ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΕΦΑΡΜΟΣΜΕΝΩΝ ΜΑΘΗΜΑΤΙΚΩΝ & ΦΥΣΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΣΧΟΛΗ ΜΗΧΑΝΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ

ΕΚΕΦΕ "ΔΗΜΟΚΡΙΤΟΣ"

ΙΝΣΤΙΤΟΥΤΟ ΝΑΝΟΕΠΙΣΤΗΜΗΣ & ΝΑΝΟΤΕΧΝΟΛΟΓΙΑΣ

ΙΝΣΤΙΤΟΥΤΟ ΠΥΡΗΝΙΚΗΣ & ΣΩΜΑΤΙΔΙΑΚΗΣ ΦΥΣΙΚΗΣ

Διατμηματικό Πρόγραμμα Μεταπτυχιακών Σπουδών Φυσικής και Τεχνολογικών Εφαρμογών

ΤΟΜΕΑΣ ΦΥΣΙΚΗΣ

ΕΡΓΑΣΤΗΡΙΟ ΠΕΙΡΑΜΑΤΙΚΗΣ ΦΥΣΙΚΗΣ ΥΨΗΛΩΝ ΕΝΕΡΓΕΙΩΝ

Οδηγός Εκμάθησης της Γλώσσας VHDL και Εφαρμογή της στην Ανάπτυξη Firmware για τις Πλακέτες MMFE8 της Αναβάθμισης του Ανιχνευτή ATLAS στο LHC του CERN

ΜΕΤΑΠΤΥΧΙΑΚΗ ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Χρήστου Μπακάλη

Επιβλέπων:

Θεόδωρος Αλεξόπουλος

Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούνιος 2017

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΕΦΑΡΜΟΣΜΕΝΩΝ ΜΑΘΗΜΑΤΙΚΩΝ ΚΑΙ ΦΥΣΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΟΜΕΑΣ ΦΥΣΙΚΗΣ
ΕΡΓΑΣΤΗΡΙΟ ΠΕΙΡΑΜΑΤΙΚΗΣ ΦΥΣΙΚΗΣ ΥΨΗΛΩΝ ΕΝΕΡΓΕΙΩΝ

Οδηγός Εκμάθησης της Γλώσσας VHDL και Εφαρμογή της στην Ανάπτυξη Firmware για τις Πλακέτες MMFE8 της Αναβάθμισης του Ανιχνευτή ATLAS στο LHC του CERN

ΜΕΤΑΠΤΥΧΙΑΚΗ ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Χρήστου Κ. Μπακάλη

Επιβλέπων: Θεόδωρος Αλεξόπουλος
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή στις 30 Ιουνίου 2017.

....	....	....
Θ. Αλεξόπουλος Καθηγητής Ε.Μ.Π.	Ε. Γαζής Καθηγητής Ε.Μ.Π.	Σ. Μαλτέζος Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούνιος 2017

....
Χρήστος Κ. Μπακάλης
Φυσικός Εφαρμογών Ε.Μ.Π.

© (2017) Εθνικό Μετσόβιο Πολυτεχνείο. *All rights reserved.*

Περίληψη

Η παρούσα μεταπτυχιακή εργασία, πραγματεύεται κατά κύριο λόγο τον προγραμματισμό των Field-Programmable Gate Arrays (FPGA), που χρησιμοποιούνται στο σύστημα των front-end electronics της αναβάθμισης του New Small Wheel (NSW) που θα πραγματοποιηθεί το 2020 στον ανιχνευτή ATLAS, στον LHC του CERN. Τα FPGA, είναι μία μεγάλη οικογένεια ολοκληρωμένων κυκλωμάτων, που μπορούν να υλοποιήσουν μία οποιαδήποτε λογική συνάρτηση, ανάλογα τις επιθυμίες του χρήστη του. Μία από τις κύριες γλώσσες προγραμματισμού των μονάδων αυτών, είναι η VHDL (VHSIC Hardware Description Language). Τα FPGA, χρησιμοποιούνται στα πρωτότυπα των πλακετών MMFE8 (MicroMegas Front-End 8), οι οποίες είναι υπεύθυνες για την περισυλλογή και επεξεργασία των πρωταρχικών ηλεκτρικών σημάτων, όπως αυτά προκύπτουν μετά τον εντοπισμό των σωματιδίων από τους ανιχνευτές MicroMegas. Στόχος της εργασίας αυτής, είναι να καθοδηγήσει έναν αναγνώστη με μηδενική εμπειρία στην ανάπτυξη firmware, και να του αποσαφηνίσει έννοιες οι οποίες μπορεί σε πρώτη φάση να φαίνονται περίπλοκες, ώστε να έχει τη δυνατότητα να κατανοεί κατ' αρχάς τη λειτουργία κυκλωμάτων που περιγράφονται από τη VHDL, και αργότερα, αποκτώντας εμπειρία, να μπορεί να σχεδιάζει από το μηδέν ολόκληρα συστήματα σε FPGA. Τα firmware αυτά μπορεί να τελούν πολυάριθμες λειτουργίες, και να βρίσκουν εφαρμογές σε πολλούς κλάδους, από την έρευνα στη φυσική υψηλών ενεργειών, μέχρι τη βιομηχανία τεχνολογιών αιχμής.

Abstract

This masters thesis, mainly discusses the programming process of the Field-Programmable Gate Arrays (FPGAs), that are being used in the front-end electronics system of the New Small Wheel (NSW) upgrade which will take place in 2020 at the ATLAS detector of LHC at CERN. FPGAs belong to a wider family of integrated circuits, that are able to implement any logic function, according to the will of their user. One of the main programming languages of these chips, is VHDL (VHSIC Hardware Description Language). FPGAs are being used in the MMFE8 (MicroMegas Front-End 8) board prototypes, which are in charge of collecting and processing the primordial signals generated from the detection of elementary particles by the MicroMegas detectors. One of the main goals of this thesis is to guide a reader with no background or prior experience in firmware developing, and clarify concepts which may at first seem complicated, in order to give him the ability to comprehend designs written in VHDL, and eventually make him able to design his own FPGA firmware from scratch. These implementations may have various functionalities, and can be used in any context, such as the research in high energy physics, or the high tech industry.

Ευχαριστίες

Σε αυτό το σημείο θα ήθελα να ευχαριστήσω όλα τα μέλη της Ομάδας Πειραματικής Φυσικής Υψηλών Ενεργειών του ΕΜΠ, για την πολύτιμη βοήθειά τους. Πιο συγκεκριμένα, θα ήθελα να ευχαριστήσω τον κύριο Αλεξόπουλο, που μου έδωσε την ευκαιρία να εργαστώ πάνω σε ένα τόσο ενδιαφέρον θέμα.

Αυτό που πρέπει να αντιληφθεί κανείς πριν ξεκινήσει την ανάγνωση της παρούσας διπλωματικής, είναι το γεγονός ότι τα FPGA και η VHDL, είναι δύο πεδία που απέχουν *αρκετά* από το πρόγραμμα σπουδών της ΣΕΜΦΕ. Η αλήθεια είναι ότι όσο όρεξη να έχει κανείς να μάθει όλα αυτά τα (ωραία) πράγματα ξεκινώντας από το μηδέν, δεν θα καταφέρει τίποτα χωρίς τη σωστή καθοδήγηση! Έτσι, θα ήθελα να ευχαριστήσω τον Πάνο και τον Πάρη, που εκτός από το να θέσουν τις βάσεις για το firmware της MMFE8, άθελά τους μου έμαθαν πολλά, διαβάζοντας και μόνο τον κώδικα που είχαν γράψει. Και η αλήθεια είναι, ότι η γνώση που έχει πλέον συσσωρευθεί από την ομάδα πάνω στις τεχνικές σχεδιασμού firmware, θα ήταν πολύ φτωχότερη, αν δεν είχαμε κάνει συζητήσεις επί συζητήσεων με τον Πάρη, πάνω σε ένα σωρό ενδιαφέροντα ζητήματα γύρω από τα FPGA και το πως λειτουργούν. Επίσης, θα ήθελα να ευχαριστήσω τον Γιώργο Ιακωβίδη, όπου οι γνώσεις του για τον τρόπο λειτουργίας του VMM, έπαιξαν σημαντικό ρόλο στην ανάπτυξη του firmware. Τέλος, θα ήθελα να ευχαριστήσω και τον Αιμίλιο, για τις διορθώσεις του πάνω στο κείμενο της διπλωματικής.

Και φυσικά, σε προσωπικό επίπεδο, θα ήθελα να ευχαριστήσω τη Δήμητρα, για την στήριξή της και την αγάπη που μου δίνει τόσο απλόχερα τα τελευταία χρόνια. Και εννοείται φυσικά, ότι δεν θα είχα φτάσει ποτέ σε αυτό το σημείο εάν δεν είχα την υποστήριξη και την αγάπη των γονιών μου. Τους οφείλω τα πάντα, και σε τελική

ανάλυση, δεν υπάρχουν λόγια για να τους εκφράσω την ευγνωμοσύνη μου.

Περιεχόμενα

I FPGA - VHDL 1

1 Field-Programmable Gate Arrays 3

1.1 Η Αρχιτεκτονική των FPGA . . . 4

1.1.1 LUTs, Flip-Flops, CLBs . . . 6

1.1.2 Hard Blocks . . . 12

1.2 Το Δίκτυο Διασυνδέσεων του FPGA . . . 19

1.3 Η Ροή της Πληροφορίας . . . 22

2 Εισαγωγή στη VHDL 29

2.1 Hello World! . . . 29

2.2 Κυκλώματα Συνδυαστικής Λογικής . . . 31

2.3 Κυκλώματα Σύγχρονης Ακολουθιακής Λογικής . . . 38

2.3.1 FDRE Flip-Flop . . . 39

2.3.2 Shift Register . . . 43

2.3.3 Counter . . . 45

2.3.4 Serializer/Deserializer . . . 48

3 Μηχανές Πεπερασμένων Καταστάσεων σε VHDL 55

3.1 Σύντομη Εισαγωγή στις FSM . . . 56

3.2 Απλές FSM σε VHDL . . . 58

3.2.1 Width Regulator . . . 59

3.2.2 Switch Debouncer . . . 65

4 Εφαρμογές σε Πραγματικά Κυκλώματα - Ζητήματα Χρονισμού 73
4.1 Μετασταθείς Καταστάσεις 74
4.1.1 Συνδυαστικά Κυκλώματα - Race Conditions 75
4.1.2 Σύγχρονα Ακολουθιακά Κυκλώματα - Metastability 78
4.2 Στατική Χρονική Ανάλυση και Vivado® 81
4.2.1 Κοινό Πεδίο Χρονισμού 81
4.2.2 Διαφορετικό Πεδίο Χρονισμού 84
4.2.3 Αντιμετωπίζοντας τα Σφάλματα Χρονισμού 86
4.2.4 Σφάλματα Χρονισμού και FSM - Deadlocking 88

II Εφαρμογές στο Πείραμα του ATLAS στο CERN 95

5 Το Πείραμα ATLAS και η Αναβάθμισή του 97
5.1 Αναζητώντας τα Στοιχειώδη Σωματίδια 97
5.1.1 Το Καθιερωμένο Πρότυπο 98
5.2 Ο Μεγάλος Επιταχυντής Αδρονίων 100
5.3 Ο Ανιχνευτής ATLAS 103
5.4 Η Αναβάθμιση του New Small Wheel 108
5.5 Τα Ηλεκτρονικά Συστήματα του New Small Wheel 112
5.6 Micromegas Front-End Board 117
5.6.1 VMM ASIC 118

6 VMM Readout/Configuration Firmware 127
6.1 Ethernet Communication - Configuration Path 129
6.1.1 Ethernet/UDP Interfacing Modules 129
6.1.2 User Logic - Configuration 136
6.2 Readout Path 144
6.2.1 VMM Readout 146
6.3 Λοιπά Υποσυστήματα 151

Παράρτημα Α΄ Πρωτόκολλα Επικοινωνίας 153
Α΄.1 Ethernet - UDP Protocol 153
Α΄.2 8b/10b Encoding 154

Παράρτημα Β΄ Ο Ανιχνευτής MicroMegas **157**

Παράρτημα Γ΄ Δείγματα Κωδίκων VHDL **161**

Βιβλιογραφία **183**

Μέρος Ι

FPGA - VHDL

1

Field-Programmable Gate Arrays

Τα FPGA *(Field-Programmable Gate Arrays)*, είναι μία οικογένεια ολοκληρωμένων κυκλωμάτων, τα οποία μπορούν να υλοποιήσουν οποιαδήποτε λογική συνάρτηση, ανάλογα τα θέλω του προγραμματιστή που τα σχεδιάζουν (ή προγραμματίζουν). Σε μία εφαρμογή, είτε αυτή περιλαμβάνει την ψηφιακή ανάλυση σήματος (Digital Signal Processing, DSP), την υλοποίηση ενός μικροεπεξεργαστή, ή το σχεδιασμό ενός δρομολογητή υψηλών απαιτήσεων, το μέσο για την επίτευξη των στόχων του σχεδιαστή του συστήματος, θα είναι σίγουρα κάποιο ολοκληρωμένο κύκλωμα. Τα FPGA προσφέρουν απαράμιλλη ευελιξία σε συνδυασμό με υψηλές επιδόσεις. Οι μονάδες FPGA, έχουν τη δυνατότητα να επαναπρογραμματίζονται όσες φορές χρειάζεται, προκειμένου μέσω μίας διαδικασίας δοκιμών και σφαλμάτων *(trial and error)* από μεριάς του σχεδιαστή, να προκύψει στο τέλος ένα firmware[1] με τις επιθυμητές λειτουργίες. Πέρα από το χαρακτηριστικό της προσαρμοστικότητας που διακατέχει τα FPGA, το οποίο τους δίδει πρωταγωνιστικό ρόλο στον τομέα του *Reconfigurable Computing*, η υψηλή τους αποδοτικότητα και ικανότητα παραλληλισμού των διεργασιών που τελούν, τα καθιστούν ιδανικά για εφαρμογές *Hardware Acceleration*. Πολλές φορές, όταν μία διεργασία που εκτελείται μέσω software[2], γίνει αρκετά πολύπλοκη και επίπονη για έναν επεξεργαστή, η λύση για την υλοποίηση του αλγορίθ-

[1]Το firmware είναι ένας αλγόριθμος γραμμένος σε γλώσσα υψηλού επιπέδου, συνήθως VHDL ή Verilog, το οποίο υλοποιείται σε κάποιο εξειδικευμένο τσιπ (π.χ. FPGA) σε μορφή λογικών πυλών ή και άλλων βασικών ηλεκτρονικών μονάδων.

[2]Το software είναι ένα σύνολο εντολών που διαχειρίζονται δεδομένα και πληροφορίες, και συνήθως υλοποιείται στην κεντρική μονάδα επεξεργασίας (Central Processing Unit, CPU) ενός Η/Υ. Μερικές γλώσσες προγραμματισμού για software είναι οι C, C++, Java, Python.

μου αυτού δεν έγκειται στη βελτιστοποίηση του αλγόριθμου στο λογισμικό, αλλά στη χρήση εξειδικευμένου hardware που εκτελεί αποκλειστικά τις εντολές αυτές. Πολλές φορές λοιπόν, χρησιμοποιούνται FPGA παράλληλα με επεξεργαστές, ώστε να επιταχυνθεί η επεξεργασία των δεδομένων, καθώς ένα FPGA μπορεί να είναι τάξεις μεγέθους γρηγορότερο από μία CPU σε συγκεκριμένες εφαρμογές (κυρίως σε όσες απαιτούν παραλληλισμό στην επεξεργασία).

Όσο αναφορά τους τομείς που χρησιμοποιούν FPGA, η ικανότητά των συσκευών αυτών να μεταχειρίζονται απλά ηλεκτρικά σήματα[3], όντας επί της ουσίας ολοκληρωμένα κυκλώματα που βαίνουν σε μία πλακέτα, τα καθιστούν ιδανικά για πολλές εφαρμογές. Χρησιμοποιούνται για παράδειγμα ως ηλεκτρονικοί εγκέφαλοι δορυφόρων, αεροπλάνων, παλμογράφων, διακομιστών (servers), μαγνητικών και αξονικών τομογράφων, συστημάτων επεξεργασίας εικόνας, οθονών, και άλλα πολλά. Οι δυνατές εφαρμογές των FPGA, δεδομένης της ευελιξίας τους, είναι αμέτρητες. Όπως θα διαπιστωθεί και στο δεύτερο μέρος της εργασίας αυτής, μία ακόμη σημαντική εφαρμογή των FPGA είναι στο πείραμα ATLAS του CERN (Κεφάλαια 5 και 6). Επειδή τα FPGA που χρησιμοποιούνται στο πείραμα του ATLAS είναι της εταιρείας Xilinx® , η οποία καταλαμβάνει το μεγαλύτερο μερίδιο της αγοράς στο χώρο, η εργασία αυτή θα επικεντρωθεί στα τσιπ της εν λόγω εταιρείας. Πάρα ταύτα, οι διαφορές ως προς τα εργαλεία ανάπτυξης firmware, και της αρχιτεκτονικής των FPGA, δεν είναι ιδιαίτερα μεγάλες από εταιρεία σε εταιρεία.

1.1 Η Αρχιτεκτονική των FPGA

Το πρώτο FPGA κατασκευάστηκε από την εταιρεία Xilinx® , το 1985 [1], και από τότε η αρχιτεκτονική των FPGA έχει αλλάξει δραματικά. Πάντα όμως, η γενικότερη φιλοσοφία υλοποίησης λογικών συναρτήσεων από τα FPGA βασίζεται στην εκμετάλλευση δύο ειδών πόρων που υλοποιούνται στη μικροδομή του από τρανζίστορ που σήμερα φτάνουν μέχρι και τα 20 nm [39]. Οι πόροι αυτοί, χωρίζονται σε δύο μεγάλες κατηγορίες: Τα *Λογικά Μπλοκ (Logic Blocks)*, και τις *Διασυνδέσεις (Interconnect)* [7, 9, 10, 14]. Τα λογικά μπλοκ αποτελούν την καρδιά των FPGA. Είναι αυτά που πραγματοποιούν τις λογικές πράξεις. Τα σήματα που διαχειρίζονται τα

[3]Αυτά τα σήματα είναι ως επί το πλείστον ψηφιακά. Πάρα ταύτα, ένα FPGA μπορεί να υλοποιήσει και ADC module ώστε να ψηφιοποιεί αναλογικά σήματα.

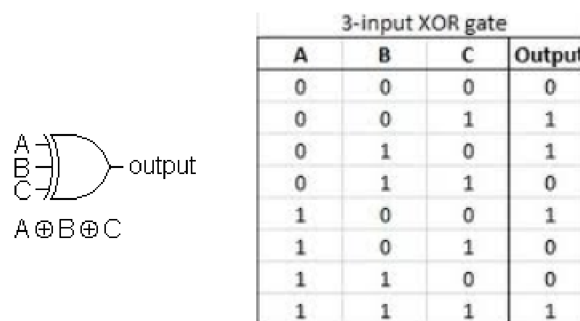
μπλοκ αυτά, μεταδίδονται μέσα στο FPGA μέσω ενός περίπλοκου δικτύου δρομολόγησης, το οποίο είναι υπεύθυνο στο να διασυνδέει τα μπλοκ μεταξύ τους. Και αυτά τα δύο είδη πόρων, είναι πλήρως προγραμματιζούμενα. Διαφορετικά προγράμματα (ή αλλιώς, *firmware*), γραμμένα στη γλώσσα VHDL την οποία συντάσσει ο σχεδιαστής του FPGA, θα ενεργοποιήσουν και διαφορετικά λογικά μπλοκ με διαφορετικό τρόπο, και θα υλοποιήσουν τις ανάλογες διασυνδέσεις προκειμένου τα ηλεκτρικά σήματα, τα προϊόντα της επεξεργασίας από τα μπλοκ δηλαδή, να ταξιδέψουν από κόμβο σε κόμβο, μέχρι να προκύψει το τελικό αποτέλεσμα στην έξοδο του ολοκληρωμένου κυκλώματος. Αυτή την άκρως πολύπλοκη διαδικασία, την υλοποιεί το λογισμικό της κάθε εταιρείας που παρέχει το FPGA. Ο χρήστης δηλαδή, απλά συντάσσει των κώδικα που θέλει να υλοποιήσει, και στη συνέχεια, το λογισμικό με κατάλληλη επεξεργασία ενεργοποιεί τα μέχρι πρότινος νεκρά μπλοκ, και υλοποιεί τις σωστές διασυνδέσεις, ώστε να πραγματοποιηθούν αυτά που έχει στο μυαλό του ο χρήστης. Και φυσικά, στα σημερινά FPGA, αυτή η διαδικασία μπορεί να επαναληφθεί άπειρες φορές.

Όταν τα πρώτα FPGA βγήκαν στην αγορά, έκαναν επανάσταση στο χώρο των *Programmable Logic Devices (PLD)*, καθώς χρησιμοποιούσαν πιο περίπλοκα λογικά μπλοκ σε σχέση με άλλα κυκλώματα της ίδιας λογικής, γεγονός που αύξησε εκθετικά την αποδοτικότητα των συσκευών αυτών. Τα πρώτα PLD της δεκαετίας του '70 [1], αποτελούντο από μία συστοιχία από απλές λογικές πύλες, οι οποίες ήταν μεταξύ τους ασύνδετες, καθώς σε όλους τους κόμβους υπήρχαν ασφάλειες (fuses). Ο σχεδιαστής, έπρεπε να αφαιρέσει τις αντίστοιχες ασφάλειες προκειμένου να ενεργοποιηθούν οι αντίστοιχες διασυνδέσεις, ώστε να υλοποιηθεί το κύκλωμα που είχε κατά νου. Στη συνέχεια αυτή η διαδικασία μπορούσε να γίνει μέσω προγραμματισμού, με τη χρήση των *Hardware Description Languages (HDL)*, οι οποίες περιέγραφαν τα κυκλώματα μέσω μίας σύνταξης κατανοητής στον άνθρωπο. Όμως τα FPGA πήγαν τη τεχνολογία των PLD πολλά βήματα παραπέρα, με την εισαγωγή μίας περίπλοκης συστοιχίας από εκλεπτυσμένα λογικά μπλοκ, διασυνδέσεων, εσωτερικών μνημών RAM, PLL για σύνθεση ρολογιών, DSP μπλοκ για ψηφιακή ανάλυση σημάτων, και I/O μπλοκ για επικοινωνία με το περιβάλλον εκτός του τσιπ. Σε αυτή την ενότητα, θα γίνει μία προσέγγιση "από μέσα προς τα έξω", δηλαδή θα μελετηθεί η δομή του FPGA, ξεκινώντας από τις βασικές μονάδες υπολογισμού που φιλοξενεί, και καταλήγοντας σε μια γενική επισκόπηση του υπό μελέτη ολοκληρωμένου

κυκλώματος.

1.1.1 LUTs, Flip-Flops, CLBs

Ο βασικός μηχανισμός υλοποίησης λογικών συναρτήσεων από τα FPGA, δε γίνεται μέσω απλών λογικών πυλών[4], αλλά μέσω *Look-Up Tables* ή *LUT*, που βρίσκονται παντού στο FPGA, μέσα στα λογικά μπλοκ. Προκειμένου να καταλάβει κανείς τον τρόπο λειτουργίας ενός LUT, θα δοθεί ένα απλό παράδειγμα από τη βιβλιογραφία [7]. Έστω ότι κάποιος θέλει να κατασκευάσει τον *Πίνακα Αληθείας (Truth Table)*, μίας πύλης XOR (Exclusive-OR) τριών εισόδων. Ο πίνακας αυτός είναι ο εξής:



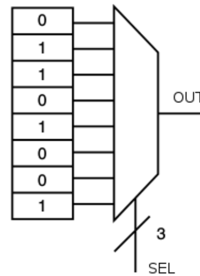
3-input XOR gate

A	B	C	Output
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Σχήμα 1.1.I: Πύλη xor τριών εισόδων μαζί με τον πίνακα αληθείας.

Το γενικό ηλεκτρονικό υποσύστημα που μπορεί να υλοποιήσει μία οποιαδήποτε λογική συνάρτηση της άλγεβρας του Boole, είναι το *LUT* [7, 14]. Ένα LUT, μπορεί να περιγραφεί και ως *Πολυπλέκτης (Multiplexer)*, ο οποίος αποτελείται από μία μνήμη (η είσοδος) N bit, και μία έξοδο ενός bit. Για το παράδειγμα του Σχήματος 1.1.I, το LUT θα έχει στη μνήμη του τις οκτώ δυνατές καταστάσεις της εξόδου, και σαν είσοδο επιλογής (SEL), θα δέχεται ένα διάνυσμα από τρία bit ($2^3 = 8$ δυνατές καταστάσεις). Η επιλογή της εξόδου, θα γίνεται μέσω του SEL, το οποίο για την πρώτη κατάσταση όπου όλοι οι είσοδοι είναι μηδέν (000) επιλέγει την πρώτη θέση της μνήμης, που είναι το λογικό μηδέν. Αν A=1, B=0, C=1 τότε SEL="101" και επιλέγεται η έκτη θέση, με έξοδο πάλι το λογικό μηδέν.

[4]Ο αναγνώστης παραπέμπεται στα αντίστοιχα κεφάλαια των [1, 2, 3] για πληροφορίες σχετικά με τις λογικές πύλες και τα λοιπά βασικά δομικά μέλη της ψηφιακής/λογικής σχεδίασης.



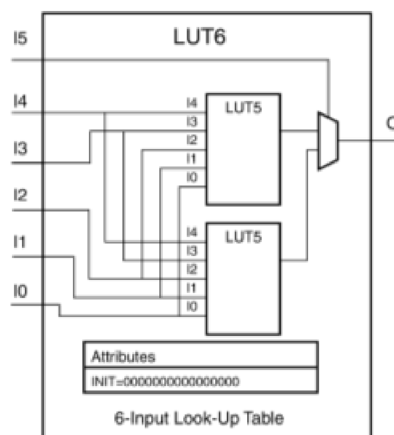
Σχήμα 1.1.ΙΙ: Το LUT ως πολυπλέκτης που υλοποιεί μία πύλη XOR τριών εισόδων [7].

Με τον τρόπο αυτό υλοποιείται η πύλη XOR. Η είσοδος στο LUT κωδικοποιείται ως το διάνυσμα της επιλογής του πολυπλέκτη, και διοχετεύεται στο κύκλωμα του LUT, το οποίο έχει στις αντίστοιχες θέσεις της μνήμης του τις πιθανές εξόδους της πύλης XOR. Προφανώς, το FPGA δεν μένει εκεί. Ο χρήστης μπορεί να κωδικοποιήσει μία πληθώρα εντολών, συντάσσοντας στην HDL της επιλογής του (συνήθως VHDL ή Verilog), οι οποίες εντολές μπορούν σαν σύνολο να μετατραπούν σε μία μήτρα από λογικές πύλες που συνδέονται μεταξύ τους με μεγάλη πολυπλοκότητα προκειμένου να δώσουν το τελικό αποτέλεσμα. Έτσι, απλούς αλγορίθμους οι οποίοι διαθέτουν εντολές που περιλαμβάνουν αποφάσεις (OR, AND, WHILE, WHEN κ.λπ.), ο *Synthesizer* (ο αντίστοιχος compiler για τις HDL) έρχεται για να τους μετατρέψει σε μία συστοιχία λογικών πυλών. Ύστερα, ορίζει το κάθε LUT του FPGA να υλοποιήσει και ένα μικρό κομμάτι αυτής της μήτρας. Τελικά, θα συνδέσει τα LUT με τέτοιο τρόπο ώστε να παραχθεί το τελικό σήμα στην έξοδο. Πολλά LUT συνδεδεμένα μεταξύ τους δηλαδή, υλοποιούν οποιαδήποτε λογική συνάρτηση.

Αν και το LUT έχει εδραιωθεί ως το βασικό εξάρτημα υπολογισμού βασικών λογικών πράξεων στα FPGA, το μέγεθος τους είναι αντικείμενο εκτενών μελετών [7, 8]. Από τη μία, τα μεγάλα LUT με πολλές εισόδους προσφέρουν περισσότερες δυνατότητες υλοποίησης λογικών συναρτήσεων ανά λογικό μπλοκ, με αποτέλεσμα να μειώνεται η ανάγκη για χρήση πολλών LUT, το οποίο με τη σειρά του ελαττώνει την πολυπλοκότητα των διασυνδέσεων, και την καθυστέρηση στα σήματα που αυτή συνεπάγεται. Όμως τα μεγάλα LUT, είναι και αυτά με τη σειρά τους αργά[5], καθώς υλοποιούνται από πιο περίπλοκους πολυπλέκτες. Επίσης, όσο μεγαλύτερα τα LUT, τόσο περισσότερος χώρος στο FPGA πάει χαμένος όταν αυτά μένουν αχρησιμοποί-

[5] Δηλαδή επιβαρύνουν με περισσότερη καθυστέρηση τα σήματα που οδηγούν.

ητα. Από την άλλη, τα μικρότερα LUT έχουν λιγότερες δυνατότητες υλοποίησης λογικών συναρτήσεων, όμως είναι πιο γρήγορα (όντας λιγότερο πολύπλοκα), γεγονός που αντισταθμίζεται από την πολυπλοκότητα των διασυνδέσεων, η οποία αυξάνεται προκειμένου μία συστοιχία από απλά LUT να φέρει σε πέρας μία εξεζητημένη λογική συνάρτηση[6]. Εκτός από το πλήθος των εισόδων των LUT, αντικείμενο έρευνας είναι και ο συνολικός αριθμός των LUT που περιλαμβάνει το κάθε λογικό μπλοκ. Σε κάθε περίπτωση, οι μελέτες αυτές ξεφεύγουν από τους στόχους του παρόντος κειμένου. Στη συνέχεια παρατίθεται το LUT *(*LUT6), που χρησιμοποιούνται σε όλα τα 7-Series FPGA της Xilinx® (Σχήμα 1.1.III).



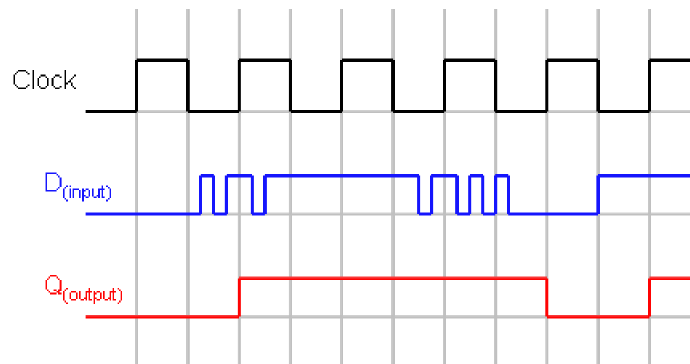
Σχήμα 1.1.III: Το LUT6. Η βασική μονάδα επεξεργασίας ενός Xilinx® Artix-7 FPGA [14].

Όπως μπορεί να διαπιστώσει ο αναγνώστης, το στοιχείο αυτό αποτελείται από δύο επιμέρους LUT πέντε εισόδων. Τα εξωτερικά pins εισόδου του LUT6 είναι έξι, και η έξοδος είναι μόνο μία. Προκειμένου να χρησιμοποιηθεί το LUT για τον υπολογισμό μίας λογικής συνάρτησης, πρέπει να οριστεί μία 64-bit δεκαεξαδική τιμή στη μεταβλητή INIT. Για παράδειγμα, η τιμή 0x8000000000000000, που αντιστοιχεί στη δυαδική συμβολοσειρά με τιμή 1 στη πρώτη θέση (Most Significant Bit (MSB)), και 0 στα υπόλοιπα 63 δυφία, κωδικοποιεί μία πύλη AND με έξι εισόδους, αφού θα δώσει στην έξοδο του LUT το λογικό ένα της πρώτης θέσης μόνο αν όλοι οι είσοδοι είναι αληθείς (που δρουν ως το SEL του πολυπλέκτη).

[6]Όπως θα διαπιστωθεί και στο Κεφάλαιο 4, οι καθυστερήσεις των σημάτων μέσα στο FPGA παίζουν πολύ σημαντικό ρόλο στην απόδοση ενός firmware.

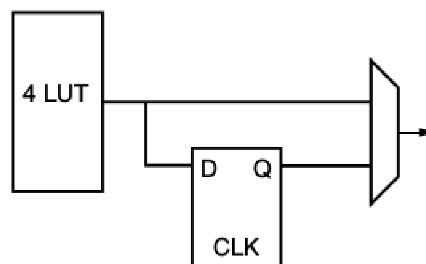
Μέχρι τώρα βέβαια, έχει γίνει αναφορά μόνο σε κυκλώματα *Συνδυαστικής Λογικής (Combinatorial Logic)* και όχι *Ακολουθιακής Λογικής (Sequential Logic)*. Στην πρώτη περίπτωση, η τιμή της εξόδου εξαρτάται μόνο από την κατάσταση των σημάτων στην είσοδο, ως έχουν σε πραγματικό χρόνο, ενώ στη δεύτερη περίπτωση, το αποτέλεσμα στην έξοδο του λογικού κυκλώματος εξαρτάται και από *προηγούμενες* καταστάσεις που έλαβαν χώρα στο κύκλωμα. Έτσι, γεννιέται η ανάγκη για αποθήκευση πληροφορίας, λειτουργία που στο επίπεδο του κυκλώματος την υλοποιούν οι *Registers*. Ο πιο απλός register, είναι η πύλη flip-flop *(Flip-Flop Gate)*. Επίσης, προκειμένου να λειτουργήσει ένα κύκλωμα ακολουθιακής λογικής, είναι απαραίτητη η ύπαρξη ενός σήματος χρονισμού δηλαδή ενός *ρολογιού*. Η γενική φιλοσοφία της flip-flop συνοψίζεται στο γεγονός ότι αλλάζει την κατάστασή της μόνο όταν αλλάζει κατάσταση και το ρολόι που την χρονίζει. Πιο συγκεκριμένα, οι flip-flop συνήθως αλλάζουν κατάσταση μόνο όταν εντοπίσουν ένα rising-edge στο ρολόι χρονισμού, δηλαδή όταν το σήμα του ρολογιού μεταβεί από το λογικό μηδέν στο λογικό ένα. Αυτό σημαίνει ότι σε οποιαδήποτε άλλη στιγμή, η έξοδος της flip-flop δεν αλλάζει και παραμένει η ίδια. Αν για παράδειγμα το ρόλοι χρονισμού έχει συχνότητα $f = 40\,Mhz$ (άρα περίοδο $T = 25\,ns$), τότε οι flip-flop που θα το χρησιμοποιούν, θα αλλάζουν κατάσταση μία φορά κάθε $25\,ns$. Παρεμπιπτόντως, αξίζει να τονιστεί ότι ως αλλαγή κατάστασης στη flip-flop, νοείται η αποθήκευση της τιμής εισόδου, το οποίο συνεπάγεται αυτόματα και προώθηση της τιμής αυτής στην έξοδο της πύλης. Άρα ανάμεσα από τα rising-edges, η οποιαδήποτε αλλαγή στα bit εισόδου, αμελείται από τη flip-flop, και η έξοδός της παραμένει ίδια (βλ. σχ. 1.1.IV). Ο τρόπος λειτουργίας της flip-flop, μπορεί να συγκριθεί με μία φωτογραφική μηχανή που παίρνει φωτογραφίες περιοδικά. Ανάμεσα από τα κλικ της μηχανής, τα γεγονότα που συμβαίνουν μπροστά από το φακό, χάνονται.

