8	9	10	11
1	(1, 2)	(3, 4)	(5, 6)	(7, 8)	(9, 10)	(11, 12)	(13, 14)	(15, 16)	(17, 18)	(19, 20)	(21, 22)
2	(1, 3)	(2, 4)	(5, 7)	(6, 8)	(9, 11)	(10, 12)	(13, 15)	(14, 16)	(17, 22)	(18, 20)	(19, 21)
3	(1, 5)	(2, 6)	(3, 7)	(4, 8)	(9, 13)	(10, 14)	(11, 15)	(12, 16)	(17, 19)	(18, 21)	(20, 22)
4	(1, 9)	(2, 17)	(3, 11)	(4, 21)	(5, 13)	(6, 14)	(7, 15)	(8, 16)	(10, 19)	(12, 22)	(18, 20)
5	(2, 5)	(3, 10)	(4, 13)	(6, 18)	(7, 19)	(8, 9)	(11, 20)	(12, 17)	(14, 21)	(15, 22)	-
6	(1, 2)	(4, 10)	(5, 11)	(6, 8)	(7, 12)	(9, 20)	(13, 17)	(14, 19)	(15, 18)	(16, 22)	-
7	(2, 6)	(3, 4)	(5, 7)	(8, 10)	(9, 14)	(11, 12)	(13, 15)	(16, 20)	(17, 18)	(19, 21)	-
8	(2, 3)	(4, 7)	(5, 6)	(8, 11)	(9, 13)	(10, 12)	(14, 15)	(16, 18)	(17, 19)	(20, 21)	-
9	(3, 4)	(6, 8)	(7, 9)	(10, 11)	(12, 17)	(13, 14)	(15, 16)	(19, 20)	-	-	-
10	(3, 5)	(4, 6)	(7, 8)	(9, 10)	(11, 13)	(12, 14)	(15, 17)	(16, 19)	-	-	-
11	(4, 5)	(6, 7)	(8, 9)	(10, 11)	(12, 13)	(14, 15)	(16, 17)	-	-	-	-
12	(5, 6)	(7, 8)	(9, 10)	(11, 12)	(13, 14)	(15, 16)	-	-	-	-	-

Table 7.4 – 22-key input 16-key output baddar22 sorting network comparison-exchange pairs

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	(1, 17)	(2, 18)	(3, 19)	(4, 20)	(5, 21)	(6, 22)	(7, 23)	(8, 24)	(9, 25)	(10, 26)	(11, 27)	(12, 28)	(13, 29)	(14, 30)	(15, 31)	(16, 32)
2	(9, 17)	(10, 18)	(11, 19)	(12, 20)	(13, 21)	(14, 22)	(15, 23)	(16, 24)	-	-	-	-	-	-	-	-
3	(5, 9)	(6, 10)	(7, 11)	(8, 12)	(13, 17)	(14, 18)	(15, 19)	(16, 20)	-	-	-	-	-	-	-	-
4	(3, 5)	(4, 6)	(7, 9)	(8, 10)	(11, 13)	(12, 14)	(15, 17)	(16, 18)	-	-	-	-	-	-	-	-
5	(2, 3)	(4, 5)	(6, 7)	(8, 9)	(10, 11)	(12, 13)	(14, 15)	(16, 17)	-	-	-	-	-	-	-	-

Table 7.5 – 32-key input 16-key output odd-even merging network comparison-exchange pairs

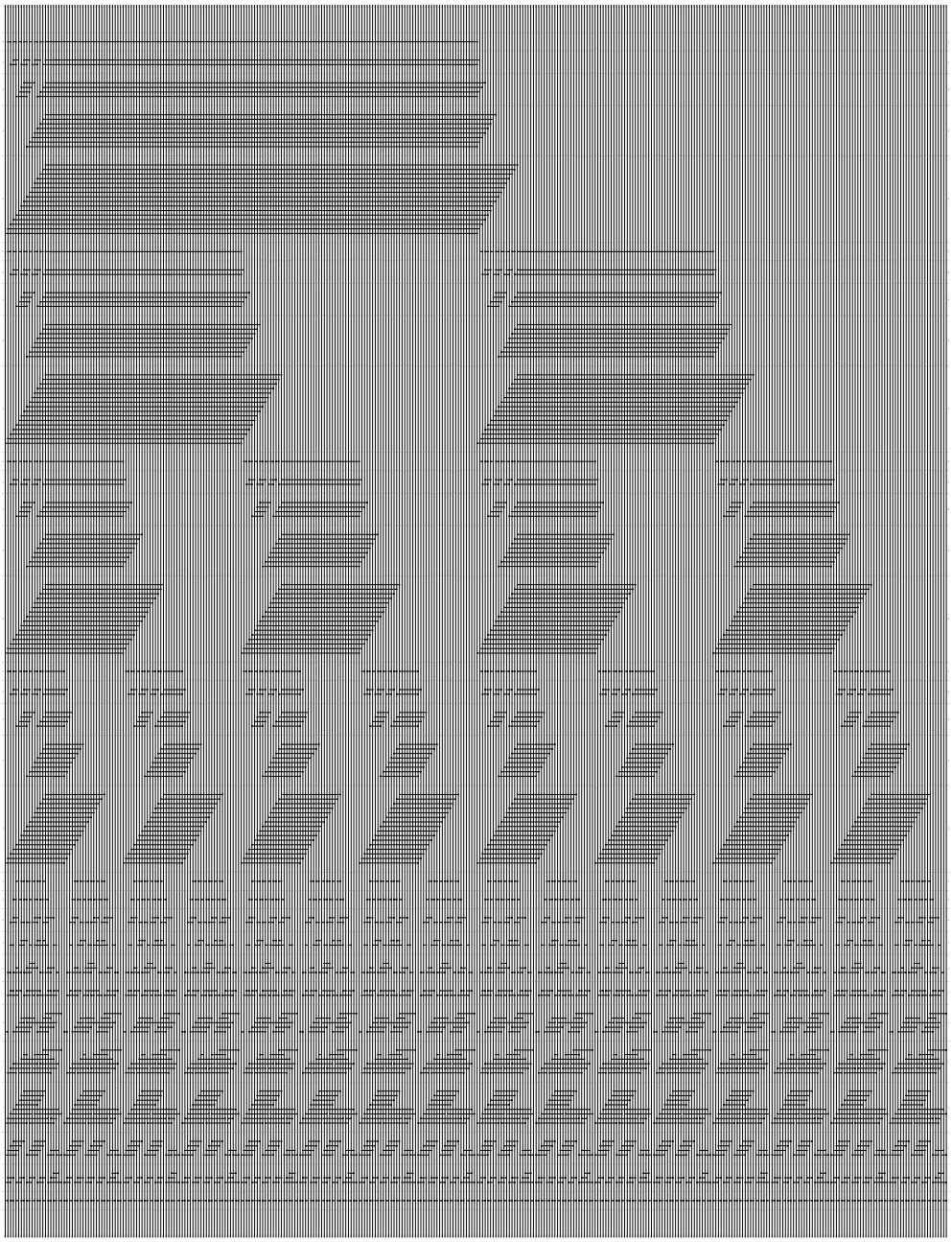


Figure 7.20 – Knuth diagram of the MUCTPI sorting network

7.11 Validation of MUCTPI sorting network

The zero-one principle implemented in the SNpy package and presented in Section 7.2.1 has been used to check the S-and-M networks, selected in the Section 7.9, against sorting and merging errors. A dataset of 2^{22} different sequences of 0s and 1s have been applied to the S-network, and the sixteen first output lines have been checked against sorting errors. No errors have been found.

The M-network has been checked against merging errors, with respect to the 16 required output elements. Every combination of two sorted sub-sequences of length 16 have been applied to the network, no errors have been found. This result demonstrates that the S-and-M networks developed in the Section 7.9 are validated to be used in the MUCTPI.

Although testing S-and-M networks alone is sufficient, as an exploratory work, it has been investigated the time needed to validate the top-level network with all S-and-M networks combined together using 2^{352} combinations of 0s and 1s. As the testing of each of the 2^{352} combinations is independent between each other, multiple combinations can be tested simultaneously. The validation has been distributed to 48 cores of a high-performance computer. It took ≈ 100 s to check 2^{20} combinations. Testing the 2^{352} combinations would take 1×10^{97} days. Even if computer technology would ever advance to the point where each proton in Earth would process data at the same speed as the high-performance computer being used, it would still take 8×10^{39} millenniums to check all the combinations.

As the validation of the network runs faster in FPGA when compared to a high-performance computer, the MUCTPI hardware can be used for validating the MUCTPI sorting network. Twenty instances of the MUCTPI sorting network running at 160 MHz can be implemented in a dedicated MUCTPI firmware version, for testing only. Opposed to the software implementation, 2^{20} combinations can be tested in ≈ 350 us in FPGA, i.e. 2.8×10^5 faster than using the high performance computer. However, it would still take 3.5×10^{91} days to test all the 2^{352} combinations.

In view that it is not possible to test all the 2^{352} combinations of 0s and 1s, 2^{30} combinations have been randomly selected. The testing of the randomly selected combinations took 1.5 days using the high-performance computer mentioned above, no errors have been found.

7.12 Summary

This chapter described the state-of-the-art in sorting networks and the optimizations for the MUCTPI.

Section 7.2 provided an introduction to merging and sorting networks. The comparison-exchange modules, Knuth diagram, and zero-one principle used in different parts of this chapter have been presented.

Section 7.3 described the well known Batcher merge-exchange sorting algorithm. Batcher innovated by comparing nonadjacent pair of keys, splitting them up to sorted sub-sequences. This technique enabled the implementation of efficient sorting networks for any value of n.

Section 7.4 described a generalization of the merge-exchange sorting algorithm with $p = 1$ that originated the odd-even merging network. An optimized version of this network has been used in the merging part of the MUCTPI sorting network.

Section 7.5 presented the sort-by-merging scheme that enables the recursive use of merging networks to generate sorting networks, such as odd-even and bitonic mergesort networks.

Section 7.6 described the investigation of either faster or more efficient sorting networks. The fastest sorting networks known in the literature for $n = \{16, 22\}$ discovered by David C. Van Voorhis and Sherenaz W. Al-Haj Baddar have been presented. An optimized version of the Baddar 22-key sorting network has been used in the sorting part of the MUCTPI sorting network.

Section 7.7 described two types of network optimizations. The first focused on optimizing away unused input or output lines. The input and output optimization have been exhaustively used to generate the results presented in Sections 7.8 and 7.9. The second optimization type focused on optimizing away unnecessary comparison-exchange modules due to pre-sorted input sub-sequences or output lines that do not need to be sorted. The pre-sorted input optimization has been investigated to reduce the number of stages of the MUCTPI sorting networks thanks to the fact that RPC and TGC sector logic modules send sorted sub-sequences of length 2 and 4, respectively. However, this optimization has not been implemented because the worst-case path delay, given by the RPC inputs, would be minimized by only one stage. In case the number of muon candidates per SL is increased in a future upgrade of the muon trigger detectors, the pre-sorted input optimization might be of higher interest for the MUCTPI application.

Section 7.8 presented a comparative study of the delay and the number of comparisons for Batcher sorting methods. It has been demonstrated that, within the Batcher sorting methods, the merge-exchange sorting network gives the lowest value of delay and number of comparisons.

Section 7.9 described the divide-and-conquer method to optimize sorting networks with $O \ll I$. The method divides a large sorting network problem into smaller sorting and merging networks. First, the input is divided into several combinations of groups with different sizes

and sorted concurrently using the Batcher merge-exchange sorting algorithm. Second, for each of these combinations, all the respective input groups are merged using a binary tree of odd-even merging networks. Then, the fastest combination options are selected. The first step of the divide-and-conquer method reduced the sorting network delay from 45 to 35 steps.

One can further optimize the sorting part if a sorting network faster than the respective Batcher merge-exchange sorting network exists. No further optimization is possible in the merging part because the odd-even merging network is optimal when the size of the sets to be merged is equal and a power-of-two value, see Section 7.4. For the MUCTPI application, one of the fastest combination options uses a 22-key input 16-key output sorting network that has been replaced by the fastest 22-key sorting network known, discovered by Sherenaz W. Al-Haj Baddar in 2009. Some of the compare-exchange operations from the Baddar sorting network and the Bacther odd-even merging network have been optimized away in view that only the 16 highest p_T muon candidates are required at the output.

Using the Baddar sorting network further reduced the total delay given by the divide-and-conquer method from 35 to 32 delay steps. The 32-step 352-key input 16-key output sorting network discovered for the MUCTPI application sorts the input data using 13 fewer steps than the 45-step 352-key Batcher merge-exchange, odd-even, or bitonic sorting networks.

Section 7.10 provided the Knuth diagram and the table of comparison-exchange pair per stage for the S-and-M networks. In addition, the block diagram and the description of their interconnectivity have been presented. A plot of the resulting network has been shown, however, the pairs can be distinguished only in the electronic version, after zooming in on the page. All the information needed to implement the MUCTPI sorting network regardless of the synthesis technique is available in Section 7.10.

Section 7.11 presented the validation of the S-network and M-network using the zero-one principle. Both sorting and merging networks have been demonstrated to sort or merge the input data with respect to the required 16 output elements.

The next chapter describes the implementation of the MUCTPI sorting network using two different synthesis techniques. It focus on the different aspects of each of the synthesis techniques to develop the same network using the same starting point, i.e. Tables 7.4 and 7.5 and Figure 7.17.

8 Implementation approaches

This chapter describes the implementation of the MUCTPI sorting network, described in Section 7.10, using the Register-Transfer Level (RTL) and the High-Level Synthesis (HLS) implementation approaches. Section 8.1 highlights the differences between RTL and HLS. Section 8.2 and Section 8.3 present the design entry, the design flow, and the implementation results for each of the implementation approaches. Section 8.4 provides a comparative study between both approaches, limited to the MUCTPI sorting unit. Section 8.5 closes the chapter with a summary.

8.1 Introduction

8.1.1 Sorting unit

The sorting unit receives information from 352 muon candidates, sorts the muon candidates with respect to their p_T, and outputs information from the 16 highest p_T muon candidates. Each of the inputs and outputs carries a data structure containing all the information from a muon candidate. In what regard the sorting unit, the data structure contains two groups of members. The first group carries the member on which the sorting is based. The second group carries all the sorting unit outputs. Some of the data structure members, such as the muon identification, must propagate through the network to uniquely identify each muon candidate in the sorting network output. Other members can either propagate through the network or be buffered externally, and multiplexed based on the muon identification number, given by the sorting network. These two design options are covered in Section 8.2.5. The two groups are defined as:

1. $\mathbf{p_T}$, the only member at which the sorting is based on.

2. **All**, which represents all the members that propagate through the network. This group includes, at least, the p_T and muon identification number.

The muon candidate data structure members are:

- **Muon identification:** Integer number ranging from 0 to 351, represented in 9 bits.
- **p_T:** Muon transverse momentum threshold, represented in 4 bits.
- **RoI:** Muon position, known as *Region-of-Interest*, represented using in 8 bits.
- **Flags:** Muon candidate flags, represented in 4 bits.

8.1.2 RTL and HLS design flows

RTL is a design abstraction that models circuits using registers as building blocks. The Register-Transfer Level (RTL) nomenclature comes from the fact that the circuit is expressed in terms of the data transfer between registers. Registers are implemented as flip-flops or latches, and the data are transferred between registers using combinational logic if needed [81]. RTL provides a higher abstraction alternative to logic-level design, i.e., building blocks from logic gates and transistor-level design, i.e., building logic gates from transistors [82].

HLS provides a higher design abstraction alternative to RTL, by omitting cycle timing details and resource types in the circuit description. The absence of such information in the design description enables a higher level of abstraction by letting the synthesizer to determine how the sequential operations are implemented [83]. The process of identifying data dependencies and mapping sequential operations into clock cycles is known as *scheduling*. The process of determining which hardware resource implements each scheduled operation is known as *binding*. In HLS, scheduling and binding are driven by optimization directives provided by the user, and information about the target device [84].

Figure 8.1 highlights the differences between the RTL and HLS design flow, in the context of the MUCTPI sorting network implementation. The blocks colored in yellow are implemented using SNpy [78], a python package for sorting networks developed by the author of this thesis. The blocks in blue and green represent the vendor-specific RTL, and HLS design flows, respectively. Both vendor-specific RTL and HLS implementation tools, i.e., Xilinx Vivado and Xilinx Vivado HLS, have been provided by the vendor of the FPGA being used in the MUCTPI. The block in purple represents the FPGA bitstream, which is a binary file that holds the FPGA configuration information for a given compilation.

The first step, i.e., generating the comparison-exchange pairs, represents all the sorting network generation and optimization steps covered in Chapter 7. This is the only step that is

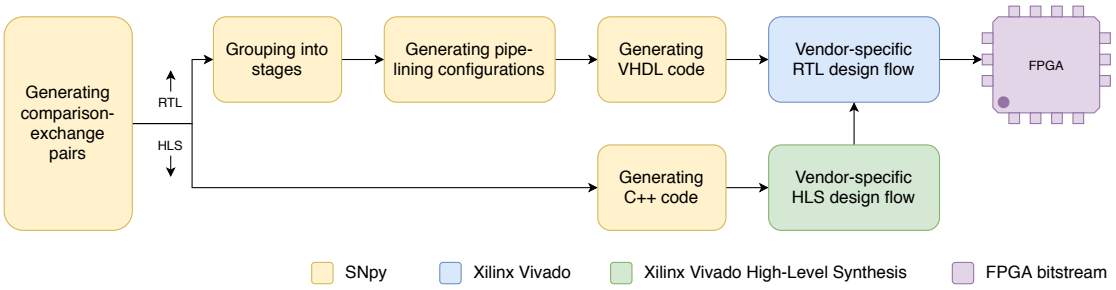


Figure 8.1 – RTL and HLS design flows

common to RTL and HLS, and for this reason, the output of this block is the common entry-point for RTL and HLS design flows. Starting from this point, the design flow can continue in two directions. The direction upward represents the remaining part of the RTL design flow. The direction downwards represents the remaining part of the HLS flow. Notice that the HLS flow also uses the vendor-specific RTL design flow to generate the FPGA bitstream.

The choice of the common entry-point for RTL and HLS design flows is based on the different ways that sorting networks can be described using software or hardware description languages. The sorting networks, when executed as single-thread software, are implemented from a uni-dimensional array of comparison-exchange operations that are executed sequentially, i.e., one after the other. The uni-dimensional array of comparison-exchange operations corresponds to the result obtained by the first block in Figure 8.1.

RTL flow

When sorting networks are described in hardware, and there is an interest in reducing the latency, all the non-overlapping comparison-exchange operations are explicitly described in parallel. Non-overlapping operations stand for all the operations that can be computed simultaneously. This is performed in the *Grouping into stages* block shown in Figure 8.1. Note that this step has already been used in Sections 7.2 and 7.7 to 7.9 to generate Knuth diagrams, optimize, and extract the delay of sorting networks. At this point, the sorting network is expressed in terms of comparison-exchange operations per stage, i.e., a bi-dimensional array.

In principle, the explicit description of the comparison-exchange pairs into stages is not required. This is because RTL FPGA implementation tools are capable of implicitly instantiating, in parallel, blocks that do not have data inter-dependence. This is the case of the non-overlapping comparison-exchange pairs. However, the explicit description of the stages has been adopted in order to enable the generation of a configurable VHSIC HDL (VHDL) [85] code that supports different pipelining configurations.

The technique of adding registers between operations in view of increasing the maximum clock frequency is known as *pipelining*. This might not be clear for the reader yet, but it is better understood after the generation of pipelining configurations, and VHDL code is covered in Section 8.2. These steps are mentioned here to emphasize that they are only present in the RTL design flow, and to highlight that the VHDL code does not need to be regenerated due to different pipelining configurations. At the end of the RTL design flow, the resulting VHDL code is implemented using the vendor-specific RTL design flow.

In principle, the explicit description of which stages are pipelined is not required, because current FPGA implementation tools are capable of performing register retiming. Register retiming is a technique that moves or rearranges registers across combinational logic in order to improve maximum operating frequency [86, 87]. This way, the registers could be placed adjacent to the combinational representation of the sorting network and then efficiently distributed across the combinational logic by the implementation tool. The implementation tool could benefit from back-annotated timing information from placing and routing to distribute the registers efficiently. However, the author of this thesis did not have success in using such register retiming techniques for the implementation of large sorting networks, such as the one implemented as part of this thesis. The use of register retiming technique from Synopsis Simplify Premier [88] and Xilinx Vivado [89] have been explored without success. In both cases, the distribution of the registers has been limited to few logic levels away from the initial position of the registers, and never being able to reach the inner-most stages of the sorting network. Experimental results exploring retiming techniques have shown that the quality of the results depends on the logic depth of the circuit, delay model, and circuit type [90]. As the use of retiming techniques is not the focus of this work, the author of this thesis decided to determine explicitly which stages of the sorting network is pipelined.

HLS flow

The bottom-part of Figure 8.1 shows the part of the design flow that is used only in the HLS option. Note that the grouping of comparison-exchange pairs into stages and generating pipelining configurations are not present in the HLS design flow. This is because the parallelism and pipelining cannot be explicitly described, in HLS, because cycling timing details are not specified in software. Then, the only option that remains is to expect that the HLS tool is able to infer the parallelism from the generated C code. HLS achieves it by analyzing the data inter-dependence within the sequential operations, and to efficiently pipeline the resulting logic.