Οι πύλες flip-flop, όπως και οι υπόλοιποι τύποι registers (π.χ. latches), υλοποιούνται με έναν συγκεκριμένο συνδυασμό λογικών πυλών. Πύλες flip-flop βρίσκονται παντού μέσα σε έναν ηλεκτρονικό υπολογιστή, και περισσότερο σε όσα ηλεκτρονικά υποσυστήματα έχουν να κάνουν με μνήμη. Τα FPGA, χρησιμοποιούν κατά κόρον πύλες flip-flop, προκειμένου να αποθηκεύσουν τα δεδομένα που προκύπτουν από τους υπολογισμούς των LUT [7, 9, 14]. Ο συνδυασμός των δύο αυτών βασικών ηλεκτρονικών υποσυστημάτων, μαζί με έναν multiplexer που επιλέγει είτε την έξοδο της flip-flop, είτε την έξοδο του LUT απευθείας, δομεί τα λογικά μπλοκ σε όλα τα FPGA



Σχήμα 1.1.IV: Χρονικό διάγραμμα μίας πύλης flip-flop. Φαίνεται πως η κατάσταση της εξόδου αλλάζει μόνο στα rising-edges του ρολογιού, και η τιμή της τότε εξισώνεται με εκείνη της εισόδου. Ανάμεσα από τις αλλαγές κατάστασης του ρολογιού από το μηδέν στο ένα, η κατάσταση του Q δεν αλλάζει. Η λογική της πύλης αυτής θα αποσαφηνιστεί περισσότερο στην ενότητα 2.3 όπου θα εξεταστεί η εφαρμογή της στη VHDL.

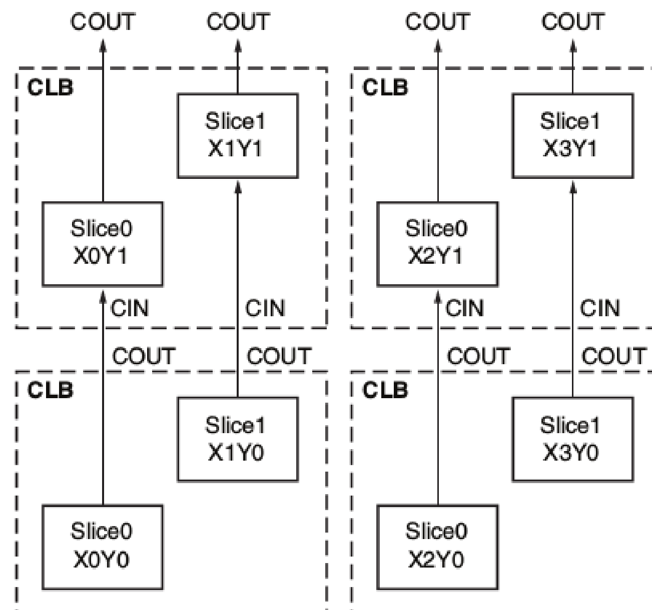
[7].



Σχήμα 1.1.V: Το πιο απλό λογικό μπλοκ [7].

Ως προς τα 7-series FPGA της Xilinx® , τα λογικά μπλοκ ονομάζονται *Configurable Logic Blocks* ή πιο απλά *CLB*. Κάθε CLB αποτελείται από ένα ζεύγος από *Slices*. Κάθε slice, διαθέτει τέσσερα LUT6, οκτώ flip-flop (ένα για κάθε LUT πέντε εισόδων που περιέχεται στο LUT6, βλ. Σχ. 1.1.III), μαζί με πολυπλέκτες. Επίσης σε ένα slice υπάρχει και ένα εξειδικευμένο μπλοκ που υλοποιεί προσθαφαιρέσεις δυαδικών. Το μπλοκ αυτό, δεν διασυνδέεται μόνο μέσα στο slice, αλλά έχει και δύο pins που μεταφέρουν το *Carry bit* (το αντίστοιχο του κρατούμενου για την πρόσθεση στο πεδίο των δυαδικών), έξω από το slice στο αντίστοιχο μπλοκ του γειτονικού CLB.

Έτσι, τα μπλοκ πρόσθεσης (που καλούνται και *Arithmetic Logic Units (ALU))*, των γειτονικών CLB διασυνδέονται και ανταλλάσσουν μεταξύ τους τα carry bits (μέσω των pins COUT και CIN), προκειμένου να κάνουν συνεργατικά τις προσθέσεις και αφαιρέσεις των δυαδικών αριθμών κατά την επεξεργασία. Η συνδεσμολογία αυτή είναι εμφανής στο Σχήμα 1.1.VI, και καλείται *Fast Carry Chain*.



Σχήμα 1.1.VI: CLB και slice. Κάθε CLB διαθέτει δύο slice, τα οποία διασυνδέονται με γειτονικά slice για διευκόλυνση των υπολογισμών σχετικών με πρόσθεση δυαδικών (η γενικότερη διασύνδεση των CLB με το δίκτυο διασύνδεσης του FPGA μελετάται σε άλλη υποενότητα). Η αλυσιδωτή αυτή διασύνδεση διευκολύνει σημαντικά τη διαδικασία υπολογισμών από το FPGA σαν σύνολο [14].

Εκτός από την υλοποίηση λογικών συναρτήσεων και αριθμητικών πράξεων, τα slice μπορούν να διαμορφωθούν με τέτοιο τρόπο ώστε να μην πραγματοποιούν πράξεις, αλλά να αποθηκεύουν προσωρινά πληροφορίες, σαν μνήμες RAM. Μία τέτοια μνήμη καλείται *Distributed RAM*, και χρησιμοποιεί λειτουργίες εγγραφής μνήμης συγχρονισμένες με το ρολόι εισόδου του slice, και αντίστοιχες λειτουργίες ανάγνωσης (δηλαδή μετάδοση του bit σε κάποια γραμμή που είναι συνδεδεμένη με το slice), με ίδιο ή διαφορετικό χρονισμό. Ένα slice μπορεί να μετατραπεί σε 256-bit RAM, αν δεσμεύσει και τα τέσσερα LUT του για αυτό το σκοπό [14]. Επίσης, τα

slice μπορούν να προγραμματιστούν με τέτοιο τρόπο ώστε να καθυστερούν τα σήματα εισόδου, για συγκεκριμένο αριθμό κύκλων ρολογιού (μέχρι 32 κύκλους). Κάθε slice δηλαδή, μπορεί να μετατρέψει τα LUT του σε flip-flops που είναι συνδεδεμένες μεταξύ τους σε σειρά και με κοινό ρολόι, και να συνδέσει και την τελική έξοδο αυτού του LUT με τις εξωτερικές flip-flop του slice. Το αποτέλεσμα είναι να δημιουργείται ένας 32-bit *Shift Register*[7] ο οποίος δύναται να καθυστερήσει το σειριακό σήμα εισόδου για ορισμένο αριθμό κύκλων. Φυσικά, δεν πρέπει να ξεχνάει κανείς και το γενικό σκοπό ύπαρξης των LUT, που είναι ο υπολογισμός λογικών συναρτήσεων. Σε αυτή τη γενική περίπτωση που υλοποιείται στην πλειοψηφία των περιπτώσεων σε ένα FPGA, το LUT μπορεί να θεωρηθεί κατά μία έννοια και ως μνήμη ROM [14] Στο τέλος του παρόντος Κεφαλαίου, μπορεί να βρεθεί το γενικό σχεδιάγραμμα του Slice, όπως αυτό δίνεται από τη Xilinx® .

1.1.2 Hard Blocks

Εκτός όμως από τα γενικά λογικά μπλοκ, σε ένα FPGA υπάρχουν και τα λεγόμενα *Hard Blocks* (εξειδικευμένα μπλοκ), τα οποία είναι υποσυστήματα που δεν χρησιμοποιούνται για γενική χρήση, αλλά είναι προκατασκευασμένα από την εταιρεία, και επιτελούν συγκεκριμένες λειτουργίες που διαφορετικά θα δέσμευαν πολλά ευέλικτα CLB για να υλοποιηθούν. Μέχρι τώρα, έχουν αναφερθεί τα ALU μέσα στα slice των CLB, τα οποία μάλιστα διασυνδέονται μεταξύ τους με τέτοιο τρόπο ώστε να ανταλλάσσουν τα bit των κρατουμένων μεταξύ τους, προκειμένου να υπολογίζουν συλλογικά τις προσθαφαιρέσεις των δεκαδικών.

Εκτός από μπλοκ για προσθαφαιρέσεις λοιπόν, υπάρχουν και μπλοκ πολλαπλασιαστών *(Multipliers)* [7, 9]. Πρόκειται για εξειδικευμένα υποσυστήματα μέσα στο FPGA, τα οποία ενεργοποιούνται όταν ο Synthesizer αντιληφθεί ότι μέσα στον κώδικα HDL υπάρχει μία λειτουργία που περιλαμβάνει πολλαπλασιασμούς. Αντί να χρησιμοποιηθεί μία συστοιχία από πολλά LUT, τα οποία θα ανταλλάσσουν μεταξύ τους πληροφορίες για να προβούν σε πολλαπλασιασμούς, τη θέση τους θα πάρει ένα ειδικό μπλοκ που πολλαπλασιάζει δυαδικά αλφαριθμητικά μεταξύ τους, απο-

[7]Shift Register είναι μία συστοιχία από πύλες flip-flop, οι οποίες επειδή είναι συνδεδεμένες σε σειρά (η έξοδος της μίας είναι η είσοδος της επόμενης) με κοινό ρολόι, καθυστερούν το σήμα της πρώτης εισόδου, από το να φτάσει στην τελική έξοδο, για τόσους κύκλους όσο και το πλήθος των πυλών. Η λειτουργία του κυκλώματος αυτού θα αποσαφηνιστεί στο Κεφάλαιο 2.

δεσμεύοντας έτσι τα CLB από αυτή τη δύσκολη δουλειά, εξοικονομώντας ταυτόχρονα χώρο. Ένα multiplier block άλλωστε δεσμεύει πολύ λιγότερο χώρο σε σχέση με τα CLB που θα απαιτούνταν για να γίνει η λειτουργία του πολλαπλασιασμού.

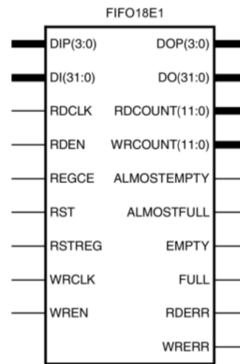
Ένα άλλο hard block που απαντάται συνεχώς στα FPGA είναι ενσωματωμένες μνήμες RAM *(Block RAM)* που βρίσκονται σε διάφορα σημεία του ολοκληρωμένου κυκλώματος [7, 14]. Εκτός από τη δυνατότητα διαμόρφωσης της λειτουργίας LUT για την υλοποίηση μίας μικρής μνήμης RAM (βλ. υποεν. 1.1.1), μέσα στο FPGA υπάρχουν και οι εξειδικευμένες μνήμες RAM, όπου αν συνυπολογιστούν όλες μαζί, μπορούν να φτάσουν τις αρκετές δεκάδες Mb σε χωρητικότητα. Η ύπαρξη τέτοιων μνημών, κρίνεται γενικά απαραίτητη για την εύρυθμη λειτουργία του FPGA, ιδίως σε περίπλοκες εφαρμογές όπου απαιτείται προσωρινή αποθήκευση πληροφορίας. Η πιο συνηθισμένη εφαρμογή των RAM είναι οι μνήμες *FIFO* μέσα στα FPGA.

First-In First-Out Memory

Η μνήμη *FIFO*[8] (First-In First-Out), χρησιμεύει ιδιαίτερα στην προσωρινή αποθήκευση (buffering) ασύγχρονων δεδομένων [1]. Όταν δεδομένα πρέπει να μεταβούν από ένα πεδίο χρονισμού *(clock domain)* σε ένα άλλο, τότε αναπόφευκτα, η ροή των δεδομένων θα αλλοιωθεί. Πιο συγκεκριμένα, αν τα δεδομένα μεταδίδονται με υπερβολικά μεγάλο ρυθμό για τις δυνατότητες του δέκτη, τότε σχεδόν πάντα χρησιμοποιείται μία FIFO για λειτουργίες buffering. Η FIFO μπορεί να έχει ένα ορισμένο πλάτος (δηλαδή τα δεδομένα εισόδου να είναι διανύσματα από δυαδικά ψηφία με συγκεκριμένο μήκος, π.χ. 16 bit), και ορισμένο βάθος (δηλαδή πλήθος διαθέσιμων θέσεων). Η χωρητικότητα της FIFO υπολογίζεται πολλαπλασιάζοντας το βάθος της μνήμης με το πλάτος της. Συνηθισμένες χωρητικότητες για τις FIFO που υλοποιούνται στα dedicated RAM blocks των 7-Series FPGA είναι $18\ kb$ και $32\ kb$. Προκειμένου να αποθηκευτούν δεδομένα εισόδου (DIN) σε μία FIFO, πρέπει να χρησιμοποιηθεί ένα ρολόι χρονισμού (WR_CLK) για το sampling των bit εισόδου και ένα σήμα *write enable* (WR_EN), το οποίο μόνο όταν είναι ψηλά τα δεδομένα εισόδου εγγράφονται στη μνήμη της FIFO. Με τη σειρά που έγιναν οι εγγραφές, τα δεδομένα δρομολογούνται και εκτός της μνήμης (η όλη διαδικασία είναι ανάλογη με την ουρά εξυπηρέτησης σε μία τράπεζα). Ανάλογα υπάρχει ένα ρολόι χρονισμού ανάγνωσης

[8] Πρόκειται ουσιαστικά για μία μνήμη RAM με βοηθητικά κυκλώματα που εξυπηρετούν στην εύκολη πρόσβαση της μνήμης [14].

(RD_CLK) και ένα σήμα *read enable* (RD_EN), που όταν είναι ψηλά, τα δεδομένα εξέρχονται από τη FIFO, με το ρυθμό του RD_CLK. Τέλος, συνήθως υπάρχουν και σήματα που ενημερώνουν το χρήστη για την κατάσταση της μνήμης (αν είναι άδεια, γεμάτη κ.ά.).



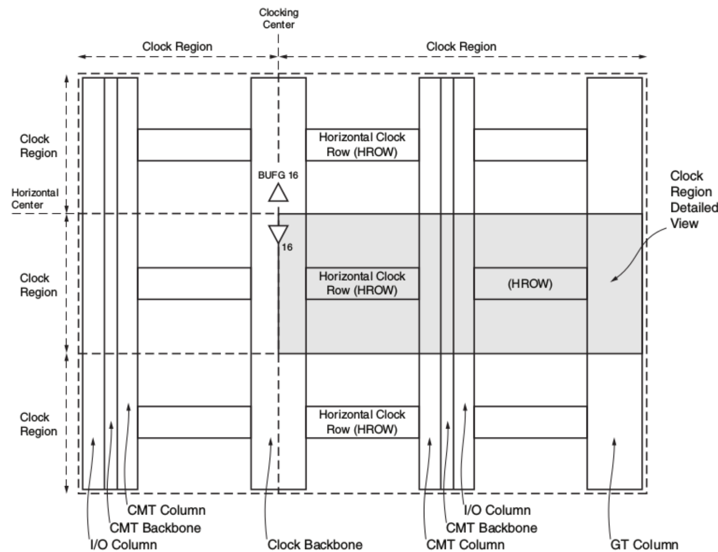
Σχήμα 1.1.VII: FIFO block των 7-Series Xilinx® FPGAs [14].

Clock Management Tiles - PLL/MMCM

Εκτός από τα μπλοκ μνήμης, τα FPGA της Xilinx® διαθέτουν και εξειδικευμένα υποσυστήματα υπεύθυνα για την παραγωγή χρονισμού για ολόκληρο το κύκλωμα. Αυτά τα μπλοκ, ονομάζονται *Clock Management Tiles*, ή *CMT* [14]. Κάθε CMT, προσφέρει συγκεκριμένα κυκλώματα, υπεύθυνα για την παραγωγή ρολογιών διαφόρων συχνοτήτων, με ελάχιστο jitter, και πολύ συγκεκριμένη φάση σε σχέση με τα εξωτερικά ρολόγια αναφοράς που τροφοδοτούν τα CMT. Τα CMT είναι οργανωμένα σε στήλες, οι οποίες είναι πολύ κοντά στα I/O blocks του FPGA. Αυτό συμβαίνει γιατί τα σήματα των ρολογιών που εισέρχονται στο FPGA από την πλακέτα, πρέπει να υφίστανται όσο το δυνατόν λιγότερη καθυστέρηση, jitter, θόρυβο, και γενικότερη παραμόρφωση πριν υποστούν επεξεργασία από το CMT. Το FPGA διαθέτει *Περιοχές Χρονισμού (Clock Regions)*, οι οποίες στα μεγάλα FPGA της σειράς 7 της Xilinx® , μπορεί να είναι και 24 τον αριθμό. Όλα τα υποσυστήματα του FPGA χρησιμοποιούν ρολόγια για να λειτουργήσουν, και ο σωστός χρονισμός κρίνεται απαραίτητος για την ομαλή λειτουργία ολόκληρου του FPGA. Κυκλώματα στην ίδια περιοχή χρονισμού, θα τροφοδοτούνται και από ρολόγια με παρόμοιες διαφορές φάσης. Μέσα

στο FPGA, υπάρχει μία συστοιχία από *Global Clock Trees* που διατρέχει τη ραχοκοκκαλιά του FPGA, και διανείμει τα σήματα των ρολογιών που παράγονται από τα CMT σε όλες τις περιοχές χρονισμού με ελάχιστες καθυστερήσεις, ενώ υπάρχουν και αντίστοιχα τοπικά δίκτυα που διανείμουν τα σήματα μέχρι και σε τρία γειτονικά clock regions [14]. Ο σχεδιαστής του FPGA, μπορεί να χρησιμοποιήσει οποιοδήποτε τύπο δρομολόγησης επιθυμεί. Προφανώς, τα πολυπλοκότερα firmware, που θα απαιτούν και περισσότερα CLB, θα απλώνονται και σε πολλές clock regions, με αποτέλεσμα να ενδείκνυται στην προκείμενη περίπτωση η χρήση του global routing. Σε κάθε περίπτωση, η προσπέλαση γίνεται με τη βοήθεια ειδικών buffers (BUFG, IBUFG, BUFGCTRL, κ.ά.) οι οποίοι δέχονται σαν είσοδο τα ρολόγια που εξέρχονται από τα CMT και έχουν σαν έξοδο τη διασύνδεση με την αντίστοιχη δρομολόγηση. Ο πιο διαδεδομένος buffer είναι ο BUFG, *(Global Clock Buffer)*, ο οποίος δέχεται ένα σήμα στην είσοδό του, συνήθως ρολόι, και το προωθεί στο global clock tree, προς διάθεση όλων των υποσυστημάτων που μπορεί να το χρειαστούν. Το κάθε CMT, αποτελείται από δύο κύρια ηλεκτρονικά κυκλώματα, τα οποία είναι υπεύθυνα για την σύνθεση παλμών χρονισμού. Πρόκειται για το *Phase-Locked Loop*, ή *PLL*, και για το *Mixed-Mode Clock Manager*, ή *MMCM*. Και τα δύο αυτά συστήματα, χρειάζονται ένα ρολόι αναφοράς για να λειτουργήσουν το οποίο προέρχεται σχεδόν πάντα από κάποιον κρύσταλλο που βαίνει πάνω στην ίδια πλακέτα όπου βρίσκεται το FPGA. Μπορεί επίσης να προέρχεται και από μία άλλη πλακέτα, η οποία μέσω κάποιου καλωδίου (π.χ. SMA), να το παρέχει στο FPGA. Μία ακόμα περίπτωση είναι το ρολόι να έχει ανακτηθεί από ένα τσιπ Ethernet PHY που το συνθέτει από τη γραμμή του Ethernet, προκειμένου να υπάρχει συγχρονισμός στην επικοινωνία μεταξύ FPGA και του δικτύου.

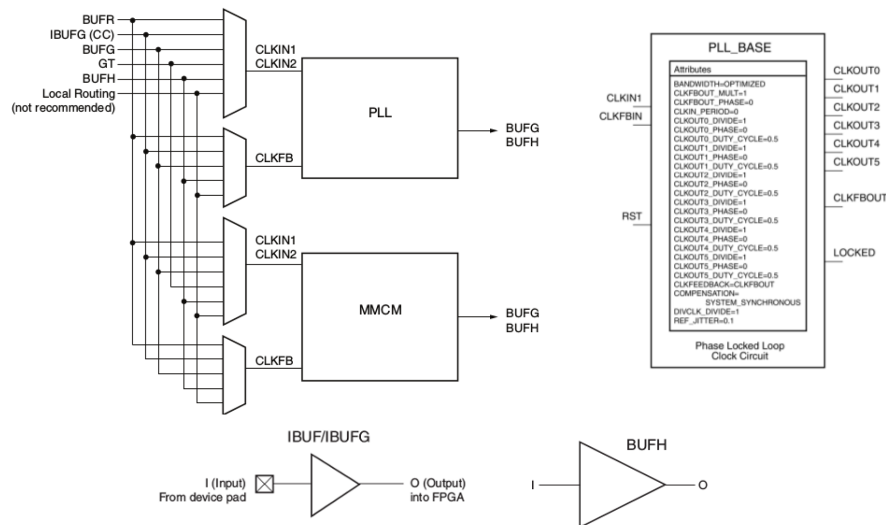
Στα παρακάτω σχήματα, απεικονίζεται σχηματικά μία clock region, με το global routing που τη διαπερνά, το CMT, και τα I/O (Σχήμα 1.1.VIII). Επίσης, απεικονίζεται και μία γενική άποψη του CMT, με τους πολυπλέκτες επιλογής ρολογιού, το PLL και δύο ενδεικτικούς buffers χρονισμού (Σχήμα 1.1.IX).



Σχήμα 1.1.VIII: Σχηματική αναπαράσταση των περιοχών χρονισμού των 7-Series FPGA μαζί με τα περιεχόμενά του. [14].

Τα PLL και MMCM, είναι κυκλώματα υπεύθυνα για τη σύνθεση ρολογιών, τα οποία μπορεί να έχουν οποιαδήποτε συχνότητα και διαφορά φάσης σε σχέση με το ρολόι αναφοράς στην είσοδό τους [1, 14]. Επίσης, λειτουργούν ως φίλτρα για το jitter, και ως βασικά υποσυστήματα σε εφαρμογές *CDR* (Clock and Data Recovery). Αξίζει επίσης να σημειωθεί ότι οι λειτουργίες των PLL και MMCM δεν παρουσιάζουν μεγάλες διαφορές. Το PLL αποτελεί ειδικότερη περίπτωση του MMCM, ως προς τις δυνατότητές του. Εδώ, θα μελετηθεί λίγο περισσότερο ο τρόπος σύνθεσης σημάτων χρονισμού από το PLL, όπως αυτός παρατίθεται στη γενική βιβλιογραφία [1].

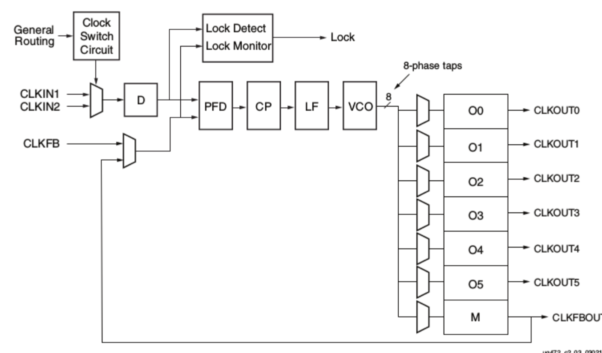
Τα κύρια υποσυστήματα ενός PLL είναι: ο *phase detector*, ο ενισχυτής *(amplifier)*, ο *voltage-controlled oscillator (VCO)* και ο πολλαπλασιαστής *(multiplier (M))*. Το PLL, δέχεται αρχικά ένα σήμα χρονισμού ως ρολόι αναφοράς (reference clock), το οποίο διοχετεύει στο VCO. Το VCO είναι ένα κύκλωμα που με τη σειρά του παράγει και αυτό ένα ρολόι, η συχνότητα και η φάση του οποίου όμως, ελέγχεται από το PLL. Όταν στην είσοδο του PLL εισέλθει ένας παλμός, το PLL με τη βοήθεια του phase detector, εντοπίζει τη διαφορά φάσης μεταξύ του ρολογιού του VCO και του παλμού εισόδου. Το κύκλωμα του phase detector εντοπίζει τη διαφορά ανάμεσα στα rising edges των δύο σημάτων, και όσο μεγαλύτερη είναι αυτή η διαφορά, παράγει αναλόγως και πλατύτερο λογικό παλμό στην έξοδό του. Αυτό σημαίνει πως όταν τα δύο



Σχήμα 1.1.IX: Αριστερά: Γενική σχηματική αναπαράσταση ενός CMT. Δεξιά: Το schematic του PLL component, όπως αυτό δίνεται για να το συμπεριλάβει ο προγραμματιστής στον κώδικά του. Κάτω παρατίθενται δύο τυπικοί clock buffers. Ο IBUFG είναι global clock buffer που λαμβάνει τα σήματα χρονισμού απ' έξω από το FPGA και τα δρομολογεί στο γενικό routing, ενώ ο BUFH λαμβάνει την έξοδο από τα CMT και διανείμει τα ρολόγια στο τοπικά δίκτυα [14].

ρολόγια είναι συμφασικά, τότε ο phase detector παράγει έναν λογικό παλμό 1 με μηδενικό πλάτος, δηλαδή ουσιαστικά, το λογικό μηδέν. Όταν όμως ο phase detector παράγει λογικούς παλμούς με μη μηδενικό πλάτος, τα σήματα αυτά θα ενισχυθούν, και θα περάσουν στο VCO, το οποίο θα αλλάξει τη συχνότητα και τη φάση του ρολογιού του, προκειμένου να ευθυγραμμίσει το ρολόι που παράγει με τη συχνότητα του παλμού εισόδου. Έτσι σιγά-σιγά, ο phase detector θα παράγει όλο και στενότερους παλμούς, όσο τα συγκρινόμενα σήματα έρχονται σε φάση. Όταν γίνουν συμφασικά και ο phase detector δεν παράγει τίποτα πια, το PLL πλέον έχει *κλειδώσει (PLL_LOCKED)*. Έτσι, το PLL *συγχρονίζει* δύο συχνότητες ρολογιών. Με αυτό τον τρόπο συνήθως γίνεται η διαδικασία του CDR, και του clock deskewing. Ο άλλος τρόπος λειτουργίας του PLL, είναι η σύνθεση συχνοτήτων από ένα ρολόι αναφοράς. Το ρολόι εισέρχεται στην είσοδο του PLL και στο VCO, και ένας πολλαπλασιαστής (η τιμή του οποίου καθορίζεται από το χρήστη), πολλαπλασιάζει με έναν ακέραιο αριθμό M, τη συχνότητα εισόδου του ρολογιού. Η συχνότητα αυτή θα είναι η συχνότητα του ρολογιού του VCO. Στη συνέχεια το πολλαπλασιασμένο αυτό ρολόι

διοχετεύεται πάλι στο PLL ως feedback, προκειμένου το PLL να κάνει συμφασικά τα δύο ρολόγια. Ύστερα, το πολλαπλασιασμένο ρολόι, μπορεί να δρομολογηθεί σε πολλά κυκλώματα που διαιρούν κατά N την πολλαπλασιασμένη συχνότητα. Έτσι, έχοντας ένα ρολόι αναφοράς με συχνότητα f_{ref}, μπορεί να παραχθεί ένα άλλο ρολόι οποιασδήποτε συχνότητας, ορίζοντας τον κοινό πολλαπλασιαστή και τον διαιρέτη αναλόγως. Για παράδειγμα, τα PLL που χρησιμοποιούνται στα Xilinx® 7-Series FPGA, έχουν έξι εξόδους (βλ. Σχ. 1.1.IX). Επομένως μπορούν να παραχθούν έξι ρολόγια με συχνότητα f_i $(i = 0, 1, ..., 5)$, σύμφωνα με τον τύπο $f_i = f_{ref}\frac{M}{N}$. Αυτή είναι και η συχνότερη χρήση του PLL στα FPGA. Κατά κανόνα, η συντριπτική πλειοψηφία των firmware που γράφονται για FPGA, περιλαμβάνουν και ένα PLL, το οποίο χάρη στην ευελιξία του, μπορεί από μία εξωτερική συχνότητα αναφοράς, να παράξει πολλά ξεχωριστά ρολόγια που δύναται να τα δρομολογήσει στα γενικά η τοπικά δίκτυα, για να χρονίσουν flip-flop, μνήμες RAM και I/O blocks. Εκτός από τα FPGA, τα PLL χρησιμοποιούνται σε όλα τα ψηφιακά κυκλώματα σήμερα, με εφαρμογές σε κινητά τηλέφωνα, μικροεπεξεργαστές, και μητρικές πλακέτες υπολογιστών [1].



Σχήμα 1.1.X: Λεπτομερής σχηματική αναπαράσταση PLL ενός 7-Series FPGA [14].

Transceivers

Εκτός από τα προαναφερθέντα hard blocks που τελούν συγκεκριμένες εργασίες μέσα στο FPGA, υπάρχουν και οι *Transceivers*[9]. Τα κυκλώματα αυτά, είναι περίπλοκα ηλεκτρονικά υποσυστήματα τα οποία υλοποιούν σειριακές διασυνδέσεις υψηλής ταχύτητας, με άλλα ολοκληρωμένα κυκλώματα εκτός του FPGA. Ο άλλος κόμβος μπορεί

[9]Transmitter+Receiver.

να βρίσκεται πάνω στην ίδια πλακέτα, ή σε διαφορετική. Σε κάθε περίπτωση, οι ταχύτητες των 7-Series Transceiver, κυμαίνονται από τα $6.6\ Gb/s$ μέχρι τα $28.05\ Gb/s$ [14], και μπορούν να υλοποιήσουν μία πληθώρα πρωτοκόλλων επικοινωνίας υψηλών ταχυτήτων, όπως PCI Express, SATA, SAS, και Ethernet μέσω οπτικής ίνας ή χαλκού. Η ύπαρξη ενός Transceiver κρίνεται απαραίτητη για την επικοινωνία του FPGA με τον έξω κόσμο, κάνοντας τη δουλειά του χρήστη ευκολότερη, αφού μπορεί να προωθήσει εντολές στο FPGA μέσω Ethernet (περισσότερα στο Κεφάλαιο 6).

1.2 Το Δίκτυο Διασυνδέσεων του FPGA

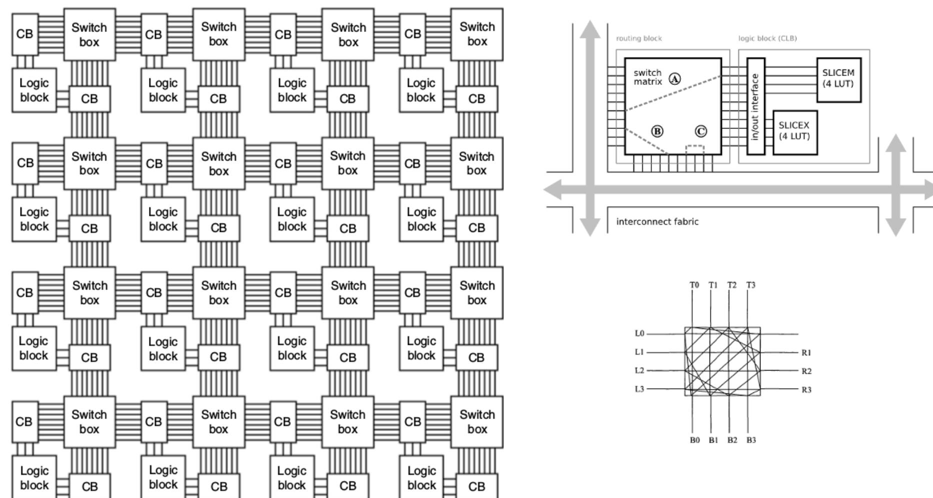
Έχοντας πλέον εμβαθύνει στη μικροδομή και στις λειτουργικότητες των λογικών μπλοκ του FPGA, τα οποία μπορεί να είναι είτε γενικής χρήσης (CLBs, Slices, LUTs, FFs), είτε ειδικής (block RAM, CMT/PLL), μένει να αποσαφηνιστεί και η αρχιτεκτονική των διασυνδέσεων μεταξύ των μπλοκ αυτών. Έχει ήδη αναφερθεί ότι ο χρονισμός του FPGA, διαδίδεται είτε σε γενικά δίκτυα (global clocking routes), είτε σε τοπικά (local clocking routes). Επίσης, τα γειτονικά slice, ανταλλάσσουν μεταξύ τους πληροφορίες που αφορούν τις προσθαφαιρέσεις των δυαδικών αλφαριθμητικών για να αυξήσουν τη γενική τους αποδοτικότητα. Όμως δεν έχει γίνει καμία αναφορά για το πως τα αποτελέσματα των LUT περνάνε σε άλλα CLB για επεξεργασία, ούτε για το πως οι έξοδοι των block RAM διαβιβάζονται στα LUT, ούτε για το πως ένα σήμα εκτός του FPGA περνάει στα λογικά του μπλοκ. Αυτή η ενότητα πραγματεύεται τις γενικές διασυνδέσεις που υφίστανται σε ένα FPGA.

Κατ' αρχάς, αξίζει να σημειωθεί ότι η συνολική επιφάνεια του FPGA, δεσμεύεται ως επί το πλείστον από το εσωτερικό δίκτυο διασυνδέσεων και όχι από τα λογικά μπλοκ, είτε αυτά είναι CLB, είτε αυτά είναι hard blocks. Πιο συγκεκριμένα, το $80 - 90\%$ της επιφάνειας του FPGA καλύπτεται από διασυνδέσεις, και μόνο το $10 - 20\%$ καλύπτεται από λογικά μπλοκ [10]. Η γενική μορφολογία των FPGA του σήμερα, είναι η λεγόμενη *Island-Style Architecture* [7]. Αυτό το μοτίβο αρχιτεκτονικής, υποδεικνύει ότι στα FPGA υπάρχει μία πληθώρα κάθετων και οριζόντιων καλωδίων που μεταφέρουν τα σήματα σε όλα τα μήκη και πλάτη του ολοκληρωμένου κυκλώματος. Τα καλώδια αυτά, διασταυρώνονται μεταξύ τους σε συγκεκριμένα σημεία-κόμβους, οι οποίοι μπορούν να προγραμματιστούν αναλόγως, προκειμένου ένα σήμα να μπορεί να κατευθυνθεί πλήρως μέσα στο FPGA. Πιο συγκεκριμένα, σε

ένα FPGA υπάρχουν δύο είδη μπλοκ που είναι υπεύθυνα για τις δρομολογήσεις των σημάτων. Αυτά είναι τα *Connection Blocks (CB)* και τα *Switch Boxes (SB)*. Τα CB είναι στενά συνδεδεμένα με τα CLB. Όλες οι είσοδοι και έξοδοι ενός CLB, πολυπλέκονται στο εσωτερικό του CB, το οποίο με τη σειρά του συνδέεται στον κεντρικό κόμβο που υλοποιεί ένα SB. Το κάθε SB, διαπερνάται από πολλά καλώδια που προέρχονται από ακόμα περισσότερα CB[10]. Η γενική δρομολόγηση των σημάτων, πραγματώνεται προγραμματίζοντας τα CB/SB, με τέτοιο τρόπο ώστε τα σήματα να διαδίδονται ακέραια μεταξύ των μπλοκ του FPGA, και έτσι ώστε να υλοποιούνται οι απαραίτητες λογικές πράξεις στο σύνολο του FPGA. Τέλος, αξίζει να σημειωθεί πως σε πολλά σημερινά FPGA, προκειμένου να αποφεύγονται φαινόμενα καθυστέρησης σημάτων, που εμφανίζονται όταν αυτά πρέπει να περάσουν μέσα από αρκετούς κόμβους πριν φτάσουν στον τελικό τους προορισμό, τα σήματα μπορούν να διαδοθούν και με μακρύτερα καλώδια *(Long Lines)* [7] που καταλήγουν σε απομακρυσμένα σημεία του FPGA. Έτσι υπάρχει η τοπική δικτύωση, που υλοποιείται με τα SB και CB, και η γενική δικτύωση, που υλοποιείται από τα Long Lines, που ξεκινούν από ένα SB, και καταλήγουν όχι σε γειτονικά, αλλά σε μακρινά SB. Η όλη αρχιτεκτονική, απεικονίζεται στο Σχήμα 1.2.I (τα Long Lines και τα Hard Blocks δεν περιλαμβάνονται).

Εκτός από τις τοπικές διασυνδέσεις μέσα στο FPGA, υπάρχουν και τα μπλοκ που υλοποιούν τη διεπαφή με το εξωτερικό του κυκλώματος. Ανάλογα το μέγεθός και τις δυνατότητές του, το FPGA μπορεί να μεταδώσει και να δεχθεί σήματα από πολλά διαφορετικά σημεία της πλακέτας στην οποία βαίνει, όπου τα επίπεδα τάσης και η γενικότερη μορφή αυτών των σημάτων, καλύπτουν ένα μεγάλο εύρος δυνατών πρωτοκόλλων και προτύπων. Τα *I/O Blocks*, είναι ακριβώς αυτά τα μπλοκ που μεταφέρουν τα σήματα από έξω, μέσα στο FPGA, και αντίστροφα. Αυτή η διαδικασία δεν είναι τόσο απλή όσο μπορεί να φανταστεί κανείς, καθώς τα διαφορετικά επίπεδα τάσεων, και η διακύμανση ως προς τη μορφολογία των παλμών που πρέπει να διαχειριστεί το FPGA στα pins του, δημιουργούν την ανάγκη για το σχεδιασμό πολλών διαφορετικών I/O Blocks, τα οποία θα πρέπει και αυτά να προγραμματίζονται, σύμφωνα με τα θέλω του χρήστη. Για παράδειγμα, τα 7-Series FPGA [14], χωρίζονται σε banks. Το κάθε bank, διαθέτει 50 I/O pins, με κοινή τάση τροφοδο-

[10]Στα FPGA της Xilinx® , το CB (που καλείται και Switch Matrix [7]) δέχεται και τα σήματα από τα περισσότερα hard blocks.



Σχήμα 1.2.I: Αριστερά: Σχηματική αναπαράσταση της αρχιτεκτονική τύπου island που εφαρμόζεται στα σημερινά FPGA. Το κάθε CLB συνδέεται με ένα CB στις εισόδους και εξόδους του. Το CB με τη σειρά του συνδέεται με δύο SB τα οποία υλοποιούν τη γενικότερη δικτύωση [7]. Πάνω Δεξιά: Η διασύνδεση του CLB με τα Switch Matrix (το αντίστοιχο SB της Xilinx®). Κάτω Δεξιά: Αναπαράσταση ενός Switch Box. Δέχεται τέσσερις εισόδους σε κάθε πλευρά του, και τις συνδέει μεταξύ τους ανάλογα με τον προγραμματισμό του.

σίας για τα pins. Στη συντριπτική τους πλειοψηφία, τα pins μπορούν να λάβουν και να μεταδώσουν τόσο διαφορικά σήματα (LVDS, RSDS, διαφορικό SSTL κ.ά.), όσο απλά μονοπολικά σήματα (LVCMOS, TTL, κ.ά.). Τα επίπεδα τάσεων που μπορεί να υποστηρίξει το κάθε pin (που είναι συνδεδεμένο με ένα I/O Block), εξαρτάται από το bank στο οποίο βρίσκεται. Ως προς αυτό το ζήτημα, υπάρχουν *High-Range (HR)* banks, άρα και pins, και *High-Performance (HP)* banks. Τα HR, υποστηρίζουν ένα μεγαλύτερο εύρος τάσεων (μέχρι $3.3\ V$) σε σχέση με τα HP που φτάνουν μέχρι τα $1.8\ V$. Τα HP interfaces όμως, είναι σχεδιασμένα με τέτοιο τρόπο ώστε να συμμετέχουν στη μετάδοση πληροφοριών με υψηλές ροές, (π.χ. Gigabit Ethernet) διατηρώντας ταυτόχρονα στο έπακρο την ακεραιότητα των σημάτων που διαχειρίζεται. Σε κάθε περίπτωση, ο σχεδιαστής μπορεί να διαχειριστεί το πρότυπο που θα ακολουθήσει ένα input ή output pin που θέλει να χρησιμοποιήσει στο σχεδιασμό του, με τη βοήθεια των *I/O buffers*. Μερικοί από αυτούς έχουν ήδη αναφερθεί προηγουμένως στο παρών Κεφάλαιο (clock buffers), αλλά φυσικά, υπάρχουν πολλοί άλλοι. Για παράδειγμα, υπάρχουν buffers που δέχονται διαφορικά σήματα από την πλακέτα

και τα μετατρέπουν σε μονοπολικά μέσα στο FPGA, και ο χρήστης μπορεί να ορίσει ελεύθερα το πρότυπο τάσεων που θα ακολουθήσει ο buffer αυτός.

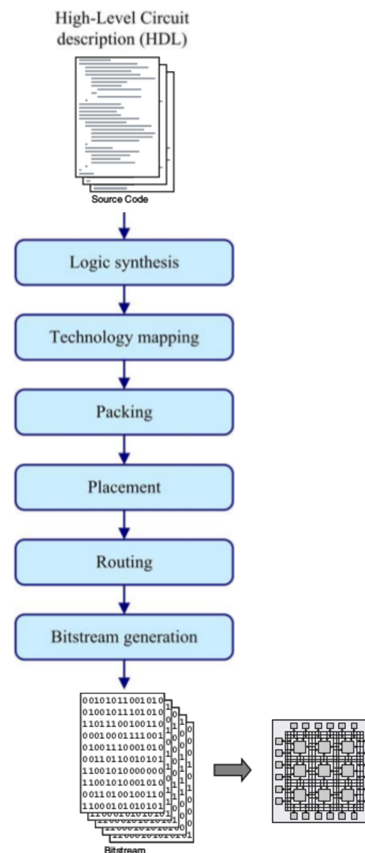
Έτσι λοιπόν, αρχίζει και ξεκαθαρίζει ο τρόπος με τον οποίο το FPGA τελεί τις λειτουργίες του. Ο σχεδιαστής, προγραμματίζει σε VHDL (ή Verilog) τον αλγόριθμο που θέλει να υλοποιήσει το FPGA, που μπορεί να είναι από μία απλή πύλη OR που θα δέχεται δύο σήματα από διακόπτες πάνω στην πλακέτα και θα διοχετεύει το αποτέλεσμα της πύλης σε μία δίοδο LED, ή ένα πιο πολύπλοκο σύστημα που θα λαμβάνει το σήμα ενός ρολογιού, θα συνθέτει μία πληθώρα συχνοτήτων μέσω ενός MMCM/PLL, και θα αποστέλλει τα περιεχόμενα μίας εσωτερικής FIFO, σειριακά εκτός της πλακέτας, με διαφορετικούς ρυθμούς, ακολουθώντας το πρότυπο LVDS. Όποιος και να είναι ο κώδικας που θα εφαρμοστεί στο FPGA, όταν αυτός πάρει την τελική του μορφή, ο Synthesizer θα αναλύσει τον εν λόγω κώδικα, και θα ενεργοποιήσει τα αντίστοιχα μπλοκ και τις διασυνδέσεις που θα μεταφέρουν τα σήματα από και προς τα διαφορετικά μπλοκ, προκειμένου να γίνουν οι επεξεργασίες των λογικών τιμών σωστά από τα LUT. Αυτή η διαδικασία, όπως μπορεί κανείς να φανταστεί δεν είναι καθόλου απλή. Για το λόγο αυτό, στη συνέχεια θα αφιερωθεί μία επιπλέον ενότητα που θα περιγράφει τον τρόπο που η πληροφορία ”ρέει” από τον κώδικα της VHDL, μέχρι τα κυκλώματα που βρίσκονται μέσα στο FPGA.

1.3 Η Ροή της Πληροφορίας

Η διαδικασία με την οποία μεταφράζεται ένας υψηλού-επιπέδου κώδικας περιγραφής hardware (HDL language), σε ενεργοποιημένα κυκλώματα και διασυνδέσεις μέσα στο FPGA, είναι αντικείμενο εκτενούς μελέτης εδώ και δεκαετίες [7, 8, 9, 10]. Σε αυτή την ενότητα, θα αναφερθεί το γενικό μοτίβο με το οποίο γίνεται η μετάφραση των κωδίκων μίας HDL, η οποία συνήθως είναι VHDL ή Verilog, σε LUT, FF, block RAM κ.λπ. Το διάγραμμα του σχήματος 1.3.I μπορεί να χρησιμοποιηθεί σαν σημείο αναφοράς για όσα θα ειπωθούν στη συνέχεια:

Αντιλαμβάνεται κανείς, ότι η γενική αποδοτικότητα του FPGA, εξαρτάται ευθέως από την ποιότητα με την οποία μεταφράζεται η πληροφορία που κρύβεται μέσα σε μία HDL, σε κυκλώματα του FPGA. Τα βήματα που ακολουθούνται είναι τα εξής [7, 10]:

- *Logic Synthesis*: Πρόκειται για τη διαδικασία κατά την οποία ο Synthesizer με-



Σχήμα 1.3.I: Διάγραμμα της ροής της πληροφορίας που διαμορφώνει τις λειτουργίες σε ένα FPGA [10, 7].

τατρέπει την HDL σε ένα σύνολο από πύλες OR, XOR, NAND κ.λπ. και πύλες flip-flop. Προφανώς, εκτός από το να κατασκευάζει αυτές τις πύλες, ο Synthesizer κάνει και στις αντίστοιχες διασυνδέσεις μεταξύ των πυλών.

- *Technology Mapping*: Μετά τη σύνθεση των συμβατικών πυλών, ο Synthesizer συλλέγει αυτές τις πύλες, και τις μετατρέπει στη φυσική μορφή των πυλών, όπως αυτές υπάρχουν στο FPGA. Δηλαδή, στην περίπτωση των Xilinx® 7-Series FPGA [14], οι λογικές πύλες γίνονται LUT, και τα σήματα εισόδου/εξόδου στον κώδικα HDL μετατρέπονται σε I/O blocks configuration.

- *Packing*: Σε αυτή τη φάση, το σύνολο από LUT και πύλες flip-flop που συνδέονται άμεσα μεταξύ τους, πακετάρονται στα CLB. Το τελικό αποτέλεσμα είναι πλέον CLB και I/O blocks.

- *Placement*: Σε αυτό το στάδιο, ο Synthesizer (ή και *Placer*) τοποθετεί μέσα στο FPGA τα διαφορετικά CLB και I/O block που έχουν προκύψει από το προηγούμενο στάδιο της διαδικασίας.

- *Routing*: Αφού γίνει η τοποθέτηση των CLB, προγραμματίζονται τα SB/CB προκειμένου να διασυνδεθούν σωστά μεταξύ τους τα CLB, ώστε το ένα να μεταφέρει το αποτέλεσμα της επεξεργασίας που έλαβε χώρα από τα LUT του, στο άλλο. Έτσι όλα τα CLB μαζί, κάνουν συλλογικούς υπολογισμούς περίπλοκων λογικών συναρτήσεων. Αυτές οι διαδικασίες τελούνται από τον *Router*. Τονίζεται ότι κάθε καλώδιο του εσωτερικού δικτύου διασυνδέσεων του FPGA, θα πρέπει στο τέλος να μεταφέρει ένα και μόνο ένα σήμα.

- *Bitstream Generation*: Όταν πλέον η τελική αρχιτεκτονική του FPGA έχει καθοριστεί "στα χαρτιά", μένει να περάσει η πληροφορία μέσα στο FPGA, προκειμένου να ενεργοποιηθούν τα αντίστοιχα LUT, hard blocks, και οι διασυνδέσεις μεταξύ τους. Αυτό γίνεται κωδικοποιώντας ένα σειριακό bitstream το οποίο εισέρχεται στο FPGA, προγραμματίζοντας τις μνήμες SRAM που βρίσκονται σε αυτό. Περισσότερα παρακάτω.

Ως επί το πλείστον, τα FPGA πλέον, είναι *SRAM-Based*. Δηλαδή, το τελικό στάδιο του προγραμματισμού τους, γίνεται από μνήμες SRAM[11] [7, 9, 10]. Κάθε προγραμματιζόμενο μέλος του FPGA, ελέγχεται από ένα κελί μνήμης SRAM. Πιο συγκεκριμένα, ένα LUT, το οποίο υπενθυμίζεται ότι έχει μία τιμή INIT με μήκος 64 bit, θα προγραμματίζεται από 64 μνήμες SRAM, η κάθε μία εκ των οποίων θα ελέγχει και μία θέση στο πεδίο INIT του LUT. Βέβαια, υπάρχουν και άλλοι τρόποι διαμόρφωσης της λειτουργίας ενός CLB ή slice, που έχουν ήδη αναφερθεί προηγουμένως (μετατροπή σε μνήμη RAM). Όλα αυτά ελέγχονται από μνήμες SRAM. Αντίστοιχα, η ενεργοποίηση και οι μικροαλλαγές στις λειτουργίες των hard blocks γίνονται επίσης από τις αντίστοιχες SRAM. Τέλος, και οι διασυνδέσεις με τη σειρά τους, ελέγχονται από μνήμες SRAM. Ανατρέχοντας στο Σχήμα 1.2.I, μπορεί κανείς να δει τη δομή του Switch Box, το οποίο μπορεί να υλοποιήσει σε εκείνη την περίπτωση δεκαέξι διαφορετικές διασυνδέσεις στους κόμβους. Υπάρχει λοιπόν μία SRAM για κάθε δυνατή διασύνδεση, για κάθε κόμβο. Όταν η διαδικασία του Routing ενεργοποιήσει κάποια διασύνδεση, τότε η αντίστοιχη SRAM στο αντίστοιχο SB, θα πάρει την τιμή 1 για να

[11]Static Random Access Memory.

κάνει τη σύνδεση.

Η απλότητα των μνημών SRAM, τις έχει καταστήσει ως την καθιερωμένη πρακτική όσο αναφορά τον προγραμματισμό των FPGA. Ο τρόπος κατασκευής τους (τεχνολογία CMOS) δε, επιτρέπει στην εταιρεία που σχεδιάζει το FPGA να εκμεταλλευθεί τις τελευταίες εξελίξεις στον τομέα αυτό [9], ο οποίος είναι εξαιρετικά δημοφιλής. Από την άλλη βέβαια, μία μνήμη SRAM χρειάζεται πέντε ή έξι τρανζίστορ για να υλοποιηθεί, ενώ σε περίπτωση που η τροφοδοσία σταματήσει, η μνήμη της σβήνεται. Αυτό από τη μία είναι καλό, καθώς αυτό ακριβώς είναι που καθιστά τα σημερινά FPGA άπειρες φορές επαναπρογραμματιζόμενα, όμως από την άλλη, προκειμένου να αποθηκευτεί το τελικό firmware στο FPGA, πρέπει να χρησιμοποιηθούν εξωτερικές μνήμες flash, οι οποίες θα αποθηκεύουν το bitstream, και θα προγραμματίζουν τις SRAM όταν το FPGA μπει σε λειτουργία.

Έτσι γίνεται πλέον σαφές το πως προγραμματίζεται ένα FPGA. Αρχικά ο προγραμματιστής συντάσσει έναν κώδικα HDL, ο οποίος μεταφράζεται πρώτα σε πύλες, μετά σε LUT και CLB, και τελικά και σε διασυνδέσεις μεταξύ αυτών των μπλοκ. Προκειμένου να υλοποιηθούν όλα αυτά, κατασκευάζεται μία ροή σειριακών ψηφιακών σημάτων (bitstream), η οποία περνάει από τον υπολογιστή στο FPGA, και προγραμματίζει όλες τις μνήμες SRAM στο FPGA, όπου το κάθε κελί μνήμης, διαχειρίζεται και ένα μικρό κομμάτι του FPGA, είτε αυτό είναι κάποιος πολυπλέκτης ή θέση μνήμης ενός LUT, είτε είναι κάποια πιθανή διασύνδεση στο δίκτυο μεταβίβασης των σημάτων μέσα στο FPGA.

Σύνοψη

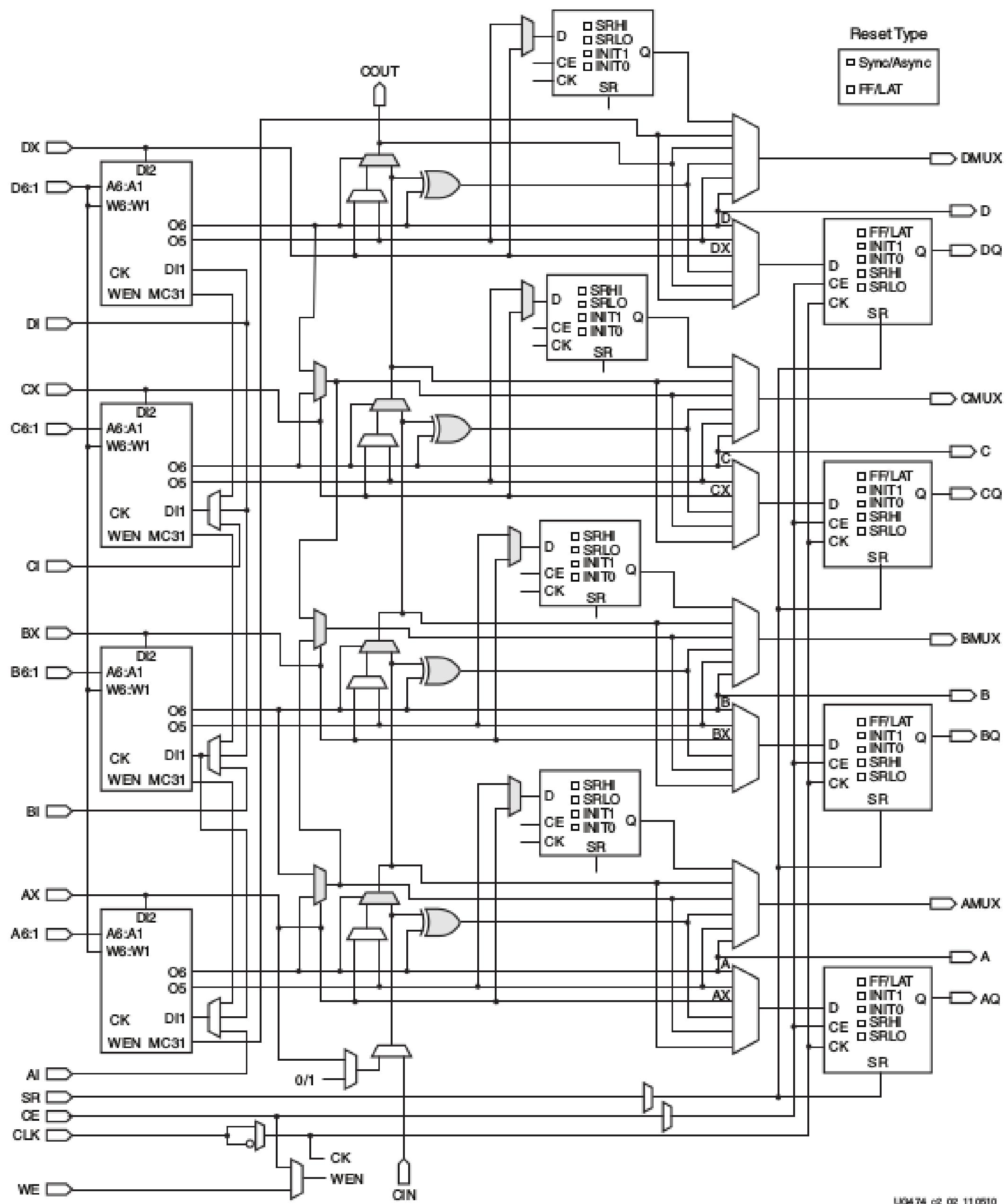
Σε αυτό το Κεφάλαιο, ξεκαθάρισε ο τρόπος λειτουργίας των FPGA. Μελετήθηκε η δομή της βασικής μονάδας επεξεργασίας του FPGA, που είναι το LUT. Τα LUT, μαζί με πολυπλέκτες και πύλες Flip-Flop, οργανώνονται σε Slice και CLB, τα οποία απαρτίζουν το κυριότερο μέρος των λογικών μπλοκ ενός FPGA. Εκτός από αυτά τα μπλοκ γενικής χρήσης όμως, υπάρχουν και εξειδικευμένα μπλοκ, τα οποία τελούν συγκεκριμένες λειτουργίες. Τέτοια μπλοκ μπορεί να εκτελούν λειτουργίες μνήμης, ή χρονισμού. Επίσης, έγινε αναφορά και στον τρόπο με τον οποίο τα λογικά μπλοκ του FPGA συνδέονται μεταξύ τους, μέσα στο τσιπ. Τέλος, αποσαφηνίστηκε και η διαδικασία μετάφρασης του κώδικα, σε υλοποίηση του κυκλώματος μέσα στο FPGA. Πριν κλείσει λοιπόν το παρών Κεφάλαιο, θα παρατεθούν μερικά νούμερα για το

Xilinx® Artix-7 FPGA xc7a200t-2fbg484, που είναι και το FPGA πάνω στο οποίο έχει αναπτυχθεί firmware για τις εφαρμογές της παρούσας εργασίας.

Device	Slices(1)	SLICEL	SLICEM	6-input LUTs	Distributed RAM (Kb)	Shift Register (Kb)	Flip-Flops
7A200T	33,650	22,100	11,550	134,600	2,888	1,444	269,200

Device	Logic Cells	Configurable Logic Blocks (CLBs)		DSP48E1 Slices(2)	Block RAM Blocks(3)			CMTs(4)	PCIe(5)	GTPs	XADC Blocks	Total I/O Banks(6)	Max User I/O(7)
		Slices(1)	Max Distributed RAM (Kb)		18 Kb	36 Kb	Max (Kb)						
XC7A200T	215,360	33,650	2,888	740	730	365	13,140	10	1	16	1	10	500

Σχήμα 1.3.II: Πίνακας με τις προδιαγραφές του Xilinx® Artix-7 FPGA xc7a200t-2fbg484 [14].



Σχήμα 1.3.III: Ένα Xilinx® 7-Series FPGA slice. Στην αριστερή στήλη διακρίνονται τα τέσσερα LUT, ενώ στα δεξιά οι τέσσερις flip-flop (που μπορούν να προγραμματιστούν και ως μνήμες latch). Στο κέντρο διακρίνεται μία σειρά από πύλες και πολυπλέκτες που υλοποιούν το κύκλωμα προσθαφαιρέσεων. Επιπλέον πολυπλέκτες επιλέγουν τις τελικές εξόδους του slice. Επίσης φαίνονται και pins για την είσοδο σημάτων χρονισμού (CLK), σημάτων εισόδου στα LUT (A6:1) και τα pins της ALU (CIN/COUT) [14].

2

Εισαγωγή στη VHDL

Τα αρχικά *VHDL*, σημαίνουν *VHSIC*[1] *Hardware Description Language*. Η VHDL, είναι μία γλώσσα περιγραφής αρχιτεκτονικής κυκλωμάτων σε υψηλό επίπεδο, η οποία χρησιμοποιείται κυρίως για τον προγραμματισμό FPGA, και για τον καθορισμό της γενικής δομής ενός ASIC, πριν αυτό μπει στη γραμμή παραγωγής [4]. Η VHDL υποστηρίζει τόσο τη μετάφραση της γλώσσας αυτής σε πραγματικά κυκλώματα μέσα στο FPGA, όσο και την προσομοίωση της απόκρισης του κυκλώματος, μέσω *simulators*. Η διαδικασία του *Synthesis* και *Implementation*, οι οποίες έχουν περιγραφεί στην ενότητα 1.3, γίνονται μέσω του Vivado® Design Suite, που είναι το IDE της Xilinx® . Επίσης, το simulation του κώδικα γίνεται και αυτό μέσω του ίδιου περιβάλλοντος. Σε αυτό το Κεφάλαιο, θα γίνει μία σύντομη εισαγωγή στη γλώσσα VHDL σε μορφή παραδειγμάτων, προκειμένου ο αναγνώστης να είναι σε θέση να κατανοήσει τους κώδικες που θα συναντήσει παρακάτω.

2.1 Hello World!

Στις περισσότερες γλώσσες προγραμματισμού, το πρώτο πράγμα που μαθαίνει κανείς, είναι να γράφει ένα πρόγραμμα που να τυπώνει στο χρήστη το κείμενο *Hello World!*. Επειδή όμως η φιλοσοφία του firmware σε σχέση με εκείνη του λογισμικού είναι εντελώς διαφορετική, το πρώτο παράδειγμα, το πρώτο *Component*, που θα εξεταστεί, θα είναι μία απλή πύλη AND.

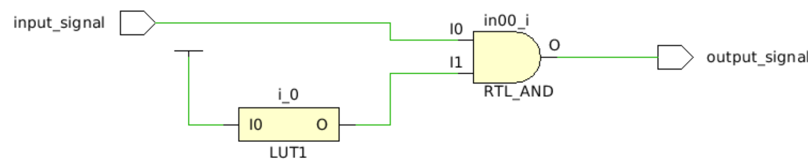
[1] Very High Speed Integrated Circuit.