The absence of these two blocks illustrates a two-folded nature of HLS. On the one hand, it simplifies the description of the MUCTPI sorting network, by removing two steps that are present only in the RTL design flow. On the other hand, it gives less control in how the design

is going to be implemented by transferring some of the designer's responsibility to the tool. The two-folded nature of HLS is covered in more detail in Section 8.4.

At the end of the HLS design flow, the C code is generated and sent to the vendor-specific HLS design flow, which synthesizes C code to a RTL description of the sorting unit. The C code generation and the HLS design flow are described in Section 8.3. The RTL design description generated by the HLS design flow is translated to an FPGA bitstream using the same RTL vendor-specific design flow used in the RTL design abstraction. In this thesis, the RTL vendor-specific design flow based on HLS-generated RTL hardware description is referred to as HLS-driven RTL design flow.

8.2 RTL implementation

8.2.1 Combinational-only sorting networks

Any sorting network can be described, concerning the functionality, using only combinational elements such as LUTs. In fact, they can be built from an array of only two unit blocks. Figure 8.2 shows the compare-exchange unit, so-called *C* unit. Each of the inputs $\langle x_1, x_2 \rangle$ is driven by the muon candidate data structure described in the introduction of Section 8.1.1.

The p_T from x_1 is compared against the p_T from x_2 in the block *"<"*, part of Figure 8.2. If the comparison result is true, block *"E"* exchanges all the members from $\langle x_1, x_2 \rangle$, i.e. all the members from x_1 are transferred to x_2, and vice-versa.

Figure 8.3 describes the bypass unit, so-called *B* unit. The *B* unit transfers the input directly to the output without any processing. This block is used when the pair of inputs $\langle x_1, x_2 \rangle$ are not compared and exchanged in a given stage.

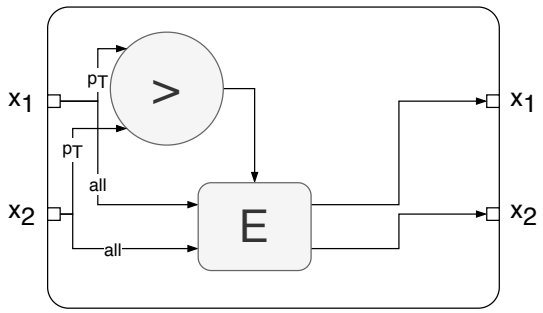


Figure 8.2 – Comparison-exchange unit



Figure 8.3 – Bypass unit

Figure 8.4 shows the implementation of a 4-key sorting network using *C* and *B* units. The 4-key sorting network is the same as the one shown in Figure 7.4, and it is duplicated in Figure 8.5 in the best interest of the reader.

The numbers in the top part of the figure indicate the respective stage of the sorting network. The inputs $\langle x_1, x_2, x_3, x_4 \rangle$ propagate from left to right through the *C* and *B* units until they are available in the respective outputs, in the right side of the figure. In stage 1, the input pairs $\langle x_1, x_2 \rangle$ and $\langle x_3, x_4 \rangle$ are compared and exchanged respectively. Note that each line, representing the network connections, carries the original input number because, after a few stages, one can easily lose track of the relationship between the connections and the originally associated input. In stage 2, the input pairs $\langle x_1, x_3 \rangle$ and $\langle x_2, x_4 \rangle$ are compared and exchanged. In stage 3, only one comparison exists, see Figure 8.5, therefore only the input pair $\langle x_2, x_3 \rangle$ is connected to the *C* unit. The remaining pair $\langle x_1, x_4 \rangle$ is directly propagated to the output using the B unit. Finally, each of the outputs from the last stage is connected to their respective sorting network output.

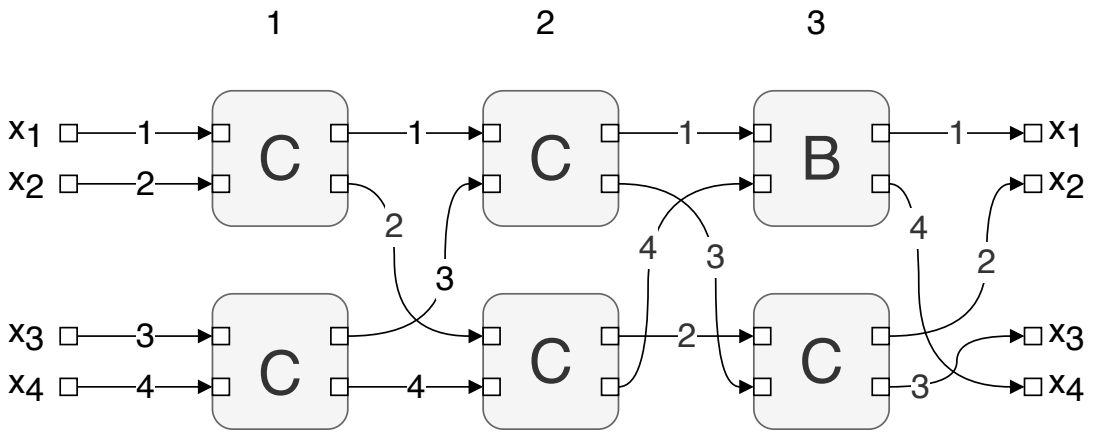


Figure 8.4 – 4-key sorting implementation

Following this example, one can generalize that a sorting network with an even number of inputs *I* can be built from an array of $\frac{I}{2} \times d^S(I)$ unit blocks such as the *C* and *B* units. For $d^S(I)$ with power-of-two values of *I*, see Equation (7.1).

8.2.2 Pipelined sorting networks

Despite that, any network can be described, concerning the functionality, using only the *C* and *B* units. However, it is impossible to implement large sorting networks, running in high clock frequencies, using only combinational elements. The reason comes from the fact that for large values of *I*, the two following issues are dominant in limiting the maximum clock frequency:

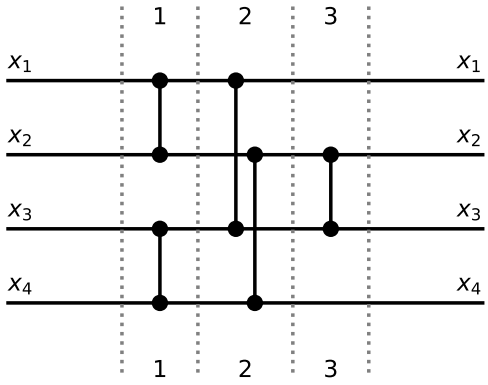


Figure 8.5 – 4-key sorting network

- **Long logic delays**, i.e., high delay values associated with the number of logic levels required to implement all the combinational elements from the *C* units.

- **Long routing delays**, i.e., long delays values associated with the routing distance between the combinational elements.

In order to complement the existing *C* and *B* units, Figures 8.6 and 8.7 introduce the *CR* and *BR* units. The only difference, compared to the previous *C* and *B* units, is that a register, abbreviated by *"R"*, is used to register the respective output in the rising edge of the clock. The clock is omitted in the block diagram for better readability. Source codes A.1 to A.3 show the VHDL description of a configurable sorting network using C, B, CR, and BR units. Notice that all VHDL source code of the MUCTPI sorting unit is shown in a dedicated appendix chapter at the end of this document.

Figure 8.8 shows the block diagram of the implementation of the 8-key merge-exchange sorting network using an array, with dimension 4×6, of *C*, *B*, *CR*, and *BR* units. The stages, input, output, and connectivity are indicated in the same way as Figure 8.4. Note that the output from stages 2, 4, and 6 are registered by using *CR*, and *BR* units instead of the *C*, and *B* units used in stages 1, 3, and 5. By pipelining stages 2, 4, and 6, the resulting sorting network can run in a clock frequency three times higher than the same network without pipelining any of the stages. This assumption assumes that the worst-case path delays for the stages (1,2), (3,4) and (5,6) are the same.

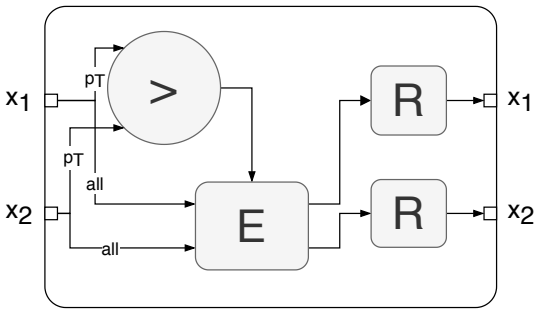


Figure 8.6 – CR unit

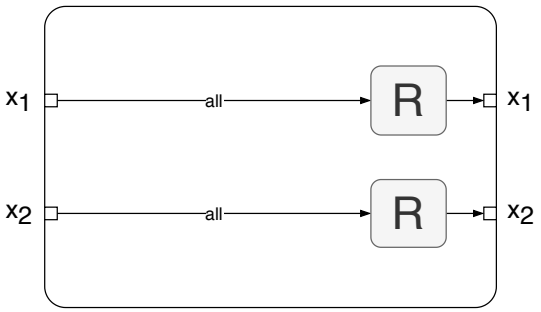


Figure 8.7 – BR unit

8.2.3 Pipelining configurations

With more stages being implemented using *CR*, and *BR* units, instead of *C*, *B* units, higher maximum frequencies are achieved. However, more registered stages come at the cost of increased latency. This leads to the optimization problem of finding the minimum number of registered stages, at which timing closure can still be achieved. Timing closure stands by having a positive slack for all the static timing analysis checks. In static timing analysis, slack is the difference between the required and arrival time between two endpoints [91].

The sorting unit in the MUCTPI is specified to run in 160 MHz with a maximum latency of 50 ns. This means that not every 32 stages of the MUCTPI sorting network can be pipelined. In fact, the MUCTPI sorting network should not exceed eight clock cycles latency, i.e., eight pipelined stages.

Table 8.1 shows different positions of the pipelining registers of the MUCTPI sorting network for a number of registered stages D ranging from 0 to 8. The rows from 0 to 8 represent each of the pipelining configurations. The columns from 0 to 31 represent the position of each of the 32 stages of the MUCTPI sorting network. Each cell filled in grey represents a stage that has been implemented using *CR*, and *BR* units i.e., pipelined stages. All the cells filled in white represent stages that have been implemented using *C*, *B* units, i.e., non pipelined stages.

The pipelined stages have been equidistant distributed whenever it is possible, given that only $D = \{1,2,4,8\}$ are divisors of 32. The last stage has been pipelined for all configurations with $D > 0$, in order to make sure that the sorting network outputs are registered. It is required that the inputs of the sorting unit are driven by registers in the upstream block of the data flow. Registering both inputs and outputs of the sorting unit guarantees that the latency is fully accounted within the sorting network, i.e., no time-borrowing from paths outside the sorting unit. The performance results for each value of D are covered in Section 8.2.9.

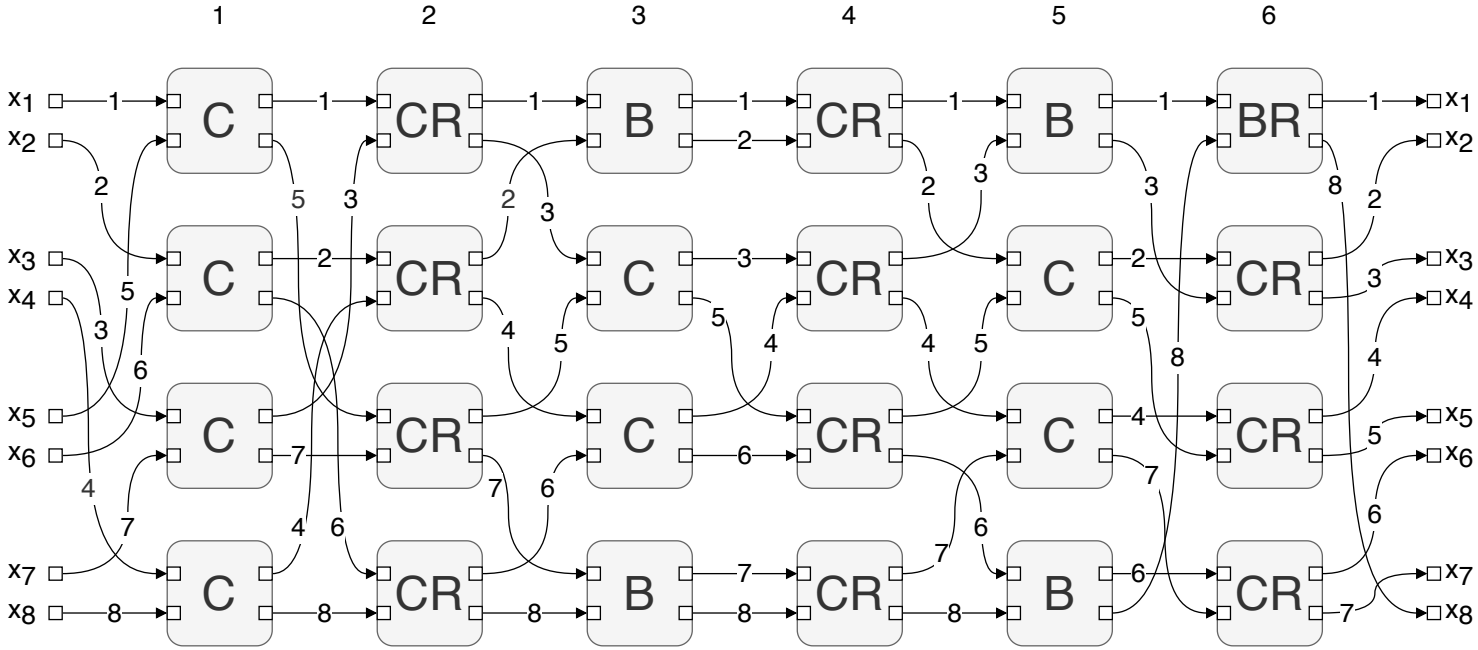


Figure 8.8 – Block diagram of the implementation of the 8-key merge-exchange sorting network

Table 8.1 – Pipelining configurations for $0 \leq D \leq 8$

D	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0																																
1																																■
2																■																■
3										■											■											■
4								■								■								■								■
5							■						■							■						■						■
6						■					■						■					■					■					■
7				■					■				■					■				■					■					■
8				■				■				■				■				■				■				■				■

8.2.4 Hierarchical options

With respect to the hierarchical organization of the MUCTPI sorting network, the two following hierarchical levels H have been investigated:

- **H = 3** : This is the higher hierarchical representation option. It implements the S-and-M networks as sub-modules in the design, and the *CR*, *BR*, *C*, and *B* units are implemented within the respective S-and-M network sub-modules. Therefore, the design is organized in the three following hierarchical levels:

  1. Top-level
  2. S-and-M networks
  3. *CR*, *BR*, *C*, and *B* units

  This option corresponds to the block diagram shown in Figure 7.17. The Knuth diagram from S-and-M networks are shown in Figures 7.18 and 7.19, respectively.

- **H = 2** : This the lower hierarchical representation option. It flattens the hierarchy from S-and-M networks, and instantiates all the *CR*, *BR*, *C*, and *B* units within the top-level. Therefore, the design is organized in the two following hierarchical levels:

  1. Top-level
  2. *CR*, *BR*, *C*, and *B* units

  This option corresponds to the Knuth diagram shown in Figure 7.20.

The higher hierarchical representation is expected to speed-up synthesis, by reusing sub-modules, but it is unclear how it impacts the overall performance. The performance results for both hierarchical options are covered in Section 8.2.9. Source code A.4 shows the sorting network VHDL description for implementation options $H = 3$ and $H = 2$.

8.2.5 Architecture options

As anticipated in Section 8.2.1, the propagation of the muon candidate data through the sorting unit can be implemented using the two following architecture options:

- **M = 0** : The entire muon candidate structure[1] propagates through the sorting network.

[1] I.e., the muon identification number ranging from 0 to 351 (9 bits), the transverse energy p_T (4 bits), the RoI position (up to 8 bits), and candidate flags (4 bits). Therefore a total of up to 25 bits propagates through the sorting network. For the width of each information, see Tables 5.1 and 5.2.

- **M = 1** : Only the muon identification and the transverse energy p_T propagates through the network[2]. The RoI and flags are buffered externally and multiplexed based on the muon identification number from each of the 16 highest p_T elements, originated from the sorting network.

For $M = 0$, the sorting unit is equivalent to the sorting network. But, for $M = 1$, the sorting unit represents the sorting network and the output multiplexor.

For the architecture option M=1, for each value of L, where L is the total latency of the sorting unit, $L-1$ clock cycles are used in the sorting network, i.e. $D = L-1$, and 1 clock cycle is used in the output multiplexor. For $M = 0$, $D = L$. The performance results for both architecture options are covered in Section 8.2.9. Source code A.5 shows the sorting unit VHDL description for implementation options $M = 0$ and $M = 1$.

8.2.6 Generating VHDL code

A significant effort has been invested in writing a configurable VHDL description of the sorting unit with support to the different delay, hierarchical, and architecture options.

Source codes A.1 to A.5 show all the VHDL files used to represent the sorting unit in all of the implementation options. Most of the VHDL code is hand crafted, except one part of the sorting network VHDL package that is automatically generated by SNpy. The automatically generated part of the VHDL package is the following:

- **Combinational-only sorting network representation**, i.e. an array of dimension $\frac{I}{2} \times d^S(I)$ describing which pairs should be either compared-exchanged or bypassed per stage. For $H = 3$, the arrays representing the S-and-M sorting networks have the dimensions 11×12 and 16×5, respectively. For $H = 2$, the single array representing the entire sorting network has the dimension 176×32.
- **Pipelining configuration**, a function that returns which stages are pipelined for each value of D. The function result is defined according to Table 8.1.

Table 8.1 is used to determine which stages are pipelined for both hierarchical options $H = 3$ and $H = 2$. For $H = 2$, each column of Table 8.1 directly corresponds to each stage of Figure 7.20. For $H = 3$, an offset is provided to the pipelining configuration function for each instance of the S-and-M networks to compensate for the fact that the sorting network is implemented in parts, see Figure 7.17. For example, all the S-networks have the offset set to 0, as they all

[2] Totaling 13 bits.

start in the first stage. The remaining four rows of M-networks start after the 12 stages of the S-network and are spaced by the five stages of the M-networks, i.e., the offset values are set to $\{12, 17, 22, 27\}$, respectively.

Notice that using a function to determine if a given stage should be pipelined or not, instead of having this information hard-coded in the sorting network representation array, enables the implementation of different values of delay D using generic parameters, instead of regenerating the VHDL sorting package for each value of D.

For the performance analysis implemented in this thesis, the sorting unit is wrapped by a block, so-called out-of-context wrapper, that implements a register for every input of the sorting unit. Without implementing this register or an equivalent input delay constraint, the logic path from the sorting unit input to the first pipelining register is not checked in Static Timing Analysis (STA), leading to inaccurate timings results. The input register is not accounted for in the value of L.

8.2.7 Vendor-specific design flow

The synthesis process has been configured to run in the *out-of-context* mode. This mode prevents I/O buffer insertion for synthesis and downstream implementation steps [92]. This enables early estimation of logic resource usage and timing performance for a given block before the remaining part of the firmware is complete. For convenience, the MUCTPI firmware with all the blocks fully implemented is referred to as final firmware.

The early estimation, using the out-of-context synthesis mode, comes at the price of inaccurate results if the actual I/O location is a dominant factor in limiting the sorting unit timing performance, once the sorting unit is integrated to the final firmware. This is particularly true for Stacked Silicon Interconnect (SSI) technology devices [93], where the device is implemented using multiple die slices, which are often referred to as Super Logic Region (SLR). The MUCTPI MSP FPGA is implemented using 3 SLRs joined by interposers. The interposer connections cause delay penalty when data cross from one SLR to another.

For instance, if the crossing of SLR regions [93] is implemented within the sorting unit because the actual I/O location, the final firmware timing performance can be different from the performance estimated using the out-of-context synthesis mode. In the out-of-context mode, the sorting unit is fully implemented within one SLR due to the fact that I/O buffers are not inserted, and the overall logic utilization is low. If all the SLR crossings are exclusively implemented in the blocks before the sorting unit, i.e., the SL interface, overlap handling, and masking units, the results from the out-of-context synthesis mode are expected to be similar to the results from the final firmware. One could still avoid SLR crossings in the sorting unit

using floorplanning. Floorplanning can limit the sorting unit implementation within a single SLR.

A second synthesis setting named *flatten_hierarchy* [89] has been investigated using the two following options:

- **R = 0** : Instructs the synthesis tool never to flatten the hierarchy. The output of synthesis has the same hierarchy as the original RTL.
- **R = 1** : When set, the synthesis tool flattens the hierarchy, perform synthesis, and then rebuild the hierarchy based on the original RTL. This value allows the quality-of-result benefit of cross-boundary optimizations, with a final hierarchy similar to the RTL for ease of analysis.

Table 8.2 shows the different values used for latency, hierarchy, architecture, and flattening options. Sixty-four implementation candidates are defined. The performance results for each of the 64 options are covered in Section 8.2.9.