VHDL Code 2.1: example_top

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

entity example_top is

port(
    -- component i/o
    input_signal  : in  std_logic;
    output_signal : out std_logic
);
end example_top;

architecture RTL of example_top is

    -- internal signal declaration
    signal internal_signal : std_logic := '0';

begin

    -- component logic
    output_signal   <= input_signal and internal_signal;
    internal_signal <= '1';

end RTL;
```

Ο αναγνώστης θα πρέπει από εδώ και στο εξής να αντιλαμβάνεται ότι κάθε κώδικας που έχει την παραπάνω μορφή, που είναι ένας ολοκληρωμένος κώδικας VHDL, πλήρως synthesizable, είναι ουσιαστικά ένα *component*. Ένα component βολεύει πολλές φορές να αναπαρίσταται σαν "μαύρο κουτί", με εισόδους και εξόδους. Ο παραπάνω κώδικας, έχει μία είσοδο και μία έξοδο, που δηλώνονται μεταξύ των γραμμών $4-11$. Πρόκειται για τα *input_signal* και *output_signal*. Το *std_logic* υποδεικνύει ότι και τα δύο είναι σήματα ενός bit, δηλαδή το κάθε ένα αντιπροσωπεύει φυσικά έναν δίαυλο μέσα στο τσιπ. Επίσης, στη γραμμή 7, φαίνεται ένα σχόλιο του κώδικα. Για να δηλώσει κανείς σχόλια στη VHDL, επισυνάπτει στην αρχή της γραμμής τους χαρακτήρες "–". Από τη γραμμή 13 και έπειτα ξεκινάει η αρχιτεκτονική του component, δηλαδή η λογική που "κρύβει" μέσα του. Πριν το *begin* στη γραμμή 18, ο χρήστης δύναται να δηλώσει εσωτερικά σήματα που δεν έχουν σχέση με τη διεπαφή του

component με άλλα κομμάτια του firmware (η οποία διεπαφή έχει δηλωθεί ήδη μεταξύ των γραμμών $4 - 11$, μέσα στο *port();*). Μπορεί να τα φανταστεί κανείς σαν εσωτερικές/κρυφές μεταβλητές. Ένα τέτοιο σήμα είναι το *internal_signal*. Προσέξτε ότι το εσωτερικό σήμα έχει αρχικοποιηθεί στο λογικό μηδέν (*:='0';*). Μετά το *begin*, ξεκινάει η κύρια λογική του component. Στη γραμμή 21, είναι φανερό ότι το σήμα εξόδου είναι το παράγωγο της λογικής πράξης *AND*, μεταξύ του σήματος εισόδου και του εσωτερικού σήματος. Το εσωτερικό σήμα όμως αν και έχει αρχικοποιηθεί στο λογικό μηδέν, είναι τελικά ίσο με το λογικό ένα, όπως είναι φανερό στη γραμμή 22. Είναι πολύ σημαντικό να γίνει αντιληπτό ότι εν προκειμένω, η σειρά δήλωσης των εντολών στις γραμμές 21, 22 *δεν έχει απολύτως καμία σημασία*. Αυτό συμβαίνει γιατί το συγκεκριμένο κύκλωμα είναι *συνδυαστικής λογικής*, δηλαδή δεν εξαρτάται από το χρόνο. Μετά τη σύνθεση, μπορεί κανείς να δει το *RTL Schematic*[2] του component, μέσω του Vivado® , όπου είναι φανερή η αρχιτεκτονική του component:



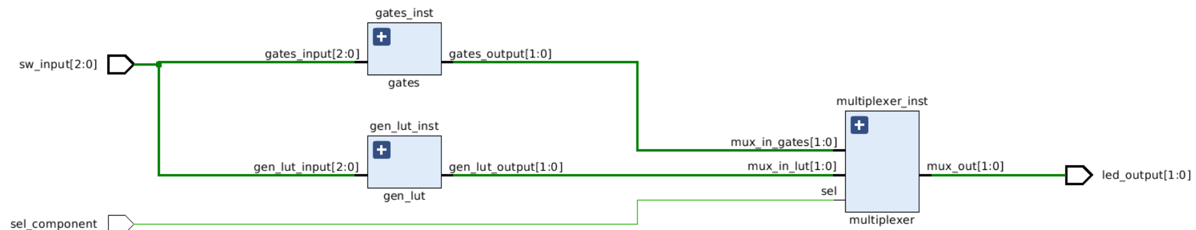
Σχήμα 2.1.I: RTL Schematic του κώδικα 2.1

2.2 Κυκλώματα Συνδυαστικής Λογικής

Σε αυτή την ενότητα, θα εξεταστούν βασικά κυκλώματα συνδυαστικής *(combinatorial)* λογικής. Πρόκειται για τα πιο βασικά κυκλώματα σε VHDL, τα οποία δεν έχουν εξάρτηση από το χρόνο, καθώς τα σήματα που παράγουν στις εξόδους τους εξαρτώνται μόνο από την τρέχουσα κατάσταση των σημάτων εισόδου, και όχι από το ιστορικό ή την αλληλουχία μεταβάσεών τους. Ο κώδικας 2.1 αντιπροσώπευε ένα τέτοιο κύκλωμα. Τέτοιοι κώδικες μέσα στο FPGA υλοποιούνται αποκλειστικά σε LUTs και στους dedicated multiplexers των Slices. Το firmware που θα περιγραφεί εδώ, θα περιέχει ένα top_level, και τρία subcomponents: μία συστοιχία από πύλες, έναν πολυπλέκτη, και ένα γενικευμένο look-up table, το οποίο είναι ουσιαστικά μία

[2]RTL: Register Transfer Level. Πρόκειται για μία δυνατότητα του Vivado® να απεικονίζει το design σε μορφή συμβατικών λογικών πυλών.

μνήμη *ROM*[3]. Το top_level του design θα περιέχει τα τρία subcomponents που προαναφέρθηκαν, ενώ οι διεπαφές του (δηλωμένες στο port();) θα είναι ουσιαστικά τα *pins* του FPGA. Ακολουθεί το RTL Schematic του top_level ώστε να αποκτηθεί μία γενική άποψη της αρχιτεκτονικής του design:



Σχήμα 2.2.I: RTL Schematic του κώδικα 2.2

Σε αυτό το παράδειγμα, υπάρχουν *τέσσερα* σήματα εισόδου. Το *sw_input [2:0]* και το *sel_component*. Το sw_input έχει κωδικοποιηθεί ως ένα *bus* ή *bundle*. Είναι δηλαδή ένα σύνολο από σήματα ενός bit (εν προκειμένω, τρία single-bit signals). Αντίστοιχα, το *led_output* είναι μία έξοδος που έχει κωδικοποιηθεί ως ένα σήμα, το οποίο όμως αποτελείται από δύο επιμέρους single-bit signals. Από το RTL Schematic, είναι φανερό ότι το προκείμενο design, δέχεται τα σήματα εισόδου (sw_input), και τα προωθεί σε δύο διαφορετικά submodules. Η εκάστοτε έξοδος που προκύπτει από την επεξεργασία των σημάτων αυτών, οδηγούνται σε έναν πολυπλέκτη, ο οποίος θα προωθεί στην έξοδο *led_output*, κάποιο από τις δύο εισόδους, ανάλογα την τιμή του *sel_component*.

Είναι φανερό ότι το παραπάνω design είναι *αρθρωτό (modular)*. Αυτός ο τρόπος σχεδιασμού firmware χρησιμοποιείται κατά κόρον, καθώς ξεκαθαρίζει τον τρόπο που ρέει η πληροφορία, ενώ επίσης κάνει και την υποστήριξη του κώδικα ευκολότερη, αφού είναι φανερό ποιο κομμάτι, ποιο component δηλαδή, κάνει τι. Ακολουθεί ο κώδικας που αντιστοιχεί στο σχήμα 2.2.I:

VHDL Code 2.2: comb_top

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity comb_top is
5
```

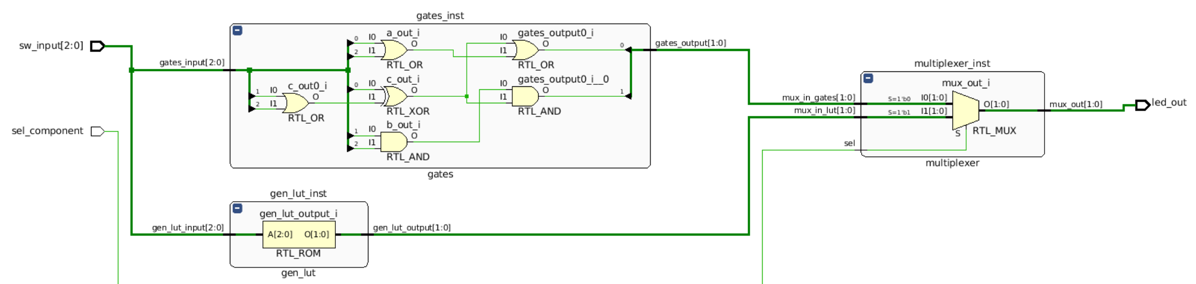
[3]Read-Only Memory.

```
6  port(
7      -- FPGA i/o
8      sw_input          : in   std_logic_vector(2 downto 0);
9      sel_component     : in   std_logic;
10     led_output        : out  std_logic_vector(1 downto 0)
11 );
12 end comb_top;
13
14 architecture RTL of comb_top is
15
16     -- submodules declaration
17     component gates
18     port(
19         gates_input       : in   std_logic_vector(2 downto 0);
20         gates_output      : out  std_logic_vector(1 downto 0)
21     );
22     end component;
23
24     component gen_lut
25     port(
26         gen_lut_input     : in   std_logic_vector(2 downto 0);
27         gen_lut_output    : out  std_logic_vector(1 downto 0)
28     );
29     end component;
30
31     component multiplexer
32     port(
33         mux_in_gates      : in   std_logic_vector(1 downto 0);
34         mux_in_lut        : in   std_logic_vector(1 downto 0);
35         sel               : in   std_logic;
36         mux_out           : out  std_logic_vector(1 downto 0)
37     );
38     end component;
39
40     -- internal signal declaration
41     signal gates_output       : std_logic_vector(1 downto 0) := "00";
42     signal gen_lut_output     : std_logic_vector(1 downto 0) := "00";
43
44 begin
45
46 -- gates module instantiation
```

```
47 gates_inst: gates
48     port map(
49         gates_input   => sw_input,
50         gates_output => gates_output
51     );
52
53 -- general LUT instantiation
54 gen_lut_inst: gen_lut
55     port map(
56         gen_lut_input    => sw_input,
57         gen_lut_output   => gen_lut_output
58     );
59
60 -- multiplexer instantiation
61 multiplexer_inst: multiplexer
62     port map(
63         mux_in_gates     => gates_output,
64         mux_in_lut       => gen_lut_output,
65         sel              => sel_component,
66         mux_out          => led_output
67     );
68
69 end RTL;
```

Στις γραμμές $6-12$ δηλώνεται η διεπαφή του comb_top, η οποία έχει περιγραφεί και παραπάνω. Είναι ουσιαστικά τα pins του FPGA. Αξίζει να προσέξει κανείς τον τρόπο δήλωσης του bus *(std_logic_vector(N downto 0))*, που δημιουργεί ένα multi-bit signal το οποίο αποτελείται από $N+1$ δυφία. Μέσα στην αρχιτεκτονική του firmware, και πριν το *begin* (γραμμές $17-38$), είναι που δηλώνονται όλα τα subcomponents που εμπεριέχονται μέσα στο comb_top και θα χρησιμοποιηθούν παρακάτω *(component declarations)*. Η σύμβαση υποδεικνύει ότι πρώτα κάποιος δηλώνει τα subcomponents, και μετά τα εσωτερικά σήματα του design (γραμμές $41-42$). Μετά από το *begin*, μπορεί κανείς να βρει τα *component instantiations* (γραμμές $47-67$). Προκειμένου να δημιουργηθεί ένα component instance, αρχικά η σύνταξη είναι *label: component name* (γραμμές $47, 54, 61$), και στη συνέχεια δηλώνονται αυτούσια τα I/O ports του εκάστοτε component με τις συνδέσεις τους, που μπορεί να είναι είτε με εσωτερικά σήματα, είτε με σήματα του port στο top_level. Ο πολυπλέκτης για παράδειγμα, έχει δύο bus inputs: *mux_in _gates* και *mux_in _lut*, τα οποία συνδέονται με τα εσωτερικά

σήματα *gates_output* και *gen_lut_output* (γραμμές 63, 64), που οδηγούν σε άλλα εσωτερικά components (γραμμές 50, 57). Επίσης, το σήμα επιλογής *sel* του πολυπλέκτη συνδέεται απευθείας με τη διεπαφή του top_level (γραμμές 9, 65), και το σήμα εξόδου του πολυπλέκτη κατευθείαν με την έξοδο *led_output* (γραμμές 10, 66). Η παραπάνω συνδεσμολογία απεικονίζεται και στο σχήμα 2.2.I, το οποίο όμως μπορεί να δώσει περισσότερες λεπτομέρειες για το κύκλωμα σαν σύνολο, επεκτείνοντας το κάθε subcomponent (βλ. σχ. 2.2.II).



Σχήμα 2.2.II: RTL Schematic του κώδικα 2.2, με τις λεπτομέρειες των subcomponets. Μπορεί κανείς να δει τη συνδεσμολογία των λογικών πυλών του *gates*, όσο και το γεγονός ότι το γενικευμένο LUT του *gen_lut* απεικονίζεται ως μνήμη ROM.

Ο κώδικας των τριών subcomponents παρατίθεται στο αντίστοιχο παράρτημα (βλ. Γ΄). Εδώ, θα παρατεθεί ένα συγκεκριμένο σημείο του κώδικα του πολυπλέκτη, το οποίο εισάγει την έννοια της *μεθόδου (process)*:

VHDL Code 2.3: multiplexer sample-combinatorial process

```
1 -- multiplexing process
2 mux_proc: process(mux_in_gates, mux_in_lut, sel)
3 begin
4     case sel is
5     when '0'    => mux_out <= mux_in_gates;
6     when '1'    => mux_out <= mux_in_lut;
7     when others => mux_out <= (others => '0');
8     end case;
9 end process;
```

Εν γένει, οι μέθοδοι στη VHDL χωρίζονται σε δύο μεγάλες κατηγορίες: σε *συνδυαστικές (combinatorial)*, και *ακολουθιακές (sequential)*. Το κριτήριο διαχωρισμού τους

έγκειται στο αν η μέθοδος είναι *χρονισμένη (clocked)* ή όχι. Η παραπάνω process, όντας μη-χρονισμένη, είναι συνδυαστική. Οι μέθοδοι χρησιμοποιούνται κατά κόρον στη VHDL, προκειμένου να υλοποιήσουν πάσης φύσεως λογική/αλγορίθμους, και κάθε firmware developer οφείλει να κατανοεί τόσο τη σύνταξή τους, όσο και τον τρόπο που λειτουργούν.

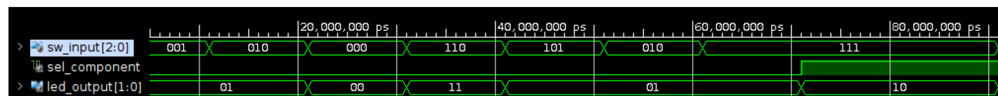
Η μέθοδος, όντας ουσιαστικά μία διαδικασία επεξεργασίας σημάτων, μπορεί να θεωρηθεί και αυτή σαν ένα μικρό subcomponent το οποίο δέχεται σήματα στις εισόδους του, και ανάλογα αλλάζει τα σήματα εξόδου. Ως προς τη σύνταξη του κώδικα, καλό είναι κατ' αρχάς η κάθε μέθοδος να διαθέτει ένα περιγραφικό *label*, που εν προκειμένω είναι το *mux_proc*. Η σύνταξη της πρώτης γραμμής μίας process είναι εν γένει: *label*:**process***(sensitivity list)* (γραμμή 2). Η *sensitivity list* στις συνδυαστικές μεθόδους, περιέχει γενικά *όλα τα σήματα εισόδου της process*. Η ύπαρξη ενός σήματος εισόδου μέσα στη λίστα, υποδεικνύει ότι όταν αυτό το σήμα αλλάξει κατάσταση, θα αλλάξει κατάσταση και το σήμα (ή τα σήματα) εξόδου της μεθόδου που εξαρτάται από αυτό. Ανάμεσα από τα *begin* και *end process* (γραμμές $3-9$) μπορεί κανείς να δει τη λογική του πολυπλέκτη, κωδικοποιημένη με *case statement* (γραμμές $4-8$). Σε αυτό το κομμάτι του κώδικα επιλέγεται το σήμα εξόδου *(mux_out)* της μεθόδου (που συμπτωματικά ταυτίζεται με το σήμα εξόδου του component), ανάλογα την κατάσταση του σήματος επιλογής *(sel)*. Ένα case statement μπορεί να εξαρτάται μόνο από ένα σήμα, που εν προκειμένω ορίζεται στη γραμμή 4. Όταν το sel είναι μηδέν, δρομολογείται στην έξοδο η είσοδος από τις πύλες (γραμμή 5), ενώ όταν το sel είναι ψηλά, δρομολογείται η είσοδος που προέρχεται από τη μνήμη ROM. Για λόγους πληρότητας, πριν το τέλος του case statement (γραμμή 8), βρίσκεται το *when others* statement, το οποίο δίνει την τιμή *"00"* στην έξοδο[4] για όλες τις άλλες καταστάσεις του *sel*[5]. Αντίστοιχη σύνταξη θα συναντήσει κανείς και στον τρόπο κωδικοποίησης της μνήμης ROM, όπου ανάλογα την κατάσταση του σήματος *gen_lut_input* αλλάζει και το σήμα εξόδου *gen_lut_output* (βλ. Παράρτημα Γ').

Το firmware που έχει περιγραφεί, μπορεί να προσομοιωθεί στο Vivado® . Κατά την

[4]Η εντολή bus_signal <= (others => '0'); αναθέτει σε ένα multi-bit signal τη τιμή μηδέν σε όλα τα επιμέρους σήματα απο το οποίο αποτελείται.

[5]Ένα σήμα *std_logic* δεν παίρνει μόνο τις λογικές τιμές μηδέν και ένα, αν και αυτές οι δύο είναι που χρησιμοποιούνται περισσότερο, όταν δηλώνει κανείς το πρότυπο *IEEE 1664* στην αρχή ενός VHDL κώδικα. Η VHDL υποστηρίζει για παράδειγμα και την τιμή 'Z' (υψηλή εμπέδηση), η οποία χρησιμοποιείται στη λογική τριών καταστάσεων, δηλαδή σε *tristate/bidirectional buffers*[17].

προσομοίωση, ο χρήστης μπορεί μέσω του γραφικού περιβάλλοντος του προαναφερθέντος προγράμματος, να αλλάζει δυναμικά τα σήματα εισόδου του design, προκειμένου να δει το αποτέλεσμα στην έξοδο. Μπορεί επίσης να αλλάξει και την κατάσταση εσωτερικών σημάτων. Εν γένει, όταν δημιουργείται ένα νέο component, *συνιστάται έντονα* να προσομοιώνεται, προκειμένου να επαληθευθεί η ορθή συμπεριφορά και λειτουργικότητά του, πριν την εφαρμογή του στο πραγματικό FPGA. Το αποτέλεσμα της προσομοίωσης θα είναι ένα *Timing Diagram*:

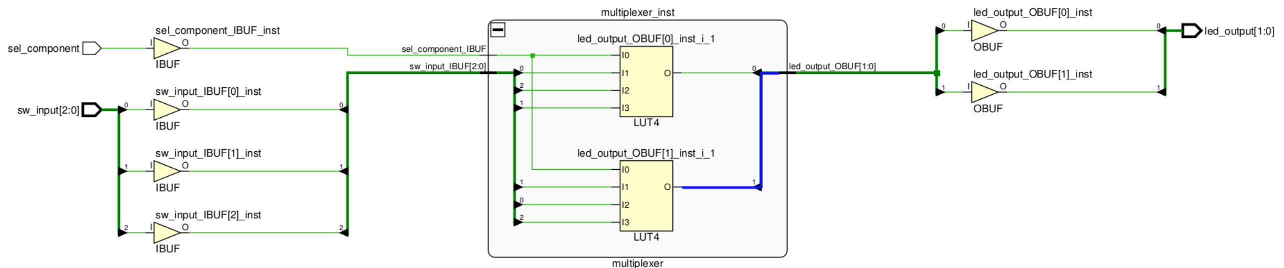


Σχήμα 2.2.III: Προσομοίωση του κώδικα 2.2. Ο χρήστης μπορεί να αλλάζει δυναμικά την τιμή των *sw_input* και *sel*, και να βλέπει άμεσα το αποτέλεσμα στο σήμα εξόδου *led_output*.

Ο αναγνώστης προτρέπεται να κατασκευάσει τον πίνακα αληθείας για τα subcomponents των πυλών και της μνήμης ROM. Αυτοί οι δύο πίνακες μπορούν σε συνδυασμό με το σήμα επιλογής *sel*, να κατασκευάσουν ένα γενικευμένο πίνακα αληθείας για όλο το firmware, και ίσως έναν πίνακα *Karnaugh*, ο οποίος μπορεί να οδηγεί σε ένα απλοποιημένο design. Συγχωνεύοντας όλα τα σήματα εισόδου (*sw_input* και *sel*), μπορεί κανείς να συγχωνεύσει αντίστοιχα και τα τρία subcomponents σε *ένα*, το οποίο απλά θα οδηγεί το σήμα εξόδου *led_output* ανάλογα την κατάσταση του συνεπτυγμένου σήματος εισόδου. Έτσι θα κωδικοποιηθεί το firmware σε μία μόνο μνήμη ROM, και κατά την εφαρμογή του design μέσα στο FPGA, το μόνο που θα γίνει είναι να υλοποιηθούν μερικά LUT με τις μεταξύ τους διασυνδέσεις ώστε να κατασκευάσουν συλλογικά αυτή τη μνήμη[6]. Με αυτή τη λογική ουσιαστικά λειτουργεί το εργαλείο της εκάστοτε εταιρείας που έχει προμηθεύσει το FPGA (για τη Xilinx® είναι το Vivado®), προκειμένου να μεταφράσει τον κώδικα του χρήστη σε FPGA components και εσωτερικό routing *(Synthesis/Implementation)*. Εκτός λοιπόν από το *RTL Schematic* που απεικονίζει το κάθε design σε συμβατικές πύλες, υπάρχει και το *Synthesized Schematic*, το οποίο προκύπτει μετά τη λογική σύνθεση του design, δηλαδή μετά την αντιστοίχηση της λογικής του firmware σε πραγματικές υποδομές που

[6]Αν ο αναγνώστης θελήσει να δημιουργήσει τον γενικευμένο πίνακα αληθείας όπως περιγράφτηκε προηγουμένως, η σύνταξη για το ισοδύναμο component μπορεί να βρεθεί στο αντίστοιχο Παράρτημα (βλ. κώδικα Γ΄.4).

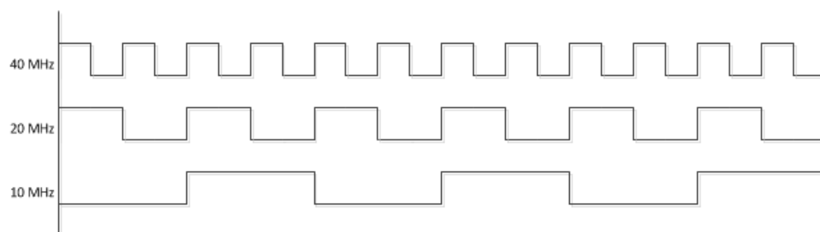
βρίσκονται μέσα στο FPGA. Το σχεδιάγραμμα του εν λόγω firmware για παράδειγμα μπορεί να το βρει κανείς στο σχ. 2.2.IV.



Σχήμα 2.2.IV: Synthesized Schematic του κώδικα 2.2. Προσέξτε ότι όλη η λογική συνεπτύχθη από το Vivado® σε δύο LUT. Επίσης, μπορεί κανείς να δει και τους I/O buffers, που το Vivado® τοποθετεί αυτόματα στα I/O ports του top_level κάθε design. Περισσότερες πληροφορίες για αυτά το component μπορεί να βρει κανείς στο [17].

2.3 Κυκλώματα Σύγχρονης Ακολουθιακής Λογικής

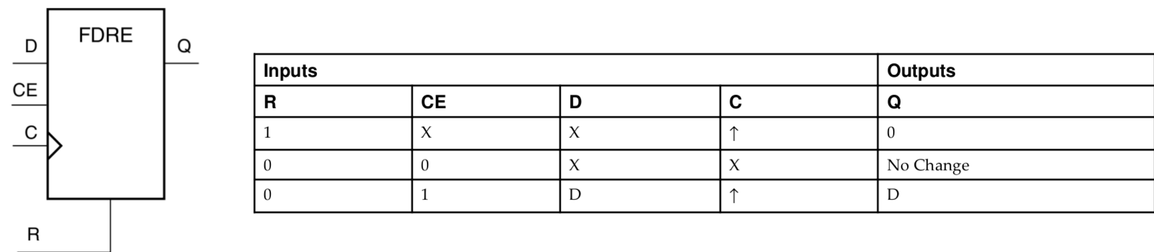
Κανένα ψηφιακό κύκλωμα υψηλών επιδόσεων δεν θα μπορούσε να λειτουργήσει χωρίς ένα σήμα χρονισμού, το οποίο υποδεικνύει ουσιαστικά τη συχνότητα με την οποία εκτελούνται διάφορες εντολές μέσα στη λογική, συντονίζοντας έτσι ολόκληρο το κύκλωμα σε έναν κοινό ρυθμό. Αυτό το σήμα καλείται *ρολόι*, *(clock)*. Σε αντίθεση με τα κυκλώματα συνδυαστικής λογικής, τα σύγχρονα design παράγουν σήματα τα οποία δεν εξαρτώνται μόνο από την τρέχουσα κατάσταση τον σημάτων εισόδου, αλλά και από το ιστορικό και την αλληλουχία μεταβάσεών τους.



Σχήμα 2.3.I: Σχηματική αναπαράσταση παλμών ρολογιού σε διάφορες συχνότητες.

2.3.1 FDRE Flip-Flop

Το βασικό συστατικό των κυκλωμάτων ακολουθιακής λογικής *(sequential logic)*, είναι η πύλη *flip-flop* (ή και *register*[7], που μεταφράζεται ως *καταχωρητής*), η οποία έχει ήδη εξετασθεί σε προηγούμενη ενότητα (βλ. 1.1.1). Όταν ο σχεδιαστής υλοποιεί εμμέσως τέτοιες πύλες στον κώδικά του, συνήθως αυτές αντιστοιχούν σε πύλες *FDRE*. Αυτή η πύλη περιγράφεται στο σχήμα 2.3.II.



Inputs				Outputs
R	CE	D	C	Q
1	X	X	↑	0
0	0	X	X	No Change
0	1	D	↑	D

Σχήμα 2.3.II: Πύλη flip-flop τύπου FDRE με σύγχρονο *Reset* και *Clock-Enable*, μαζί με τον λογικό της πίνακα [17].

Κάθε slice στα Xilinx® 7-Series FPGAs διαθέτει από οκτώ μονάδες αποθήκευσης [14], οι οποίες ως επί το πλείστον είναι FDREs. Ο κώδικας που υλοποιεί μία απλή πύλη flip-flop έχει ως εξής:

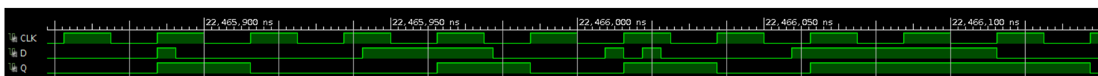
VHDL Code 2.4: ff_top

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity ff_top is
5  port(
6      CLK : in  std_logic;
7      D   : in  std_logic;
8      Q   : out std_logic
9  );
10 end ff_top;
11
12 architecture RTL of ff_top is
```

[7]Εν γένει, ως register καλείται και η πύλη *latch*, η οποία και αυτή λειτουργεί σαν καταχωρητής δεδομένων, αλλά με διαφορετικό τρόπο. Στη συνέχεια της εργασίας αυτής, όταν θα γίνεται αναφορά για register, θα νοείται η πύλη flip-flop.

```
13
14 begin
15
16 -- simple flip-flop/register process
17 ff_proc: process(CLK)
18 begin
19     if(rising_edge(CLK)) then
20         Q <= D;
21     end if;
22 end process;
23
24 end RTL;
```

Ο παραπάνω κώδικας, υλοποιεί εμμέσως *(inferring)*, μία πύλη flip-flop FDRE. Στη συνέχεια, θα αναλυθεί ο κώδικας γραμμή προς γραμμή, όμως για την ώρα θα δοθεί μία γενική εικόνα της λογικής. Η πύλη δέχεται σαν είσοδο το σήμα *D* και περνάει την κατάστασή του στην έξοδο *Q*, *μόνο τη στιγμή* που το σήμα του ρολογιού *CLK* μεταβαίνει από το λογικό μηδέν στο ένα. το *Q* παραμένει στην ίδια κατάσταση μέχρι την επόμενη μετάβαση του ρολογιού από το μηδέν στο ένα (χρονικό διάστημα που ταυτίζεται με την περίοδο του ρολογιού). Αυτή είναι ουσιαστικά η φιλοσοφία της συντριπτικής πλειοψηφίας των registers που χρησιμοποιούνται σε όλα τα κυκλώματα ψηφιακής λογικής, από τα FPGA μέχρι την CPU ενός ηλεκτρονικού υπολογιστή. Όσο αναφορά τα δύο σήματα ελέγχου του register, εν προκειμένω το *Reset* είναι πάντα μηδέν, και το *Clock-Enable* είναι πάντα ένα. Εν γένει όμως, τα δύο αυτά σήματα ελέγχουν τη συμπεριφορά της πύλης, με τέτοιο τρόπο όπως φαίνεται στον πίνακα του σχήματος 2.3.II. Το reset επαναφέρει την έξοδο στο λογικό μηδέν, ανεξαρτήτως των άλλων σημάτων εισόδου, ενώ το *D* οδηγείται στο *Q* μόνο κατά την άφιξη του μετώπου του ρολογιού, και μόνο αν το *CE* είναι ψηλά. Ελέγχοντας δηλαδή την κατάσταση του σήματος *CE*, μπορεί κανείς να *αποθηκεύσει* πληροφορία μέσω του register.



Σχήμα 2.3.III: Προσομοίωση του κώδικα 2.4.

Ο τρόπος κωδικοποίησης μίας σύγχρονης μεθόδου μπορεί να μελετηθεί στις γραμμές $17-22$. Η συγκεκριμένη μέθοδος, είναι η πρώτη που εμφανίζεται το *if clause*.

Σε αυτό το σημείο είναι καλό να ανοιχθεί μία παρένθεση για την επεξήγηση της σύνταξης του *if*:

VHDL Code 2.5: if

```
1 if (statement_0) then
2     -- signal assignments_0
3 elsif (statement_1) then
4     -- signal assignments_1
5 else
6     -- signal assignments_else
7 end if;
```

Στις γραμμές $1, 3$ κάνει κανείς τη δήλωση ελέγχου, προκειμένου να εκτελεστούν οι εντολές που βρίσκονται μέσα στο *if clause*. Οι δηλώσεις ελέγχου (*statement_0* και *statement_1* εδώ), μπορεί να έχουν τη μορφή *sig_a = '0' OR sig_b = '1'* κ.λπ. Η σύγκριση των υποθετικών αυτών σημάτων εισόδου sig_a και sig_b, μπορεί να γίνει με το λογικό μηδέν/ένα ή και με άλλα σήματα. Αν η υπόθεση του *statement_0* ισχύει, τότε θα εκτελεστούν οι αντίστοιχες εντολές, και δεν θα γίνει έλεγχος των υπολοίπων υποθέσεων. Αντίστοιχα, αν δεν ισχύει καμία από τις δύο υποθέσεις, θα εκτελεστούν οι εντολές του *else*. Όπως και στον προγραμματισμό λογισμικού δηλαδή, έχει σημασία η σειρά κωδικοποίησης των ελέγχων.

Επιστρέφοντας στον κώδικα 2.4 της σύγχρονης λογικής, μπορεί να δει κανείς το *if clause* που ουσιαστικά δηλώνει έμμεσα την πύλη flip-flop. Αυτή είναι η γραμμή 19. Το *rising_edge(CLK)* είναι που κατασκευάζει έναν register με είσοδο το *D* και έξοδο το *Q*. Εν γένει, οποιοδήποτε σήμα εξόδου βρίσκεται μέσα στο *if(rising_edge(CLK))then*, υλοποιείται ως καταχωρητής. Τέλος, αξίζει να σημειωθεί ότι σε τέτοιες μεθόδους, οποιοδήποτε σήμα εισόδου βρίσκεται ανάμεσα από τις γραμμές $19 - 22$, δεν χρειάζεται να μπει στην *sensitivity list*, εν αντιθέσει με τις συνδυαστικές μεθόδους. Το μόνο σήμα που πρέπει να δηλωθεί στη λίστα είναι το ρολόι, και αυτό γιατί ουσιαστικά τα σήματα εξόδου μπορούν να αλλάξουν *μόνο όταν αλλάζει κατάσταση το ρολόι* (εν προκειμένω, αφού είναι *rising_edge* register, στην μετάβαση από το μηδέν στο ένα).

Αν θελήσει κάποιος να κωδικοποιήσει και τα σήματα ελέγχου του register, όπως έχουν αναφερθεί στο Σχ. 2.3.ΙΙ, τότε ο κώδικας θα έχει ως εξής:

VHDL Code 2.6: FDRE_top

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity FDRE_top is
5  port(
6      CLK : in  std_logic;
7      RST : in  std_logic;
8      CE  : in  std_logic;
9      D   : in  std_logic;
10     Q   : out std_logic
11 );
12 end FDRE_top;
13
14 architecture RTL of FDRE_top is
15
16 begin
17
18 -- inferred FDRE with all side features
19 fdre_proc: process(CLK)
20 begin
21     if(rising_edge(CLK)) then
22         if(RST = '1') then
23             Q <= '0';
24         else
25             if(CE = '1') then
26                 Q <= D;
27             else
28                 null;
29             end if;
30         end if;
31     end if;
32 end process;
33
34 end RTL;
```

Ο παραπάνω κώδικας έχει σαν στόχο να υλοποιήσει εμμέσως μία πύλη *FDRE* όπως περιγράφεται στο Σχ. 2.3.II. Προκειμένου να συμβεί αυτό, πρέπει κατ'αρχάς το σήμα *Reset* να είναι σύγχρονο, επομένως ο έλεγχος του σήματος πρέπει να βρίσκεται κάτω από τον έλεγχο *rising_edge* του ρολογιού (γραμμές 21 − 23). Κοιτάζοντας τον λογικό πίνακα της πύλης *FDRE* στο αντίστοιχο σχήμα, μπορεί να δει κανείς ότι το *Reset*

προηγείται λογικά των υπόλοιπων σημάτων εισόδου της πύλης. Αν δηλαδή είναι ίσο με το λογικό ένα, η έξοδος της πύλης θα είναι πάντα μηδέν. Αυτό εκφράζεται κωδικοποιώντας τον έλεγχο *if(RST = '1')then* πριν από κάθε άλλο έλεγχο στη μέθοδο. Αν το *Reset* είναι μηδέν, η μέθοδος μπαίνει στις εντολές που βρίσκονται κάτω από το *else* (γραμμές $24-30$). Όπως έχει αναφερθεί, μόνο αν το σήμα ενεργοποίησης του ρολογιού είναι ψηλά, η είσοδος περνάει στην έξοδο κατά την άφιξη του μετώπου του σήματος του ρολογιού. Αυτό εκφράζεται στις γραμμές $25, 26$. Αν το σήμα αυτό δεν είναι αληθές, τότε εκτελείται το *else* του ελέγχου του σήματος (γραμμές $27, 28$). Εκεί έχει δηλωθεί το *null*, δηλαδή αν μπει η μέθοδος σε εκείνο το κομμάτι, δεν αλλάζει τίποτα στο σήμα εξόδου *Q*. Αυτή η λογική συμβαδίζει με τη δεύτερη γραμμή του λογικού πίνακα της πύλης (βλ. σχ. 2.3.II). Ο χρήστης προτρέπεται να προσομοιώσει τον απλό αυτό design προκειμένου να αντιληφθεί όλες τις λειτουργίες του βασικού register των Xilinx® FPGA Devices.

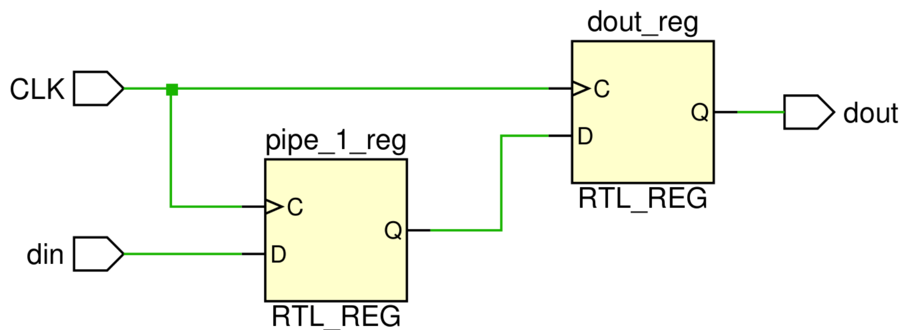
2.3.2 Shift Register

Μία άλλη εφαρμογή κυκλωμάτων σύγχρονης λογικής είναι οι *Shift Registers* ή *Pipelines*. Σε αυτή την εφαρμογή, πολλοί *registers* συνδέονται μεταξύ τους σε αλληλουχία *(cascading)*, προκειμένου να καθυστερήσουν ένα σήμα από το να αλλάξει κατάσταση για όσα στάδια έχει ορίσει ο σχεδιαστής. Ένας κώδικας με ένα στάδιο καθυστέρησης παρατίθεται εδώ μαζί με το *RTL Schematic*:

VHDL Code 2.7: shift_reg_1

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity pipeline_1 is
5  port(
6      CLK    : in  std_logic;
7      din    : in  std_logic;
8      dout   : out std_logic
9  );
10 end pipeline_1;
11
12 architecture RTL of pipeline_1 is
13
14 -- first pipeline stage
```

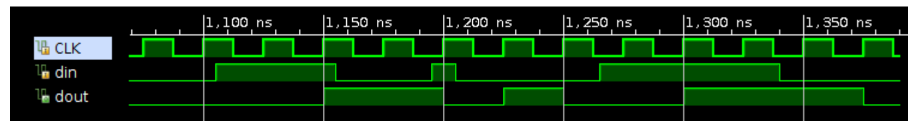
```
15 signal pipe_1 : std_logic := '0';
16
17 begin
18
19 -- pipelining process
20 pipeline_proc: process(CLK)
21 begin
22     if(rising_edge(CLK)) then
23         pipe_1  <= din;
24         dout    <= pipe_1;
25     end if;
26 end process;
27
28 end RTL;
```



Σχήμα 2.3.IV: RTL Schematic του κώδικα 2.7.

Η διαφορά σε σχέση με τους προηγούμενους κώδικες είναι ότι έχει δηλωθεί και ένα εσωτερικό σήμα, το *pipe_1*, το οποίο ουσιαστικά αντιστοιχεί σε έναν ακόμα register, όπως μπορεί κανείς να δει στη γραμμή 23. Το αποτέλεσμα της καταχώρησης του σήματος, οδηγείται σε μία ακόμα flip-flop, που δηλώνεται στη γραμμή 24. Έτσι κωδικοποιεί κανείς έναν shift register με δύο στάδια. Η προσομοίωση του κυκλώματος αυτού απεικονίζεται στο σχήμα 2.3.V.

Ο αναγνώστης προτρέπεται να προσθέσει και άλλα στάδια καθυστέρησης στον δοθέντα κώδικα, προκειμένου να διαπιστώσει πόση καθυστέρηση μπορεί να εισάγει στο σύστημα. Αξίζει επίσης να σημειωθεί ότι οι δύο αυτοί registers θα μπορούσαν να είχαν κωδικοποιηθεί σε δύο διαφορετικές μεθόδους, χωρίς καμία διαφορά στο



Σχήμα 2.3.V: Προσομοίωση του κώδικα 2.7. Προσέξτε ότι το σήμα εξόδου *dout* έχει καθυστέρηση δύο κύκλων σε σχέση με την αλλαγή κατάσταση του σήματος εισόδου *din*, ακριβώς όσος είναι δηλαδή και ο αριθμός των flip-flops που υλοποιεί το κύκλωμα.

αποτέλεσμα[8]. Ο κώδικας αυτός παρατίθεται στο αντίστοιχο παράρτημα (βλ. κώδικα Γ΄.5 στο Παράρτημα Γ΄).

2.3.3 Counter

Μία πολύ σημαντική εφαρμογή της γλώσσας VHDL, είναι η υλοποίηση *μετρητών (counters)*. Οι μετρητές βρίσκουν χρησιμότητα σε κάθε firmware. Μπορεί για παράδειγμα κάποιος να θέλει να ελέγξει την ποιότητα μίας γραμμής επικοινωνίας στέλνοντας μέσω ενός πομπού, πακέτα με τις διαδοχικές τιμές ενός μετρητή, και να περιμένει ο δέκτης να αποκωδικοποιεί τις τιμές αυτές σωστά. Επίσης, μία ακόμα συνήθης εφαρμογή ενός μετρητή, είναι να χρησιμοποιείται ουσιαστικά σαν μετρητής χρόνου. Ο τρόπος κωδικοποίησης αυτών των παραδειγμάτων θα ξεκαθαριστεί παρακάτω στην παρούσα ενότητα.

Στο σημείο αυτό καλό είναι να αναφερθεί ότι εκτός από το *std_logic* και το *std_logic_vector(N downto 0)* ως δηλώσεις σημάτων, είναι πολύ σύνηθες να χρησιμοποιούνται τα *unsigned(N downto 0)* και *integer*. Το πρώτο κωδικοποιεί ένα *unsigned vector*[9], το οποίο επιτρέπει αριθμητικές πράξεις μεταξύ αυτού και σημάτων ίδιου είδους (προσθαφαιρέσεις και πολλαπλασιασμοί), ενώ το δεύτερο είναι ένας ακέραιος αριθμός. Προφανώς, και τα δύο από τα προαναφερθέντα σήματα, είναι κάτι περισσότερο αφηρημένο σε σχέση με το τι υλοποιείται μέσα στο FPGA. Και οι δύο από αυτούς τους τύπους σημάτων είναι υποκατηγορίες του *std_logic_vector*, δηλαδή υλοποιούνται ως multi-bit signals. Το πρώτο παράδειγμα παρατίθεται στον εξής κώδικα:

[8]Αυτό υποδεικνύει ότι δύο διαφορετικές μέθοδοι λειτουργούν *παράλληλα*, ακριβώς όπως δύο ξεχωριστές γραμμές κώδικα συνδυαστικής λογικής.

[9]Unsigned σημαίνει ότι το σήμα αυτό μπορεί να πάρει μόνο θετικές τιμές.

VHDL Code 2.8: counter_simple

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

entity counter_simple is
port(
    clk     : in  std_logic;
    set     : in  std_logic;
    rst     : in  std_logic;
    cnt_val : out std_logic_vector(7 downto 0)
);
end counter_simple;

architecture RTL of counter_simple is

    -- internal signal, integer implementation
    signal cnt_val_int : integer range 0 to 255 := 0;

begin

-- counter implementation process
counter_proc: process(CLK)
begin
    if(rising_edge(CLK)) then
        if(rst = '1') then
            cnt_val_int <= 0;
        else
            if(set = '1') then
                cnt_val_int <= cnt_val_int + 1;
            else
                null;
            end if;
        end if;
    end if;
end process;

    -- convert the internal integer signal to an std_logic_vector
    cnt_val <= std_logic_vector(to_unsigned(cnt_val_int, 8));

end RTL;
```

Σε αυτό τον κώδικα, ο μετρητής υλοποιείται μέσω *integer*, που είναι ένα εσωτερικό σήμα και δηλώνεται στη γραμμή 17. Εδώ φαίνεται μία λειτουργία της VHDL που ο σχεδιαστής οφείλει να εκμεταλλεύεται κάθε φορά που υλοποιεί έναν ακέραιο αριθμό, και αυτή είναι η δυνατότητα καθορισμού *εύρους* τιμών που μπορεί να πάρει ένας ακέραιος που ορίζεται μέσα στον κώδικα. Εν προκειμένω, οι δυνατές τιμές εκτείνονται από το 0 μέχρι το 255 (με αρχικοποίηση στο μηδέν), επομένως, ο ακέραιος αυτός θα κωδικοποιηθεί στην πραγματικότητα ως ένα *std_logic_vector* με *οκτώ* δυφία. Γνωρίζοντας εκ των προτέρων την εφαρμογή του κάθε μετρητή, ο σχεδιαστής μπορεί να ορίσει τη μέγιστη τιμή που θα παίρνει, επομένως και το εύρος. Είναι πολύ σημαντικό να ορίζεται το εύρος (πάντα έχοντας στο μυαλό την αντιστοίχηση των αριθμών από το δυαδικό στο δεκαδικό σύστημα[10]), ώστε να είναι γνωστό ακριβώς πόσους πόρους του FPGA δεσμεύει ο μετρητής.

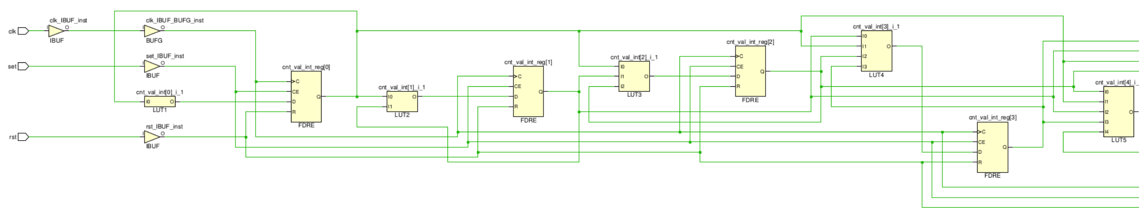
Έχοντας αυτό κατά νου, ο υπόλοιπος κώδικας δεν διαφέρει και πολύ από αυτά που έχουν αναφερθεί παραπάνω στην παρούσα ενότητα. Το σήμα *Reset* επαναφέρει τον μετρητή στο μηδέν, ενώ αν το σήμα *Set* δεν είναι ψηλά, τότε ο μετρητής δεν αλλάζει τιμή. Στη γραμμή 29 είναι που δηλώνεται η αύξηση της τιμής του ακεραίου κατά ένα, η οποία αύξηση θα συμβαίνει σε κάθε clock tick του ρολογιού που χρονίζει το μετρητή. Η μόνη καινούργια πληροφορία που εισάγει αυτό το δείγμα κώδικα είναι στη γραμμή 38, όπου ο ακέραιος αριθμός μετατρέπεται πρώτα σε *usnigned*, και μετά σε *std_logic_vector*, προκειμένου να βγει από το top_level ως output. Θεωρείται καλή πρακτική οι διεπαφές των components να αποτελούνται μόνο από *std_logic* και *std_logic_vector*[11], επομένως αμέσως απαιτείται η μετατροπή του εσωτερικού σήματος του μετρητή, που είναι ακέραιος, σε διάνυσμα από bits. Στο αριστερό μέλος της εντολής της γραμμής 38 βρίσκεται ένα *std_logic_vector(7 downto 0)*, επομένως οτιδήποτε βρίσκεται στα δεξιά του βέλους, πρέπει να γίνει map σε οκτώ δυφία, γεγονός που είναι φανερό στο τέλος της γραμμής. Το κομμάτι *(to_unsigned(cnt_val_int, 8))*, μετατρέπει τον ακέραιο αριθμό σε *unsigned* μήκους οκτώ bits, το οποίο μετατρέπεται

[10]Δηλαδή ότι ένα vector με 2 bits μπορεί να μετράει μέχρι το 3, ένα με 3 bits μπορεί να μετράει μέχρι το 7, ένα με 8 bits μέχρι το 255, κ.ο.κ.

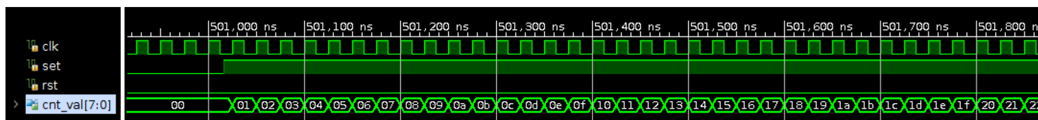
[11]Σε κάποια project, γίνεται χρήση μονάδων που είναι γραμμένα σε VHDL μαζί με άλλα modules που είναι γραμμένα σε Verilog. Επειδή οι διεπαφές των Verilog components/modules είναι ουσιαστικά μόνο *std_logic* και *std_logic_vector*, αν θελήσει κανείς να μιλήσει χρησιμοποιώντας την ονοματολογία της VHDL, είναι καλό όλες οι διεπαφές των VHDL components να είναι τέτοιου είδους.

αμέσως σε *std_logic_vector*. Οι μετατροπές αυτές είναι δυνατές μέσω της βιβλιοθήκης που ορίζεται στη γραμμή 3 του κώδικα.

Ενδιαφέρον παρουσιάζει το *Synthesized Schematic* του παραπάνω κυκλώματος. Το τελικό υλοποιημένο κύκλωμα θα είναι κάποια *cascaded* LUT, το κάθε ένα εκ των οποίων δέχεται στις εισόδους του κάποια bits του μετρητή. Ανάμεσα από τα LUT, βρίσκονται FDRE, οι οποίες με κάθε 'χτύπο' του ρολογιού clk, περνάνε στο επόμενο LUT την νέα τιμή του μετρητή, ο οποίος υλοποιεί τη πρόσθεση, και με τη σειρά του περνάει το αποτέλεσμα στην επόμενη FDRE, κ.ο.κ. Αυτό σημαίνει ότι ο μετρητής θα αυξάνει κατά ένα σε κάθε *rising_edge* του ρολογιού που χρονίζει το κύκλωμα. Αν για παράδειγμα χρησιμοποιείται ένα ρολόι $f = 40\,Mhz$, κάθε $25\,ns$ ο μετρητής θα αυξάνει κατά ένα.



Σχήμα 2.3.VI: Κομμάτι από το Synthesized Schematic του κώδικα 2.8.



Σχήμα 2.3.VII: Simulation του κώδικα 2.8.

Ο παραπάνω κώδικας θα μπορούσε να είχε γραφτεί λίγο διαφορετικά, αν αντί για ακέραιο αριθμό για την υλοποίηση του μετρητή χρησιμοποιούταν απευθείας ένα *unsigned* vector μήκους οκτώ bits. Ο ισοδύναμος αυτός κώδικας παρατίθεται στο αντίστοιχο Παράρτημα (βλ. κώδικα Γ΄.6 στο Γ΄).

2.3.4 Serializer/Deserializer

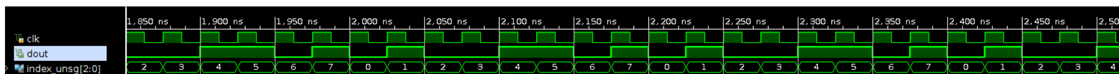
Τα περισσότερα πρωτόκολλα επικοινωνίας μεταξύ ολοκληρωμένων κυκλωμάτων είναι *σειριακά*. Αυτό σημαίνει πως οι πληροφορίες μεταδίδονται μόνο σε μία γραμμή,

ενώ αρκετές φορές ξεχωριστά από τη γραμμή δεδομένων, ο πομπός μεταδίδει το ρολόι της επικοινωνίας, προκειμένου ο δέκτης να το χρησιμοποιήσει για τη σωστή αποκωδικοποίηση των πληροφοριών που του στέλνονται. Άλλες φορές, ο τρόπος κωδικοποίησης της γραμμής δεδομένων (όπως συμβαίνει π.χ. στο *Ethernet*), επιτρέπει την *ανάκτηση του ρολογιού* από μέρους του δέκτη. Στο παράδειγμα που ακολουθεί, θα σχεδιαστεί αρχικά ένας πομπός, που θα λειτουργεί ουσιαστικά ως *serializer*, δηλαδή θα δέχεται ένα διάνυσμα από bits, και θα τα μεταδίδει σε μία γραμμή, ένα δυφίο τη φορά, σύμφωνα με ένα ρολόι αναφοράς, που θα καθορίζει ουσιαστικά την ταχύτητα της επικοινωνίας. Στη συνέχεια θα παρατεθεί ο κώδικας για έναν *deserializer*, ο οποίος θα παίζει το ρόλο του δέκτη. Θα ανακτά το ρολόι που του στέλνει ο πομπός, και μέσω αυτού θα προβαίνει σε δειγματοληψία της γραμμής επικοινωνίας, κατασκευάζοντας έτσι ένα *std_logic_vector* από τα bits που καταχωρεί. Ακολουθεί ο κώδικας του serializer μαζί με το simulation του:

VHDL Code 2.9: serializer

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.all;
4
5  entity serializer is
6  port(
7      clk     : in  std_logic;
8      dout    : out std_logic
9  );
10 end serializer;
11
12 architecture RTL of serializer is
13
14     signal   index_unsg : unsigned(2 downto 0) := (others => '0');
15     constant dout_vec   : std_logic_vector(7 downto 0) := "01011001";
16
17 begin
18
19 -- serializing process
20 ser_proc: process(clk)
21 begin
22     if(rising_edge(clk))then
23         case index_unsg is
```

```
24            when "000"   => dout <= dout_vec(0);
25            when "001"   => dout <= dout_vec(1);
26            when "010"   => dout <= dout_vec(2);
27            when "011"   => dout <= dout_vec(3);
28            when "100"   => dout <= dout_vec(4);
29            when "101"   => dout <= dout_vec(5);
30            when "110"   => dout <= dout_vec(6);
31            when "111"   => dout <= dout_vec(7);
32            when others => dout <= '0';
33        end case;
34    end if;
35 end process;
36
37 -- counting process
38 cnt_proc: process(clk)
39 begin
40     if(rising_edge(clk))then
41         index_unsg <= index_unsg + 1;
42     end if;
43 end process;
44
45 end RTL;
```



Σχήμα 2.3.VIII: Προσομοίωση του κώδικα 2.9.

Το παραπάνω design έχει απλοποιηθεί ώστε να μη διαθέτει *set/reset*, και απλά να περνάει το *dout_vec*, bit by bit, στην έξοδο *dout*. Η επιλογή του bit που θα μεταδοθεί γίνεται στη μέθοδο μεταξύ των γραμμών $20-35$, η οποία είναι ουσιαστικά ένας σύγχρονος πολυπλέκτης. Με κάθε *rising_edge* του ρολογιού *clk*, ο μετρητής αυξάνει κατά ένα (γραμμές $38-43$), και επειδή είναι δηλωμένος σαν *unsigned vector*, όταν φτάσει στη μέγιστη τιμή του (το "111"), στον επόμενο κύκλο θα πάρει την τιμή "000" *(rollover)*. Προφανώς, το μήκος του *index_unsg* έχει επιλεγεί έτσι ώστε η μέγιστη τιμή του να ταυτίζεται με την τελευταία θέση του *std_logic_vector* που μεταδίδεται σειριακά. Επίσης, ο κώδικας έχει δύο μεθόδους, οι οποίες αν και λειτουργούν με το ίδιο ρολόι (άρα θα μπορούσαν και να είχαν συγχωνευθεί σε μία), είναι χωρισμένες

αφού οι συμβάσεις που ακολουθούνται υποδεικνύουν να χωρίζονται οι processes ανάλογα με το έργο που επιτελούν. Στο παράρτημα μπορεί να βρεθεί ένας ισοδύναμος κώδικας, όπου δεν χρησιμοποιείται πολυπλέκτης για την επιλογή του σήματος εξόδου, αλλά γίνεται χρήση ακέραιου αριθμού (βλ. κώδικα Γ΄.7 στο Παράρτημα Γ΄). Ο κώδικας του *deserializer* από την άλλη, θα έχει ως εξής:

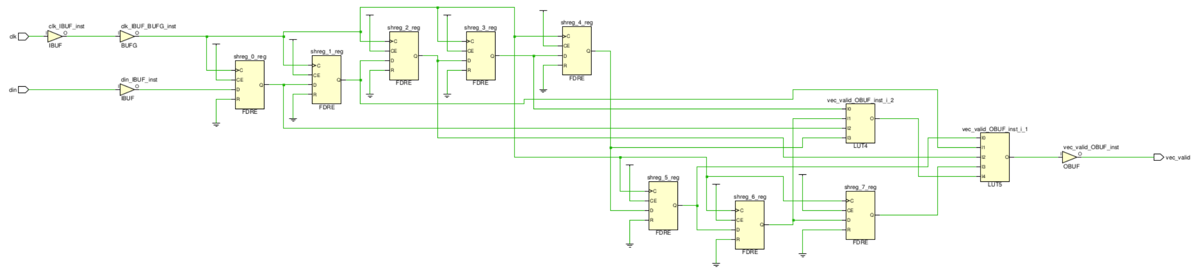
VHDL Code 2.10: deserializer

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.all;
4
5  entity deserialzer is
6  port(
7      clk         : in  std_logic;
8      din         : in  std_logic;
9      vec_valid   : out std_logic
10 );
11 end deserialzer;
12
13 architecture RTL of deserialzer is
14
15     signal  shreg_0 : std_logic := '0';
16     signal  shreg_1 : std_logic := '0';
17     signal  shreg_2 : std_logic := '0';
18     signal  shreg_3 : std_logic := '0';
19     signal  shreg_4 : std_logic := '0';
20     signal  shreg_5 : std_logic := '0';
21     signal  shreg_6 : std_logic := '0';
22     signal  shreg_7 : std_logic := '0';
23     signal  din_des : std_logic_vector(7 downto 0):= (others => '0');
24
25 begin
26
27 -- deserializer/shift register
28 des_proc: process(clk)
29 begin
30     if(rising_edge(clk))then
31         shreg_0 <= din;
32         shreg_1 <= shreg_0;
33         shreg_2 <= shreg_1;
```

```
34          shreg_3 <= shreg_2;
35          shreg_4 <= shreg_3;
36          shreg_5 <= shreg_4;
37          shreg_6 <= shreg_5;
38          shreg_7 <= shreg_6;
39      end if;
40  end process;
41
42  -- vectorize the shift register interconnections
43      din_des <= shreg_0 & shreg_1 & shreg_2 & shreg_3
44               & shreg_4 & shreg_5 & shreg_6 & shreg_7;
45
46  -- if the serializer vector is found, assert the valid signal
47  check_proc: process(din_des)
48  begin
49      case din_des is
50      when "01011001" => vec_valid <= '1';
51      when others     => vec_valid <= '0';
52      end case;
53  end process;
54
55  end RTL;
```

Πρόκειται για έναν *shift register*, ο οποίος αποτελείται από οκτώ καταχωρητές, οι διασυνδέσεις των οποίων φτιάχνουν ένα διάνυσμα από bits, οκτώ θέσεων, με όνομα *din_des*. Το διάνυσμα αυτό σχηματίζεται στις γραμμές 43, 44 ενώ ο shift register ορίζεται στη σύγχρονη μέθοδο των γραμμών $28 - 40$. Τέλος, υπάρχει και μία συνδυαστική μέθοδος η οποία ελέγχει το διάνυσμα *din_des* και το συγκρίνει με το διάνυσμα που μεταδίδει ο serializer. Αν η μετάδοση της πληροφορίας είναι σωστή, τότε κάθε οκτώ κύκλους του ρολογιού, το σήμα *vec_valid* θα παίρνει τη λογική τιμή ένα. Ο αναγνώστης παραπέμπεται στο Παράρτημα (συγκεκριμένα στον κώδικα Γ΄.8), όπου θα βρει έναν διαφορετικό, πιο συμπαγή τρόπο ορισμού του shift register.

Εάν οι παραπάνω δύο κώδικες συνδεθούν μεταξύ τους, δηλαδή αν τους παραχθεί ένα κοινό ρολόι, και αν οδηγηθεί η έξοδος του ενός στην είσοδο του άλλου, τότε θα διαπιστωθεί η ορθή λειτουργία της μεταξύ τους επικοινωνίας (το σήμα *vec_valid* θα παίρνει τη λογική τιμή ένα ανά οκτώ κύκλους). Ο χρήστης προτρέπεται να κατασκευάσει ένα ακόμα αρχείο VHDL, το οποίο θα περιέχει τους serializer/deserializer



Σχήμα 2.3.IX: Synthesized Schematic του κώδικα 2.10.

(θα είναι δηλαδή ένας *wrapper*), και θα υλοποιεί τη μεταξύ τους επικοινωνία.

Σύνοψη

Σε αυτό το Κεφάλαιο, μελετήθηκαν κάποιες βασικές αρχές της VHDL με τη χρήση απλών παραδειγμάτων. Ο αναγνώστης παραπέμπεται στα [4, 6] για περαιτέρω εμβάθυνση.

3

Μηχανές Πεπερασμένων Καταστάσεων σε VHDL

Μέχρι τώρα έχουν μελετηθεί κυκλώματα συνδυαστικής και ακολουθιακής λογικής, και εν γένει, αυτοί είναι οι τύποι κυκλωμάτων που μπορεί να σχεδιάσει κανείς μέσω της VHDL. Το μεγαλύτερο μέρος των designs βέβαια, αποτελείται από χρονισμένα ακολουθιακά κυκλώματα, τα οποία προτιμούνται λόγω της ευελιξίας που προσφέρουν, αφού για να υλοποιηθεί ακόμα και ο πιο βασικός αλγόριθμος, απαιτείται η ύπαρξη ρολογιού για την καταχώρηση του ”ιστορικού” του κυκλώματος. Πάρα ταύτα, τα σύγχρονα design που έχουν μελετηθεί μέχρι τώρα, δεν μπορεί να πει κανείς ότι υλοποιούσαν κάποιον περίπλοκο αλγόριθμο. Ο λόγος είναι ότι η καθιερωμένη μέθοδος κωδικοποίησης περισσότερο πολύπλοκων ακολουθιακών κυκλωμάτων, είναι μέσω μοντελοποίησης σε *Μηχανές Πεπερασμένων Καταστάσεων (Finite State Machines (FSMs))*. Η FSM είναι μία αφηρημένη έννοια λογικής, στην οποία ορίζεται ένα σύνολο *καταστάσεων (states)*, με την παραδοχή ότι η FSM μπορεί να βρίσκεται σε μόνο *μία* κατάσταση ανά χρονική στιγμή[1]. Εκτός από τις καταστάσεις, ο χρήστης πρέπει να ορίσει και τις συνθήκες *μετάβασης (transition)*, από τη μία κατάσταση στην άλλη, οι οποίες συνήθως καθορίζονται από τις τιμές των σημάτων εισόδου της FSM, και από την τρέχουσα κατάσταση στην οποία βρίσκεται. Ένα ρολόι χρονισμού, καθορίζει κάθε πότε αλλάζει (ή όχι) η κατάσταση της FSM: αν η συνθήκη μετάβασης πληρούται, τότε στο επόμενο *rising_edge* του ρολογιού, η FSM

[1]Για τις FSM που υλοποιούνται σε ψηφιακά κυκλώματα, είναι πιο σωστό να πει κανείς ότι η FSM μπορεί να βρίσκεται σε μόνο μία κατάσταση, σε κάθε περίοδο του ρολογιού που χρονίζει την FSM.

θα μεταβεί στην επόμενη κατάσταση. Τέλος, η FSM θα έχει και κάποια σήματα εξόδου, τα οποία μπορεί είτε να εξαρτώνται αμιγώς από την τρέχουσα κατάσταση (FSM τύπου *Moore*), ή και από τις τιμές των σημάτων εισόδου σε συνδυασμό με την τρέχουσα κατάσταση (FSM τύπου *Mealy*).

3.1 Σύντομη Εισαγωγή στις FSM

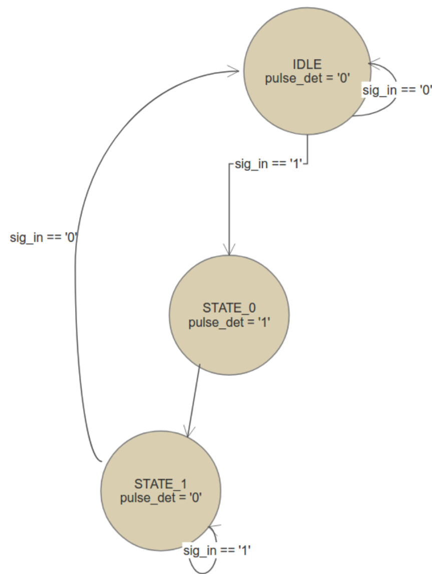
Έχοντας πάρει μία γενική ιδέα για το τι είναι μία Μηχανή Πεπερασμένων Καταστάσεων, η συνέχεια θα αφιερωθεί σε σχηματικές αναπαραστάσεις των FSM. Ο πιο συνηθισμένος τρόπος απεικόνισης FSM είναι μέσω του *Διαγράμματος Μετάβασης Καταστάσεων (State Transition Diagram (STD))*. Έστω μία απλή FSM, η οποία αποτελείται από τρεις καταστάσεις: *IDLE* (που είναι και η αρχική), *STATE_0* και *STATE_1*[2]. Η FSM δέχεται ένα σήμα εισόδου που ονομάζεται *sig_in*, και παράγει ένα σήμα εξόδου, το *pulse_det*. Για τη γραφική αναπαράσταση της λογικής της FSM ο αναγνώστης παραπέμπεται στο *STD*[3], του σχήματος 3.1.I.

Καθώς η FSM βρίσκεται στην κατάσταση *IDLE*, περιμένει το σήμα εισόδου *sig_in* να πάρει τη λογική τιμή ένα, ώστε να μεταβεί στην κατάσταση *STATE_0*. Αν κάτι τέτοιο δε συμβεί, τότε θα παραμείνει στο *IDLE*. Στην κατάσταση *STATE_0*, το σήμα εξόδου *pulse_det* γίνεται ένα, και στον επόμενο κύκλο πραγματοποιείται μία μετάβαση στην κατάσταση *STATE_1*, όπου το σήμα εξόδου γίνεται μηδέν. Η FSM θα παραμείνει σε αυτή την κατάσταση μέχρι το σήμα εισόδου να επανέλθει στο λογικό μηδέν, όπου θα ξαναγυρίσει στο *IDLE*. Η μετάβαση από την *STATE_0* στην *STATE_1* επειδή δεν εξαρτάται από κάποιο σήμα εισόδου χαρακτηρίζεται ως *unconditional transition*. Η FSM που μόλις περιγράφηκε είναι τύπου Moore, καθώς το σήμα εξόδου *pulse_det* εξαρτάται μόνο από την κατάσταση στην οποία βρίσκεται η FSM, και όχι από την τιμή του σήματος εισόδου *σε συνδυασμό* με την τρέχουσα κατάσταση. Αν θελήσει κανείς να εφαρμόσει αυτό το μοντέλο, που είναι τύπου Mealy, τότε η FSM θα είναι σχεδιασμένη λίγο διαφορετικά, γεγονός που φαίνεται στο σχήμα 3.1.II.

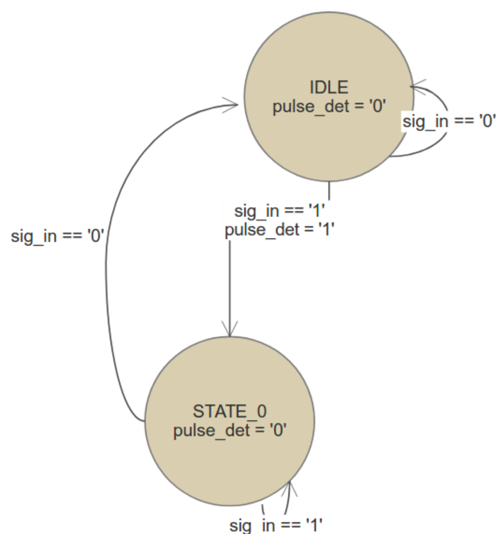
Είναι φανερό ότι η FSM τύπου Mealy μπορεί να είναι πιο οικονομική από άποψη

[2]Οι συμβάσεις που ακολουθούνται υποδεικνύουν τη χρήση κεφαλαίων γραμμάτων για τη δήλωση των καταστάσεων στις FSM.

[3]Τα διαγράμματα αυτά κατασκευάστηκαν με το on-line λογισμικό του ιστότοπου *https://cloud.smartdraw.com*.



Σχήμα 3.1.I: STD μίας απλής FSM τύπου Moore. Στα βέλη όπου συμβολίζονται οι μεταβάσεις από τη μία κατάσταση στην άλλη, σημειώνεται η συνθήκη μετάβασης (εξ ου και το ”==”), ενώ μέσα στους κύκλους των καταστάσεων σημειώνεται και η τιμή των σημάτων εξόδου όταν η FSM βρίσκεται στην εν λόγω κατάσταση.



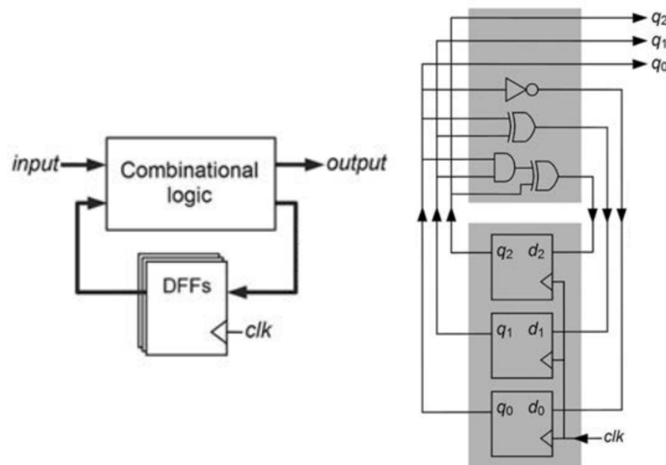
Σχήμα 3.1.II: STD του σχήματος 3.1.I αλλά με αρχιτεκτονική τύπου Mealy.

πλήθους καταστάσεων. Κάτι τέτοιο είναι ένας γενικός κανόνας που διέπει τη σχέση μεταξύ των δύο τρόπων σχεδιασμού FSM. Η διαφορά είναι ότι στην FSM τύπου

Mealy το σήμα εξόδου μπορεί να αλλάξει ενώ η FSM βρίσκεται στο *IDLE*, αν το σήμα εισόδου γίνει ένα. Μετά, στην *STATE_0*, το σήμα εξόδου θα γίνει πάλι μηδέν. Το τελικό αποτέλεσμα θα είναι δηλαδή το ίδιο για το *pulse_det*: θα μένει ψηλά πάντα για έναν κύκλο του ρολογιού που χρονίζει την FSM.

3.2 Απλές FSM σε VHDL

Αν μέχρι τώρα δεν έχει ξεκαθαριστεί ο τρόπος με τον οποίο λειτουργούν οι μηχανές πεπερασμένων καταστάσεων, τα παραδείγματα κωδικοποίησης των προαναφερθέντων μηχανών σε VHDL είναι σίγουρο πως θα ρίξουν λίγο περισσότερο φως στην κατάσταση. Κατ' αρχάς, στο hardware γενικότερα, άρα και στα FPGA, οι FSM υλοποιούνται από κυκλώματα τόσο συνδυαστικής, όσο και σύγχρονης λογικής. Η σύγχρονη λογική υλοποιείται για να αποθηκεύσει την κατάσταση της FSM *(state register)*, ενώ η συνδυαστική λογική χρειάζεται για να λάβει αποφάσεις για τις μεταβάσεις των καταστάσεων, και για την κατάσταση των σημάτων εξόδου (βλ. σχ. 3.2.I).



Σχήμα 3.2.I: Γενική επισκόπηση της αρχιτεκτονικής κάθε FSM σε χαμηλό επίπεδο. Αριστερά απεικονίζεται η γενική ιδέα που διέπει τη συνδεσμολογία κάθε FSM. Φαίνεται πως οι λογικές πράξεις του συνδυαστικού κυκλώματος οδηγούνται πάλι πίσω στη σύγχρονη λογική των state registers ως *feedback*, προκειμένου στον επόμενο κύκλο του ρολογιού η FSM να μεταβεί στην επόμενη κατάσταση. Δεξιά φαίνεται ο τρόπος υλοποίησης ενός απλού μετρητή με μορφή FSM [5].