Table 8.2 – RTL implementation options and values

Option	Values
L	$1 \leq L \leq 8$
M	$\{0, 1\}$
H	$\{3, 2\}$
R	$\{0, 1\}$

8.2.8 Design verification

The self-checking functional simulation testbench has been written in Python using the Cocotb functional verification framework [94]. The same testbench is used for all the implementation options shown in Table 8.2, except for different values of R. The option R is defined in the synthesis configuration as it does not depend on the RTL description under test. Random muon candidates for 100.000 BCs are generated and connected to the sorting unit input. Then, the following tests are performed:

1. **Simulation model check:** The random muon candidates are connected to a Python simulation model of the sorting network. The simulation model inherits SNpy methods to compute the expected sorting network output. Then, the output of the sorting network is compared to the output given by the simulation model. The entire muon candidate information from the 16 highest p_T muon candidates, i.e., muon candidate number, p_T, RoI, and flags, are checked for errors.

2. **p_T-only check:** This test compares the 16 highest p_T values given by the sorting unit against the 16 highest p_T values using a builtin Python sorting function. This test is independent of the simulation model, and it is used as a sanity check. However, it is limited because the muon candidate number, RoI, and flags are not checked.

3. **Latency check:** Using simulation timestamps, the phase offset between the input and output is checked against the expected latency value for each value of L.

The sorting unit has been checked against errors using the three tests for all the implementation options. No errors have been found.

8.2.9 Implementation results

Tables 8.3 and 8.4 show the RTL implementation results, of the MUCTPI sorting unit, for $1 \leq L \leq 4$ and $5 \leq L \leq 8$, respectively, where L represents the total delay in the sorting unit, see Sections 8.2.3 and 8.2.5. M represents the architecture option, see Section 8.2.5. H represent the number of hierarchical levels, see Section 8.2.4. R represent the synthesis flatten hierarchy option, see Section 8.2.7.

The Worst Negative Slack (WNS) is the worst slack of all the timing paths for max delay analysis. It can be positive or negative. The Total Negative Slack (TNS) is the sum of all WNS violations when considering only the worst timing violation between two endpoints. The TNS value can be 0 ns when all timing constraints are met for max delay analysis, or negative when there are timing violations. The Worst Hold Slack (WHS) is the worst slack of all the timing paths for min delay analysis. It can also be positive or negative. A design reaches timing closure when all timing requirements, such as WNS and WHS are positive for all Process Voltage Temperature (PVT) corners [93, 91]. In Tables 8.3 and 8.4, negative values of WNS, and TNS are highlighted in red. Power represents the estimated dissipated power in Watts (W). LUT, FF, and LUTR represent the utilization of LUT, flip-flops, and LUT RAMs. The LUT RAM specifies how many of the LUTs are being used as memory. The LUT RAM is indicated separately because only a subset of the LUTs can be used as memory elements, such as shift registers and distributed memories [95]. Δ_{TS} and Δ_{TI} represent the synthesis and implementation processing time, respectively. The time is formatted using the ISO 8601 extended format [96]. Implementation options indicated by "-" did not complete synthesis after one month processing time.

Timing performance

An implementation can only be safely used if timing closure is achieved. For the MUCTPI sorting unit, the hold timing analysis is successful for all the implementation options discussed

Table 8.3 – RTL implementation results for $1 \leq L \leq 4$

L	M	H	R	WNS	TNS	WHS	Power	LUT	FF	LUTR	Δ_{TS}	Δ_{TI}
1	0	3	0	-15.02	-5574.77	0.09	7.01	100855	6034	0	00:21:01	00:58:31
			1	-17.34	-6547.35	0.05	7.91	60378	6034	0	00:21:19	00:46:00
		2	0	-	-	-	-	-	-	-	-	-
			1	-	-	-	-	-	-	-	-	-
	1	3	0	-21.1	-7396.16	0.09	5.49	60652	6034	0	00:15:32	02:02:50
			1	-21.86	-7841.56	0.05	6.22	55060	6034	0	00:16:44	02:22:25
		2	0	-21.57	-7535.19	0.21	5.49	60699	6034	0	00:29:09	12:30:44
			1	-28.14	-9649.24	0.05	6.49	60455	6034	0	00:28:38	24:09:38
2	0	3	0	-5.79	-16178.55	0.09	6.96	98301	9146	0	00:20:46	00:53:17
			1	-6.52	-18950.93	0.1	7.65	61231	9146	0	00:19:52	00:45:47
		2	0	-5.53	-15961.54	0.05	6.93	98462	9157	0	72:48:29	00:54:46
			1	-6.18	-18225.33	0.05	7.44	72399	9157	0	72:35:57	00:53:00
	1	3	0	-14.41	-5496.41	0.04	5.05	63030	10656	0	00:15:55	00:55:54
			1	-15.88	-10374.03	0.05	5.81	55055	10947	0	00:15:44	00:47:40
		2	0	-	-	-	-	-	-	-	-	-
			1	-	-	-	-	-	-	-	-	-
3	0	3	0	-1.92	-9469.74	0.06	6.62	73507	13567	0	00:19:22	00:47:45
			1	-2.57	-12087.92	0.07	7.47	63163	13565	1	00:20:28	00:46:11
		2	0	-1.78	-8113.61	0.05	6.51	73680	13616	1	38:47:49	00:55:25
			1	-2.15	-11337.58	0.05	7.37	74694	13616	1	40:50:12	00:55:34
	1	3	0	-4.5	-6689.91	0.04	5.02	62277	16331	0	00:17:55	00:46:57
			1	-5.71	-10548.47	0.04	5.78	55063	16649	1	00:17:17	00:46:49
		2	0	-4.6	-7043.31	0.04	5.08	62585	16332	0	72:42:47	00:48:28
			1	-5.57	-10217.15	0.04	5.73	56460	16652	5	72:11:06	00:47:43
4	0	3	0	0.01	0	0.05	6.39	69663	16740	0	00:22:25	00:39:58
			1	-0.52	-1609.48	0.06	7.21	59326	16737	1	00:22:00	00:44:29
		2	0	0.02	0	0.04	6.42	67995	16774	1	26:01:10	00:45:42
			1	-0.47	-975.85	0.05	7.34	67138	16724	25	27:17:57	00:50:41
	1	3	0	-1.38	-2661.11	0.04	5.02	59492	14136	4224	00:17:26	00:45:20
			1	-1.64	-4558.7	0.04	5.71	58139	14397	4225	00:18:21	00:47:41
		2	0	-0.98	-1954.83	0.04	5.03	59553	14149	4225	39:52:58	00:49:50
			1	-1.98	-5498.06	0.05	5.8	61097	14418	4237	40:31:22	00:43:55

here. However, the setup timing analysis results, i.e., WNS and TNS, are highly dependent on L, because, if more stages of the logic are pipelined, the implementation tool has more timing slack to accommodate the logic and routing delays. Secondarily, the results show a dependence on the architecture option. Implementation options without the multiplexor, $M = 0$, present a WNS up to 3 times higher, compared to the equivalent option with the multiplexor, $M = 1$. Notice that higher WNS represents better timing performance. The implementation option $M = 0$ has an additional clock cycle for the sorting network, compared to the option $M = 1$, see Section 8.2.5. It has been observed that the combinational delay added by propagating the entire muon candidate information through the network, with $M = 0$, is lower than the clock period allocated to the multiplexor, with $M = 1$.

Table 8.4 – RTL implementation results for $5 \leq L \leq 8$

L	M	H	R	WNS	TNS	WHS	Power	LUT	FF	LUTR	Δ_{TS}	Δ_{TI}
5	0	3	0	0.37	0	0.08	6.3	63593	20281	0	00:21:12	00:35:18
			1	0.36	0	0.06	7.25	62280	20277	1	00:22:55	00:35:22
		2	0	0.16	0	0.05	6.32	64401	20311	1	15:48:26	00:36:54
			1	0.04	0	0.05	7.22	61425	20186	49	16:33:34	00:41:42
	1	3	0	0.06	0	0.04	5.02	56818	15591	4224	00:17:53	00:37:28
			1	-0.42	-220.17	0.04	5.61	53696	15889	4225	00:17:50	00:42:44
		2	0	0.01	0	0.04	4.96	57958	15583	4225	23:31:01	00:38:21
			1	0.03	0	0.04	5.68	59333	15896	4237	23:32:48	00:42:50
6	0	3	0	0.9	0	0.05	6.13	56395	24147	0	00:22:31	00:34:22
			1	0.7	0	0.04	6.84	53216	24142	1	00:21:22	00:33:57
		2	0	0.54	0	0.05	6.16	56567	24223	1	13:22:25	00:36:19
			1	0.66	0	0.05	6.93	59281	23950	97	13:07:54	00:35:52
	1	3	0	0.46	0	0.04	4.93	50741	17251	4224	00:18:59	00:32:54
			1	0.02	0	0.05	5.58	54394	17581	4225	00:18:22	00:37:28
		2	0	0.67	0	0.04	4.87	50597	17316	4225	15:46:06	00:33:54
			1	0.45	0	0.04	5.59	54490	17550	4250	16:23:50	00:38:51
7	0	3	0	1.14	0	0.05	6.17	56342	28644	0	00:22:35	00:33:51
			1	0.79	0	0.05	6.97	56359	28638	1	00:22:42	00:33:16
		2	0	1.08	0	0.04	6.07	56336	28704	1	13:12:33	00:36:24
			1	0.55	0	0.05	6.76	65765	28192	191	13:28:43	00:40:00
	1	3	0	0.63	0	0.04	4.86	48350	18964	4224	00:18:55	00:32:09
			1	0.61	0	0.04	5.35	49684	19262	4225	00:18:50	00:40:22
		2	0	0.66	0	0.04	4.87	48303	19022	4225	13:26:41	00:38:50
			1	0.44	0	0.04	5.35	52831	19202	4274	13:43:12	00:37:51
8	0	3	0	1.61	0	0.04	6.12	56335	31984	1	00:23:59	00:33:45
			1	1.29	0	0.05	6.63	57272	31979	1	00:23:37	00:32:31
		2	0	1.36	0	0.05	6.03	56336	32103	1	11:26:42	00:39:59
			1	1.28	0	0.04	6.66	64134	31590	191	10:14:37	00:37:38
	1	3	0	0.89	0	0.04	4.82	48283	20999	4224	00:20:08	00:32:33
			1	0.75	0	0.04	5.36	50649	21262	4225	00:20:06	00:32:09
		2	0	0.8	0	0.04	4.83	48309	21021	4225	13:29:24	00:39:47
			1	0.76	0	0.04	5.28	53886	21187	4274	12:22:54	00:39:40

Thirdly, disabling the cross-boundary synthesis optimization setting, $R = 0$, grants a small additional slack, that has been decisive in achieving timing closure for $L = 4$. However, in this case, timing closure has been achieved by a very low margin of only 20 ps, which most likely would not be achieved with higher FPGA utilization. The out-of-context project benefits from a very low FPGA utilization, which is not the case anymore after the sorting unit is integrated to the MUCTPI trigger firmware. The higher utilization reduces routing options that limit timing performance. For this reason, the two implementation options, with $L = 4$, that achieved timing closure are ignored in this work.

Implementation options with $L \geq 5$ present satisfactory timing performance. Notice that increasing L further does not help much in increasing the timing slack, indicating the existence of a timing performance plateau. This is because, for higher values of L, the routing interconnect delays become dominant when compared to the logic delay. In addition, the implementation optimization effort is reduced for the paths that have already closed timing. The observed timing performance plateau is similar to what has been observed in other works, such as a study on pipelining architectures for FPGA-based multipliers [97].

Finally, the hierarchical option H shows an ambiguous impact on the timing performance. For most of the cases, the influence is considered to be very low. However in few cases, for instance with $\{D = 5; M = 0\}$, the option $H = 3$ outperformed the option $H = 2$ with a significant margin of up to 350 ps of positive slack. In fact, the lowest-latency implementation option that reached best timing performance is $\{D = 5; M = 0; H = 3; R = 0\}$. This option has a very good WNS value of 370 ps. Moreover, the hierarchical option has a powerful impact on the synthesis and implementation time, which is covered in the next subsection.

Synthesis and implementation time

Most of the implementation tools available today can perform very well for the majority of circuit descriptions that a user provides typically. However, for some cases, depending on whether the synthesis tool can reuse or not units in a design, a significant effect in synthesis processing time has been observed. This condition holds even when the resulting logic is minimal compared to the available FPGA resources.

A huge dependence on the synthesis and implementation time to the hierarchical level has been observed for the implementation of the MUCTPI sorting unit. In some cases, the synthesis time reached prohibitive values, taking up to ≈ 220 times longer to be completed, using $H = 2$, compared to the equivalent option, using $H = 3$. Notice that there is no influence from the available FPGA resources because the overall utilization is always lower than 8.5%.

It has been understood that representing the network with more hierarchical levels, i.e., using S-and-M sub-modules, provides much lower synthesis time compared to the options that implement all the *CR*, *BR*, *C*, and *B* within the same hierarchical level. The only exception is when none of the sorting network stages is pipelined, which is the case with $\{L = 1; M = 1; H = 2\}$ that presents a low synthesis time for both $H = 3$ and $H = 2$. However, both are penalized by a very long implementation time. The results indicate that the synthesis time is further penalized depending on whether it exists or not pipelined stages in the sorting network, even if the pipelined stage offsets are fixed in the design description and register retiming optimization is disabled in the implementation tool.

This penalization is higher for lower values of D, for $D > 0$. For instance, synthesis is not even completed, after 30 days, when the sorting network is implemented with a single pipelined stage. This corresponds to implementation options $\{L = 1; M = 0; H = 2\}$ and $\{L = 2; M = 1, H = 2\}$.

Fortunately, if the option $H = 3$ is used, synthesis is always completed within ≈ 20 minutes. This satisfactory synthesis time comes with no timing performance penalty. Therefore, it is the preferred hierarchical option to be used in the RTL description of the MUCTPI sorting unit.

Resources utilization and power

The total LUT, and FF available in the MUCTPI MSP FPGA are 1,182,240, and 2,364,480, respectively [19]. The sorting unit LUT utilization ranges from 48283 (4.1%) to 100855 (8.5%), and the FF utilization ranges from 6034 (0.3%) to 32103 (1.4%), both depending from the implementation option. For all the cases, the LUT and FF utilization does not exceed 8.5% and 1.4%, respectively.

The LUT utilization is higher for lower values of L, because the implementation tool duplicates logic to improve timing. For higher values of L, not as many LUTs need to be copied, because timing is more relaxed. If one analyses only the options with more relaxed timing, i.e., $L \geq 5$, the LUT usage variation is much lower, ranging from 48283 (4.1%) to 63593 (5.4%). Notice that the usage of LUTs as memory, i.e., LUT RAM, is higher for the options with the multiplexor, i.e., $M = 1$. This is because shift registers are implemented to buffer the RoI and flags information, from the muon candidates, until the sorting network result is available.

If register duplication is disabled, only the registers explicitly described in the design are implemented. Register duplication is a synthesis optimization technique that duplicates registers in critical paths in order to ease timing [98]. Therefore, the FF usage depends only on L and the width of the inputs and outputs. But in practice, register duplication is often used, and many registers are duplicated. For this reason, the FF usage observed in this work does not have a linear dependence with L. However, even with register duplication enabled, the FF usage is very low for all the implementation options. The low FF percentage usage value is thanks to the fact that current FPGAs are rich in storage elements.

The estimated dissipated power is dependent on the overall usage of LUTs and FFs. Secondarily, the implementation options, with $R = 0$, are more power-efficient, even in the cases that the utilization is slightly higher, compared to the analogous option with $R = 1$. This is not very well understood but might be related to the fact that the techniques used in the synthesis cross-boundary optimizations, at least with the settings used in this work, reduced the logic usage at the price of higher dissipated power. An example of such trade-off is when the synthesis tools

implement Finite State Machine (FSM) and counters using one-hot state encoding, which increases logic usage but reduces toggling-rate, and consequently the power [99].

Therefore, taking into account all the factors covered here, the lowest-latency implementation option with the best timing performance is $\{D = 5; M = 0; H = 3; R = 0\}$.

8.3 HLS implementation

This section describes the HLS description of the sorting unit and the respective design flow. Sections 8.3.1 to 8.3.6 describe the HLS source code of the sorting unit. Only portions of the code are shown to reduce the amount of code printed in the thesis document. For example, include statements are omitted, and the network pairs header is truncated.

Notice that HLS data types, used in Section 8.3.1, and the optimization directives, used in Sections 8.3.1, 8.3.2 and 8.3.4 to 8.3.6, are targeted to Xilinx Vivado HLS, but similar directives also exist in other HLS tools, such as Mentor Graphics Catapult HLS [100], and Intel HLS Compiler [101].

8.3.1 Data Structure

Source code 8.1 shows the C description of the muon candidate data structure, introduced in Section 8.1.1. Lines 9-12 defines the width of each of the data members. Each of the data member types is defined in lines 14 to 17 using the *ap_uint* data type, which is an arbitrary precision unsigned integer data type. The arbitrary precision data types are beneficial because C-based native data types are all on 8-bit boundaries (8, 16, 32, 64 bits) [84].

Next, lines 19 to 30 show data structure type definitions, so-called *ielement_t* and *oelement_t*, representing the input and output of the sorting unit, respectively. Different data types are used for input and output because the muon identification number is not needed in the input. The muon identification number is not needed because it can be extracted from the array index of the muon candidate inputs.

8.3.2 Comparison-exchange unit

Source code 8.2 shows the C description of the comparison-exchange unit. Line 1 shows the method definition. In HLS, the port direction is inferred automatically. A function parameter that is only read within the function is inferred as input. On the other hand, a function parameter that is only written within the function is inferred as output. In case a function parameter is both read and written within the function, an input and an output port are

```
9  const int PT_WIDTH = 4;   // transverse momentum (pt) width
10 const int ID_WIDTH = 9;   // identification (id) number width
11 const int ROI_WIDTH = 8;  // region of interest (roi) width
12 const int FLG_WIDTH = 4;  // flags width
13
14 typedef ap_uint<PT_WIDTH> mpt_t;    // pt arbitrary precision type
15 typedef ap_uint<ID_WIDTH> mid_t;    // id arbitrary precision type
16 typedef ap_uint<ROI_WIDTH> mroi_t;  // roi arbitrary precision type
17 typedef ap_uint<FLG_WIDTH> mflg_t;  // flags arbitrary precision type
18
19 typedef struct { // struct for each input muon candidate
20     mpt_t pt;    // transverse momentum
21     mroi_t roi;  // region of interest
22     mflg_t flg;  // flags
23 } ielement_t;    // name of struct type: ielement_t
24
25 typedef struct { // struct for each output muon candidate
26     mid_t id;    // muon identification number
27     mpt_t pt;    // transverse momentum
28     mroi_t roi;  // region of interest
29     mflg_t flg;  // flags
30 } oelement_t;    // name of struct type: oelement_t
```

Source Code 8.1 – Muon candidate structure definition

implemented. One can set a function parameter as constant to make sure it is never assigned within the function, preventing it from being implemented as an output. The first function parameter is the pointer to the muon candidate array[3]. HLS translates this pointer to an input and an output because it is read and written in the function. Next, two more constant parameters represent the pair of input integers $\{a, b\}$. This pair corresponds to the pair of inputs that are compared-exchanged in a given method call.

When the C code includes a hierarchy of sub-functions, the final RTL design includes a hierarchy of modules or entities that have a one-to-one correspondence with the original C function hierarchy. All instances of a function use the same RTL implementation or block. Line 5 shows the *HLS INLINE* directive, which is the first optimization directive described in this thesis. This optimization directive removes the function as a separate entity in the hierarchy. This optimization prevents reusing a single block for all the instances of this function, which would

[3] In C, an array name is a constant pointer to the first element of the array.

```
2  // C unit method: input/output 352 muon candidates, input pair {a,b}
3  void compare_exchange(oelement_t data[I], const int a, const int b)
4  {
5      #pragma HLS INLINE              // removing function hierarchy
6      oelement_t t;                   // temporary swap variable
7      if (data[a].pt < data[b].pt) {  // pt comparator
8          // swapping data[a] and data[b]
9          t = data[a];
10         data[a] = data[b];
11         data[b] = t;
12     }
```

Source Code 8.2 – Comparator-exchange unit

increase latency [84]. In HLS, optimizations directives are described using one of the following two options:

- **Source code (#pragma):** Optimization directive inserted directly in the C source code. Normally, it is recommended when a given directive is common for all the implementation options.
- **Directive file (TCL command):** Optimization directive inserted in a Tool Command Language (TCL) file. It is recommended when a given directive changes among different implementation options.

The use of such optimization directives eases the design exploration by exploiting pre-compiled HLS libraries that are able to generate several implementation variations, also known as *HLS solutions*, without changing the C source code.

The lines 6-11 describe the comparison-exchange unit functionality, shown in Figure 8.2. If the p_T value of the first input is lower than the second, all the data members from both inputs are swapped. The swap operation uses a temporary variable to hold the data from the first element, then the data from the second element is assigned to the first. Finally, the data in the temporary variable is assigned to the second element.

Notice that describing the CR, B, and BR units are not needed because parallelism and cycle details are not explicitly defined in the C source code, but inferred in the scheduling HLS synthesis step.

8.3.3 Network pairs header

Source code 8.3 shows a portion of the sorting unit header file containing the number of sorting network inputs and outputs, and all 2528 comparison-exchange pairs of the MUCTPI sorting network, see Table 7.3 and the Knuth diagram shown in Figure 7.20. This header file has been automatically generated using SNpy. A constant array with dimension 2528×2 stores all the 2528 pairs of constant integers $\{a, b\}$. The pairs $\{a, b\}$ are assigned to each of the 2528 calls of the *compare_exchange* function, described in Section 8.3.2.

```
4  const int I = 352;          // number of muon candidate inputs
5  const int O = 16;           // number of muon candidate outputs
6
7  const int np = 2528;        // number of compare-exchange pairs
8  const int pairs[np][2] =    // bi-dimensional array of pairs {a,b}
9  {
10 {0,1},                      // compare-exchange pair 0001
11 {2,3},                      // compare-exchange pair 0002
12 {4,5},                      // compare-exchange pair 0003
13 {6,7},                      // compare-exchange pair 0004

   (...)

2534 {9,10},                   // compare-exchange pair 2525
2535 {11,12},                  // compare-exchange pair 2526
2536 {13,14},                  // compare-exchange pair 2527
2537 {15,176}                  // compare-exchange pair 2528
2538 };
```

Source Code 8.3 – Comparison-exchange operations

8.3.4 Top-level without multiplexor

Source code 8.4 shows the C description of the sorting unit with $M = 0$, i.e., all the muon candidate information propagates through the sorting network, and no output multiplexor is needed.

Line 14 shows the method definition with the input and outputs defined as arrays with the length I and O, respectively. I and O are defined in Source code 8.3. Even in the cases that input and output have the same size and data types, it is not recommended to share the same pointer for input and outputs at the top-level. This is to avoid the so-called write-after-read anti-dependence that limits pipelining performance [84].

```
13 }
14 // sorting unit method: 352 candidate inputs and 16 outputs
15 void sorting_unit(const ielement_t idata[I], oelement_t odata[O])
16 {   // block, port, and array directives, see text for more information
17     #pragma HLS INTERFACE ap_ctrl_hs port=return
18     #pragma HLS INTERFACE ap_none port=idata
19     #pragma HLS INTERFACE ap_none register port=odata
20     #pragma HLS ARRAY_PARTITION variable = idata complete dim = 1
21     #pragma HLS ARRAY_PARTITION variable = odata complete dim = 1
22     #pragma HLS ARRAY_PARTITION variable = data complete dim = 1
23
24     // copying input and id to internal muon candidate array
25     oelement_t data[I];             // internal candidate array
26     for (int i = 0; i < I; i++) {   // loop through 352 inputs
27         #pragma HLS UNROLL          // implementing loop in parallel
28         data[i].pt = idata[i].pt;   // read pt from input
29         data[i].roi = idata[i].roi; // read roi from input
30         data[i].flg = idata[i].flg; // read flg from input
31         data[i].id = i;             // read id from loop index
32     }
33
34     // applying the 2528 compare-exchange operations
35     for (int i = 0; i < np; i++) {  // loop through 2528 pairs
36         #pragma HLS UNROLL          // implementing loop in parallel
37         compare_exchange(data, pairs[i][0], pairs[i][1]); // C unit
38     }
39
40     // copying 16 highest-pt candidate information to output
41     for (int i = 0; i < O; i++) {   // loop through 16 outputs
42         #pragma HLS UNROLL          // implementing loop in parallel
43         odata[i] = data[i];         // copy candidate information
44     }
```

Source Code 8.4 – Top-level sorting unit when $M = 0$

Line 16 sets the block-level interface protocol to *ap_ctrl_hs*, which implements a hand-shake protocol using the following ports:

- **ap_start**: Input that acts as a data valid port, indicating that the block can process the input data.

- **ap_ready**: Output that indicates when the block is ready to accept new inputs.
- **ap_idle**: Output indicating that input data has been received and the unit is busy processing data.
- **ap_done**: Output that indicates when output data are valid, and can be read by the downstream block.

Lines 17 and 18 set the port-level interface protocols to *ap_none* for input and output ports, respectively. This interface protocol implements wire ports with no associated handshake signal. Each structure member of each array index is mapped to an individual port. If a handshake protocol is associated with the input or output array, handshake signals would be implemented for each individual port, instead of a single signal for the complete input and output array. The additional parameter called register, present only for the output, indicates that the individual output port is registered.

In HLS, arrays are synthesized into block RAM by default. Lines 19-21 make sure the arrays are partitioned into individual registers to improve access to data and remove block RAM bottlenecks. Lines 23 to 31 read the p_T, RoI, and flags from the input and the muon identification number from the array index, and assign them to an internal variable that stores all the muon candidate information.

In HLS, loops in the C functions are kept rolled by default. This way, synthesis creates the logic for implementing one iteration of the loop, and the same logic is reused for each loop iteration of the loop, sequentially. The optimization directive *HLS UNROLL* is used in lines 26, 35, and 41 to make sure the respective loops are unrolled, allowing all the iterations to occur in parallel. In fact, all the loop iterations are implemented in parallel only when no data dependencies are identified.

Lines 33 to 37 implement the 2528 calls for the *compare_exchage* function. Each call feeds the same data array, assigning a different $\{a, b\}$ pair for each iteration. After all the 2528 iterations, the 16 highest p_T elements are placed in the top 16 array index positions.

Lines 39 to 43 assign the top 16 array indexes to the output port. The output contains all the muon candidate information for the 16 highest p_T muon candidates.

8.3.5 Top-level with multiplexor

Source codes 8.5 and 8.6 show the portions of the code that are different, with respect to Source codes 8.4 and 8.5, when the multiplexor is used.

```
19 typedef struct { // struct that propagates through the network when M=1
20     mid_t id;    // muon identification number
21     mpt_t pt;    // transverse momentum
22 } element_t;     // name of struct type: element_t
```

Source Code 8.5 – Reduced muon candidate structure definition when $M = 1$

The lines 19-23 of Source code 8.5 shows the definition of the reduced muon data structure, *element_t*, that is propagated through the sorting network when the output multiplexor is used. This data structure contains only the muon sector identification number and p_T.

```
23     // copying only pt and id to internal muon candidate array
24     element_t data[I];              // internal candidate array
25     for (int i = 0; i < I; i++) {   // loop through 352 inputs
26         #pragma HLS UNROLL          // implementing loop in parallel
27         data[i].pt = idata[i].pt;   // read pt from input
28         data[i].id = i;             // read id from loop index
29     }

 (...)

37     // copying 16 highest-pt candidate information to output
38     int id_temp;                    // temporary id variable
39     for (int i = 0; i < O; i++) {   // loop through 16 outputs
40         #pragma HLS UNROLL          // implementing loop in parallel
41         id_temp = data[i].id;       // id of Nth pt-highest muon
42         odata[i].id = data[i].id;   // read id from network
43         odata[i].pt = data[i].pt;   // read pt from network
44         odata[i].roi = idata[id_temp].roi; // multiplex input roi
45         odata[i].flg = idata[id_temp].flg; // multiplex input flags
46     }
```

Source Code 8.6 – Top-level sorting unit when $M = 1$

The lines 23 to 29 of Source code 8.6 show that only the muon identification and the p_T are assigned to the data array that drives the comparison exchanges loop, shown in Source code 8.4. Lines 37 to 46 of Source code 8.6 read the p_T and muon identification number from the comparison exchanges loop output, and the RoI and flags multiplexed from the input data.

8.3.6 Exploring different solutions

Source code 8.7 shows the TCL commands used to define two solution-specific optimization directives.

```
1 # setting minimum and maximum latency requirement
2 set_directive_latency -min $L -max $L "sorting_unit"
3 # setting iteration interval requirement
4 set_directive_pipeline -II $II "sorting_unit"
```

Source Code 8.7 – HLS solution-specific directives

Line 2 shows the minimum and maximum *sorting_unit* latency requirement for each solution. The latency requirement is defined in the same range as in the RTL design flow, i.e. $1 \leq L \leq 8$.

By default, functions are not pipelined. This means that when a function is reused for several iterations of a loop, the function is not able to receive new data every clock cycle. A pipelined function or loop can process new inputs every N clock cycles, where N is the Iteration Interval (II). Line 4 constrains the *sorting_unit* with the pipeline optimization directive using a solution-specific value for II.

The II has not been discussed in the RTL design flow because it is difficult to write a generic RTL description that reuses instances of the compare-exchange unit according to the value of II. In the MUCTPI, the compare-exchange operations can be reused because new input data are received only in 1 out of 4 clock cycles. This is because the sorting unit runs in 160 MHz, and the bunch crossing rate is 40 MHz. In HLS, relaxing II is achieved by setting a single optimization directive. Then, one more implementation option is added, i.e., $II = 4$, to the design exploration. On the other hand, the hierarchical options $H = 3$ and $H = 2$ make less sense in the HLS design flow, because all the compare-exchange units are flattened.

Similar, to the RTL design flow, the options $R = 0$ and $R = 1$ are used to explore how cross-boundary optimization options influence the implementation results of the HLS-driven RTL description. Table 8.5 shows all the values for $L, II, M,$ and R implementation options. The combination of these values result in 64 HLS solutions.

8.3.7 Vendor-specific design flow

Figure 8.9 [84] shows the Xilinx Vivado HLS design flow, which is the HLS vendor-specific design flow used in this thesis. The top part shows all the source files provided by the user, i.e:

Table 8.5 – HLS implementation options and values

Option	Values
L	$1 \leq L \leq 8$
II	$\{1,4\}$
M	$\{0,1\}$
R	$\{0,1\}$

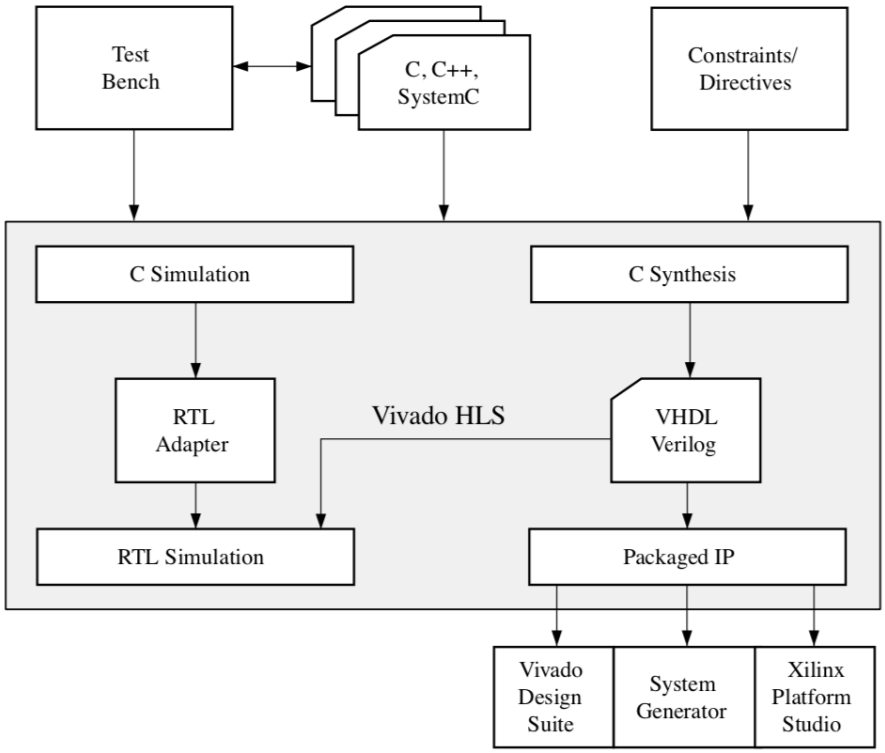


Figure 8.9 – Xilinx Vivado HLS design flow

- **C function:** Primary input to Vivado HLS, it describes the HLS software description of the circuit to be implemented. It can be written in C, C++, or SystemC
- **C test bench:** Software routines and data files used to test the functionality of the C function.
- **Directives:** Optimization directives written in a TCL file, see Sections 8.3.2 and 8.3.6
- **Constraints:** TCL statements that defines clock period, clock uncertainty, and HLS-driven RTL synthesis options. In HLS, the clock uncertainty is used to over-constrain the C synthesis, in an effort to accommodate routing delays in the HLS-driven RTL design flow. C synthesis is defined later in this section.

For the sorting unit, the HLS software description of the circuit and the testbench are written in C. The constraint and directives are defined using TCL. Sections 8.3.1 to 8.3.5 describe the C function and the optimization directive file. The C test bench and constraint files are omitted in the thesis document but can be accessed at [78].

The center of the figure shows the processes and results generated using Xilinx Vivado HLS. On the left and right sides are described the HLS simulation and synthesis features of Vivado HLS.

The HLS synthesis is performed using the two following steps:

1. **C synthesis** is the process that synthesizes the design to an RTL implementation, i.e., converting software to hardware representation. The results are generated using VHDL and Verilog hardware description languages.
2. **Packaged IP** interfaces the RTL generated files to the HLS-driven RTL vendor-specific design flow. The path through the Xilinx Vivado Design Suite is used in this thesis. The System Generator and Xilinx Platform Studio paths are not covered here.

The C synthesis, RTL simulation, and Packaged IP processes are solution-specific and have to be executed for each solution being investigated. A TCL script has been written to control Vivado HLS in the implementation of the 64 solutions, described in Section 8.3.6.

For the performance analysis implemented in this thesis, each of the 64 solutions is wrapped by an out-of-context wrapper, in the HLS-driven RTL vendor-specific design flow. This wrapper is equivalent to the out-of-context wrapper used in the RTL design flow. The reason why using it is described in Section 8.2.6. Analogous to the RTL design flow, the input register implemented in the out-of-context wrapper is not accounted for in the value of L.

The HLS simulation is performed using the two following steps:

1. **C simulation** is an early verification mechanism to check if the result from the C function is correct prior to C synthesis.
2. **RTL simulation** runs after C synthesis, and checks the result from the RTL description using a RTL simulator. The built-in Xilinx Vivado Simulator is used in this thesis.

Both C and RTL simulation shares the same C test bench file. All the software pieces to interface the C testbench to the RTL simulator are automatically generated by the RTL adapter, part of the Xilinx Vivado HLS functionality.

SNpy has been used to generate random stimulus and golden reference files of the sorting unit to drive the C and RTL simulation steps. The same stimulus and golden reference files, containing random muon candidate information corresponding to 100.000 bunch crossings, have been used to check against errors all the 64 solutions. No errors have been found in both C and RTL simulations.

8.3.8 Implementation results

Tables 8.6 and 8.7 show the MUCTPI sorting network HLS implementation results for $1 \leq L \leq 4$ and $5 \leq L \leq 8$, respectively. Both tables are subdivided into the three following column groups:

1. **Options:** Value for the implementation options L, M, II and R, described in Section 8.3.6, used in each of the 64 HLS solutions.

2. **HLS:** Results obtained in the HLS desgin flow. II' represents the actual II obtained during C synthesis. In HLS, the actual II can be different to the requested II based on data dependencies [84]. WNS, LUT, and FF represent timing performance and logic usage estimates obtained during C synthesis. The three metrics are described in Section 8.2.9.

3. **HLS-driven RTL:** Results obtained in the HLS-driven RTL design flow using the hardware description of the sorting unit generated in the HLS design flow. The metrics WNS, TNS, WHS, Power, LUT, FF, LUTR, Δ_{TS}, and Δ_{TI} are described in Section 8.2.9.

Negative values of WNS, and TNS are highlighted in red.

Timing performance

Following the same principle used in RTL, an implementation option is valid when the WNS and WHS values are positive. WHS is positive for all the 64 sorting unit implementation options. Therefore it is not discussed further in this section. Both the HLS WNS estimate and the actual HLS-driven RTL WNS value are highly dependent of L, because, if more stages of the logic are pipelined, the implementation tool has more timing slack to accommodate the logic and routing delays. The early HLS estimation of WNS is reasonably accurate for some implementation options, but in some cases, it has been over-pessimistic when compared to the actual WNS after the HLS-driven RTL implementation. The HLS estimation of WNS has to be considered carefully not to discard implementation options based only on the timing estimates given by C synthesis. One has to take into account only the final timing results provided by the HLS-driven RTL design flow.

Table 8.6 – HLS implementation results for $1 \leq L \leq 4$

Options				HLS				HLS-driven RTL								
L	M	II	R	II'	WNS	LUT	FF	WNS	TNS	WHS	Power	LUT	FF	LUTR	Δ_{TS}	Δ_{TI}
1	0	1	0	1	-23.12	134521	402	-21.12	-7909.24	0.08	8.11	73329	6036	0	00:23:59	00:50:44
			1	1	-23.12	134521	402	-22.51	-8472.54	0.13	8.28	73599	6038	0	00:25:40	00:46:12
		4	0	2	-23.12	134532	402	-20.23	-7522.05	0.05	8.01	73575	6052	0	00:24:21	00:53:34
			1	2	-23.12	134532	402	-21.02	-7913.68	0.06	8.07	72861	6042	0	00:26:43	00:48:25
	1	1	0	1	-24.53	138457	402	-28.25	-10357.58	0.23	6.24	65504	6046	0	00:19:30	08:15:06
			1	1	-24.53	138457	402	-23.36	-8328.31	0.15	6.47	65195	6045	0	00:22:53	00:53:55
		4	0	2	-24.53	138468	402	-36.57	-12005.19	0.06	6.67	67691	6041	0	00:21:30	08:09:28
			1	2	-24.53	138468	402	-26.06	-9253.09	0.06	6.76	69415	6043	0	00:22:19	00:49:54
2	0	1	0	1	-11.27	134521	9066	-8.78	-11507.93	0.06	7.46	73013	14570	0	00:27:02	00:35:02
			1	1	-11.27	134521	9066	-7.83	-13639.70	0.06	7.30	69597	14557	0	00:22:22	00:43:00
		4	0	3	-11.27	134538	9066	-7.39	-11226.14	0.05	6.05	66662	14558	0	00:20:50	00:42:29
			1	3	-11.27	134538	9066	-6.89	-11390.01	0.05	6.17	67907	14555	0	00:23:26	00:48:05
	1	1	0	1	-12.69	138457	8168	-17.74	-19462.54	0.05	6.23	53674	13676	0	00:16:36	08:39:20
			1	1	-12.69	138457	8168	-11.51	-14718.08	0.04	6.06	57962	13686	0	00:20:17	00:48:48
		4	0	3	-12.69	138474	3944	-14.17	-16477.29	0.06	5.57	55487	9454	0	00:23:30	09:02:42
			1	3	-12.69	138474	3944	-13.27	-16152.58	0.08	5.64	55951	9455	0	00:26:37	00:39:42
3	0	1	0	1	-5.08	134521	12151	-2.39	-1869.97	0.05	6.51	57130	17721	0	00:19:11	00:37:36
			1	1	-5.08	134521	12151	-2.19	-1924.01	0.04	6.54	57263	17720	0	00:21:42	00:45:01
		4	0	4	-5.08	134544	9289	-3.13	-2378.44	0.05	4.48	56203	14884	0	00:17:06	00:40:50
			1	4	-5.08	134544	9289	-2.75	-2792.73	0.04	4.58	57212	14876	0	00:18:19	00:40:41
	1	1	0	1	-5.59	138457	14076	-10.05	-5595.34	0.02	5.66	50444	19578	0	00:17:20	08:26:37
			1	1	-5.59	138457	14076	-7.03	-5403.16	0.04	5.57	50895	19568	0	00:18:25	08:00:30
		4	0	4	-5.59	138480	5628	-7.38	-4895.43	0.04	4.71	53736	11123	0	00:26:20	03:59:41
			1	4	-5.59	138480	5628	-5.85	-4292.17	0.04	4.79	52378	11127	0	00:20:48	01:19:19
4	0	1	0	1	-2.86	134521	15378	-1.04	-410.06	0.04	6.41	54430	20911	0	00:19:20	00:30:54
			1	1	-2.86	134521	15378	-0.25	-11.52	0.04	6.52	57197	20911	0	00:25:41	00:41:22
		4	0	4	-2.86	134566	11155	-0.46	-62.50	0.04	5.10	55110	16693	0	00:24:41	00:32:01
			1	4	-2.86	134566	11155	-0.52	-133.72	0.05	5.11	55804	16694	0	00:20:21	00:38:34
	1	1	0	1	-2.86	160985	55891	-3.54	-1807.18	0.05	5.56	51798	16376	4224	00:18:12	02:32:32
			1	1	-2.86	160985	55891	-3.41	-1545.52	0.05	5.66	52788	16381	4224	00:21:07	00:51:04
		4	0	4	-2.86	138502	6597	-3.35	-1573.54	0.05	4.88	49026	12149	0	00:20:08	01:09:17
			1	4	-2.86	138502	6597	-2.79	-1210.14	0.04	4.84	49950	12146	0	00:16:59	00:47:46