3.2.1 Width Regulator

Έτσι λοιπόν, συνδυάζοντας τις ήδη αποκτημένες γνώσεις από το προηγούμενο κεφάλαιο, και ακολουθώντας την μεθοδολογία όπως επισημαίνεται στη βιβλιογραφία [4, 5, 6], μπορεί κανείς να κωδικοποιήσει την Moore-FSM της προηγούμενης ενότητας, σε VHDL, ως εξής:

VHDL Code 3.1: Moore_FSM

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.all;
4
5  entity fsm_moore_1 is
6  port(
7      clk          : in  std_logic;
8      sig_in       : in  std_logic;
9      pulse_det    : out std_logic
10 );
11 end fsm_moore_1;
12
13 architecture RTL of fsm_moore_1 is
14
15     -- state declaration
16     type stateType is (IDLE, STATE_0, STATE_1);
17     signal pr_state, nx_state : stateType := IDLE;
18
19     signal pulse_det_i : std_logic := '0';
20
21 begin
22
23 -- Moore FSM state register
24 moore_fsm_state_reg: process(clk)
25 begin
26     if(rising_edge(clk))then
27         pr_state <= nx_state;
28     end if;
29 end process;
30
31 -- Moore FSM combinatorial logic
32 moore_fsm_comb: process(pr_state, sig_in)
```

```
33 begin
34     case pr_state is
35     when IDLE =>
36         pulse_det_i  <= '0';
37         if(sig_in = '1')then
38             nx_state <= STATE_0;
39         else
40             nx_state <= IDLE;
41         end if;
42
43     when STATE_0 =>
44         pulse_det_i <= '1';
45         nx_state     <= STATE_1;
46
47     when STATE_1 =>
48         pulse_det_i  <= '0';
49         if(sig_in = '0')then
50             nx_state <= IDLE;
51         else
52             nx_state <= STATE_1;
53         end if;
54
55     when others =>
56         nx_state <= IDLE;
57     end case;
58 end process;
59
60 -- Output register (optional)
61 reg_output: process(clk)
62 begin
63     if(rising_edge(clk))then
64         pulse_det <= pulse_det_i;
65     end if;
66 end process;
67
68 end RTL;
```

Ξεκινώντας από τις γραμμές $6-10$, μπορεί κανείς να δει ότι η FSM δέχεται σαν είσοδο το ρολόι των state registers, και το *sig_in*, ενώ έχει σαν μοναδική έξοδο το *pulse_det*. Εν συνεχεία, καταχωρείται η λίστα των καταστάσεων που μπορεί να έχει

η FSM (γραμμή 16). Ουσιαστικά, σε αυτή τη γραμμή, ορίζεται ένας νέος τύπος σήματος, ο οποίος έχει τρεις δυνατές τιμές, που είναι οι καταστάσεις της FSM (*IDLE, STATE_0, STATE_1*)[4]. Στη συνέχεια (γραμμή 17), ορίζεται το σήμα της κατάστασης αυτό καθ' αυτό, το οποίο είναι του τύπου της γραμμής 16, και αρχικοποιείται στο *IDLE*. Αργότερα θα γίνει φανερό γιατί χρειάζονται δύο σήματα για την κατάσταση της FSM.

Στις γραμμές 24 – 29, βρίσκεται η σύγχρονη μέθοδος όπου καταχωρείται σε κάθε *rising_edge* του ρολογιού η κατάσταση της FSM. Το σήμα *nx_state* είναι αυτό που υπολογίζεται *από* το κύκλωμα της συνδυαστικής λογικής, ενώ το σήμα *pr_state* είναι εκείνο που *οδηγείται* στη συνδυαστική λογική προκειμένου αυτή να κάνει τις απαραίτητες λογικές πράξεις για τη συνέχεια. Στις γραμμές 32 – 58 βρίσκεται το κομμάτι της συνδυαστικής λογικής της FSM, κωδικοποιημένο σα μέθοδος, με κύριο συστατικό το *case statement*, που λαμβάνει υπόψιν του το σήμα που καταχωρείται από την σύγχρονη λογική *(pr_state)*. Στις γραμμές 35 – 41, βλέπει κανείς τι συμβαίνει όταν η FSM βρίσκεται στο *IDLE*. Αν το σήμα εισόδου γίνει ένα, τότε στον επόμενο κύκλο του ρολογιού η FSM θα βρεθεί στην *STATE_0* (γραμμές 43 – 45). Σε αυτή την κατάσταση, το σήμα εξόδου γίνεται ένα, και στον επόμενο κύκλο η FSM μεταβαίνει στην *STATE_1* (γραμμές 47 – 53), όπου το σήμα εξόδου γίνεται και πάλι μηδέν, και λαμβάνεται η απόφαση μετάβασης πίσω στο *IDLE*. Ο αναγνώστης παραπέμπεται πίσω στο σχήμα 3.1.I, προκειμένου να διαπιστώσει την συμφωνία μεταξύ του κώδικα και του διαγράμματος της FSM. Στις γραμμές 61 – 66 βρίσκεται ένας *output register*, δηλαδή ένας καταχωρητής που θα φροντίσει το σήμα εξόδου να είναι απόλυτα συγχρονισμένο με το ρολόι που χρονίζει τη λογική. Η προσθήκη τέτοιων μεθόδων είναι αρκετά συνηθισμένη, καθώς βοηθάει στο να ορίζεται με σαφήνεια η συμπεριφορά του σήματος εξόδου σε σχέση με το ρολόι αναφοράς. Πολλές φορές όμως, η προσθήκη καταχωρητών για τα σήματα εξόδου δεν είναι απαραίτητη, ή σε μερικές περιπτώσεις είναι ακόμα και ανεπιθύμητη. Περισσότερα για το συγκεκριμένο ζήτημα θα αναφερθούν στη συνέχεια της παρούσας εργασίας.

Ο παραπάνω τρόπος κωδικοποίησης FSM είναι ο πιο διαδεδομένος στη βιβλιογραφία, καθώς ξεκαθαρίζει απόλυτα το πως οδηγούνται τα σήματα μεταξύ των μεθό-

[4]Στο FPGA, η κάθε κατάσταση αντιστοιχεί σε μία και μοναδική τιμή ενός *std_logic_vector*. Περισσότερες λεπτομέρειες για το πως κωδικοποιείται η κατάσταση μίας FSM σε χαμηλό επίπεδο μπορούν να βρεθούν στο Κεφάλαιο 4.

δων που αποτελούν την FSM. Εκτός από τη συγκεκριμένη μεθοδολογία, υπάρχει και μία περισσότερο οικονομική, ως προς το πλήθος γραμμών κώδικα, αλλά πολλές φορές και ως προς τον αριθμό πόρων που δεσμεύονται στο FPGA από την FSM. Η συνέχεια της παρούσας ενότητας θα αφιερωθεί στο να παρουσιαστεί αυτή η μεθοδολογία (η οποία έχει σα βάση την περιγραφή της FSM από μία και μόνο process), και να τεθεί σε σύγκριση με την προηγούμενη:

VHDL Code 3.2: Moore_FSM_simplified

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.all;
4
5  entity fsm_moore is
6  port(
7      clk          : in  std_logic;
8      sig_in       : in  std_logic;
9      pulse_det    : out std_logic
10 );
11 end fsm_moore;
12
13 architecture RTL of fsm_moore is
14
15     type stateType is (IDLE, STATE_0, STATE_1);
16     signal state : stateType := IDLE;
17
18 begin
19
20 -- simple Moore FSM
21 moore_fsm_proc: process(clk)
22 begin
23     if(rising_edge(clk))then
24         case state is
25         when IDLE =>
26             pulse_det <= '0';
27             if(sig_in = '1')then
28                 state <= STATE_0;
29             else
30                 state <= IDLE;
31             end if;
32
```

```
33          when STATE_0 =>
34              pulse_det   <= '1';
35              state       <= STATE_1;
36
37          when STATE_1 =>
38              pulse_det   <= '0';
39              if(sig_in = '0') then
40                  state <= IDLE;
41              else
42                  state <= STATE_1;
43              end if;
44
45          when others => state <= IDLE;
46          end case;
47      end if;
48  end process;
49
50  end RTL;
```

Η ειδοποιός διαφορά μεταξύ των κωδίκων 3.2 και 3.1, έγκειται στο γεγονός ότι στη δεύτερη περίπτωση, όλη η FSM βρίσκεται κωδικοποιημένη μέσα στο *if(rising_edge(clk))* του ρολογιού της (γραμμές 23 – 47), σε μία και μόνο μέθοδο. Αυτό έχει σαν αποτέλεσμα κατ' αρχάς να χρειάζεται μόνο ένα σήμα για την περιγραφή της κατάστασης, που εν προκειμένω ονομάζεται *state*. Μάλιστα, αν συγκρίνει κανείς τη συνδυαστική μέθοδο του κώδικα 3.2 με τις γραμμές 25 – 46 του κώδικα 3.1, θα διαπιστώσει ότι η διαφορά είναι ότι πλέον μόνο ένα σήμα ορίζει την κατάσταση της FSM. Εδώ όμως, είναι σημαντικό να τονιστεί μία σημαντική λεπτομέρεια: ανατρέχοντας στην ενότητα 2.3, ο αναγνώστης μπορεί να φρεσκάρει τη μνήμη του, και να θυμηθεί ότι *όλα* τα σήματα εξόδου[5] που εμπεριέχονται μέσα στο *if(rising_edge(clk))* μίας μεθόδου, αποτελούν ουσιαστικά έναν *flip-flop register*. Αυτό έχει σαν συνέπεια στον προκείμενο τρόπο κωδικοποίησης της FSM, *όλα* τα σήματα εξόδου της FSM να είναι συγχρονισμένα με το ρολόι *clk* που χρονίζει την αλλαγή των καταστάσεων, και αυτό το γεγονός, σε ορισμένες εφαρμογές μπορεί να οδηγεί σε "σπατάλη" πόρων του FPGA, καθώς δεν χρειάζεται (ή σε πολύ λίγες περιπτώσεις είναι και ανεπιθύμητο) πάντα τα σήματα εξόδου να είναι συγχρονισμένα με το ρολόι της FSM[6]. Σε κάθε περίπτωση,

[5] Δηλαδή όλα τα σήματα που βρίσκονται αριστερά μίας εντολής $a <= b$.

[6] Παρόμοιο σχόλιο έχει γίνει και για τις γραμμές 60 – 65 του κώδικα 3.1. Υπενθυμίζεται ότι η

για το υπόλοιπο της παρούσας εργασίας, θα προτιμάται ο δεύτερος, πιο οικονομικός τρόπος κωδικοποίησης στις FSM. Έτσι, στη συνέχεια θα παρατεθεί ο κώδικας για την ίδια FSM, αλλά με αρχιτεκτονική τύπου Mealy:

VHDL Code 3.3: Mealy_FSM

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.all;
4
5  entity fsm_mealy is
6  port(
7      clk         : in  std_logic;
8      sig_in      : in  std_logic;
9      pulse_det   : out std_logic
10 );
11 end fsm_mealy;
12
13 architecture RTL of fsm_mealy is
14
15     type stateType is (IDLE, STATE_0);
16     signal state : stateType := IDLE;
17
18 begin
19
20 -- simple Mealy FSM
21 mealy_fsm_proc: process(clk)
22 begin
23     if(rising_edge(clk))then
24         case state is
25
26         when IDLE =>
27             if(sig_in = '1')then
28                 pulse_det   <= '1';
29                 state       <= STATE_0;
30             else
31                 pulse_det   <= '0';
32                 state       <= STATE_0;
33             end if;
```

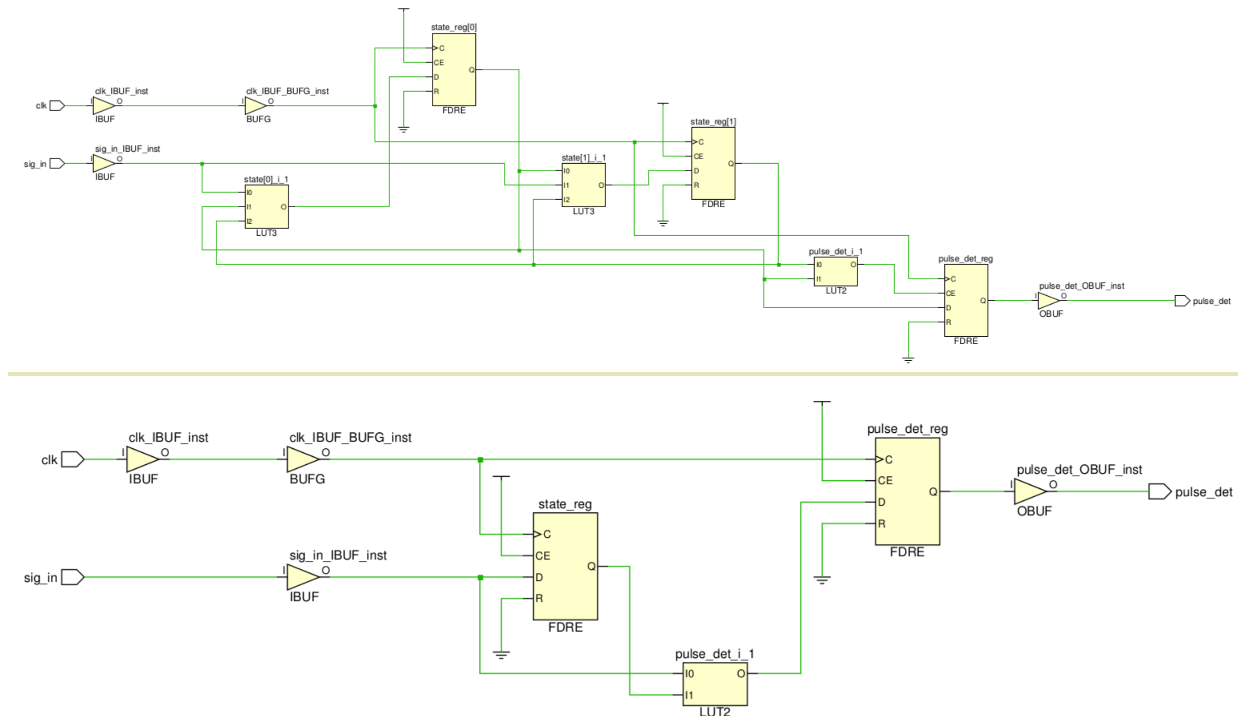
χρονική συμπεριφορά των σημάτων σε πραγματικά κυκλώματα θα μελετηθεί εκτενέστερα στο Κεφάλαιο 4.

```
34
35            when STATE_0 =>
36                pulse_det <= '0';
37                if(sig_in = '0')then
38                    state <= IDLE;
39                else
40                    state <= STATE_0;
41                end if;
42
43            when others => state <= IDLE;
44            end case;
45        end if;
46    end process;
47
48    end RTL;
```

Όπως είναι φανερό, το πλήθος των καταστάσεων είναι μικρότερο, καθώς δεν απαιτείται μία κατάσταση μόνο η οποία θα έχει σαν σκοπό να οδηγεί το σήμα εξόδου στο λογικό ένα (όπως συμβαίνει με τη Moore-FSM, στην κατάσταση *STATE_0*), αφού η Mealy-FSM αφήνει περιθώρια να αλλάξει τιμή το σήμα εξόδου και στο *IDLE* (γραμμές $27 - 33$). Το αποτέλεσμα είναι ένα ακόμα πιο οικονομικό κύκλωμα, αφού οι λιγότερες καταστάσεις θα υλοποιήσουν και λιγότερους state registers, άρα και τα κυκλώματα στο συνδυαστικό κομμάτι της FSM θα είναι πιο απλά, καθώς θα εξαρτώνται από λιγότερα inputs. Ενώ επειδή θα χρειαστεί να οδηγήσουν λιγότερους state registers, δεν θα απαιτείται να έχουν και τόσες εξόδους. Αυτό είναι εμφανές στα Synthesized Schematics των δύο τρόπων κωδικοποίησης των FSM, που παρατίθενται στο σχήμα 3.2.II. Τέλος, η προσομοίωση μέσω του Vivado® θα δείξει και τη διαφορά στη χρονική απόκριση των δύο FSM (βλ. σχ. 3.2.III).

3.2.2 Switch Debouncer

Εν γένει, οι FSM χρησιμοποιούνται κατά κόρον στα μεγάλα design, καθώς επιτρέπουν μία διαισθητική προσέγγιση σε πληθώρα προβλημάτων που πρέπει να επιλυθούν. Το παράδειγμα της προηγούμενης υποενότητας, αντιπροσώπευε ουσιαστικά ένα κύκλωμα το οποίο εντόπιζε έναν παλμό εισόδου, με τυχαίο πλάτος, και παρήγαγε έναν παλμό εξόδου διάρκειας λογικού ένα ίσου με έναν κύκλο ρολογιού

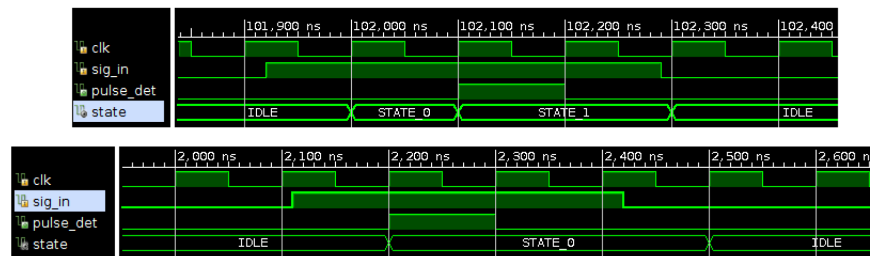


Σχήμα 3.2.II: Πάνω: Synthesized Schematic του κώδικα 3.2 (Moore-FSM). Κάτω: Synthesized Schematic του κώδικα 3.3 (Mealy-FSM). Προσέξτε τα LUT που αντιπροσωπεύουν τη συνδυαστική λογική, πως συνδέονται με τους state registers στις εισόδους και στις εξόδους τους, πάντα σε συμφωνία με τη γενικότερη ιδέα που απεικονίζεται στο σχήμα 3.2.I. Αξίζει επίσης να σημειωθεί το γεγονός ότι η Moore-FSM φαίνεται να δεσμεύει περισσότερη λογική σε σχέση με την Mealy-FSM.

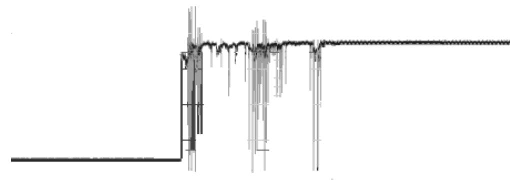
αναφοράς. Κάτι τέτοιο ίσως και να μην απαιτούσε την κωδικοποίηση FSM για να υλοποιηθεί. Όμως, αυτή η FSM αποτέλεσε μία εισήγηση στη φιλοσοφία που διέπει την αρχιτεκτονική και τη γενικότερη λογική των κυκλωμάτων αυτών. Προφανώς, ακόμα και σε αυτό το στάδιο της εργασίας, μπορεί να δοθεί ένα παράδειγμα FSM με πρακτική εφαρμογή. Ακολουθεί λοιπόν η FSM ενός *Switch Debouncer*.

Το να πατάει κανείς ένα κουμπί μίας ηλεκτρονικής συσκευής θεωρείται κάτι το τετριμμένο. Όμως στα ψηφιακά κυκλώματα, ελλοχεύει ο πολύ σημαντικός κίνδυνος του *switch bouncing* [1, 5], ο οποίος απεικονίζεται στο Σχήμα 3.2.IV.

Αυτό που συμβαίνει συνήθως, είναι οι δύο επαφές του διακόπτη να βρίσκουν και να χάνουν την μεταξύ τους σύνδεση από 10 μέχρι 100 φορές μέσα σε ένα χιλιο-



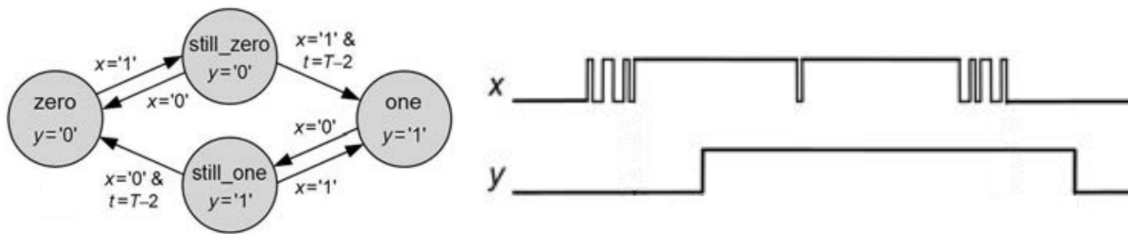
Σχήμα 3.2.III: Πάνω: Προσομοίωση της Moore-FSM. Κάτω: Προσομοίωση της Mealy-FSM, όπου φαίνεται πως το σήμα *pulse_det* αλλάζει τιμή έναν κύκλο πιο πριν σε σχέση με τη Moore-FSM. Το πλάτος του παλμού όμως διαρκεί και στις δύο περιπτώσεις για έναν κύκλο του ρολογιού.



Σχήμα 3.2.IV: Διάγραμμα τάσης προς χρόνο τη στιγμή κλεισίματος ενός συμβατικού διακόπτη.

στό του δευτερολέπτου [1], λόγω μηχανικών ατελειών. Φανταστείτε τώρα ότι κάποιος χρησιμοποιεί την έξοδο αυτού του διακόπτη για να αυξήσει την τιμή ενός μετρητή. Αν ο μετρητής αυξάνει κατά ένα μετά από κάθε μετάβαση του διακόπτη από ανοιχτό στο κλειστό, τότε το αποτέλεσμα θα είναι ο μετρητής να μην αυξάνει κατά ένα, αλλά...κατά πολύ περισσότερο. Για το λόγο αυτό, μαζί με κάθε διακόπτη, κατασκευάζεται και ένα βοηθητικό κύκλωμα που προβαίνει στο λεγόμενο *switch debouncing*. Τέτοια κυκλώματα μπορούν να βρεθούν εύκολα στη βιβλιογραφία, όμως εν προκειμένω, θα σχεδιαστεί μία απλή FSM, η οποία θα δέχεται σαν είσοδο την αλλαγή τάσης που προκαλεί το κλείσιμο του διακόπτη, και στην έξοδο θα δίνει ένα σταθερό σήμα.

Η συγκεκριμένη FSM ελέγχει την κατάσταση του σήματος εισόδου, και μόλις αυτό αλλάξει, τότε ένας μετρητής ξεκινάει να αυξάνει. Αν αυτός φτάσει στο όριό του και το σήμα εισόδου έχει παραμείνει στην αλλαγμένη κατάσταση, τότε μόνο το σήμα εξόδου θα πάρει την τιμή του σήματος εισόδου. Έτσι διασφαλίζεται ότι το σήμα εξόδου θα είναι μία σταθεροποιημένη έκδοση του σήματος εισόδου, αφού η FSM



Σχήμα 3.2.V: Αριστερά: Διάγραμμα μετάβασης καταστάσεων μίας FSM που προβαίνει σε switch debouncing. Δεξιά: Η κυματομορφή του επιθυμητού αποτελέσματος (x είναι το σήμα εισόδου ενώ y είναι το σήμα εξόδου)[5].

ουσιαστικά ”περιμένει”, για να δει αν όντως το σήμα εισόδου έχει πάρει σταθερή τιμή και δεν αλλάζει απότομα. Ο κώδικας που θα υλοποιήσει μία τέτοια FSM έχει ως εξής:

VHDL Code 3.4: Debouncing_FSM

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.all;
4
5  entity debouncer is
6  port(
7      clk     : in  std_logic;
8      sig_in  : in  std_logic;
9      sig_out : out std_logic
10 );
11 end debouncer;
12
13 architecture RTL of debouncer is
14
15     component counter_simple
16     port(
17         clk     : in  std_logic;
18         set     : in  std_logic;
19         rst     : in  std_logic;
20         cnt_val : out std_logic_vector(7 downto 0)
21     );
22     end component;
23
24     type stateType is (ZERO, STILL_ZERO, ONE, STILL_ONE);
```

```
25      signal state : stateType := ZERO;
26
27      signal rst_cnt : std_logic := '0';
28      signal set_cnt : std_logic := '0';
29      signal cnt_val : std_logic_vector(7 downto 0) := (others => '0');
30
31  begin
32
33  -- switch debouncing FSM
34  counter_proc: process(clk)
35  begin
36      if(rising_edge(clk))then
37          case state is
38
39          when ZERO =>
40              rst_cnt <= '1';
41              set_cnt <= '0';
42              sig_out <= '0';
43
44              if(sig_in = '1')then
45                  state <= STILL_ZERO;
46              else
47                  state <= ZERO;
48              end if;
49
50          when STILL_ZERO =>
51              rst_cnt <= '0';
52              set_cnt <= '1';
53              sig_out <= '0';
54
55              if(cnt_val = "11111111" and sig_in = '1')then
56                  state <= ONE;
57              elsif(sig_in = '0')then
58                  state <= ZERO;
59              else
60                  state <= STILL_ZERO;
61              end if;
62
63          when ONE =>
64              rst_cnt <= '1';
65              set_cnt <= '0';
```

```
66              sig_out <= '1';
67
68              if(sig_in = '0')then
69                  state <= STILL_ONE;
70              else
71                  state <= ONE;
72              end if;
73
74          when STILL_ONE =>
75              rst_cnt <= '0';
76              set_cnt <= '1';
77              sig_out <= '1';
78
79              if(cnt_val = "11111111" and sig_in = '0')then
80                  state <= ZERO;
81              elsif(sig_in = '1')then
82                  state <= ONE;
83              else
84                  state <= STILL_ONE;
85              end if;
86
87          when others =>
88              rst_cnt <= '0';
89              set_cnt <= '0';
90              sig_out <= '0';
91              state   <= ZERO;
92
93          end case;
94      end if;
95  end process;
96
97  counter_inst: counter_simple
98      port map(
99          clk     => clk,
100         set     => set_cnt,
101         rst     => rst_cnt,
102         cnt_val => cnt_val
103     );
104
105 end RTL;
```

Το μόνο καινούργιο που θα μπορούσε να πει κανείς ότι βλέπει σε αυτό τον κώδικα, είναι το γεγονός ότι χρησιμοποιείται ένα subcomponent, το οποίο υλοποιεί έναν απλό μετρητή. Ο κώδικας αυτού του μετρητή έχει παρατεθεί στην ενότητα 2.3.3. Σε κάθε κατάσταση, η FSM ελέγχει πλήρως τον μετρητή, αυξάνοντάς τον σε κάθε χτύπο του ρολογιού όταν χρειάζεται, ή επαναφέροντάς τον στο μηδέν όταν η τιμή του σήματος εισόδου μοιάζει να είναι σταθερή. Ο αναγνώστης παροτρύνεται να προσομοιώσει τον κώδικα αυτό, προβαίνοντας σε γρήγορες αλλαγές στο σήμα εισόδου, ώστε να αναπαράξει το φαινόμενο του switch bouncing. Σε κάθε περίπτωση, η έξοδος της FSM θα είναι ένα σταθερό σήμα, που είναι και ο λόγος ύπαρξης αυτής της λογικής.

Σύνοψη

Σε αυτό το Κεφάλαιο, έγινε μία σύντομη εισαγωγή στον τρόπο με τον οποίο μπορεί κανείς να κωδικοποιήσει απλές FSM μέσω της γλώσσας VHDL. Αυτοί οι κώδικες, μεταφράζονται από εξειδικευμένο λογισμικό σε πύλες συνδυαστικής λογικής και σε καταχωρητές, προκειμένου να υλοποιήσουν την FSM μέσα στο FPGA. Τα απλά παραδείγματα που συνάντησε ο αναγνώστης όμως, δεν αρκούν για να ξεκινήσει την κωδικοποιεί μηχανές πεπερασμένων καταστάσεων που θα είναι αρκετά αξιόπιστες για να χρησιμοποιηθούν σε πραγματικά design. Στο επόμενο Κεφάλαιο όμως, που θα διαπιστωθεί η διαφορά της θεωρίας από την πράξη, μπορεί να βρει κανείς τις πληροφορίες που θα τον βοηθήσουν να κατασκευάζει αποδοτικά ψηφιακά κυκλώματα.

4

Εφαρμογές σε Πραγματικά Κυκλώματα - Ζητήματα Χρονισμού

Στη μέχρι τώρα ανάλυση της εργασίας αυτής, έχουν μελετηθεί συνδυαστικά κυκλώματα, καθώς και σύγχρονα ακολουθιακά κυκλώματα που λειτουργούν με μία συχνότητα ρολογιού αναφοράς. Συνήθως, η διαδικασία που ακολουθείται για να επιλυθεί ένα πρόβλημα, περιλαμβάνει το σχεδιασμό ενός αλγορίθμου, και εν συνεχεία μίας Μηχανής Πεπερασμένων Καταστάσεων (FSM), η οποία θα υλοποιεί τον αλγόριθμο αυτό, σε πραγματικό κύκλωμα. Η FSM, αποτελείται από πύλες flip-flop οι οποίες καταχωρούν την κατάσταση που βρίσκεται το σύστημα, και από πύλες συνδυαστικής λογικής, που επικοινωνούν τόσο με τους καταχωρητές της κατάστασης, όσο και με εξωτερικά σήματα, προκειμένου να καθορίσουν την επόμενη κατάσταση της FSM. Κάτι τέτοιο όμως, θα μπορούσε να αναπτυχθεί και με μία γλώσσα λογισμικού, και όχι μόνο με τη VHDL (ή τη Verilog). Τι είναι λοιπόν αυτό που κάνει αυτές τις δύο γλώσσες να διαφέρουν από όλες τις υπόλοιπες, πέρα από το προφανές, ότι δηλαδή οι HDL υλοποιούνται σε ένα FPGA, ενώ οι γλώσσες software σε μία κεντρική μονάδα επεξεργασίας; Τι είναι αυτό που κάνει τελικά το έργο ενός FPGA designer να διαφέρει τόσο πολύ από ενός software developer;

Η απάντηση έχει τις ρίζες της στο γεγονός ότι οι γλώσσες περιγραφής hardware, αφήνουν πλήρη ελευθερία στο πως μπορεί να σχεδιαστεί ένα κύκλωμα που υλοποιεί έναν αλγόριθμο, σε πολύ χαμηλό επίπεδο, και συγκεκριμένα σε επίπεδο καταχωρητών και λογικών πυλών. Αυτή την ελευθερία δεν μπορεί να την προσφέρει μία γλώσσα ανάπτυξης λογισμικού, και αυτή ακριβώς η ελευθερία είναι που δίνει

τη δυνατότητα στις γλώσσες ανάπτυξης firmware να βελτιστοποιήσουν στο μέγιστο τη διαδικασία υλοποίησης ενός αλγορίθμου. Με τη μεγάλη δύναμη όμως...έρχονται και οι ευθύνες, που για να τις επωμιστεί κανείς, πρέπει κατ'αρχάς να γνωρίζει *τι ακριβώς* είναι αυτό που έχει γράψει σε VHDL, δηλαδή σε τι πύλες και καταχωρητές θα μεταφραστεί ο κώδικάς του, και πως αυτά τα επιμέρους στοιχεία θα συνδέονται μεταξύ τους. Το επόμενο βήμα από αυτό, θα μπορέσει να παρθεί μόνο αν γίνουν αντιληπτοί οι εγγενείς περιορισμοί των ψηφιακών κυκλωμάτων μεγάλης κλίμακας, που πηγάζουν κατά κύριο λόγο από δύο γεγονότα:

- Η αλλαγή τάσης στο ένα άκρο ενός αγωγού, θα μεταδοθεί στο άλλο του άκρο *σε πεπερασμένο χρόνο*.

- Αν η κατάσταση των εισόδων μία πύλης αλλάξει, η μεταβολή κατάστασης του σήματος (ή των σημάτων) εξόδου της πύλης θα λάβει χώρα *σε πεπερασμένο χρόνο*.

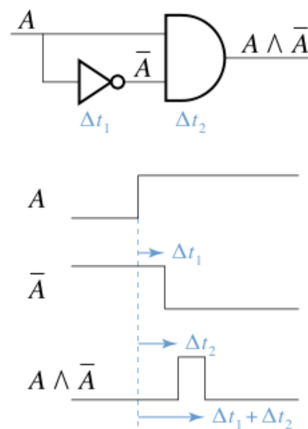
Αποδεχόμενοι λοιπόν ότι οι αλλαγές καταστάσεων στα πραγματικά κυκλώματα δεν λαμβάνουν χώρα ακαριαία, μένει να αναφερθούν οι συνέπειες που προκύπτουν από αυτό το γεγονός.

4.1 Μετασταθείς Καταστάσεις

Προκειμένου ένα κύκλωμα ψηφιακής λογικής να λειτουργήσει σωστά κάτω από όλες τις πιθανές συνθήκες και μεταβολές τάσεων, ο σχεδιαστής οφείλει να λάβει υπόψιν του τις καθυστερήσεις που είναι συνυφασμένες με όλα τα στοιχεία που αποτελούν το σύστημα. Αν η καθυστέρηση μίας λογικής διασύνδεσης είναι υπερβολικά μεγάλη ή μικρή, τότε υπάρχει περίπτωση ένα σήμα του κυκλώματος να λάβει λάθος τιμή σε κάποιο κρίσιμο χρονικό σημείο, προκαλώντας έτσι στιγμιαία, ή ακόμα και σε ακραίες περιπτώσεις μόνιμα, σφάλματα. Αυτά τα σφάλματα καλούνται *σφάλματα χρονισμού (timing errors)* [12]. Όλα τα ψηφιακά κυκλώματα είναι ευαίσθητα σε τέτοια σφάλματα, και μόνο μετά την πλήρη εξάλειψη της πιθανότητας εμφάνισης τέτοιων σφαλμάτων, μπορεί να εξασφαλίσει κανείς την ομαλή λειτουργία ενός design κάτω από όλες τις δυνατές συνθήκες.