Table 8.7 – HLS implementation results for $5 \leq L \leq 8$

Options				HLS				HLS-driven RTL								
L	**M**	**II**	**R**	**II'**	**WNS**	**LUT**	**FF**	**WNS**	**TNS**	**WHS**	**Power**	**LUT**	**FF**	**LUTR**	Δ_{TS}	Δ_{TI}
5	0	1	0	1	-0.85	134521	17568	0.40	0.00	0.04	6.37	54291	23144	0	00:17:26	00:24:50
			1	1	-0.85	134521	17568	0.54	0.00	0.04	6.33	54216	23144	0	00:17:22	00:24:32
		4	0	4	-0.85	134568	10073	0.13	0.00	0.04	4.04	56348	15675	0	00:19:28	00:26:44
			1	4	-0.85	134568	10073	0.24	0.00	0.04	3.96	54017	15675	0	00:16:25	00:26:07
	1	1	0	1	-0.84	160985	57012	-2.26	-651.56	0.04	5.66	51354	17504	4224	00:17:02	02:20:03
			1	1	-0.84	160985	57012	-2.39	-675.54	0.04	5.67	52423	17498	4224	00:20:55	07:42:51
		4	0	4	-0.84	138504	11692	-0.63	-107.41	0.05	4.45	48141	17236	0	00:16:22	00:49:01
			1	4	-0.84	138504	11692	-1.00	-182.66	0.04	4.45	48648	17235	0	00:16:33	01:13:16
6	0	1	0	1	0.83	134521	18392	0.50	0.00	0.04	6.27	52301	23968	0	00:19:23	00:24:22
			1	1	0.83	134521	18392	0.51	0.00	0.04	6.33	52815	23968	0	00:19:59	00:24:41
		4	0	4	0.81	134568	11020	0.35	0.00	0.05	4.47	51219	16621	0	00:19:10	00:25:30
			1	4	0.81	134568	11020	0.33	0.00	0.04	4.44	51457	16621	0	00:16:58	00:26:11
	1	1	0	1	0.79	160985	57604	-0.53	-44.08	0.04	5.61	51194	18124	4224	00:14:53	00:54:10
			1	1	0.79	160985	57604	0.04	0.00	0.04	5.53	51228	18124	4224	00:14:57	00:35:35
		4	0	4	0.79	138504	12164	-0.27	-2.88	0.05	4.40	46335	17745	0	00:15:29	00:55:11
			1	4	0.79	138504	12164	0.04	0.00	0.05	4.41	46864	17745	0	00:15:46	00:26:13
7	0	1	0	1	0.83	134521	18796	0.64	0.00	0.04	6.32	53134	24374	0	00:22:25	00:25:46
			1	1	0.83	134521	18796	0.54	0.00	0.04	6.29	53069	24374	0	00:19:27	00:26:14
		4	0	4	0.81	134559	11000	0.51	0.00	0.05	4.64	52489	16603	0	00:16:13	00:27:01
			1	4	0.81	134559	11000	0.29	0.00	0.05	4.64	52811	16603	0	00:16:31	00:26:33
	1	1	0	1	0.83	160985	57797	0.14	0.00	0.04	5.50	50419	18330	4224	00:16:54	01:50:51
			1	1	0.83	160985	57797	0.20	0.00	0.04	5.53	50947	18330	4224	00:17:10	00:33:30
		4	0	4	0.83	138495	12245	0.12	0.00	0.04	4.41	46432	17839	0	00:17:17	02:12:08
			1	4	0.83	138495	12245	0.52	0.00	0.04	4.39	46818	17865	0	00:16:31	00:31:57
8	0	1	0	1	0.83	135033	20120	0.41	0.00	0.05	6.42	52949	24773	0	00:18:00	00:26:34
			1	1	0.83	135033	20120	0.72	0.00	0.03	6.30	52688	24773	0	00:17:35	00:25:34
		4	0	4	0.81	134566	11487	0.31	0.00	0.05	4.68	54463	17090	0	00:17:38	00:26:42
			1	4	0.81	134566	11487	0.38	0.00	0.04	4.68	53873	17090	0	00:18:11	00:26:30
	1	1	0	1	0.83	161049	58315	0.46	0.00	0.04	5.59	50499	18734	4224	00:18:17	00:28:58
			1	1	0.83	161049	58315	0.37	0.00	0.04	5.61	50595	18760	4224	00:18:01	00:26:29
		4	0	4	0.83	138502	12636	0.62	0.00	0.04	4.36	45869	18262	0	00:17:04	00:26:16
			1	4	0.83	138502	12636	0.62	0.00	0.04	3.88	45896	18259	0	00:17:29	00:26:41

Secondarily, similarly to what was observed in RTL, the timing performance is also dependent on whether the architecture option with the multiplexor is selected or not. The timing performance is always better for the option that the entire muon candidate information propagates through the sorting unit, which relieves the use of the output multiplexor.

Thirdly, the timing performance depends on the II. When II is relaxed to 4, the LUT and/or FF usage is reduced at the price of lower timing performance. Finally, oppose to what has been seen in RTL, the option $R = 1$ outperforms, in terms of timing, the option $R = 0$, for most of the HLS solutions. This performance gain is not understood by the author of this thesis, given that the hardware description generated by HLS is expressed in a single hierarchy level within a single file.

The first set of implementation options that reaches timing closure has $L = 5$ and $M = 0$. Notice that the respective HLS estimates did not indicate timing closure for those implementation options. The lowest-latency implementation option that reached best timing performance is $\{L = 5; M = 0; II = 1; R = 1\}$. This option has a very good WNS value of 540 ps. For higher values of L, the WNS value does not improve much, similarly to what was observed in the RTL design flow and in other works [97].

Synthesis and implementation time

The Δ_{TS} and Δ_{TI} elapsed times have not been an issue in the HLS-driven RTL design flow. Notice that Δ_{TS} and Δ_{TI} does not include the time taken in C synthesis. All the HLS solutions are synthesized, in the HLS-driven RTL design flow, in less than 30 min, each. Most of the HLS solutions are implemented, in the HLS-driven RTL design flow, in less than 1h, and none of them exceeded 10h.

The C synthesis elapsed time, i.e., the time taken to synthesize the software representation of the sorting unit to RTL, is not discussed here because of inaccurate elapsed time information from the Xilinx Vivado HLS logs. As a matter of fact, each of the implementation options has been translated to RTL description in few minutes, for higher values of L, or a couple of hours, for lower values of L.

The C synthesis processing time is less critical compared to Δ_{TS} and Δ_{TI} elapsed times, because, the HLS flow does not have to be re-executed after the RTL description of the sorting unit is generated. On the other hand, the HLS-driven RTL flow is re-executed every time the MUCTPI firmware is modified. If RTL synthesis time needs to be reduced, the sorting unit can be integrated into the MUCTPI firmware as an out-of-context block. In this case, RTL synthesis reruns only in case the block itself is modified.

Resources utilization and power

The HLS estimates for LUT and FF usage diverge significantly to the final usage values at the end of the HLS-driven RTL design flow. In some cases, HLS reported a usage three times higher than the actual value. These results indicate that one should avoid making design decisions based on HLS usage estimates.

Given that the total number of LUT and FF available in the MUCTPI MSP FPGA are 1,182,240, and 2,364,480, respectively [19]. The sorting unit LUT utilization ranges from 45869 (3.9%) to 73599 (6.2%), and the FF utilization ranges from 6036 (0.3%) to 24773 (1.0%), both depending on the implementation option.

The LUT utilization follows a very similar pattern to the one found in the RTL design flow, i.e., higher usage values for lower values of L, and lower variation for high values of L. The reason behind this pattern is described in Section 8.2.9. Similarly to the RTL design flow, the FF usage is low for all the implementation options, i.e., never exceeds 1% of the total number of registers available in the device.

The implementation option II has an influence on the overall logic utilization and dissipated power. The lower toggling rate expected at the input, for $II = 4$, is not expressed in terms of design constraints in the HLS-driven RTL design flow. Therefore, the dissipated power for $II = 4$ is overestimated here, and it is dependent only on the overall logic utilization. The requested $II = 4$ has not been achieved for $L \leq 3$ due to data dependencies. Therefore, relaxing II did not result in reducing the overall logic usage and power for $L \leq 3$. For $L \geq 4$, all the implementation options achieved the requested II value, i.e., $II' = II$. For this reason, an overall logic usage and power reduction have been observed only for $\{L \geq 4; II = 4\}$. For example, for $\{L = 5; M = 0; R = 1\}$, the option with $II = 4$ has $\approx 30\%$ lower FF usage, and dissipates $\approx 40\%$ less power than the analogous option with $II = 1$. However, this logic resource usage reduction came together with the timing performance penalty.

Therefore, taking into account all the factors covered here, the lowest-latency implementation option with the best timing performance is $\{L = 5; M = 0; II = 1; R = 1\}$.

8.4 Comparative study

This section presents a comparative study on the design effort, performance, and implementation processing time between RTL and HLS design abstractions. This study is limited to the MUCTPI sorting unit, investigated in this thesis. Therefore, extending the conclusions obtained here to different applications should be carefully analyzed in a case-by-case basis.

8.4.1 Design exploration effort

The design exploration effort is defined in this thesis as the one-time cost to design, verify, and implement the sorting unit using the RTL and HLS approaches. Only the design effort presented in this chapter is considered in this study. This is because the ideas resulting from the work presented in Chapter 7 have been used in both RTL and HLS implementation approaches, i.e., RTL and HLS have been designed from the same starting point.

The author of this thesis estimates that designing the sorting unit using the RTL approach took, at least, 10 times longer than using the HLS approach. The gain in development time can be illustrated by the complexity differences between the HLS description, shown in Source codes 8.1 to 8.4 and 8.6, and the RTL description, shown in Source codes A.1 to A.5.

HLS exempts the user from entering many design characteristics that have to be explicitly described in RTL, such as schedule of operations, resource allocation, cycle details, pipeline registers, FSM encoding, etc. This allows designers to focus on their design work as compared to detailed and mechanical RTL implementation tasks. In addition, less expert knowledge is needed, which enables different types of professionals, such as software engineers, to participate in firmware development actively.

Concerning the implementation of the MUCTPI sorting network, the parallelism and pipelining configurations have been explicitly described for each of the implementation options $0 \leq L \leq 8$ when using the RTL approach. The HLS description benefits from the fact that design characteristics, such as parallelism, cycle details, and logic resources, are not explicitly described in the C source code. Instead, they are inferred in the scheduling and binding synthesis steps. The requested, resulting characteristics, such as the total latency and II, are configured using design directives in the C source code and the directive TCL file, shown in Source code 8.7. This exempts the engineer from entering many design details that reduce the design effort and enables smoother design exploration, given that many of the implementation options can be explored without changing the source code.

HLS does not only provide a fast and high-quality path to RTL, but it also enables verifying it earlier. The HLS design approach enables the detection of functional errors in an early stage of the design flow, i.e., before C synthesis. The same C testbench code used to check the C source code is also used to test the HLS-driven RTL description, using an automatically generated RTL adaptor. This RTL adaptor creates stimulus data files, creates the required interconnecting logic, and checks the results using the software testbench. In the RTL approach, functional testing can only be performed after parallelism, and cycle details are expressed in the RTL description. Moreover, the users have to write their test benches using hardware description languages and build the RTL adaptor themselves. In brief, HLS provides a highly productive path to a high-quality well-verified RTL implementation.

Using the RTL approach, some implementation options required prohibitive synthesis and implementation elapsed times, and some others have not been completed after 30 days. The RTL code generated by HLS is synthesized significantly faster than the RTL description, written by the author of this thesis.

Design exploration of the MUCTPI sorting unit has been faster in the HLS approach because of the following reasons:

- The sorting unit HLS description is significantly simpler than the RTL description because cycle and resource details are not specified in the HLS source code.

- Design verification is more straightforward and performed earlier, i.e., before C synthesis. Moreover, HLS testbench is more comfortable to write because one can use software languages. In the RTL approach, a testbench written in software could only be used after integrating a third-party tool to the design flow.

- The overall elapsed time for synthesis and implementation is shorter in the HLS approach. Notice that when evaluating many implementation options, one can compare the possibilities quicker if synthesis and implementation times are shorter. The elapsed synthesis and implementation time are also crucial for implementation options that are not going to be selected in the end, because the performance comparison can only be completed after all the implementation results are available.

8.4.2 Performance metrics

Table 8.8 shows the implementation results from the best RTL and HLS implementation options. Both RTL and HLS descriptions achieved equivalent latency performances, i.e. both descriptions generate a working design with $L = 5$. However, for the same value of L, the best HLS implementation option, i.e. $\{L = 5, M = 0, II = 1, R = 1\}$, achieved better timing performance than the best RTL option, i.e. $\{L = 5, M = 0, H = 3, R = 0\}$. The best HLS WNS value is $\approx$ 50% greater than the best RTL value, i.e. 540 ps and 370 ps for HLS and RTL, respectively.

The increased slack comes with slightly lower logic usage and the same estimated dissipated power compared to the best RTL option. The best HLS implementation option dissipates $\approx 6.3\ W$ and requires 54,216 LUTs and 23,144 FFs, while the best RTL implementation option also dissipates $\approx 6.3\ W$, and requires 63,593 and 20,281 FFs, representing a reduction of $\approx$ 15% in LUTs and an increase of $\approx$ 12% in FFs.

Table 8.8 – Best RTL and HLS implementation options

Option	WNS	TNS	WHS	Power	LUT	FF	LUTR
RTL $\{L = 5, M = 0, H = 3, R = 0\}$	0.37	0	0.08	6.3	63593	20281	0
HLS $\{L = 5, M = 0, II = 1, R = 1\}$	0.54	0	0.04	6.3	54216	23144	0

8.5 Summary

This chapter described RTL and HLS implementation approaches in the context of the MUCTPI sorting unit implementation. Section 8.1 presented the interface of the sorting unit to the remaining part of the firmware, the muon candidate data structure, and an overview of RTL and HLS design approaches. Their differences and similarities have been explored together with an introduction to the required design effort for each implementation approach.

Section 8.2 described the RTL design flow using gradual steps. It covered the VHDL design entry, starting from the combinatorial-only sorting network, covering pipelined sorting networks, their respective configurations, different hierarchical and architectures options, and finishing at the generation of the VHDL code. Then, the respective vendor-specific design and verification flow have been presented. All the 64 implementation options have been checked for functional errors using RTL simulation. No errors have been found.

Section 8.2.9 presented the RTL implementation results, which concluded that the lowest-latency implementation option that reached best timing performance is $\{L = 5, M = 0, H = 3, R = 0\}$. It has been demonstrated that the implementation option M is the main design parameter affecting latency and timing performance, followed by H, and R. Secondarily, the effect of such design parameters in implementation time and logic resource usage has been discussed.

Section 8.3 described the HLS design flow covering the C design entry, including the data structure, the comparison-exchange unit, the header containing all the comparison-exchange operations, and the top-level files for implementation options M. At the same time, the code was being described, HLS concepts, such as the resulting hierarchy of the RTL based on C sub-functions, optimization directives, write-after-read anti-dependence, interface protocols, and loop unrolling have been presented. Next, a summary of all the implementation options, so-called HLS solutions, and an overview of HLS vendor-specific design flow has been covered.

Section 8.3.8 presented the HLS implementation results, which concluded that the lowest-latency implementation option that reached best timing performance is $\{L = 5, M = 0, II = 1, R = 1\}$. Similarly to RTL, it has been demonstrated that the latency and timing performance is mainly limited by the implementation option M, followed by II and R, respectively. Their impact in the implementation time and in the logic resources have been covered.

Section 8.4 presented a comparative study between RTL and HLS design abstractions, highlighting that both implementation options achieved the same latency performance, i.e. $L = 5$. The best WNS value of the HLS approach is ≈ 50% greater than the best RTL value, i.e. 540 ps and 370 ps for HLS and RTL, respectively. The increased WNS slack comes with a slightly lower logic usage and same dissipated power, when compared to the best RTL option. The best HLS implementation option dissipates ≈ 6.3 W and requires 54,216 LUTs and 23,144 FFs, while the best RTL implementation option also dissipates ≈ 6.3 W, but requires 63,593 and 20,281 FFs, representing a reduction of ≈ 15% in LUTs and an increase of ≈ 12% in FFs, see Table 8.8.

The HLS approach required much less effort, expert knowledge, and device-specific information to achieve slightly better results than the RTL approach. Design characteristics, such as schedule of operations, resource allocation, cycle details, pipeline registers, and FSM encoding, have been automatically inferred by the HLS tool in the scheduling and binding synthesis steps. This allows designers to focus on their design work as compared to detailed and mechanical RTL implementation tasks.

With more time for the design work, the engineer has more time to explore new architecture options and perform it earlier in the design stages. For example, by only changing an optimization directive value in the HLS approach, two different II options have been explored for the MUCTPI sorting network implementation. In this case, it has been quickly discovered that increasing II does not contribute to reduce logic resource usage and/or improve timing. Much more effort would have been required to explore different values of II in the RTL approach.

HLS does not only provide a fast and high-quality path to RTL, but it also enables verifying it earlier. One can catch functional errors before C synthesis. The earlier verification comes with no extra cost because the same C testbench used to check the C source code is also used to test the HLS-driven RTL description. The improved verification requires lower effort than the one needed in the RTL approach because a software language can be used without requiring any third-party tool. Moreover, a RTL adaptor is automatically generated to ease the interface to stimulus and golden reference files. In brief, HLS provides a highly productive path to a high-quality well-verified RTL implementation.

9 Conclusions and Outlook

This thesis document presented the upgrade of the first-level trigger system of ATLAS in order to keep the trigger output below the manageable rate of 100 kHz. To cope with the increasing luminosity, the trigger systems have to become more selective, which is achieved by routing more information from the detector to the trigger system and by processing larger parts of this information together. These two requirements introduce new challenges in the data transfer and processing of trigger systems, such as higher bandwidth and integration level. Both challenges have to be addressed, ensuring that both, hardware and firmware have low and fixed latency, and are reliable.

A summary of the results achieved is presented below.

- **Part I - Data Transfer:**
  1. Software packages to automate the testing of hundreds of high-speed serial links.
  2. MUCTPI high-speed serial links BER $< 9 \times 10^{-16}$ with a $CL = 95\%$.
  3. FPGA MGT latency of ≈ 50 ns and latency uncertainty of 3.125 ns.
  4. IP to synchronize data from 208 SL inputs with low and fixed latency. The total data transfer and synchronization latency is below 125 ns.
- **Part II - Data Processing:**
  1. Software framework to generate, optimize, combine, plot, and write VHDL and C descriptions of sorting and merging networks.
  2. MUCTPI sorting network with 13 fewer steps than the 45-step 352-key Batcher merge-exchange, odd-even, or bitonic sorting networks.
  3. MUCTPI sorting network with a very low latency value of 31.25 ns using both RTL and HLS approaches independently.

Sections 9.1 and 9.2 present the conclusions from Parts I and II, respectively. Finally, Section 9.3 presents the outlook of this Ph.D. work.

9.1 Data transfer

The first achievement of this Ph.D. work is the feasibility demonstration of using the Xilinx UltraScale transceivers, and the 14 Gb/s Broadcom MiniPODs in the MUCTPI application. The proof of concept has been demonstrated based on error-free BER tests, wide eye-diagrams, and error-free SL synchronization using the MUCTPI demonstrator. The MUCTPI demonstrator consists of a Xilinx VCU-108 evaluation board, a custom double-width FMC, so-called MPOD FMC, respective FPGA firmware, and low-level software. The demonstrator has also been used to validate the TTC information reception hardware and firmware, the measurement of online statistical eye diagrams, and the synchronization of the SL inputs from the recovered clock to the system clock domain for combined data processing, see Section 3.2.

The testing of all ≈ 330 high-speed serial links, per board, for all MUCTPI prototypes, has been automatized using two Python packages. Due to the very high number of high-speed serial connections in the MUCTPI, reading the schematics thoroughly to extract the interconnectivity, the pin assignments, and the link polarities is very difficult, time-consuming, and susceptible to human errors. The first package extracts connectivity from the back-annotated PCB netlist to generate VHDL wrappers, placements & polarity constraints, and netlist verification reports. The second package manages BER tests by generating TCL scripts to automate the mapping between links, configuring their respective polarities, running the BER tests, and eye-scan measurements. Moreover, plotting eye-diagrams, running eye-mask checks, generating horizontal, vertical, and area opening histograms, and compile all the results into a report, see Section 3.4.

The two Python packages have also been used to detect accidental polarity inversion of differential lines in the MUCTPI schematics. These errors have been discovered and fixed before the first PCB had been produced. Besides, the automatically generated VHDL wrappers and placing constraints have been used for other firmware developments in all the MUCTPI FPGAs. Moreover, these tools have also been used to create eye-diagrams from the high-speed serial links of the Barrel Calorimeter Processor board, part of the CMS Phase-II upgrade.

BER values for all on-board and off-board MUCTPI links running at 12.8 Gb/s have been measured as 9×10^{-16} with a $CL = 95\%$. This result is excellent, meaning that the BER value is lower than one error per day with a confidence level of 95%, see Section 3.5.1. This is acceptable as it corresponds to only one potential fake trigger or lost event per day.

A wide horizontal eye-diagram opening of 76% has been measured from the optical output of one of the L1Topo outputs running at 11.2 Gb/s using a high-speed oscilloscope equipped with an optical-to-electrical converter, see Section 3.5.2.

An excellent result has been measured from the eye mask compliance test. All the on-board and off-board MUCTPI links running at 6.4 Gb/s, SL bit-rate for the Phase-I upgrade, and 12.8 Gb/s, used as a stress-test, passed the eye mask compliance check for all MUCTPI prototypes, see Section 3.5.5.

An eye-diagram opening area comparative study has been used to illustrate performance results over all the MUCTPI SL links. It has been measured an average 15% opening area difference between two transceiver types running at 6.4 Gb/s in the first MUCTPI prototype. Fortunately, only one transceiver type is used for the next two prototypes. For MUCTPI V2 and V3, it has been measured a great improvement in the worst-case and average area opening values, when compared to V1. For both prototypes, the worst-case value increased from ≈ 55% to ≈ 70%, and the average value increased from ≈ 67% to ≈ 75%. For 12.8 Gb/s, the opening area has been measured ranging from ≈ 40% up to ≈ 62%, see Section 3.5.4. The smaller eye-diagram opening is not an issue because the measured BER is very low for all links.

It is a formidable result that, even when running at twice the bit-rate required for the Phase-I upgrade, all the SL links have passed the mask compliance test, and the BER value is measured to be lower than one error per day with $CL = 95\%$.

A latency measurement test system, based on a Kintex Ultrascale FPGA development kit, was developed to optimize the data-path and clock-fabric transceiver configuration in the interest of low and fixed latency. It has been measured a TX to RX transceiver latency of ≈ 50 ns, and uncertainty of 3.125 ns. The transmitter and receiver settings are used in the MUCTPI trigger data path, and the transmitter settings are used in the RPC and TGC firmware. The latency value of ≈ 50 ns is excellent and provides enough extra latency for the SL synchronization and data processing. The latency uncertainty of 3.125 ns is very low. Hence it can be absorbed by the synchronizer IP without causing any synchronization error, see Section 4.3.1.

A synchronization IP was designed to transfer data from the recovered clock of each of 208 MUCTPI FPGA on-chip transceivers to the system clock domain for combined data processing. This unit does not only synchronizes the SL data with low and fixed latency, but it also absorbs the latency uncertainty from the FPGA transceivers. This functionality is achieved by loading fixed parameters that are obtained in a one-time calibration procedure. Notice that the phase relationship between the system clock domain and each of the recovered clocks is unknown due to the length mismatch among the clock and data optical fibers and the part-to-part skew of each of the sector logic module components. Still, it is fixed because the length and the skew are time-invariant.

A comprehensive functional simulation for the synchronization IP has been implemented to check the design for errors, and elaborate the calibration procedure, see Section 5.5.6. Moreover, to measure the minimum and maximum latency read pointer offsets, measure the error-free latency variation limits, and measure the synchronization latency minimum and maximum limits. This simulation demonstrated that the latency variation tolerance is higher than the latency uncertainty measured for the MUCTPI high-speed serial links, see Section 5.5.

Integration tests with RPC and TGC subsystems, using up to 12 serial links, demonstrated that the synchronization IP operates error-free after resetting and power cycling the MUCTPI and the sector logic interfaces. The overall data transfer and synchronization latency from the transmitter to the receiver system clock domain has been measured to be lower than 125 ns, which is within the expected range given by the transceiver latency measurements and the synchronizer functional simulation, see Section 5.6.

9.2 Data processing

It was demonstrated that the MUCTPI Run 2 sorting algorithm cannot be extended from 26 and 2 to 352 and 16 input and output candidates, respectively, required for Run 3, see Section 6.3. Therefore, a solution based on sorting networks, the fastest practical method to sort data in hardware, has been conceived. Given that sorting networks are also data oblivious algorithms, adopting sorting networks seemed very suitable for the MUCTPI. Data oblivious algorithms feature fixed latency because they perform a fixed number of operations regardless of the input data pattern. The use of sorting networks fulfilled, at the same time, low and fixed latency requirements.

A Python package was developed for generating, optimizing, combining, plotting, and writing HDL and C descriptions of sorting and merging networks, see Chapters 7 and 8. Existing sorting networks have been optimized by removing compare-exchange operations from unused inputs, unused outputs, pre-sorted input ranges, and outputs ranges that are not required to be sorted, see Section 7.7.1. A comparative study that used some of these optimizations demonstrated that within the Batcher sorting methods with the number of elements $n \in \mathbb{Z} \mid 2^1 \leq n \leq 2^9$, the merge-exchange algorithm gives the lowest delay value without requiring more comparators than any of the others Batcher methods, see Section 7.8.

Next, a divide-and-conquer method was developed, see Section 7.9, to optimize sorting networks with $O \ll I$. The method divides a large sorting network problem into smaller sorting and merging networks. First, the input is divided into several combinations of groups with different sizes and sorted concurrently using the Batcher merge-exchange sorting algorithm. Second, for each of these combinations, all the respective input groups are merged using a binary tree of odd-even merging networks. Then, the fastest combination options are selected.

In an optional second step, one can further optimize the sorting part if a sorting network faster than the respective Batcher merge-exchange sorting network exists. No further optimization is possible in the merging part because the odd-even merging network is optimal when the size of the sets to be merged is equal and with a power-of-two value, see Section 7.4. For the MUCTPI application, one of the fastest combination options uses a 22-key input 16-key output sorting network that has been replaced by the fastest 22-key sorting network known, discovered by Sherenaz W. Al-Haj Baddar in 2009. Some of the compare-exchange operations from the Baddar sorting network and the Bacther odd-even merging network have been optimized away in view that only the 16 highest p_T muon candidates are required at the output. The optimized sorting and merging networks, part of the MUCTPI sorting network, are referred to as S-and-M networks.

Using the Baddar sorting network further reduced the total delay given by the divide-and-conquer method from 35 to 32 delay steps. The 32-step 352-key input 16-key output sorting network discovered for the MUCTPI application sorts the input data using 13 fewer steps than the 45-step 352-key Batcher merge-exchange, odd-even, or bitonic sorting networks. Although the results presented here are obtained in the scope of the MUCTPI sorting network, the divide-and-conquer method should be applicable for other sorting network problems when $O \ll I$.

The divide-and-conquer method requires generating, optimizing, and combining sorting and merging networks of different sizes for each of the combinations at which the networks can be combined. This extensive process has been implemented to get early comparative complexity and performance results for each combination option. This study accelerates the firmware development flow because a first step performance analysis occurs before hardware implementation. This way, only the selected option is implemented in hardware.

Before the hardware implementation, the MUCTPI sorting network was validated in software using the two following steps. First, the S-and-M networks have been tested alone for the complete set of possible input combinations using the zero-one principle, i.e., 2^{22} input combinations for the S network, and every combination of two sorted sub-sequences of length 16 for the M network, no errors have been found. Second, the entire MUCTPI sorting network. i.e., with all S-and-M network instances, has been tested using a randomly selected subset of length 2^{30} out of a total of 2^{352} input combinations using the zero-one principle. No errors have been found, see Section 7.11.

It has been estimated that testing all the 2^{352} input combinations of the MUCTPI sorting network would take 1×10^{97} days using a high-performance computer. Even if computer technology would ever advance to the point where each proton in Earth would process data at the same speed as the high-performance machine being used, it would still take 8×10^{39} millenniums to check all the combinations.

Next, the MUCTPI sorting network was developed independently using the RTL and HLS approaches. Configurable VHDL and C codes have been developed to describe the MUCTPI sorting network with different latency, architecture, hierarchical, and iteration interval options. The latency represents the total time required to output the sorted list with the 16 highest p_T muon candidates. The architecture options define if either the complete or only a subset of the muon candidate data propagate through the network. In the second case, the data subset that does not propagate through the sorting network is buffered externally and multiplexed based on the sorting network output. The hierarchical option, only applicable to the RTL approach, defines if the compare exchange operations from each of the S-and-M networks are described using sub-modules or if they are all described together in a single level of the hierarchy. The iteration interval, only applicable to the HLS approach, enables the option of relaxing the design throughput requirement in view that the input data arrive at the bunch crossing rate, and the logic runs four times faster. For both RTL and HLS design flows, an option of flattening or keeping the netlist hierarchy during RTL synthesis was explored.

For each of the RTL and HLS approaches, 64 implementation options have been explored using an automated design flow capable of controlling the FPGA implementation tool and extract the timing analysis results, logic resource usage, estimated power, and elapsed times, see Tables 8.3, 8.4, 8.6 and 8.7. RTL and HLS approaches have been able to implement the MUCTPI sorting network with a very low latency value of 31.25 ns. Both approaches presented similar performance results, with a slight advantage to HLS in terms of timing slack and logic resource usage.

The best WNS value of the HLS approach is ≈ 50% greater than the best RTL value, i.e. 540 ps and 370 ps for HLS and RTL, respectively. The increased WNS slack comes with a slightly lower logic usage and same dissipated power, when compared to the best RTL option. The best HLS implementation option dissipates ≈ 6.3 W and requires 54,216 LUTs and 23,144 FFs, while the best RTL implementation option also dissipates ≈ 6.3 W, but requires 63,593 and 20,281 FFs, representing a reduction of ≈ 15% in LUTs and an increase of ≈ 12% in FFs, see Section 8.4.

The HLS approach presented many advantages concerning the design effort when compared to the RTL approach. For example, the author of this thesis estimates that designing the MUCTPI sorting network using HLS took at least ten times less than the time needed when using the RTL approach. Also, the design verification has been simplified in the HLS design flow, given that the HLS testbench is written using a software language without requiring a third-party tool. Finally, the elapsed time for the tool to generate implementation results have been much shorter using HLS and gives better timing and logic resource results for most cases. For example, for implementing RTL options with $\{L = 5; M = 0; H = 2\}$ took up to ≈ 24 h, however the equivalent HLS option with $\{L = 5; M = 0; II = 1\}$ took less than 1 h. On the timing performance, HLS provided a WNS value of at least 400 ps and the RTL design flow reached a

maximum WNS value of 160 ps. More drastically, some RTL options have not been finished implementation after 30 days. However, no HLS implementation option took more than 10 h to finish. In fact, most of them have been implemented in less than 1 h.

9.3 Outlook

The experience from testing the MUCTPI high-speed serial links allows one to reuse some of the ideas and the tools developed to have a complete performance report of systems with hundreds of high-speed connections per board by using BER tests, mask compliance checks, and area opening histograms. Moreover, one can inherit the advantage that most of the testing tasks have been automated. This exempts the designer from thoroughly reading the schematics to extract the inter-connectivity, pin assignments, and link polarities, hence avoiding mechanical and time-consuming tasks that are also susceptible to human errors.

The latency optimized data-path and clock-fabric FPGA MGT configurations can be reused in other applications with latency requirements in the order of tens of ns. The synchronization IP and the experience from its thorough functional verification can be applied in other similar synchronization solutions that are also required to absorb latency uncertainty from MGT transceivers.

The know-how acquired with the implementation of the MUCTPI sorting unit opens the way for using sorting networks combined with the divide-and-conquer method in other applications at which low-latency sorting is needed.

The experience from implementing the MUCTPI sorting network using the HLS approach allows one in the future to use HLS for other similar algorithms that can also benefit from the highly-capable HLS ability to infer parallelism, cycle details, and required logic elements from the C source code. Moreover, with the advantage of requiring much less design effort, enabling early testing, and providing equiparable performance results compared to the RTL approach. Some other successful examples of using HLS in HEP are machine learning [102], overlap muon track finding [103], Kalman filtering [104], and jet and energy sum computation [104]. In general, HLS can significantly reduce the design effort in developing new algorithms for trigger systems in HEP.

A RTL description of the sorting unit

Source code A.1 shows the sorting network VHDL package file. It contains: First, the definition of constants, records, and array types. Secondly, method that returns the sorting network pairs. Finally, method that returns pipelining configurations.

Most of the file content is generated by SNpy [78]. Only the beginning of each sorting network pairs and pipelining configurations definitions are shown. The portion of the file printed here contains mainly the part of the file designed and manually written by the author of this thesis. The complete file, i.e. including the automatically generated part from SNpy, is ≈ 40 times longer, containing ≈ 220,000 characters.

Source code A.2 shows the VHDL description of the C, B, CR, and BR units. The generic parameter *pass_through* defines if a comparison-exchange or bypass unit is implemented, and *output_register* defines if an output register is implemented.

Source code A.3 shows the generic sorting network VHDL description for any even value of I. Therefore, this file is used both in the options $H = 3$, implementing the S-and-M units, and also for $H = 2$, implementing the flat MUCTPI sorting network.

Source code A.4 shows two VHDL architectures representing implementation options $H = 3$, names as *hier* and $H = 2$, named as *flat*. The VHDL architecture *flat* is only an wrapper to Source code A.3, while the VHDL architecture *hier* implements the hierarchical instantiation of the the S-and-M units, and the respective pipelining configuration offset, following the block diagram shown in Figure 7.17.

Source code A.5 shows the top-level sorting unit VHDL wrapper file. It contains the option to add the input register for performance analysis, instantiates $H = 3$ or $H = 2$ VHDL architecture of Source code A.4, and implements the $M = 0$ and $M = 1$ implementation options.

Source code A.1 – Sorting Network package file (truncated)

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use IEEE.math_real.all;
4
5  package csn_pkg is
6
7    constant MUON_NUMBER : integer := 352;
8    constant IDX_WIDTH   : integer := integer(ceil(log(
       real(MUON_NUMBER), real(2))));
9    constant PT_WIDTH    : integer := 4;
10   constant ROI_WIDTH   : integer := 8;
11   constant FLAGS_WIDTH : integer := 4;
12   constant in_word_w   : integer := PT_WIDTH +
       ROI_WIDTH + FLAGS_WIDTH;
13   constant out_word_w  : integer := PT_WIDTH +
       ROI_WIDTH + FLAGS_WIDTH + IDX_WIDTH;
14
15   type muon_type is record
16     idx   : std_logic_vector(IDX_WIDTH - 1 downto 0);
17     pt    : std_logic_vector(PT_WIDTH - 1 downto 0);
18     roi   : std_logic_vector(ROI_WIDTH - 1 downto 0);
19     flags : std_logic_vector(FLAGS_WIDTH - 1 downto
         0);
20   end record;
21
22   type muon_sort_type is record
23     pt  : std_logic_vector(PT_WIDTH - 1 downto 0);
24     idx : std_logic_vector(IDX_WIDTH - 1 downto 0);
25   end record;
26
27   type muon_a is array (natural range <>) of
       muon_type;
28   type muon_sort_a is array (natural range <>) of
       muon_sort_type;
29
30   type cmp_cfg is record
31     a : natural;
32     b : natural;
33     p : boolean;
34   end record;
35
36   -- has to be array of array instead of (x,y) array
       because of issues with synplify
37   type pair_cmp_cfg is array (natural range <>) of
       cmp_cfg;
38   type cfg_net_t is array (natural range <>) of
       pair_cmp_cfg;
39   type stages_a is array (natural range <>) of
       boolean;
40
41   function to_array(data : std_logic_vector; N :
       integer) return muon_a;
42   function to_stdv(muon  : muon_a; N : integer)
       return std_logic_vector;
43
44   --type cfg_net_t is array (natural range <>,
       natural range <>) of cmp_cfg;
45   function get_cfg(I         : integer) return
       cfg_net_t;
46   function get_stg(I         : integer; D : integer)
       return stages_a;
47
48   constant empty_cfg : cfg_net_t := (
49     ((a => 0, b => 1, p => false), (a => 2, b => 3, p
         => false)),
50     ((a => 0, b => 2, p => false), (a => 1, b => 3, p
         => false)),
51     ((a => 1, b => 2, p => false), (a => 0, b => 3, p
         => true))
52     );
53
54 end package csn_pkg;
55
56 package body csn_pkg is
```

```
57
58  function get_cfg(I : integer) return cfg_net_t is
59  begin
60    case I is
61      -- Sherenaz W. Al-Haj Baddar 22-key 12-step
        SORTING network
62      when 22 => return (
63        ((a   => 20, b => 21, p => false), (a => 18,
          b => 19, p => false), (a => 16, b => 17, p
          => false),  truncated (...)
64        );
65      -- M=16,N=16 Batcher Odd-even MERGING network,
        the two 16-key input sequences have to be
        sorted
66      when 32 => return (
67        ((a   => 0, b => 16, p => false), (a => 8, b
          => 24, p => false), (a => 4, b => 20, p =>
          false),  truncated (...)
68    -- Flat MUCTPI sorting netowrk
69      when 352 => return (
70        ((a    => 20, b => 21, p => false), (a => 42,
            b => 43, p => false), (a => 64, b => 65, p
          => false),  truncated (...)
71      when others => return empty_cfg;
72    end case;
73  end function get_cfg;
74
75  function get_stg(I : integer; D : integer) return
      stages_a is
76  begin
77    case I is
78      -- pipeline options for a total of 32
        comparison stages
79      when 352 =>
80        case D is
81          when 0 =>
82      return (false, false, false, false,
          truncated (...)
83            -- total number of registered stages: 0.
84          when 1 =>
85      return (false, false, false, false,
          truncated (...)
86            -- total number of registered stages: 1.
87          truncated (...)
88          when 8 =>
89           return (false, false, false, true,
              truncated (...)
90            -- total number of registered stages: 8.
91          when others =>
92           null;
93        end case;
94
95      when others => return (false, false);
96
97    end case;
98  end function get_stg;
99
100 end package body csn_pkg;
```

Source code A.2 – C, B, CR, and BR units VHDL description

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4  use work.csn_pkg.all;
5
6  entity csn_cmp is
7    generic(ascending       : boolean := False;
8            pass_through    : boolean := False;
9            output_register : boolean := False
10          );
11   port(
12     clk : in  std_logic;
13     a_i : in  muon_type;
14     b_i : in  muon_type;
15     a_o : out muon_type;
16     b_o : out muon_type
17   );
18 end entity csn_cmp;
19
20 architecture rtl of csn_cmp is
21
22   signal a_o_comb : muon_type;
23   signal b_o_comb : muon_type;
24
25 begin
26
27   process(all)
28   begin
29     if pass_through then
30       a_o_comb <= a_i;
31       b_o_comb <= b_i;
32     else
33       if (a_i.pt > b_i.pt) = ascending then
34         b_o_comb <= a_i;
35         a_o_comb <= b_i;
36       else
37         a_o_comb <= a_i;
38         b_o_comb <= b_i;
39       end if;
40     end if;
41   end process;
42
43   out_g : if output_register generate
44     process(clk)
45     begin
46       if rising_edge(clk) then
47         a_o <= a_o_comb;
48         b_o <= b_o_comb;
49       end if;
50     end process;
51   else generate
52     a_o <= a_o_comb;
53     b_o <= b_o_comb;
54   end generate out_g;
55
56 end architecture rtl;
```

Source code A.3 – Generic sorting network VHDL description

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  use work.csn_pkg.all;
6
7  entity csn is
8    generic(
9      I     : natural := 16;
10     O     : natural := 16;
11     delay : natural := 3;
```

```
                Off    : natural := 0
  );
  port(
    clk          : in  std_logic;
    sink_valid   : in  std_logic;
    source_valid : out std_logic;
    muon_i       : in  muon_a(0 to I - 1);
    muon_o       : out muon_a(0 to O - 1)
  );
end entity csn;

architecture RTL of csn is

  constant cfg_net : cfg_net_t := get_cfg(I);
  constant stages  : stages_a  := get_stg(352,delay);

  type net_array_t is array (natural range <>) of
    muon_a(0 to I - 1);

  signal net_array   : net_array_t(0 to cfg_net'
    length);
  signal valid_array : std_logic_vector(0 to cfg_net'
    length);

begin

  net_array(0) <= muon_i;
  valid_array(0) <= sink_valid;

  stage_g : for stage in 0 to cfg_net'high generate
    pair_g : for pair in 0 to I / 2 - 1 generate
      -- sorting network stage
      csn_cmp_inst : entity work.csn_cmp
        generic map(
          ascending       => False,
          pass_through    => cfg_net(stage)(pair).p,
          output_register => stages(stage+Off)
        )
        port map(
          clk => clk,
          a_i => net_array(stage)(cfg_net(stage)(pair
            ).a),
          b_i => net_array(stage)(cfg_net(stage)(pair
            ).b),
          a_o => net_array(stage + 1)(cfg_net(stage)(
            pair).a),
          b_o => net_array(stage + 1)(cfg_net(stage)(
            pair).b)
        );