4.1.1 Συνδυαστικά Κυκλώματα - Race Conditions

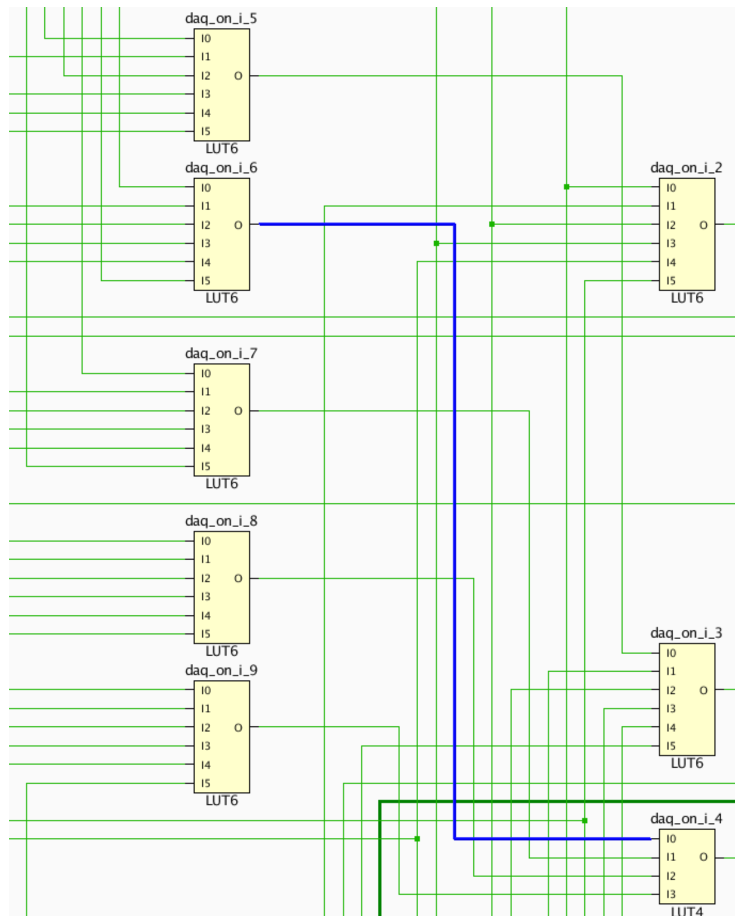
Συγκρίνοντας τις δύο μεγάλες οικογένειες ψηφιακών κυκλωμάτων, τα συνδυαστικά και τα σύγχρονα ακολουθιακά, μπορεί κανείς να υποθέσει ότι τα συνδυαστικά, όντας ανεξάρτητα του χρόνου, δεν είναι επιρρεπή σε σφάλματα χρονισμού. Αυτό όμως είναι μία λανθασμένη υπόθεση, καθώς *όλες* οι λογικές πύλες είναι επιρρεπείς σε τέτοιου είδους προβλήματα. Στα συνδυαστικά κυκλώματα μάλιστα, τα χρονικά σφάλματα ονομάζονται και *race conditions*. Ένα πολύ απλό παράδειγμα μπορεί να φανεί στο σχήμα 4.1.I.



Σχήμα 4.1.I: Race condition σε μία πύλη AND. Προφανώς, αν οι πύλες ήταν ιδανικές, τότε η έξοδος της AND θα ήταν πάντα μηδέν. Όμως επειδή ο αναστροφέας εισάγει μία καθυστέρηση Δt_1, όταν η κατάσταση του σήματος εισόδου (A) αλλάξει, το συμπληρωματικό $(\overline{A})$ του θα φτάσει με διαφορά χρόνου Δt_1 στην είσοδο της AND σε σχέση με αυτό. Το αποτέλεσμα είναι να εμφανίζεται στην έξοδο της AND το λογικό ένα για πολύ μικρό χρονικό διάστημα, γεγονός ανεπιθύμητο και μη συμβατό με τη λογική του κυκλώματος. Αξίζει να σημειωθεί ότι και η πύλη AND εισάγει μία καθυστέρηση της τάξης Δt_2 ως προς την ανανέωση του σήματος εξόδου.

Τέτοια φαινόμενα συμβαίνουν κατά κόρον στα ψηφιακά κυκλώματα συνδυαστικής λογικής, και γίνονται περισσότερο αισθητά όσο μεγαλώνει η κλίμακα του design. Σε ένα FPGA, πολλά LUT θα συνδέονται μεταξύ τους για να τελέσουν μία περίπλοκη συνδυαστική συνάρτηση με πολλές εισόδους και εξόδους. Καθώς όμως μεγαλώνει το πλήθος των λογικών κελιών και των μεταξύ τους διασυνδέσεων, τόσο φθίνουν και οι διαθέσιμες συνδεσμολογίες μεταξύ των κελιών αυτών. Έτσι δημιουργούνται

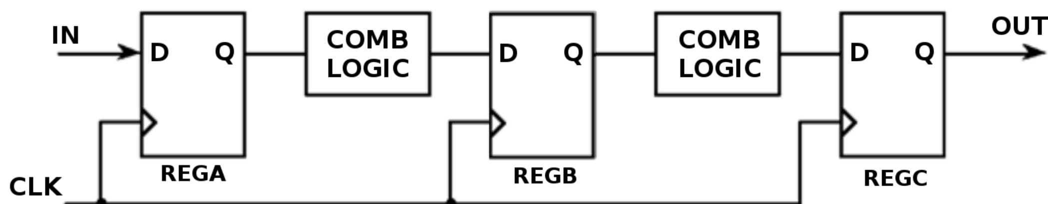
προβλήματα *συνωστισμού (congestion)* μέσα στο FPGA, το οποίο συνεπάγεται ότι θα υπάρχουν και διαφοροποιήσεις μεταξύ των καθυστερήσεων στα επιμέρους σήματα. Όσο αυτές οι χρονικές διαφορές μεγαλώνουν, τόσο αυξάνει και η πιθανότητα να δημιουργούνται σφάλματα χρονισμού, που εν προκειμένω καλούνται και *glitches*. Στην εικόνα 4.1.II μπορεί να δει κανείς ένα κομμάτι design όπου μερικά LUT συνδέονται μεταξύ τους, ενώ το συγκεκριμένο κομμάτι απεικονίζεται στο πραγματικό FPGA στην εικόνα 4.1.III. Ένα τέτοιο κύκλωμα θα διατρέχει κίνδυνο εμφάνισης glitch σε διάφορες εξόδους των LUT του.



Σχήμα 4.1.II: Implemented schematic μερικών αλληλοσυνδεδεμένων LUT.

Σχήμα 4.1.III: Πραγματικό κύκλωμα σε FPGA του schematic από το σχήμα 4.1.II. Κάτω φαίνονται τα CLB του FPGA fabric (βλ. και σχήμα 1.3.III), των οποίων τα LUT συνδέονται μεταξύ τους (τα άσπρα βέλη υποδεικνύουν τη συνδεσμολογία). Πάνω μπορεί κανείς να δει το πραγματικό routing που συνδέει τα λογικά κελιά μεταξύ τους. Πρώτα το σήμα οδηγείται στο τοπικό router, ο οποίος θα προωθήσει το σήμα στο κατάλληλο LUT. Σε αυτό το σχήμα μπορεί κανείς να δει το πρόβλημα του συνωστισμού, που προκαλείται όταν πολλά λογικά κελιά καλούνται να συνδεθούν μεταξύ τους. Τότε είναι που τα race conditions στα συνδυαστικά κυκλώματα κυριαρχούν, και μπορεί να κρίνουν την τύχη ολόκληρου του design.

Δυστυχώς, τα glitches δεν μπορούν να αποφευχθούν εντελώς, όμως αυτό που μπορεί να γίνει είναι να περιοριστούν σε συγκεκριμένες περιοχές του κυκλώματος, όπου υπάρχουν μόνο συνδυαστικά κελιά, και πριν περάσουν σε κάποιο άλλο κομμάτι του design, να καταχωρούνται από πύλες flip-flop. Αυτή η τεχνική ονομάζεται *pipelining* και χρησιμοποιείται κατά κόρον ώστε να μειώνονται οι κίνδυνοι των race conditions που προκύπτουν από κυκλώματα συνδυαστικής λογικής μεγάλης κλίμακας (βλ. σχ. 4.1.IV).



Σχήμα 4.1.IV: Pipelined design.

4.1.2 Σύγχρονα Ακολουθιακά Κυκλώματα - Metastability

Ο κίνδυνος του λεγόμενου *metastability*, δηλαδή το να πέσει μία πύλη flip-flop σε *μετασταθή* κατάσταση είναι ο πλέον χαρακτηριστικός όταν μιλάει κανείς για χρονικά σφάλματα σε ένα ψηφιακό κύκλωμα ακολουθιακής λογικής. Μέχρι τώρα στην ανάλυση αυτής της εργασίας, είχε ληφθεί η παραδοχή ότι οι flip-flop registers είναι ιδανικοί. Κάτι τέτοιο όμως δεν έχει σχέση με την πραγματικότητα, καθώς υπάρχουν τρεις τιμές που χαρακτηρίζουν την κάθε πύλη flip-flop:

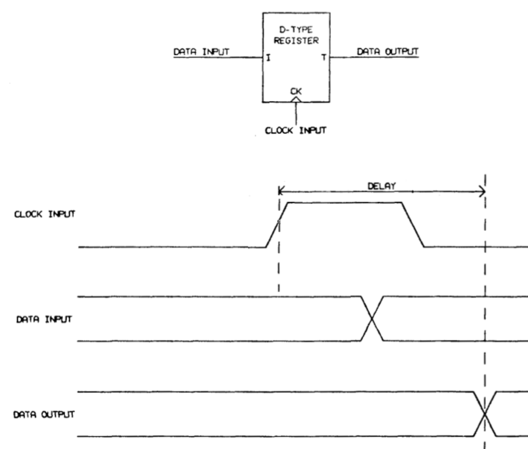
- *Setup Time:* Πρόκειται για ένα χρονικό διάστημα *πριν* από το rising edge του παλμού του ρολογιού, μέσα στο οποίο *δεν* πρέπει να αλλάξει κατάσταση το σήμα εισόδου (D). Συμβολίζεται με t_s .

- *Hold Time:* Πρόκειται για ένα χρονικό διάστημα *μετά* από το rising edge του παλμού του ρολογιού, μέσα στο οποίο *δεν* πρέπει να αλλάξει κατάσταση το σήμα εισόδου (D). Συμβολίζεται με t_h .

- *Delay Time:* Πρόκειται για το χρονικό διάστημα που απαιτείται για να ανανεωθεί η έξοδος Q της πύλης flip-flop, σε σχέση με το πιο πρόσφατο rising edge του ρολογιού. Συμβολίζεται με t_d . Καλείται επίσης και *clock-to-output delay.*

Αν εξετάσει κανείς τις προσομοιώσεις των προηγούμενων κεφαλαίων (το σχήμα 2.3.III για παράδειγμα), θα προσέξει ότι την *ίδια* στιγμή που εμφανίζεται το μέτωπο του παλμού του ρολογιού, η κατάσταση του Q στην πύλη flip-flop αλλάζει. Κάτι τέτοιο δε συμβαδίζει με την πραγματικότητα, αφού υπάρχει το *delay time*. Ο χρόνος που απαιτείται για να αλλάξει τιμή το σήμα εξόδου, είναι το t_d .

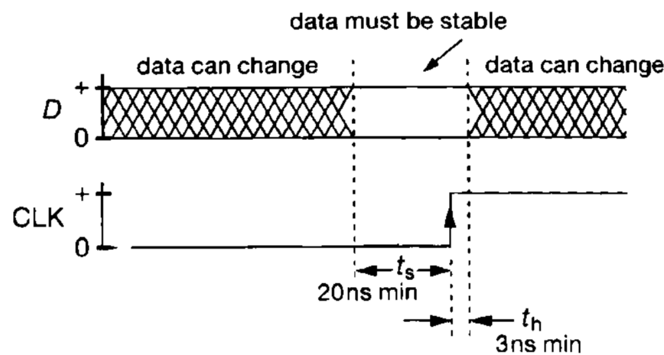
Ως προς τις τιμές setup/hold, το άθροισμα των χρόνων t_s και t_h ορίζει ένα *παράθυρο (aperture window)*, μέσα στο οποίο το σήμα εισόδου *δεν* πρέπει να αλλάξει κατάσταση, προκειμένου η flip-flop να λειτουργήσει σωστά, δηλαδή *ακριβώς* μετά το χρόνο t_d , να έχει περάσει την τιμή του D στο Q.



Σχήμα 4.1.V: Η τιμή του σήματος εξόδου καθυστερεί να ανανεωθεί, και δεν αλλάζει άμεσα τη στιγμή που εμφανίζεται το rising edge του ρολογιού [12].

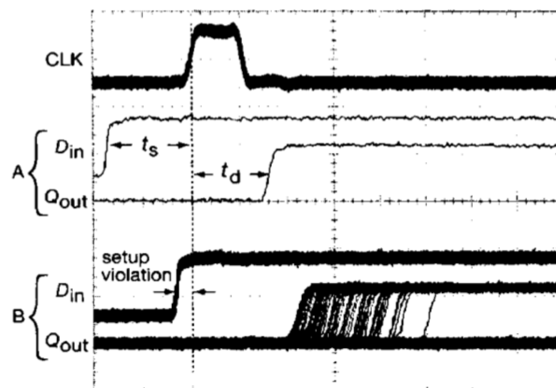
Το ερώτημα που προκύπτει αμέσως όμως είναι το εξής: Τι θα συμβεί αν παραβιαστεί η απαίτηση του setup/hold, δηλαδή τι θα συμβεί αν αλλάξει κατάσταση του σήματος εισόδου μέσα στο παράθυρο της flip-flop; Η απάντηση είναι ότι σε αυτή την περίπτωση η flip-flop θα μπει σε *μετασταθή (metastable)* κατάσταση, δηλαδή ουσιαστικά *δεν θα μπορεί να αποφασίσει* αν πρέπει να περάσει το λογικό ένα ή το λογικό μηδέν στην έξοδό της. Αν γίνει μετάβαση από το μηδέν στο ένα μέσα στο παράθυρο της flip-flop, τότε η έξοδος θα πάρει την τιμή του λογικού ένα, απλά σε χρόνο μεγαλύτερο του t_d . Αν για παράδειγμα στη 74HC74[1] flip-flop παραβιαστεί η συνθήκη setup/hold, τότε το σήμα εξόδου θα πάρει την σωστή τιμή σε χρόνο μεγαλύτερο του t_d . Επίσης, αυτός ο επιπλέον χρόνος δεν μπορεί να προσδιοριστεί με σαφήνεια. Στο

[1] $t_d = 16$ ns και παράθυρο 23 ns.



Σχήμα 4.1.VI: Μία πύλη flip-flop με κωδικό 74HC74, η οποία έχει $t_s = 20\,\text{ns}$ και $t_h = 3\,\text{ns}$. Το παράθυρό της θα έχει συνολικό άνοιγμα 23 ns, και το σήμα εισόδου D δεν πρέπει να αλλάξει κατάσταση μέσα σε αυτό το παράθυρο [1].

σχήμα 4.1.VII δίνεται μία γενική εικόνα των σημάτων εισόδου-εξόδου της flip-flop όταν παραβιάζεται η συνθήκη του setup/hold.



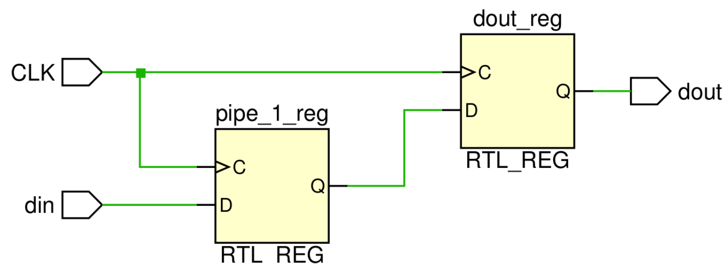
Σχήμα 4.1.VII: Σήματα εισόδου/εξόδου μαζί με το ρολόι σε μία 74HC74 flip-flop. Η κλίμακα του χρόνου είναι 20 ns. Στην περίπτωση A, δεν παραβιάζεται η συνθήκη του setup, καθώς η τιμή του σήματος εισόδου αλλάζει πριν το t_s, και επομένως το σήμα εξόδου λαμβάνει την τιμή ένα μετά από t_d. Στην περίπτωση B όμως, το σήμα εισόδου μεταβάλλεται από μηδέν σε ένα μέσα στο χρόνο t_s, με αποτέλεσμα η πύλη να μπαίνει σε μετασταθή κατάσταση. Το σήμα εξόδου αλλάζει μετά από $18 - 35\,\text{ns}$, λόγω της παραβίασης της συνθήκης setup. Στη συγκεκριμένη μέτρηση ελήφθησαν δεδομένα για 2 δευτερόλεπτα [1].

4.2 Στατική Χρονική Ανάλυση και Vivado®

Μέχρι τώρα, ο κίνδυνος των χρονικών σφαλμάτων έχει μελετηθεί σε μικροσκοπικό επίπεδο, και όχι σε κλίμακα μεγάλων και πολύπλοκων κυκλωμάτων. Ο μόνος τρόπος να γίνει αντιληπτός ο αντίκτυπος που μπορεί να έχει ένα χρονικό σφάλμα, είναι να γίνει μία σύντομη αναφορά στον τρόπο που το Vivado® προβαίνει στη *στατική χρονική ανάλυση (static timing analysis)* ενός design, προκειμένου να αποφανθεί αν αυτό είναι επιρρεπές σε χρονικά σφάλματα ή όχι.

4.2.1 Κοινό Πεδίο Χρονισμού

Η στατική χρονική ανάλυση πραγματώνεται αμέσως μετά τη σύνθεση. Έστω ένα πολύ απλό design, το οποίο έχει μελετηθεί στην υποενότητα 2.3.2, και υλοποιείται από τον κώδικα 2.7. Πρόκειται για δύο flip-flop registers, όπου η έξοδος του ενός συνδέεται με την είσοδο του άλλου, ενώ έχουν και κοινό ρολόι. Οι δύο αυτοί registers επομένως βρίσκονται σε *κοινό πεδίο χρονισμού (common/single clock domain)*. Τέλος, το schematic τους παρατίθεται εδώ για διευκόλυνση (βλ. σχ. 4.2.Ι).



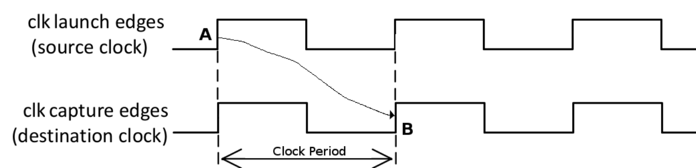
Σχήμα 4.2.Ι: RTL Schematic του κώδικα 2.7.

Όταν η έξοδος του pipe_1 _reg αλλάξει κατάσταση, για παράδειγμα από το λογικό μηδέν στο λογικό ένα, τότε προκειμένου να μην υπάρξει χρονικό σφάλμα, πρέπει ο dout_reg να δειγματίσει αυτό το λογικό ένα πριν από το t_s του. Μετά τη σύνθεση, το Vivado® θα αντιστοιχίσει αυτούς τους δύο synthesized registers σε πραγματικούς registers στο FPGA fabric (εν προκειμένω, FDREs), και μετά θα αναλάβει να τους τοποθετήσει σε κάποιο σημείο μέσα στο τσιπ, και να τους ενώσει μεταξύ τους. Το που ακριβώς θα μπουν, και πόση πρέπει να είναι η απόσταση των διασυνδέσεών τους, καθορίζεται από το λεγόμενο *slack*. Αν το slack είναι θετικό, τότε η διασύνδεση

(ή και *timing path*), κρίνεται ασφαλής. Αν από την άλλη προκύψει αρνητική τιμή, τότε το Vivado® θα επανατοποθετήσει τους καταχωρητές μέχρι η τιμή του slack να γίνει θετική. Μία απλοποιημένη σχέση του slack έχει ως εξής[2]. [15]:

$$\text{Slack} = \text{ClockPeriod} - \text{ClockUncertainty} - \text{SetupTime} - \text{DatapathDelay} \tag{4.2.1}$$

Η τιμή του *Clock Uncertainty* είναι η αβεβαιότητα προσδιορισμού της ακριβής θέσης των μετώπων του ρολογιού, δηλαδή το *jitter*. Το *Setup Time* είναι εγγενής ιδιότητα των flip-flop που εξετάζει η μηχανή ανάλυσης χρονισμού του Vivado® . Η Xilinx® παρέχει registers υψηλής ποιότητας, με $t_s < 100\,\text{ps}$[3]! Η τιμή του *Datapath Delay* είναι ουσιαστικά ο χρόνος που απαιτείται για να μεταδοθεί η αλλαγή τάσης από το Q pin του register που στέλνει το σήμα, στο D pin του καταχωρητή που το λαμβάνει, μαζί με το t_d, που είναι εγγενής ιδιότητα των καταχωρητών, όπως το t_s.



Σχήμα 4.2.II: Το ρολόι των δύο καταχωρητών είναι κοινό, επομένως η κυματομορφή είναι ίδια. Πάνω βρίσκεται η μορφή του ρολογιού όπως "το βλέπει" ο πρώτος καταχωρητής που στέλνει το σήμα (pipe_1 _reg), και κάτω η αντίστοιχη κυματομορφή για τον καταχωρητή που λαμβάνει το σήμα (dout_reg). Αν στο χρονικό σημείο A σταλεί μία αλλαγή κατάστασης, τότε στο χρονικό σημείο B ο δεύτερος register θα καταχωρήσει αυτή την αλλαγή. Η χρονική απόσταση μεταξύ των δύο διαδοχικών rising edges προφανώς ταυτίζεται με την περίοδο του κοινού ρολογιού.

Όταν οι καταχωρητές χρονίζονται από το ίδιο ρολόι, όπως συμβαίνει εδώ, τότε αυτό που καθορίζει κατά κύριο λόγο το αν θα προκύψει θετικό slack ή όχι, είναι η *περίοδος* του κοινού ρολογιού. Ένα αργό ρολόι, θα δίνει αυτόματα μεγάλη τιμή στο *Clock Period*, που είναι ο μόνος θετικός όρος της σχέσης 4.2.1. Ένα γρήγορο ρολόι όμως, θα δώσει μικρότερο *Clock Period*, μειώνοντας έτσι τα περιθώρια στο slack να πάρει

[2]Για την πλήρη σχέση ο αναγνώστης παραπέμπεται στο [15].
[3]Παρ' όλα αυτά, οι περιπτώσεις όπου μπορεί να εμφανιστούν μετασταθείς καταστάσεις σε ένα Xilinx® FPGA δεν είναι αμελητέες.

θετική τιμή. Σημαντικό ρόλο επίσης παίζει και το jitter, καθώς μία κακής ποιότητας πηγή χρονισμού θα αυξάνει την αβεβαιότητα του ρολογιού που συγχρονίζει τους καταχωρητές. Έχοντας αυτές τις τιμές κατά νου, το Vivado® υπολογίζει το περιθώριο που μπορεί να δώσει στο μήκος της διασύνδεσης μεταξύ των καταχωρητών (δηλαδή την τιμή του *Datapath Delay*).

Το τελικό συμπέρασμα επομένως, είναι το εξής: Σε γενικές γραμμές, όταν γίνεται χρήση ρολογιών μεγάλης συχνότητας, οι καταχωρητές που χρονίζονται από αυτό το ρολόι θα τοποθετηθούν κοντά μεταξύ τους, ενώ για χρονισμό χαμηλής συχνότητας, τα περιθώρια είναι λιγότερο αυστηρά καθώς το slack παίρνει πιο εύκολα θετική τιμή. Σε αυτή την περίπτωση οι καταχωρητές μπορούν να έχουν μεγαλύτερη απόσταση μεταξύ τους.

Φυσικά όμως, υπάρχουν και περισσότερο περίπλοκα design, πέρα από δύο registers που έχουν συνδεθεί μεταξύ τους. Αν λάβει κανείς υπόψιν του το σχήμα 4.1.IV, θα μπορέσει να φανταστεί το επόμενο επίπεδο ως προς την πολυπλοκότητα της στατικής χρονικής ανάλυσης. Εν δυνάμει, οι καταχωρητές REGA και REGB είναι συνδεδεμένοι μεταξύ τους, ακριβώς όπως στο προηγούμενο παράδειγμα του σχήματος 4.2.I. Όμως ανάμεσά τους δεν βρίσκεται απλά ένας αγωγός που υλοποιεί τη διασύνδεση, αλλά κελιά συνδυαστικής λογικής, δηλαδή LUTs. Προφανώς η τιμή του slack εξακολουθεί να υπακούει τη σχέση 4.2.1, όμως πλέον, το αν θα προκύψει αρνητική τιμή ή όχι, δεν εξαρτάται μόνο από την περίοδο του κοινού ρολογιού CLK, αλλά και από την πολυπλοκότητα της συνδυαστικής λογικής. Κάθε λογικό κελί θα προσθέτει και καθυστέρηση[4], και όσο περισσότερες λογικές πράξεις πρέπει να γίνουν, τόσο θα αυξάνεται η τιμή του *Datapath Delay* στη σχέση 4.2.1. Επίσης, όσο αυξάνεται η πολυπλοκότητα της συνδυαστικής λογικής, τόσο περισσότερο routing δεσμεύεται στην περιοχή που υλοποιείται το κύκλωμα, με αποτέλεσμα να γίνεται όλο και πιο δύσκολο να συνδεθούν με σωστό τρόπο οι καταχωρητές μεταξύ τους, έτσι ώστε να μην προκαλούνται σφάλματα χρονισμού.

Υπάρχουν περιπτώσεις στις οποίες το Vivado® αδυνατεί να δώσει θετική τιμή στο slack για κάποιο timing path, δοθέντων όλων των παραμέτρων της σχέσης 4.2.1, εκτός προφανώς από το *Datapath Delay* το οποίο είναι στον έλεγχο του *placer* και του *router*[5]. Σε αυτή την περίπτωση, στο τέλος της διαδικασίας του implementation,

---

[4]Θυμηθείτε το σχήμα 4.1.I όπου ένας απλός αναστροφέας επιβαρύνει με καθυστέρηση το σήμα.

[5]Βλ. ενότητα 1.3.

το Vivado® αναφέρει τις προβληματικές διασυνδέσεις, και την αρνητική τιμή του slack. Εδώ είναι που ο σχεδιαστής του κυκλώματος έχει την ευθύνη να αλλάξει την τιμή αυτή σε θετική, και αυτό μπορεί να το κάνει π.χ. χαμηλώνοντας τη συχνότητα του ρολογιού, ή απλοποιώντας τη λογική που δημιουργεί μεγάλο *Datapath Delay*.

Αρνητικό slack για κάποιο timing path σημαίνει ένα εκ των παρακάτω:

- Έχει παραβιαστεί η συνθήκη setup/hold για τον register που βρίσκεται στο τέλος του timing path, δηλαδή για αυτόν που δέχεται το σήμα στο D pin του.
- Για μία διασύνδεση κοινού πεδίου χρονισμού, εννοείται ότι όταν στέλνονται δεδομένα από τον έναν καταχωρητή στον άλλο, ο δέκτης θα λαμβάνει τα δεδομένα στο επόμενο rising edge του ρολογιού από εκείνο που εστάλησαν (ομοίως με το σχ. 4.2.II). Για αρνητικό slack, υπάρχει περίπτωση να μην παραβιάζεται η συνθήκη setup/hold, αλλά να "αργεί" τόσο πολύ το σήμα που να δειγματίζεται από ένα από τα επόμενα rising edges του ρολογιού του δέκτη.

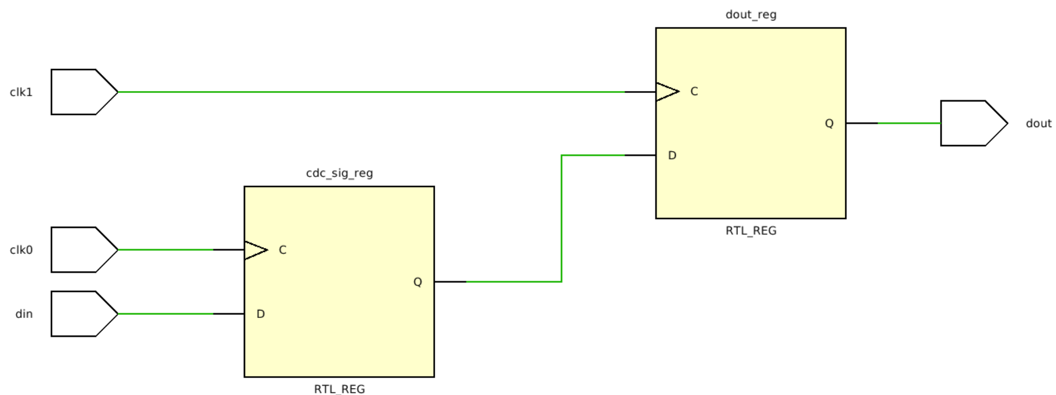
Οποιαδήποτε από τις δύο περιπτώσεις και να ισχύει, τα προβλήματα που θα μπορούσαν να προκληθούν είναι πολυάριθμα. Αν για παράδειγμα ένα timing path που ανήκει σε κάποιο κύκλωμα που υλοποιεί έναν μετρητή έχει αρνητικό slack, υπάρχει πάντα ο κίνδυνος ο μετρητής αυτός να μην αυξάνει κατά ένα με κάθε rising edge του ρολογιού που χρονίζεται. Επίσης, αν το κύκλωμα που τροφοδοτεί με το σήμα *WR_EN* μία μνήμη FIFO (βλ. ενότητα 1.1.2) έχει αρνητικό slack, σημαίνει ότι υπάρχει κίνδυνος να μην καταχωρούνται σωστά τα δεδομένα στη μνήμη, το οποίο θα οδηγήσει σε αλλοίωση των δεδομένων *(data corruption)*.

4.2.2 Διαφορετικό Πεδίο Χρονισμού

Τι θα συμβεί όμως αν δύο διαδοχικοί registers, όπως αυτοί του σχήματος 4.2.I είναι χρονισμένοι από διαφορετικά ρολόγια, τα οποία όμως παράγονται από το ίδιο MMCM[6]; Εδώ είναι που τα πράγματα γίνονται περισσότερο περίπλοκα, και συμβαίνει αυτό που ονομάζεται *Clock Domain Crossing (CDC)*. Στο CDC, ένα σήμα που δημιουργείται από κάποιον register συγχρονισμένο με ένα ρολόι clk0, καταχωρείται

[6]Αν δύο ρολόγια έχουν κοινή πηγή, τότε το Vivado® μπορεί να προβεί σε υπολογισμούς για τον υπολογισμό του slack. Αν όχι, τότε η κατάσταση περιπλέκεται περισσότερο, και θα γίνει αντιληπτό στη συνέχεια.

από έναν register συγχρονισμένο από ένα ρολόι clk1. Το schematic αυτού του κυκλώματος απεικονίζεται στο 4.2.III, ενώ ο VHDL κώδικας που το περιγράφει βρίσκεται στο αντίστοιχο Παράρτημα (βλ. κώδικα Γ΄.9).



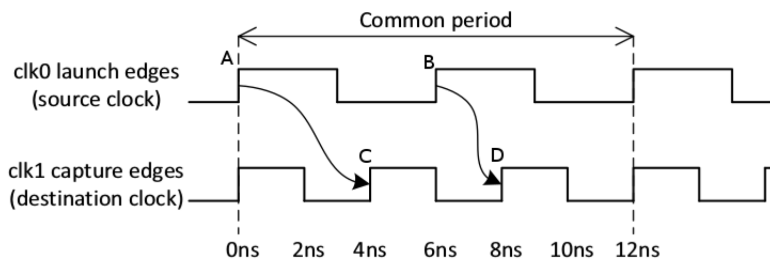
Σχήμα 4.2.III: RTL Schematic δύο διαδοχικών καταχωρητών που ανήκουν σε διαφορετικά πεδία χρονισμού.

Πλέον, επειδή τα ρολόγια είναι διαφορετικά, η εξίσωση 4.2.1 θα μετατραπεί σε:

$$\text{Slack} = \text{CaptureEdgeTime} - \text{LaunchEdgeTime} - \text{ClockUncertainty} - \text{SetupTime} - \text{DatapathDelay} \tag{4.2.2}$$

Το Vivado® θεωρεί ένα χρονικό σημείο μηδέν στο οποίο τα rising edges των ρολογιών συμπίπτουν, και μετά υπολογίζει την *κοινή περίοδο (common period)* των δύο ρολογιών, δηλαδή το χρονικό διάστημα που περνάει μέχρι τα μέτωπά τους να συμπίπτουν ξανά. Μέσα σε αυτό το διάστημα, το Vivado® βρίσκει το χειρότερο σενάριο, που καθορίζεται από τη διαφορά των *Capture Edge Time (CET)* και *Launch Edge Time (LET)*. Όταν αυτή η διαφορά γίνει ελάχιστη, τότε το slack του συγκεκριμένου *timing path* θα υπολογιστεί για αυτή την περίπτωση και μόνο, αφού αυτή διατρέχει και το μεγαλύτερο κίνδυνο να παρουσιάσει σφάλμα χρονισμού. Οι υπόλοιποι όροι της εξίσωσης 4.2.2 είναι ίδιοι με την προηγούμενη εξίσωση. Οι κυματομορφές των δύο ρολογιών απεικονίζονται στο σχήμα 4.2.IV, όπου γίνεται και περισσότερο ξεκάθαρη η παραπάνω περιγραφή.