      -- valid flags
      valid_g : if stages(stage+Off) generate
        process(clk)
        begin
          if rising_edge(clk) then
            valid_array(stage + 1) <= valid_array(
              stage);
          end if;
        end process;
      else generate
        valid_array(stage + 1) <= valid_array(stage);
      end generate valid_g;

    end generate pair_g;
  end generate stage_g;

  muon_o <= net_array(cfg_net'length)(muon_o'range);
  source_valid <= valid_array(cfg_net'length);

end architecture RTL;
```

Source code A.4 – $H = 3$ and $H = 2$ sorting network wrapper VHDL description

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  use work.csn_pkg.all;
6
7  entity csn_net is
8    generic(
9      I : natural := 352;
10     O : natural := 16;
11     D : natural := 3  -- delay in clock cycles for
         pipeline register
12     );
13
14   port(
15     clk          : in  std_logic;
16     sink_valid   : in  std_logic;
17     source_valid : out std_logic;
18     muon_i       : in  muon_a(0 to I - 1);
19     muon_o       : out muon_a(0 to O - 1)
20     );
21 end entity csn_net;
22
23 architecture hier of csn_net is
24
25   constant R   : natural := 16;
26   constant I_s : natural := 22;
27   constant I_m : natural := 32;
28
29   type muon_2d is array (natural range <>) of muon_a
       (0 to O - 1);
30   signal muon_R  : muon_2d(0 to R-1);
31   signal muon_M1 : muon_2d(0 to R/2-1);
32   signal muon_M2 : muon_2d(0 to R/4-1);
33   signal muon_M3 : muon_2d(0 to R/8-1);
34
35   signal source_valid_r  : std_logic_vector(0 to R-1)
       ;
36   signal source_valid_m1 : std_logic_vector(0 to R
       /2-1);
37   signal source_valid_m2 : std_logic_vector(0 to R
       /4-1);
38   signal source_valid_m3 : std_logic_vector(0 to R
       /8-1);
39
40
41 begin
42
43   -- sorting step
44   R_g : for Ri in 0 to R-1 generate
45     csn_inst : entity work.csn
46       generic map(
47         I     => I_s,
48         O     => O,
49         delay => D,
50         Off   => 0
51         )
52       port map(
53         clk          => clk,
54         sink_valid   => sink_valid,
55         source_valid => source_valid_r(Ri),
56         muon_i       => muon_i(Ri*I_S to ((Ri+1)*I_s
             -1)),
57         muon_o       => muon_R(Ri)
58         );
59
60   end generate R_g;
61
62   -- merging step 1
63   M1_g : for Mi in 0 to R/2-1 generate
64     csn_inst : entity work.csn
65       generic map(
66         I     => I_m,
```

```
        O     => O,
        delay => D,
        Off   => 12
        )
      port map(
        clk              => clk,
        sink_valid       => source_valid_r(2*Mi),
        source_valid     => source_valid_m1(Mi),
        muon_i(0 to 15)  => muon_R(2*Mi),
        muon_i(16 to 31) => muon_R(2*Mi+1),
        muon_o           => muon_M1(Mi)
        );

  end generate M1_g;

  -- merging step 2
  M2_g : for Mi in 0 to R/4-1 generate
    csn_inst : entity work.csn
      generic map(
        I     => I_m,
        O     => O,
        delay => D,
        Off   => 17
        )
      port map(
        clk              => clk,
        sink_valid       => source_valid_m1(2*Mi),
        source_valid     => source_valid_m2(Mi),
        muon_i(0 to 15)  => muon_M1(2*Mi),
        muon_i(16 to 31) => muon_M1(2*Mi+1),
        muon_o           => muon_M2(Mi)
        );

  end generate M2_g;

  -- merging step 3
  M3_g : for Mi in 0 to R/8-1 generate
    csn_inst : entity work.csn
      generic map(
        I     => I_m,
        O     => O,
        delay => D,
        Off   => 22
        )
      port map(
        clk              => clk,
        sink_valid       => source_valid_m2(2*Mi),
        source_valid     => source_valid_m3(Mi),
        muon_i(0 to 15)  => muon_M2(2*Mi),
        muon_i(16 to 31) => muon_M2(2*Mi+1),
        muon_o           => muon_M3(Mi)
        );

  end generate M3_g;

  -- merging step 4
  csn_inst : entity work.csn
    generic map(
      I     => I_m,
      O     => O,
      delay => D,
      Off   => 27
      )
    port map(
      clk              => clk,
      sink_valid       => source_valid_m3(0),
      source_valid     => source_valid,
      muon_i(0 to 15)  => muon_M3(0),
      muon_i(16 to 31) => muon_M3(1),
      muon_o           => muon_o
      );

end architecture hier;

architecture flat of csn_net is

```

```
143   signal muon_cand      : muon_sort_a(0 to I - 1);
144   signal muon_stage_b   : muon_sort_a(0 to O - 1);
145   signal source_valid_a : std_logic_vector(0 to 3);
146   signal sink_valid_int : std_logic;
147   signal sink_valid_b   : std_logic;
148   signal source_valid_b : std_logic;
149
150   type mux_int_a_t is array (natural range <>) of
        integer range 0 to I - 1;
151   signal mux_int_a : mux_int_a_t(0 to O - 1);
152
153   type muon_2d is array (natural range <>) of muon_a
        (0 to I - 1);
154   signal muon_int : muon_2d(0 to D);
155
156 begin
157
158   csn_inst : entity work.csn
159     generic map(
160       I     => I,
161       O     => O,
162       delay => D,
163       Off   => 0
164       )
165     port map(
166       clk          => clk,
167       sink_valid   => sink_valid,
168       source_valid => source_valid,
169       muon_i       => muon_i,
170       muon_o       => muon_o
171       );
172
173 end architecture flat;
```

Source code A.5 – $M = 0$ and $M = 1$ sorting network wrapper VHDL description

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  use work.csn_pkg.all;
6
7  entity csn_sort_v2 is
8    generic(
9      I      : natural := 352;
10     O      : natural := 16;
11     D      : natural := 3;  -- delay in clock cycles
         for pipeline register
12     in_reg : natural := 0;
13     mux    : natural := 1;
14     flat   : natural := 1
15     );
16   port(
17     clk          : in  std_logic;
18     sink_valid   : in  std_logic;
19     source_valid : out std_logic;
20     muon_i       : in  muon_a(0 to I - 1);
21     muon_o       : out muon_a(0 to O - 1)
22     );
23 end entity csn_sort_v2;
24
25 architecture rtl of csn_sort_v2 is
26
27   constant DN : natural := D -mux*1;
28
29   signal muon_cand     : muon_a(0 to I - 1);
30   signal muon_cand_int : muon_a(0 to I - 1);
31   signal muon_i_int    : muon_a(0 to I - 1);
32   signal muon_stage_b  : muon_a(0 to O - 1);
33
```

```
34    signal source_valid_a : std_logic_vector(0 to 3);
35    signal sink_valid_int : std_logic;
36    signal sink_valid_b   : std_logic;
37    signal source_valid_b : std_logic;
38
39    type mux_int_a_t is array (natural range <>) of
        integer range 0 to I - 1;
40    signal mux_int_a : mux_int_a_t(0 to O - 1);
41
42    type muon_2d is array (natural range <>) of muon_a
        (0 to I - 1);
43    signal muon_int : muon_2d(0 to DN);
44
45  begin
46    -- assigning constant id to input
47    id_g : for id in 0 to I - 1 generate
48      muon_cand(id).idx <= std_logic_vector(to_unsigned
          (id, IDX_WIDTH));
49      muon_cand(id).pt  <= muon_i(id).pt;
50      roi_flags_g : if mux = 0 generate
51        muon_cand(id).roi   <= muon_i(id).roi;
52        muon_cand(id).flags <= muon_i(id).flags;
53      end generate roi_flags_g;
54    end generate id_g;
55
56    -- registering input if desired
57    in_reg_g : if in_reg = 0 generate
58      muon_i_int     <= muon_i;
59      sink_valid_int <= sink_valid;
60      muon_cand_int  <= muon_cand;
61    else generate
62      process (clk) is
63      begin
64        if rising_edge(clk) then
65          muon_i_int     <= muon_i;
66          sink_valid_int <= sink_valid;
67          muon_cand_int  <= muon_cand;
68        end if;
69      end process;
70    end generate in_reg_g;
71
72    -- instantiating network
73    net_g : if flat = 1 generate
74      csn_net_1 : entity work.csn_net(flat)
75        generic map (
76          I => I,
77          O => O,
78          D => DN)
79        port map (
80          clk          => clk,
81          sink_valid   => sink_valid_int,
82          source_valid => source_valid_b,
83          muon_i       => muon_cand_int,
84          muon_o       => muon_stage_b);
85    else generate
86      csn_net_1 : entity work.csn_net(hier)
87        generic map (
88          I => I,
89          O => O,
90          D => DN)
91        port map (
92          clk          => clk,
93          sink_valid   => sink_valid_int,
94          source_valid => source_valid_b,
95          muon_i       => muon_cand_int,
96          muon_o       => muon_stage_b);
97    end generate net_g;
98
99    mux_g : if mux = 1 generate
100     -- with mux
101     -- delaying input and source_valid
102     process(all)
103     begin
104       muon_int(0) <= muon_i_int;
105       if rising_edge(clk) then
```

```
106           -- delaying muon input (keeping full
                thoughput which is actually not necessay)
107           -- should be
108           for i in 1 to DN loop
109             muon_int(i) <= muon_int(i - 1);
110           end loop;
111           source_valid <= source_valid_b;
112         end if;
113       end process;
114       -- 1 stage mux
115       o_g : for id in 0 to 0 - 1 generate
116         process(all)
117         begin
118           if not is_x(muon_stage_b(id).idx) then
119             mux_int_a(id) <= to_integer(unsigned(
                  muon_stage_b(id).idx));
120           end if;
121           if rising_edge(clk) then
122             -- avoiding mux for idx and pt as it goes
                  through the network
123             muon_o(id).idx    <= muon_stage_b(id).idx;
124             muon_o(id).pt     <= muon_stage_b(id).pt;
125             -- using mux for roi and flags as it does
                  not goes through the network
126             muon_o(id).roi    <= muon_int(DN)(mux_int_a(
                  id)).roi;
127             muon_o(id).flags <= muon_int(DN)(mux_int_a(
                  id)).flags;
128           end if;
129         end process;
130       end generate o_g;
131     else generate
132       -- no mux
133       muon_o       <= muon_stage_b;
134       source_valid <= source_valid_b;
135     end generate mux_g;
136
137   end architecture rtl;
```

Bibliography

[1] J. W. Lockwood et al. «A Low-Latency Library in FPGA Hardware for High-Frequency Trading (HFT)». In: *2012 IEEE 20th Annual Symposium on High-Performance Interconnects*. 2012, pp. 9–16. DOI: 10.1109/HOTI.2012.15.

[2] B. Ramesh, A. D. George, and H. Lam. «Real-time, low-latency image processing with high throughput on a multi-core SoC». In: *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. Sept. 2016, pp. 1–7. DOI: 10.1109/HPEC.2016.7761645.

[3] The ATLAS Collaboration et al. «The ATLAS Experiment at the CERN Large Hadron Collider». en. In: *Journal of Instrumentation* 3.08 (2008), S08003. ISSN: 1748-0221. DOI: 10.1088/1748-0221/3/08/S08003. URL: http://stacks.iop.org/1748-0221/3/i=08/a=S08003 (visited on 04/26/2016).

[4] Georges Aad et al. *Technical Design Report for the Phase-I Upgrade of the ATLAS TDAQ System*. Tech. rep. Sept. 2013. URL: https://cds.cern.ch/record/1602235 (visited on 04/26/2016).

[5] Jörg Stelzer and the ATLAS collaboration. «The ATLAS High Level Trigger Configuration and Steering: Experience with the First 7 TeV Collision Data». en. In: *Journal of Physics: Conference Series* 331.2 (Dec. 2011), p. 022026. ISSN: 1742-6596. DOI: 10.1088/1742-6596/331/2/022026. URL: http://stacks.iop.org/1742-6596/331/i=2/a=022026?key=crossref.4ded91f1ddda6418ea9bdbc558c2106d (visited on 03/28/2019).

[6] P. B. Amaral et al. «The ATLAS Level-1 central trigger processor». In: *14th IEEE-NPSS Real Time Conference, 2005*. June 2005, 4 pp.–. DOI: 10.1109/RTC.2005.1547406.

[7] S. Ask et al. «The ATLAS central level-1 trigger logic and TTC system». en. In: *Journal of Instrumentation* 3.08 (2008), P08002. ISSN: 1748-0221. DOI: 10.1088/1748-0221/3/08/P08002. URL: http://stacks.iop.org/1748-0221/3/i=08/a=P08002 (visited on 04/26/2016).

[8] H.C. van der Bij et al. «S-LINK, a Data Link Interface Specification for the LHC Era». In: *1996 IEEE Nuclear Science Symposium. Conference Record*. Vol. 1. Nov. 1996, 465–469 vol.1. DOI: 10.1109/NSSMIC.1996.591032.

[9] R. Cranfield et al. «The ATLAS ROBIN». en. In: *Journal of Instrumentation* 3.01 (Jan. 2008), T01002–T01002. ISSN: 1748-0221. DOI: 10.1088/1748-0221/3/01/T01002. URL: https://doi.org/10.1088%2F1748-0221%2F3%2F01%2Ft01002 (visited on 04/17/2019).

[10] R. Caputo et al. «Upgrade of the ATLAS Level-1 trigger with an FPGA based Topological Processor». In: *2013 IEEE Nuclear Science Symposium and Medical Imaging Conference (2013 NSS/MIC)*. Oct. 2013, pp. 1–5. DOI: 10.1109/NSSMIC.2013.6829555.

[11] Stefan Haas. *Overlap Handling of the MUCTPI Octant Module*. Mar. 2011. URL: https://edms.cern.ch/ui/file/1134525/2.2/MIOCT_Overlap_Handling_Rev2_2.pdf (visited on 02/07/2017).

[12] Marcos Vinícius Silva Oliveira et al. «The ATLAS Level-1 Muon Topological Trigger Information for Run 2 of the LHC». en. In: *Journal of Instrumentation* 10.02 (2015), p. C02027. ISSN: 1748-0221. DOI: 10.1088/1748-0221/10/02/C02027. URL: http://stacks.iop.org/1748-0221/10/i=02/a=C02027 (visited on 01/09/2017).

[13] CERN. *Project Schedule | HL-LHC Industry*. 2019. URL: https://project-hl-lhc-industry.web.cern.ch/content/project-schedule (visited on 08/01/2020).

[14] L. Rossi and O. Brüning. *Introduction to the HL-LHC Project*. en. 2015. DOI: 10.1142/9789814675475_0001. URL: https://cds.cern.ch/record/2130736 (visited on 04/17/2019).

[15] PICMG. *PICMG® 3.0 Revision 3.0 AdvancedTCA® Base Specification*. 2008. URL: https://cds.cern.ch/record/1159877?ln=en (visited on 11/16/2017).

[16] «IEEE Standard for a Versatile Backplane Bus: VMEbus». In: *ANSI/IEEE Std 1014-1987* (1987), 0_1–. DOI: 10.1109/IEEESTD.1987.101857.

[17] Marcos Vinícius Silva Oliveira. «The ATLAS Level-1 Muon Topological Trigger Information for Run 2 of the LHC». PhD thesis. Juiz de Fora: Federal University of Juiz de Fora, Feb. 2015. URL: https://drive.google.com/file/d/0B7wt7DnUWp7hM3haa3E3dDhlZFk/view?usp=sharing%5C&usp=embed_facebook (visited on 01/10/2017).

[18] Marcos Vinícius Silva Oliveira et al. «The ATLAS Muon to Central Trigger Processor Interface Upgrade for the Run 3 of the LHC». In: *2017 IEEE Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC)*. Oct. 2017, pp. 1–5. DOI: 10.1109/NSSMIC.2017.8532707.

[19] Xilinx. *UltraScale Architecture and Product Overview*. 2016. URL: https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf (visited on 02/03/2017).

[20] Avago. *MiniPOD™AFBR-814VXYZ, AFBR-824VXYZ 14 Gb/s Data Sheet*. 2013.

[21] IEEE. «IEEE 802.3-2015 Standard for Ethernet». In: *IEEE Std 802.3-2015 (Revision of IEEE Std 802.3-2012)* (Mar. 2016), pp. 1–4017. DOI: 10.1109/IEEESTD.2016.7428776.

[22] Jonathan Valdez and Jared Becker. «Understanding the I2C Bus». en. In: (2015), p. 8.

[23] Tektronix. *Bridging the Gap Between BER and Eye Diagrams — A BER Contour Tutorial*. 2010. URL: http://download.tek.com/document/65W_26019_0_Letter.pdf.

[24] IEEE. *IEEE - The World's Largest Technical Professional Organization Dedicated to Advancing Technology for the Benefit of Humanity*. URL: https://www.ieee.org/ (visited on 06/28/2020).

[25] Maxim. «Statistical confidence levels for estimating error probability». In: *Maxim Engineering Journal* 37 (2000), pp. 12–15. URL: https://pdfserv.maximintegrated.com/en/ej/EJ37.pdf (visited on 06/05/2019).

[26] NetTest. *Qualifying SDH/SONET Transmission Path*. 2004. URL: https://docplayer.net/23881527-Qualifying-sdh-sonet-transmission-path.html (visited on 06/05/2019).

[27] Dennis Derickson and Marcus Müller. *Digital Communications Test and Measurement: High-Speed Physical Layer Characterization*. en. Pearson Education, Dec. 2007. ISBN: 978-0-13-279721-4.

[28] NetTest. *Qualifying SDH/SONET Transmission Path*. 2004. URL: https://docplayer.net/23881527-Qualifying-sdh-sonet-transmission-path.html (visited on 06/05/2019).

[29] Lecroy. *WaveRunner 6 Zi Oscilloscopes 400 MHz –4 GHz*. July 2019. URL: https://cdn.teledynelecroy.com/files/pdf/waverunner-6zi-datasheet.pdf (visited on 06/27/2019).

[30] Xilinx. *UltraScale Architecture GTY Transceivers User Guide*. Dec. 2016. URL: https://www.xilinx.com/support/documentation/user_guides/ug578-ultrascale-gty-transceivers.pdf.

[31] Internationale Elektrotechnische Kommission. *Fibre Optic Communication Subsystem Test Procedures - Part 2-2: Digital Systems - Optical Eye Pattern, Waveform and Extinction Ratio Measurement*. en. 2012. ISBN: 978-2-8322-0420-7.

[32] Xilinx. *UltraScale Architecture GTH Transceivers User Guide*. Oct. 2016. URL: http://www.xilinx.com/support/documentation/user_guides/ug576-ultrascale-gth-transceivers.pdf (visited on 03/23/2016).

[33] Xilinx. *UltraScale Architecture GTY Transceivers User Guide*. Dec. 2016. URL: https://www.xilinx.com/support/documentation/user_guides/ug578-ultrascale-gty-transceivers.pdf.

[34] Xilinx. *VCU118 Evaluation Board User Guide*. en. 2018.

[35] Marcos Vinícius Silva Oliveira. *MPOD FMC Schematics and PCB Layout*. 2015. URL: https://edms.cern.ch/ui/%5C#!master/navigator/item?P:1162692124:1812207066:subDocs (visited on 08/04/2020).

[36] FS. *MTP/MPO Trunk Cables Datasheet*. 2020. URL: https://img-en.fs.com/file/datasheet/mtp-mpo-trunk-cables-datasheet.pdf (visited on 08/21/2020).

[37] Texas Instruments. *CDCE62005 Clock Generator, Jitter Cleaner with Integrated Dual VCOs Data Sheet*. 2016. URL: https://www.ti.com/lit/ds/symlink/cdce62005.pdf?ts=1596554755034%5C&ref_url=https%5C%253A%5C%252F%5C%252Fwww.google.com%5C%252F (visited on 08/04/2020).

[38] Silicon Labs. *Si5338 I2C Programmable Any-Frequency Any-Output Quad Clock Generator*. 2015. URL: https://www.silabs.com/documents/public/data-sheets/Si5338.pdf (visited on 08/04/2020).

[39] LEMO. *LEMO Unipole and Multipole Connectors*. 2020. URL: https://www.lemo.com/catalog/ROW/UK_English/unipole_multipole.pdf (visited on 08/04/2020).

[40] Finisar. *OC-3 IR-1/STM S-1.1 RoHS Compliant Pluggable SFP Transceiver*. 2015. URL: https://www.finisar.com/sites/default/files/downloads/finisar_ftlf1323p1btr_oc-3_ir-1_stm_s-1.1_rohs_compliant_pluggable_sfp_transceiver_product_specification_rev_b1.pdf (visited on 08/04/2020).

[41] Broadcom. *AFBR-79EQDZ40 Gigabit Ethernet & InfiniBand QSFP+ Pluggable, Parallel Fiber-Optics Module*. 2014. URL: https://docs.broadcom.com/doc/AV02-2924EN_DS_AFBR-79EQDZ_2014-09-03 (visited on 08/04/2020).

[42] Molex. *RF/Microwave Products*. 2016. URL: http://www.literature.molex.com/SQLImages/kelmscott/Molex/PDF_Images/987651-2321.PDF (visited on 08/04/2020).

[43] Motorola. *SPI Block Guide*. 2003. URL: https://web.archive.org/web/20150413003534/http://www.ee.nmt.edu/~teare/ee308l/datasheets/S12SPIV3.pdf (visited on 08/04/2020).

[44] Philips Semiconductors. *I2C Manual*. Mar. 2003. URL: https://www.nxp.com/docs/en/application-note/AN10216.pdf (visited on 08/04/2020).

[45] Xilinx. *IBERT for UltraScale GTH Transceivers*. Aug. 2016.

[46] Xilinx. *IBERT for UltraScale GTY Transceivers*. Aug. 2016.

[47] Marcos Vinícius Silva Oliveira. *PCBpy: A Cadence Allegro PCB schematics parser and verification tool*. July 2018. URL: https://github.com/mvsoliveira/PCBpy (visited on 06/05/2019).

[48] Marcos Vinícius Silva Oliveira. *IBERTpy: A Python package for running IBERT Eye scan in Vivado, ploting eye diagrams with mathplotlib and compiling results with LaTeX*. Oct. 2018. URL: https://github.com/mvsoliveira/IBERTpy (visited on 06/05/2019).

[49] Stephen Goadhouse and Nikitas Loukas. *CMS Barrel Calorimeter Read-out and Trigger Primitive Generation*. en. 2020. URL: https://indico.cern.ch/event/863071/contributions/3738875/ (visited on 06/14/2020).

[50] Keysight. *Keysight DSAV334A 33 GHz Infiniium V-Series Oscilloscope*. fr-FR. Aug. 2019. URL: https://www.keysight.com/en/pdx-x202209-pn-DSAV334A/infiniium-v-series-oscilloscope-33-ghz-4-analog-channels?cc=FR&lc=fre (visited on 06/27/2019).

[51] Keysight. *Keysight N7004A 33 GHz Optical-to-Electrical Converter*. fr-FR. Aug. 2019. URL: https://www.keysight.com/en/pd-2746451-pn-N7004A/33-ghz-optical-to-electrical-converter?cc=FR&lc=fre (visited on 06/27/2019).

[52] FS. *FHD MTP Cassettes Datasheet*. 2020. URL: https://img-en.fs.com/file/datasheet/fhd-mtp-mpo-lc-sc-cassette-datasheet.pdf (visited on 07/22/2020).

[53] JDSU. *JDSU MAP Precision Attenuator Product Brief*. June 2019. URL: http://img.wl95.com/upload/file/2018-05-05/050953580b4c.pdf (visited on 06/26/2019).

[54] VIAVI. *OLP-85 and -85P SmartClass Fiber inspection-ready optical power meters*. eng. May 2015. URL: https://www.viavisolutions.com/en-us/products/smartclass-fiber-olp-85-85p-inspection-ready-optical-power-meters (visited on 06/26/2019).

[55] TIA. *TIA/EIA-568-B.3*. 2000. URL: https://www.csd.uoc.gr/~hy435/material/TIA-EIA-568-B.3.pdf (visited on 07/12/2020).

[56] Xilinx. *KC705 Evaluation Board for the Kintex-7 FPGA*. Apr. 2016. URL: http://www.xilinx.com/support/documentation/boards_and_kits/kc705/ug810_KC705_Eval_Bd.pdf.

[57] Xilinx. *VCU108 Evaluation Board User Guide*. Apr. 2016. URL: http://www.xilinx.com/support/documentation/boards_and_kits/vcu108/ug1066-vcu108-eval-bd.pdf.

[58] Silicon Labs. *Si5330/34/35/38 Evaluation Board User Guide*. 2011. URL: https://www.silabs.com/documents/public/user-guides/Si5338-EVB.pdf (visited on 08/06/2019).

[59] R. Giordano and A. Aloisio. «Fixed-Latency, Multi-Gigabit Serial Links With Xilinx FPGAs». In: *IEEE Transactions on Nuclear Science* 58.1 (Feb. 2011), pp. 194–201. ISSN: 0018-9499. DOI: 10.1109/TNS.2010.2101083.

[60] X. Liu, Q. X. Deng, and Z. K. Wang. «Design and FPGA Implementation of High-Speed, Fixed-Latency Serial Transceivers». In: *IEEE Transactions on Nuclear Science* 61.1 (2014), pp. 561–567. ISSN: 0018-9499. DOI: 10.1109/TNS.2013.2296301.

[61] Clifford E. Cummings. *Clock Domain Crossing (CDC) Design & Verification Techniques Using SystemVerilog*. 2008.

[62] Xilinx. «Vivado Design Suite User Guide: Implementation». en. In: (2019), p. 192.

[63] Mentor. *ModelSim: Sophisticated FPGA Verification*. en. 2019. URL: https://www.mentor.com/products/fv/modelsim/ (visited on 08/12/2019).

[64] Python Core Team. *Python: A dynamic, open source programming language. Python Software Foundation*. en. 2019. URL: https://www.python.org/ (visited on 08/12/2019).

[65] Cocotb Core Team. *Coroutine Co-simulation Test Bench*. original-date: 2013-06-12T20:07:15Z. Aug. 2019. URL: https://github.com/cocotb/cocotb (visited on 08/12/2019).

[66] Wes McKinney. «Data Structures for Statistical Computing in Python». In: 2010, pp. 51–56. URL: http://conference.scipy.org/proceedings/scipy2010/mckinney.html (visited on 08/12/2019).

[67] S. van der Walt, S. C. Colbert, and G. Varoquaux. «The NumPy Array: A Structure for Efficient Numerical Computation». In: *Computing in Science Engineering* 13.2 (Mar. 2011), pp. 22–30. ISSN: 1521-9615. DOI: 10.1109/MCSE.2011.37.

[68] J. D. Hunter. «Matplotlib: A 2D Graphics Environment». In: *Computing in Science Engineering* 9.3 (May 2007), pp. 90–95. ISSN: 1521-9615. DOI: 10.1109/MCSE.2007.55.

[69] Joseph Bulone and Roger Sabbagh. «A Pragmatic Approach to Metastability-Aware Simulation». en. In: (2014), p. 8.

[70] Xilinx. *Aurora 64B/66B v12.0 LogiCORE IP Product Guide*. en. 2019.

[71] D. Berge et al. *The ATLAS Level-1 Muon to Central Trigger Processor Interface*. 2007. DOI: 10.5170/CERN-2007-007.453,10.5170/CERN-2007-007.453.

[72] Xilinx. *UltraScale FPGA Product Selection Guide*. 2019. URL: https://www.xilinx.com/support/documentation/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf (visited on 09/09/2019).

[73] Donald Ervin Knuth. *The art of computer programming*. en. Addison-Wesley series in computer science and information processing. Reading, Mass: Addison-Wesley Pub. Co, 1973. ISBN: 978-0-201-03809-5.

[74] K. E. Batcher. «Sorting networks and their applications». en. In: *Proceedings of the April 30–May 2, 1968, spring joint computer conference on - AFIPS '68 (Spring)*. Atlantic City, New Jersey: ACM Press, 1968, p. 307. DOI: 10.1145/1468075.1468121. URL: http://portal.acm.org/citation.cfm?doid=1468075.1468121 (visited on 03/11/2019).

[75] Sherenaz W. Al-Haj Baddar and Kenneth W. Batcher. *Designing sorting networks: a new paradigm*. en. OCLC: ocn778707417. New York, NY: Springer, 2011. ISBN: 978-1-4614-1850-4 978-1-4614-1851-1.

[76] M. Ajtai, J. Komlós, and E. Szemerédi. «An 0(n log n) sorting network». In: ACM, Dec. 1983, pp. 1–9. ISBN: 978-0-89791-099-6. DOI: 10.1145/800061.808726. URL: http://dl.acm.org/citation.cfm?id=800061.808726 (visited on 09/21/2019).

[77] Sherenaz Al-Haj Baddar. *Finding Faster Sorting Networks: Using Sortnet*. English. Saarbrücken: VDM Verlag, Oct. 2009. ISBN: 978-3-639-17800-5.

[78] Marcos Vinícius Silva Oliveira. *SNpy: A Python Package for Generating, Plotting, Optmizing, and Generating HDL Description of Sorting Networks*. Nov. 2019. URL: https://github.com/mvsoliveira/SNpy (visited on 12/05/2019).

[79] Daniel Bundala et al. «Optimal-Depth Sorting Networks». In: *Journal of Computer and System Sciences* 84 (2014), pp. 185–204. ISSN: 00220000. DOI: 10.1016/j.jcss.2016.09.004. arXiv: 1412.5302.

[80] V. E. Alekseev. «Sorting Algorithms with Minimum Memory». en. In: *Cybernetics* 5.5 (Sept. 1969), pp. 642–648. ISSN: 0011-4235, 1573-8337. DOI: 10.1007/BF01267888.

[81] Sanjay Churiwala and Sapan Garga. *Principles of VLSI RTL Design: A Practical Guide*. en. New York: Springer, 2011. ISBN: 978-1-4419-9295-6 978-1-4419-9296-3.

[82] Frank Vahid. *Digital Design with RTL Design, VHDL, and Verilog*. en. John Wiley & Sons, Mar. 2010. ISBN: 978-0-470-53108-2.

[83] A. Takach. «High-Level Synthesis: Status, Trends, and Future Directions». In: *IEEE Design Test* 33.3 (June 2016), pp. 116–124. ISSN: 2168-2356. DOI: 10.1109/MDAT.2016.2544850.

[84] Xilinx. *Vivado Design Suite User Guide: High-Level Synthesis*. en. 2020. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug871-vivado-high-level-synthesis-tutorial.pdf (visited on 06/14/2020).

[85] IEEE. «IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002) IEEE Standard VHDL Language Reference Manual». en. In: (2009), p. 640.

[86] Babette van Antwerpen et al. «Register Retiming Technique». US7120883B1. Oct. 2006. URL: https://patents.google.com/patent/US7120883/en (visited on 05/03/2020).

[87] Charles E. Leiserson and James B. Saxe. «Retiming Synchronous Circuitry». en. In: *Algorithmica* 6.1 (June 1991), pp. 5–35. ISSN: 1432-0541. DOI: 10.1007/BF01759032.

[88] Synopsis. *Synplify Pro and Premier Datasheet*. 2018. URL: https://www.synopsys.com/cgi-bin/verification/dsdla/docsdl/synplify-pro-premier-ds.pdf?file=synplify-pro-premier-ds.pdf (visited on 05/03/2020).

[89] Xilinx. *Vivado Design Suite User Guide - Synthesis*. 2017. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug901-vivado-synthesis.pdf (visited on 04/23/2018).

[90] G. De Micheli. «Synchronous Logic Synthesis: Algorithms for Cycle-Time Minimization». In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 10.1 (Jan. 1991), pp. 63–73. ISSN: 1937-4151. DOI: 10.1109/43.62792.

[91] Xilinx. *Vivado Design Suite User Guide: Design Analysis and Closure Techniques*. en. 2020. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug906-vivado-design-analysis.pdf (visited on 06/14/2020).

[92] Xilinx. *Vivado Design Suite User Guide: Hierarchical Design*. en. 2019. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug905-vivado-hierarchical-design.pdf (visited on 06/14/2020).

[93] Xilinx. *UltraFast Design Methodology Guide for the Vivado Design Suite*. en. 2019. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug949-vivado-design-methodology.pdf (visited on 06/14/2020).

[94] Cocotb Core Team. *Coroutine Co-Simulation Test Bench*. cocotb. Aug. 2019. URL: https://github.com/cocotb/cocotb (visited on 08/12/2019).

[95] Xilinx. «UltraScale Architecture Configurable Logic Block User Guide». en. In: (2017), p. 58. URL: https://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf (visited on 06/14/2020).

[96] ISO. «ISO 8601-1:2019». en. In: (2019). URL: https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/07/09/70907.html (visited on 05/01/2020).

[97] Xianyang Jiang et al. «Performance Effects of Pipeline Architecture on an FPGA-Based Binary32 Floating Point Multiplier». en. In: *Microprocessors and Microsystems* 37.8, Part D (Nov. 2013), pp. 1183–1191. ISSN: 0141-9331. DOI: 10.1016/j.micpro.2013.08.007.

[98] M. Leverington and K. N. Shemdin. *Principles of Timing in FPGAs*. English. 1 edition. CreateSpace Independent Publishing Platform, Jan. 2017. ISBN: 978-1-5428-1585-7.

[99] Xilinx. *XST User Guide*. 2008. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx10/books/docs/xst/xst.pdf (visited on 05/02/2020).

[100] Mentor Graphics. *Catapult High-Level Synthesis*. en. URL: https://www.mentor.com/hls-lp/catapult-high-level-synthesis/ (visited on 06/13/2020).

[101] Intel. *Intel High-Level Synthesis Compiler*. en. URL: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html (visited on 06/13/2020).

[102] Javier Duarte et al. «Fast Inference of Deep Neural Networks in FPGAs for Particle Physics». In: *Journal of Instrumentation* 13.07 (July 2018), P07027–P07027. ISSN: 1748-0221. DOI: 10.1088/1748-0221/13/07/P07027. arXiv: 1804.06913.

[103] Wojciech M. Zabołotny. «Implementation of OMTF Trigger Algorithm with High-Level Synthesis». In: *Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2019*. Vol. 11176. International Society for Optics and Photonics, Nov. 2019, p. 1117641. DOI: 10.1117/12.2536258.

[104] S. Summers, A. Rose, and P. Sanders. «Using MaxCompiler for the High Level Synthesis of Trigger Algorithms». en. In: *Journal of Instrumentation* 12.02 (Feb. 2017), pp. C02015–C02015. ISSN: 1748-0221. DOI: 10.1088/1748-0221/12/02/C02015.



Marcos Oliveira

Senior FPGA Engineer

English (fluent) – French (B1-B2) – Portuguese (native)

marcos.oliveira@epfl.ch
+41 78 820 75 70
Geneva, Switzerland
https://cern.ch/marcos.oliveira
https://linkedin.com/in/marcosvsoliveira

FPGA Engineer with ten years of experience in FPGA designs using different Xilinx and Altera devices. Experience in low-latency synchronous systems, high-speed serial links, automated design flow & testing, and SoC design.

AREAS OF EXPERTISE and SKILLS

- Experienced with specification, description (**VHDL/Verilog & HLS**), simulation (**Modelsim & Vivado Simulator**), implementation (**Xilinx** & **Altera**) and testing (**In-System Verification & Evaluation Kit Prototyping**) of digital electronic circuits
- Implementation of **FPGA firmware** and **low-level software** for interfacing with hardware devices (In-Hardware Verification, control, data readout, etc.). Includes firmware and software development to support the following devices/interfaces: Multi-gigabit transceivers, DDR3, SRAM & Flash memories, Gigabit Ethernet, I2C, PMBus and Slave Serial FPGA configuration.
- **Embedded Software & Hardware Acceleration for SoC** FPGAs (Altera Cyclone V SoC FPGA, Xilinx Zynq-7000 SoC FPGA) Includes experience in hardware acceleration for computation of 2-D correlation coefficient using SoC FPGAs.
- **PCB design**, automated netlist verification and integration with FPGA design flow
- Computer **Programming/Script Languages** (Python, C/C++, MATLAB, Shell Script, TCL, etc.)
- Analysis of **Finite Word-Length Effects** in hardware implementation of linear and non-linear functions

RELEVANT EXPERIENCE

https://cern.ch/marcos.oliveira

- 2020-Present: Senior Fellow Electronic Engineer - ATLAS Liquid Argon Calorimeter at CERN (Switzerland)
  - Static timing analysis and latency optimization
  - Latency-uncertainty characterization and mitigation
  - Review of printed circuit board schematics and layout
- 2011-2019: Electronic Engineer - ATLAS Level-1 Central Trigger Processor at CERN (Switzerland)
  - FPGA design of low-latency high-throughput digital circuits for the ATLAS Level-1 Trigger System
  - Implementation of sorting networks using RTL and HLS synthesis approaches
  - Verification using functional & timing simulation, in-system verification, evaluation kit prototyping and measurement in laboratory
  - Design of FPGA-based high-speed serial links (synchronizing 100+ links per FPGA)
  - Optical link characterization (automating characterization of 300+ transceivers per board)
  - PCB design & integration with the FPGA design flow
  - Software design for automated PCB schematic verification & board testing
  - Low-level software for board controlling and monitoring
    - More information on https://cern.ch/marcos.oliveira
- 2014-2014: Research project at the Postgraduate Program in Electrical Engineering Federal University of Juiz de Fora
  Partial-time project working remotely
  - Design and hardware implementation of Artificial Neural Networks for event classification
  - Performance and Finite Word-Length Effects Analysis to specify the Neural Network topology and quantization parameters
- 2008-2011: Research and development project at Power Line Communication Modem Project – UFJF (Brazil)
  Specification and implementation of DSP algorithms for the physical layer of the first Latin American PLC modem
  - FPGA design of IP blocks for the physical layer of the first Latin American PLC Modem
  - Functional & timing simulation of IP blocks
  - Evaluation kit prototyping
  - OFDM system simulation in MATLAB

EDUCATION

- 2016-Present: **PhD in Electrical Engineering**
  École Polytechnique Fédérale de Lausanne (Switzerland) / **Microelectronic Systems** Laboratory
  PhD thesis: Low-Latency High-Bandwidth Circuit and System Design for Trigger Systems in High Energy Physics Experiments
  Investigation on low-latency data synchronization of 100+ high-speed serial links in a single FPGA, design of low-latency digital circuits, high-level synthesis, and hardware acceleration in SoC FPGAs (https://cds.cern.ch/record/2296209)
  Average score of exams: 5.88 / 6
- 2012-2015: **Master in Electrical Engineering**
  Federal University of Juiz de Fora (Brazil) / **Digital Signal Processing** Laboratory
  Master thesis: *The ATLAS Level-1 Muon Topological Trigger Information for Run 2 of the LHC* (http://cds.cern.ch/record/2634056)
  Feasibility studies and implementation of the firmware upgrade of the ATLAS Muon-to-Central-Trigger-Processor Interface (MUCTPI). Introducing the processing of trigger objects position in the event selection system.
  Average score of exams: 98.75 / 100

- 2006-2011: **Bachelor in Electrical Engineering**
  Federal University of Juiz de Fora (Brazil) / **Digital Signal Processing** Laboratory
  Bachelor's thesis: Timing Synchronization Technique for OFDM Systems (http://cern.ch/marcos.oliveira/documents/TFC.pdf)
  Performance analysis and implementation in FPGA of timing synchronization techniques for OFDM systems
  Average score of exams: 78.37 / 100

LANGUAGES

- **English** (fluent)
- **French** (B1-B2)
- Portuguese (mother tongue)

AWARDS and DISTINCTIONS

- **ATLAS author** thanks to relevant contributions to the ATLAS detector
- **Placed first** in the selection to be part of the first Latin American PLC modem project
- Selected as the **assistant teacher** for the Digital Electronics and Logic Circuits course at UFJF

RELATED COURSES and SEMINARS

- 2018**:** Vivado Design Suite Tutorial: **High-Level Synthesis**
- 2017: **Embedded systems** & **Real-time embedded systems**
- 2016: **PLLs and clock & data recovery**
- 2014: **Signal Processing** Special Topics: Statistical and Adaptive Processing
- 2013: **Digital Filters**: Design and implementation of IR and FIR digital filters
- 2013: **Probabilistic and Stochastic Processes**
- 2012: **Computational Intelligence** Special Topics
- 2012: General and Professional **French** at Université Ouverte de Genève
- 2009: **Quartus II Complete Design Flow** at DHW Engenharia e Representação, Brazil

PUBLICATIONS and PAPERS

https://orcid.org/0000-0003-2285-478X

- Since 2018, collaborative author of ~200 journal papers on ATLAS Physics results. List available at https://inspirehep.net/authors/1267586.
- SILVA OLIVEIRA M. V., HAAS S et al. "The Muon to Central Trigger Processor Interface for the Upgrade of the ATLAS Muon Trigger for Run-3", presented in Topical Workshop on Electronics for Particle Physics, October 2018 (**Primary Author**)
- SILVA OLIVEIRA M. V., HAAS S et al. *"The ATLAS Muon to Central Trigger Processor Interface Upgrade for the Run 3 of the LHC"*, published in the conference record of the IEEE Nuclear Sciense Symposium and Medical Imaging Conference, October 2017 (**Primary Author**)
- SILVA OLIVEIRA M. V., HAAS S et al. *"The ATLAS Level-1 Muon Topological Trigger Information for Run 2 of the LHC"*, published in IOP Journal of Instrumentation, February 2015, JINST 10 C02027 (**Primary Author**)
- SCHMIEDEN K., SILVA OLIVEIRA M. V. et al. *"Upgrade of the ATLAS Central Trigger for LHC Run 2"*, published in IOP Journal of Instrumentation, February 2015, JINST 10 C02030
- GHIBAUDI M., SILVA OLIVEIRA M. V. et al. *"Hardware and firmware developments for the upgrade of the ATLAS Level-1 Central Trigger Processor"*, published in IOP Journal of Instrumentation, January 2014, JINST 9 C0135
- SILVA OLIVEIRA M. V., PERALVA B. S., ANDRADE FILHO L. M., SANTIAGO A. C. *"Project and Implementation of a detection system with adjustable false alarm probability"*, published in Congresso Brasileiro de Automática, 2012, Campina Grande (**Primary Author**)
- LEMOS G. F. C., SILVA OLIVEIRA M. V., CAMPOS F. P. V., ANDRADE FILHO L. M., RIBEIRO M. V. *"A Low-Cost Implementation of High Order Square M-QAM Detection/Demodulation in a FPGA Device",* published in ITS 2010, Manaus.

OTHER INFORMATION

- 2018: Creator of **SNpy**, **PCBPy** and **IBERTPy** projects available in PyPI and GitHub
  - SNpy: Sorting Networks Python HDL Utilities. More information at https://github.com/mvsoliveira/SNpy
  - PCBpy: Integrates the PCB to FPGA design flow. More information at https://github.com/mvsoliveira/PCBpy
  - IBERTpy: Automatic of high-speed serial links. More information https://github.com/mvsoliveira/IBERTpy
- 2016-2019: President of the Portuguese-speaking Geneva Methodist Church – Volunteering activity
- 2011: Responsible for the Hardware Description Language course for the PLC modem project at the University and teacher of the modules **Verilog Basics & Advanced**
- 2002-2019: Pianist at Methodist Church (Switzerland and Brazil) - Volunteering activity