Όταν βρεθεί η μικρότερη διαφορά χρόνων, τότε το Vivado® , όπως στην προηγούμενη περίπτωση, θα προσπαθήσει να υλοποιήσει τη βέλτιστη διασύνδεση, προκειμένου η εξίσωση 4.2.2 να πάρει θετική τιμή. Στην προκείμενη περίπτωση όπου υπάρ-



Σχήμα 4.2.IV: Κυματομορφές των ρολογιών που χρονίζουν τους καταχωρητές του σχήματος 4.2.III. Απεικονίζεται η κοινή περίοδος, και το χειρότερο δυνατό σενάριο κατά το οποίο ο ένας καταχωρητής θα στέλνει δεδομένα στον άλλο. Στην μία περίπτωση, για τα edges A->C, $\text{CET} - \text{LET} = 4\,\text{ns}$ ενώ για τα B->D, $\text{CET} - \text{LET} = 2\,\text{ns}$ [15].

χουν δύο πεδία χρονισμού, οι πιθανότητες το slack να προκύψει αρνητικό είναι μεγαλύτερες σε σχέση με το αν το ρολόι ήταν κοινό, καθώς εδώ παίζει σημαντικό ρόλο η *σχέση* μεταξύ των δύο ρολογιών. Αν για παράδειγμα τα δύο ρολόγια έχουν αρκετά διαφορετικές συχνότητες, τότε η διαφορά CET – LET σίγουρα θα πάρει πολύ μικρή τιμή στο χειρότερο σενάριο, αναγκάζοντας έτσι το Vivado® να τοποθετήσει τους δύο καταχωρητές σε όσο το δυνατό μικρότερη απόσταση, με την ελπίδα να μην υπάρξει σφάλμα χρονισμού.

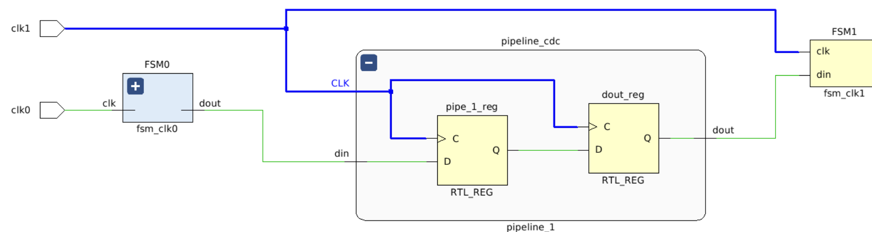
4.2.3 Αντιμετωπίζοντας τα Σφάλματα Χρονισμού

Έχει τονιστεί προηγουμένως ότι αυτοί οι υπολογισμοί μπορούν να γίνουν μόνο αν τα δύο ρολόγια έχουν κοινή πηγή, καθώς μόνο τότε το Vivado® μπορεί να βρει την κοινή τους περίοδο. Αν τα δύο ρολόγια έχουν διαφορετικές πηγές[7], ή ακόμα και αν έχουν κοινή πηγή, η διαφορά συχνοτήτων των ρολογιών είναι τόσο μεγάλη που δεν βρίσκεται κοινή περίοδος μετά από 1000 κύκλους και των δύο ρολογιών, τότε τα δύο ρολόγια θεωρούνται *ασύγχρονα (asynchronous/unexpendable clocks)* [15]. Όταν καταχωρητές που ανήκουν σε αυτά τα πεδία χρονισμού ανταλλάσσουν σήματα μεταξύ τους, τα σφάλματα χρονισμού είναι αναπόφευκτα. Σύμφωνα με τη βιβλιογραφία [13, 14, 15, 18], *ειδικά κυκλώματα συγχρονισμού* που εξασφαλίζουν ότι η λογική του πεδίου χρονισμού που υποδέχεται το σήμα θα δεχτεί ένα σήμα απόλυτα συγχρονισμένο με το ρολόι της, είναι απαραίτητο να τοποθετούνται ενδιάμεσα σε αυτά

[7] Δηλαδή αν προέρχονται από δύο διαφορετικά κυκλώματα χρονισμού που οδηγούνται στο FPGA, ή αν δεν έχουν κοινό MMCM

τα κρίσιμα *timing paths*.

Όταν τα σήματα που μεταβαίνουν από το ένα πεδίο χρονισμού στο άλλο είναι single-bit[8], τότε το πιο απλό κύκλωμα συγχρονισμού είναι ένα διπλό pipeline stage, όπως φαίνεται στο σχήμα 4.2.V



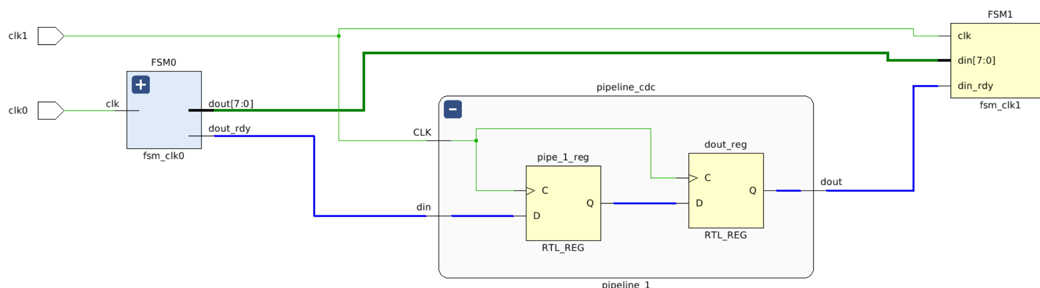
Σχήμα 4.2.V: RTL Schematic δύο διαδοχικών καταχωρητών που παρεμβάλλονται μεταξύ δύο FSM, οι οποίες χρονίζονται από δύο διαφορετικά ρολόγια που είναι μεταξύ τους ασύγχρονα. Ο *pipe_1 _reg* θα δέχεται ένα σήμα συγχρονισμένο με το ρολόι clk0, επομένως είναι σίγουρο ότι θα δημιουργούνται συνεχώς παραβιάσεις του setup/hold time, με αποτέλεσμα το σήμα εξόδου του *pipe_1 _reg* να έχει άγνωστο χρονικό προφίλ (βλ. Q_{out} στο σχ. 4.1.VII). Το πιθανότερο όμως είναι ότι η τιμή του σήματος θα έχει σταθεροποιηθεί πριν το επόμενο rising edge του ρολογιού, και έτσι το *dout_reg* θα παρέχει στη δεύτερη FSM ένα καλά συγχρονισμένο σήμα με το clk1.

Όταν ένα multi-bit (data bus) σήμα μεταβαίνει από ένα πεδίο χρονισμού σε κάποιο άλλο, τότε ο ασφαλέστερος τρόπος να περάσουν όλα τα επιμέρους σήματα στο πεδίο που τα δέχεται, είναι με μία FIFO. Η FIFO μπορεί να λειτουργήσει με δύο διαφορετικά ρολόγια, ένα ρολόι με το οποίο γράφει δεδομένα, και ένα με το οποίο διαβάζει. Έτσι δρα ουσιαστικά σαν "γέφυρα" που ενώνει τα δύο πεδία χρονισμού. Αυτό που χρειάζεται από τη μεριά του χρήστη είναι να σχεδιαστεί η επιπλέον λογική που θα οδηγεί τα σήματα *WR_EN* και *RD_EN* που γράφουν και διαβάζουν δεδομένα στη FIFO.

Η παραπάνω προσέγγιση εξασφαλίζει την ακεραιότητα των δεδομένων, αλλά επίσης δεσμεύει και αρκετή λογική μέσα στο FPGA, καθώς η FIFO θα αντιστοιχεί σε ένα στοιχείο μνήμης RAM μαζί με τη βοηθητική λογική η οποία υλοποιεί τους μετρητές και τα σήματα ελέγχου *(EMPTY, FULL)* που αποτελούν τη FIFO. Επίσης, η επιπλέον λογική του χρήστη που θα οδηγεί τα σήματα *WR_EN* και *RD_EN* της FIFO

[8] Στην περίπτωση για παράδειγμα που δύο FSM χρονισμένες από διαφορετικά ρολόγια ανταλλάσσουν σήματα ελέγχου μεταξύ τους.

θα δεσμεύουν και αυτά με τη σειρά τους κάποια LUT και FFs/Latches. Μία απλοποιημένη μέθοδος, θα είναι να μη συγχρονίζεται το data bus αυτό καθ' αυτό, αλλά μόνο ένα σήμα ελέγχου το οποίο θα περνάει από το ένα πεδίο χρονισμού στο άλλο. Αυτό το σήμα ελέγχου θα οδηγείται από το πεδίο χρονισμού που στέλνει τα σήματα, με μία αυθαίρετη καθυστέρηση σε σχέση με τα δεδομένα του bus, και θα συγχρονίζεται από ένα διπλό pipeline stage, όπως στην προηγούμενη περίπτωση. Η καθυστέρηση θα μπαίνει ώστε να εξασφαλίζεται ότι όλα τα επιμέρους σήματα του bus έχουν σταθεροποιηθεί[9], και ότι οι registers που τα δέχονται στο άλλο πεδίο χρονισμού θα έχουν βγει από τυχόν μετασταθείς καταστάσεις. Λόγω του αυθαίρετου αυτής της καθυστέρησης, η συγκεκριμένη μέθοδος θεωρείται λιγότερο αξιόπιστη σε σχέση με την προηγούμενη που έκανε χρήση της FIFO, όμως είναι πιο απλή στην υλοποίησή της. Στο σχήμα 4.2.VI μπορεί κανείς να πάρει μία γενική ιδέα όσο αναφορά την αρχιτεκτονική του κυκλώματος αυτού.



Σχήμα 4.2.VI: Συγχρονίζοντας ένα σήμα με πολλά bit (εν προκειμένω έχει μήκος ενός byte) από ένα πεδίο χρονισμού σε άλλο. Ουσιαστικά οι δύο FSM ανταλλάσσουν ένα σήμα *handshaking* (din_rdy και dout _rdy) το οποίο σηματοδοτεί ότι η τιμή του multi-bit bus έχει ανανεωθεί. Μόνο αυτό το σήμα συγχρονίζεται, και παίρνει την τιμή ένα με μία καθυστέρηση σε σχέση με την αλλαγή της τιμής του data bus.

4.2.4 Σφάλματα Χρονισμού και FSM - Deadlocking

Μέχρι τώρα έχουν αναφερθεί κατά κύριο λόγο μόνο γενικά παραδείγματα για προβλήματα που μπορεί να δημιουργηθούν από μετασταθείς καταστάσεις και λοιπά σφάλματα χρονισμού σε ένα FPGA. Μία πολύ συγκεκριμένη κατηγορία σφαλμά-

[9]Ως *σταθεροποίηση* του data bus νοείται η ταύτιση των λογικών επιπέδων και στα δύο άκρα του data bus, για όλα τα επιμέρους σήματα.

των συνυφασμένων με παραβιάσεις των χρονικών περιορισμών, είναι το λεγόμενο *FSM Deadlocking*. Προκειμένου να μελετηθεί αυτή η πολύ ενδιαφέρουσα (και επικίνδυνη) κατηγορία σφαλμάτων, θα ήταν ίσως χρήσιμο να γίνει πρώτα μία νύξη για το πως κωδικοποιούνται οι καταστάσεις των FSM σε επίπεδο καταχωρητών.

Ανατρέχοντας στην ενότητα 3.2, και στην εικόνα 3.2.I, μπορεί να δει κανείς το γενικό μοτίβο που διέπει την αρχιτεκτονική των FSM. Τι θα συμβεί λοιπόν, αν ένα σήμα εισόδου σε κάποια FSM, δεν είναι συγχρονισμένο με το ρολόι που τη χρονίζει; Έστω ότι ένα κομμάτι του κώδικα που περιγράφει την FSM1 της εικόνας 4.2.V έχει ως εξής:

VHDL Code 4.1: FSM1

```
1  case state is
2      when ST_IDLE =>
3          -- ...state transision clauses...
4          -- ...signal assertions...
5      when ST_0 =>
6          -- ...state transision clauses...
7          -- ...signal assertions...
8      when ST_1 =>
9          -- ...state transision clauses...
10         -- ...signal assertions...
11     when ST_2 =>
12         if(din = '1')then
13             state <= ST_3;
14         else
15             state <= ST_2;
16         end if;
17     when ST_3 =>
18         -- ...state transision clauses...
19         -- ...signal assertions...
20 end case;
```

Η συγκεκριμένη FSM διαθέτει πέντε δυνατές καταστάσεις, οι οποίες θα αντιστοιχούν σε κάποιους *state registers*, ενώ οι αποφάσεις για τις μεταβάσεις από τη μία κατάσταση στην άλλη θα λαμβάνεται από κυκλώματα συνδυαστικής λογικής. Το Vivado® διαθέτει διάφορους τρόπους αντιστοίχησης της εκάστοτε κατάστασης της FSM σε λογικά επίπεδα των καταχωρητών της κατάστασης. Οι τρεις πιο διαδεδομένοι τρόποι *κωδικοποίησης (FSM Encoding)* καταστάσεων είναι οι [14, 18]:

- *One-Hot:* Κάθε κατάσταση θα έχει μόνο έναν state register στο λογικό ένα, με τους άλλους να είναι στο μηδέν.
- *Gray Code:* Οι καταστάσεις θα ακολουθούν μία αρίθμηση σύμφωνα με τους μετρητές τύπου Gray.
- *Sequential:* Οι καταστάσεις θα ακολουθούν μία αρίθμηση σύμφωνα με συμβατικούς δυαδικούς μετρητές.

Στο προκείμενο παράδειγμα, οι καταστάσεις θα κωδικοποιούνται ως εξής:

State	One-Hot	Gray Code	Sequential
ST_IDLE	10000	000	000
ST_0	01000	001	001
ST_1	00100	011	010
ST_2	00010	010	011
ST_3	00001	110	100

Ο κάθε τρόπος κωδικοποίησης χαρακτηρίζεται από διάφορα πλεονεκτήματα και μειονεκτήματα [5]. Ο *One-Hot* για παράδειγμα, μπορεί να λειτουργήσει για μεγάλες ταχύτητες ρολογιού, όμως δεσμεύει και τους περισσότερους state registers. Η ανοχή του στο χρονισμό οφείλεται στο γεγονός ότι επειδή η κάθε κατάσταση χαρακτηρίζεται ουσιαστικά από ένα bit, η συνδυαστική λογική της FSM απλοποιείται σημαντικά, άρα δεσμεύονται σχετικά λίγα LUT και διασυνδέσεις. Για το λόγο αυτό ο one-hot χρησιμοποιείται περισσότερο από το Vivado® σε σχέση με τους άλλους τρόπους κωδικοποίησης. Οι *Gray* και *Sequential* από την άλλη, δεσμεύουν λιγότερους καταχωρητές για να αποθηκεύσουν την κατάσταση, όμως σε σχέση με τον *One-Hot* απαιτούν περισσότερο πολύπλοκη λογική για τη μετάβαση από τη μία κατάσταση στην άλλη. Συνήθως, το Vivado® λαμβάνει υπόψιν του την ταχύτητα του ρολογιού που χρονίζει τους state registers σε συνδυασμό με το πλήθος των καταχωρητών της κατάστασης, προκειμένου να αποφασίσει τον βέλτιστο τρόπο κωδικοποίησης, ώστε να μην προκύπτουν σφάλματα χρονισμού. Σε ένα rising edge του ρολογιού που χρονίζει την FSM[10], οι state registers θα στείλουν στο συνδυαστικό κύκλωμα την κατάσταση στην οποία βρίσκονται. Η συνδυαστική λογική θα λάβει υπόψιν της την κατάσταση σε συνδυασμό με τα σήματα εισόδου, οδηγώντας έτσι στους state registers το νέο τους

[10] βλ. σχ. 3.2.I

bit, προκειμένου στον επόμενο κύκλο να γίνει (ή να μη γίνει) κάποια μετάβαση σε άλλη κατάσταση. Αυτό σημαίνει ότι το εν λόγω σήμα θα πρέπει να φτάσει στους state registers *πριν* από το επόμενο rising edge του ρολογιού, *και* εκτός του παραθύρου των state registers, ώστε να μη προκληθεί κάποιο σφάλμα χρονισμού. Η κρίσιμη ερώτηση λοιπόν, είναι η εξής: Τι θα συμβεί αν για οποιοδήποτε λόγο προκληθεί σφάλμα χρονισμού σε κάποιον κύκλο λειτουργίας της FSM;

Έστω ότι η FSM του παραδείγματος έχει κωδικοποιηθεί με τρόπο *One-Hot*, και βρίσκεται στην κατάσταση *ST_2*. Αν το σήμα εισόδου *din* γίνει ένα, τότε ο τέταρτος register θα πάρει στον επόμενο κύκλο την τιμή μηδέν, και ο πέμπτος θα πάρει την τιμή ένα, προκειμένου να γίνει η μετάβαση στην *ST_3*, δηλαδή $00010 => 00001$. Σύμφωνα με το σχήμα 4.2.V, το *din* πριν φτάσει στη συνδυαστική λογική της FSM1 περνάει από δύο στάδια συγχρονισμού, τα οποία "φέρνουν" το σήμα στο πεδίο χρονισμού του ρολογιού της FSM1. Έτσι το σήμα θα μπει στο συνδυαστικό κύκλωμα, το οποίο με τη σειρά του θα στείλει στον τέταρτο και πέμπτο καταχωρητή τις νέες τιμές. Επειδή το *din* είναι συγχρονισμένο με το ρολόι της FSM1, θα φτάσει στο συνδυαστικό κύκλωμα σε σωστό χρονικό σημείο, με αποτέλεσμα να αποφεύγεται έτσι κάποιο σφάλμα χρονισμού. Τι θα συμβεί όμως αν αφαιρεθεί το κύκλωμα συγχρονισμού από το σχήμα 4.2.V, με αποτέλεσμα η FSM1 να λαμβάνει ένα σήμα εισόδου ασύγχρονο για αυτήν;

Το πιθανότερο είναι ότι σε έναν τυχαίο κύκλο θα συμβεί κάποιο χρονικό σφάλμα, καθώς η συνδυαστική λογική θα στείλει την ανανεωμένη κατάσταση στους καταχωρητές με κακό timing, δηλαδή *μέσα στο παράθυρό τους*. Έτσι θα στείλουν καθυστερημένα την αλλαγή κατάστασης στη συνδυαστική λογική, αφού η μεταστα-θής κατάσταση προκαλεί ακριβώς αυτό[11]. Έτσι μπορεί για παράδειγμα στον επόμενο κύκλο οι καταχωρητές να μην κάνουν τη μετάβαση $00010 => 00001$, αλλά $00010 => 00011$, δηλαδή ο τέταρτος καταχωρητής να μείνει στο λογικό ένα λόγω της καθυστέρησης που προκάλεσε η μετασταθής κατάσταση. Όμως το κύκλωμα της FSM *δεν έχει σχεδιαστεί για να βρεθεί ποτέ στην κατάσταση* 00011. Όταν η FSM καταλήξει σε μία κωδικοποίηση που δεν αντιστοιχεί σε καμία πραγματική κατάσταση, αυτό προκαλεί το λεγόμενο *FSM Deadlock*. Η FSM θα παραμείνει εσαεί σε αυτή την άγνωστη κατάσταση, μη μπορώντας να βγει από αυτήν, αφού το συνδυαστικό κύκλωμα αδυνατεί να κάνει οποιαδήποτε μετάβαση με αφετηρία αυτή την άγνωστη

[11] βλ. σχ. 4.1.VII

κατάσταση/κωδικοποίηση. Εκτός από αυτό τον τύπο απόλυτου deadlocking, υπάρχει και ένας άλλος, στον οποίο όταν κάποιος καταχωρητής πέσει σε μετασταθή κατάσταση, μπορεί να οδηγήσει την FSM σε κάποια state που είναι μεν υπαρκτή, αλλά είναι εντελώς διαφορετική από την επιθυμητή. Μία πλήρως άσχετη μετάβαση κατάστασης μπορεί να δημιουργήσει πολλά προβλήματα, καθώς σε ένα μεγάλο design όπου πολλές FSM επικοινωνούν μεταξύ τους, μία απρόβλεπτη μετάβαση κάποιας FSM μπορεί να προκαλέσει εξίσου απρόβλεπτες μεταπτώσεις καταστάσεων σε άλλα κυκλώματα.

Όπως αντιλαμβάνεται κανείς, ο συγκεκριμένος κίνδυνος πρέπει να αποφεύγεται πάση θυσία, καθώς αν μία FSM βρεθεί σε μία κατάσταση τη λάθος χρονική στιγμή σε σχέση με άλλες του ίδιου design, ή αν παγιδευτεί σε μία κατάσταση που δεν υπάρχει καν, τότε διακυβεύεται όλη η λογική μέσα στο κύκλωμα. Προκειμένου να μην εμφανίζονται αυτά τα άκρως επικίνδυνα σφάλματα, ο σχεδιαστής οφείλει να:

- Σχεδιάζει τις FSM όσο πιο απλές γίνεται, με λίγες καταστάσεις (πολλοί state registers σημαίνει και περίπλοκο συνδυαστικό κύκλωμα που τους συνοδεύει), ώστε να μην χρειάζονται τα σήματα να ταξιδεύουν μεγάλες αποστάσεις μέσα στο FPGA fabric.
- Να περιορίζει τη χρήση πολλών πεδίων χρονισμού. Λίγα ρολόγια σε ένα design σημαίνει και λιγότερο περίπλοκη δουλειά για το Vivado® που προσπαθεί να εξαλείψει τα χρονικά σφάλματα που προκύπτουν όταν σήματα στέλνονται από το ένα πεδίο χρονισμού στο άλλο.
- Όπου χρειάζεται, να προσθέτει κυκλώματα συγχρονισμού.

Σύνοψη

Σε αυτό το Κεφάλαιο πραγματοποιήθηκε η μετάβαση από τα θεωρητικά και ιδεατά κυκλώματα που μελετώνται στις προσομοιώσεις, στα πραγματικά κυκλώματα που υπάρχουν μέσα στο FPGA. Έγινε μία πρώτη γνωριμία με την έννοια των *μετασταθών καταστάσεων (metastable states)* και των *race conditions* που επηρεάζουν τα σύγχρονα και συνδυαστικά κυκλώματα αντίστοιχα. Ο χρόνος *setup/hold* είναι μία εγγενής ιδιότητα των πυλών *flip-flop*, η οποία θέτει περιορισμούς στη διαδικασία τοποθέτησης και αλληλοσύνδεσης των καταχωρητών του FPGA. Τον πιο σημαντικό ρόλο όμως τον παίζει το ρολόι που χρονίζει τους καταχωρητές. Όσο μεγαλύτερη εί-

ναι η συχνότητα, τόσο πιο δύσκολο το έργο του Vivado® στο να τοποθετήσει και να συνδέσει τους registers ώστε να είναι άψογα συγχρονισμένοι μεταξύ τους. Η κατάσταση χειροτερεύει περαιτέρω, όταν ένα design έχει πολλά πεδία χρονισμού. Το εργαλείο της Xilinx® όμως, αναφέρει στο τελευταίο στάδιο της εφαρμογής του design (μετά το placing και το routing) τις τιμές του *slack*. Αυτές είναι οι τιμές του *Worst Negative Slack*, δηλαδή αναφέρει το *timing path* που διατρέχει μεγαλύτερο κίνδυνο να εμφανίσει σφάλμα χρονισμού, και του *Total Negative Slack*, που αντιπροσωπεύει το άθροισμα όλων των επιμέρους αρνητικών slack. Η τελευταία τιμή δίνει μία γενική εικόνα για την υγεία του design. Ο σχεδιαστής οφείλει να λαμβάνει σοβαρά υπόψιν του αυτές τις τιμές, και να αλλάζει το σχεδιασμό του FPGA αναλόγως, προκειμένου να μην συναντήσει σοβαρά σφάλματα, όπως το FSM deadlocking, ή και την αλλοίωση των δεδομένων που εγγράφονται στις μνήμες του FPGA.

Μέρος ΙΙ

Εφαρμογές στο Πείραμα του ATLAS στο CERN

5

Το Πείραμα ATLAS και η Αναβάθμισή του

Στο παρών κεφάλαιο, μετά από μία σύντομη νύξη για τις εδραιωμένες θεωρίες περί Στοιχειωδών Σωματιδίων, θα πραγματοποιηθεί μία γενική επισκόπηση του πειράματος ATLAS που διεξάγεται στο Ευρωπαϊκό Κέντρο Πυρηνικών Ερευνών (CERN: Conseil Européenne pour la Recherche Nucléaire), και της αναβάθμισης του New Small Wheel (NSW) και των ηλεκτρονικών του, που είναι και το ουσιαστικό κίνητρο για την εργασία αυτή.

5.1 Αναζητώντας τα Στοιχειώδη Σωματίδια

Η Φυσική των Στοιχειωδών Σωματιδίων, λειτουργεί έχοντας σαν κύριο άξονα την εξής ερώτηση: ”Από τι είναι φτιαγμένη η ύλη;”, και αποπειράται να απαντήσει σε αυτό το ερώτημα, μελετώντας τις δομές των στοιχειωδών σωματιδίων σε όσο το δυνατόν περισσότερο θεμελιώδες επίπεδο. Είναι αρκετά εντυπωσιακό το γεγονός ότι η ύλη του Σύμπαντος αποτελείται τελικά από μικροσκοπικά κομμάτια συστατικών, τα οποία όταν ανασυνδυάζονται μεταξύ τους με διαφορετικούς τρόπους, κατασκευάζουν κάθε φορά και μία διαφορετική έκφανση της ύλης.

Οι ρίζες της εικασίας ότι τα πάντα αποτελούνται από μικρά κομμάτια δομικών λίθων ύλης, μπορούν να εντοπιστούν ακόμα και στα χρόνια των αρχαίων Ελλήνων Φιλοσόφων, με κύριο υποστηρικτή τον Δημόκριτο (460-370 π.Χ.), ο οποίος πίστευε ότι

όλα γύρω του αποτελούντο από μικρά αδιαίρετα τμήματα βασικής ύλης, τα άτομα. Οι σκέψεις αυτές, αν και επρόκειτο μόνο για μεταφυσικές εικασίες, έδωσαν κίνητρο σε πολλούς σύγχρονους επιστήμονες να εκτελέσουν σημαντικά πειράματα για τον προσδιορισμό και τη μελέτη των βασικών τμημάτων της ύλης. Οι πιο καρποφόρες μελέτες έγιναν από τα τέλη του 19ου αιώνα, μέχρι και τα μέσα του 20ου, με μεγάλα ονόματα Φυσικών να εμπλουτίζουν το γνωστικό υπόβαθρο για τα στοιχειώδη σωματίδια. Για παράδειγμα, το 1897, ο J.J. Thomson ανακάλυψε το ηλεκτρόνιο, ενώ το 1899 ο E. Rutherford με το πείραμα σκέδασης σωματιδίων άλφα σε λεπτά φύλλα χρυσού, ενίσχυσε τις θεωρίες που ήθελαν την ύλη να αποτελείται από μικροσκοπικά άτομα. Επίσης, η θεωρητική συμβολή του A. Einstein το 1905 με την ανάλυση του φωτοηλεκτρικού φαινομένου που εξήγησε την κβάντωση της ηλεκτρομαγνητικής ακτινοβολίας, και η ταυτόχρονη ανάπτυξη της κβαντομηχανικής, έδωσαν τις βάσεις για μία ολοκληρωμένη θεωρία για τα στοιχειώδη σωματίδια.

5.1.1 Το Καθιερωμένο Πρότυπο

Το τελικό αποτέλεσμα όλων αυτών των επίπονων μελετών, είναι αυτό που ονομάζουν σήμερα οι Φυσικοί *Καθιερωμένο Πρότυπο* (Standard Model of Particle Physics). Πρόκειται για μία γενική θεωρία που εξηγεί με συνέπεια τη δομή των βασικών μορφών ύλης, και τη φύση των μεταξύ τους αλληλεπιδράσεων. Σύμφωνα με αυτό το πρότυπο, η ύλη είναι κατασκευασμένη αποκλειστικά από συνδυασμούς *quark* (κουάρκ), και λεπτονίων. Τα quark, η πιο βασική μονάδα ύλης, ζευγαρώνουν είτε ανά δύο για να δημιουργήσουν τα μεσόνια (mesons), ή ανά τρία για να δημιουργήσουν τα βαρυόνια (baryons). Τα quark και τα λεπτόνια, αλληλεπιδρούν μεταξύ τους είτε με ηλεκτρομαγνητικές αλληλεπιδράσεις, είτε με ισχυρές ή ασθενείς πυρηνικές αλληλεπιδράσεις. Οι φορείς των αλληλεπιδράσεων αυτών είναι τα φωτόνια (γ), τα $W^{\pm}$, Z^{0} μποζόνια, και τα γλουόνια. Όλες αυτές οι εξωτικές οντότητες μπορούν να ομαδοποιηθούν στο Σχήμα 5.1.I:

Αξίζει να σημειωθεί, ότι η ύλη σήμερα αποτελείται αποκλειστικά από quark και λεπτόνια πρώτης γενιάς. Αυτά είναι τα up και down quark, και τα ηλεκτρόνια με τα νετρίνο (neutrino) τους. Τα πρωτόνια και τα νετρόνια σχηματίζονται από τα quark αυτά, ενώ τα ηλεκτρόνια περιτριγυρίζουν τους διαφορετικούς συνδυασμούς πρωτονίων και νετρονίων για να σχηματίσουν τελικά τα άτομα, τα χημικά στοιχεία από τα οποία είναι φτιαγμένα όλα όσα μπορεί να φανταστεί κανείς: από τους αστέρες,

Elementary Particles

	I	II	III	Force Carriers
Quarks	u up	c charm	t top	g gluon
Quarks	d down	s strange	b bottom	γ photon
Leptons	ν_e e neutrino	ν_μ μ neutrino	ν_τ τ neutrino	W W boson
Leptons	e electron	μ muon	τ tau	Z Z boson

3 → I II III ← Generations

Σχήμα 5.1.I: Η λίστα με τα δομικά στοιχεία της ύλης σύμφωνα με το Καθιερωμένο Πρότυπο.

μέχρι τις πρωτεΐνες στα κύτταρα των έμβιων όντων. Υπάρχουν όμως και αναρίθμητα άλλα σωματίδια, τα οποία σχηματίζονται από τα πιο βαριά quark, αλλά είναι σχεδόν αδύνατο να παρατηρηθούν, καθώς οι υπάρχουσες συνθήκες στο σημερινό Σύμπαν δεν επιτρέπουν στα σωματίδια αυτά να υφίστανται για τόσο χρονικό διάστημα ώστε να είναι παρατηρήσιμα.

Σύμφωνα λοιπόν με τις καθιερωμένες αντιλήψεις που θέλουν το πρώιμο Σύμπαν να είναι θερμό και με υψηλές ενέργειες, ο στόχος της σύγχρονης φυσικής στοιχειωδών σωματιδίων πλέον ξεκαθαρίζεται: Προκειμένου να μελετηθούν σε βάθος οι δομές και οι αλληλεπιδράσεις μέσα στην ύλη, πρέπει να ξεπεραστούν οι χαμηλές ενέργειες που διέπουν το σημερινό Σύμπαν, ένα Σύμπαν ψυχρό, και εν τέλει...βαρετό σε σχέση με πριν από πολλά δισεκατομμύρια χρόνια, όπου υπήρχε μία πληθώρα σωματιδίων, και που οι δυνάμεις μεταξύ των δεν ήταν διακριτές όπως σήμερα, αλλά όπως δείχνουν πρόσφατες μελέτες, μάλλον ήταν ενοποιημένες [19]. Έτσι λοιπόν, λόγω της ανάγκης για εξερεύνηση σε όλο και μικρότερες τάξεις μεγέθους, για τη μελέτη όλο και βαρύτερων και σπανιότερων σωματιδίων που θα συμπληρώσουν το παζλ του Καθιερωμένου Προτύπου, γεννήθηκε η ανάγκη για όλο και *Υψηλότερες Ενέργειες*. Έτσι οι επιστήμονες ξεκίνησαν να μελετούν θεωρητικά τις Υψηλές Ενέργειες, και να κατασκευάζουν συσκευές που θα τους επιτρέψουν να τις προσεγγίσουν. Μία τέτοια συσκευή, είναι και ο Μεγάλος Επιταχυντής Αδρονίων, που βρίσκεται στο CERN.

5.2 Ο Μεγάλος Επιταχυντής Αδρονίων

Γενικά, μπορεί να πει κανείς ότι οι επιταχυντές είναι τα ”μικροσκόπια” για τους Φυσικούς Υψηλών Ενεργειών [21]. Ιστορικά, ως πρώτος επιταχυντής μπορεί να θεωρηθεί ο καθοδικός σωλήνας που κατασκεύασε το 1895 ο Röntgen για να παράξει ακτίνες X, όπου μία διαφορά δυναμικού επιτάχυνε ηλεκτρόνια σε ενέργειες της τάξης των μερικών keV [1]. Σήμερα, οι ενέργειες που αγγίζουν οι επιταχυντές είναι της τάξης των TeV, και δεν επιταχύνουν μόνο ηλεκτρόνια, αλλά και πρωτόνια ή και βαρέα ιόντα. Στα πειράματα αυτά, μία δέσμη σωματιδίων επιταχύνεται είτε σε ευθεία, είτε σε κυκλική τροχιά με τη βοήθεια ηλεκτρομαγνητών που παράγουν πεδία αρκετά ισχυρά ώστε να αυξάνουν την ταχύτητα των σωματιδίων ενώ ταυτόχρονα τα κατευθύνουν σε αυστηρά καθορισμένες τροχιές. Τελικά τα σωματίδια αυτά συγκρούονται είτε σε σταθερούς στόχους, είτε με άλλες δέσμες σωματιδίων, για να παράξουν βαρέα σωματίδια υψηλών ενεργειών, που στη συνέχεια μελετώνται από τους ερευνητές.

Ο Μεγάλος Επιταχυντής Αδρονίων (Large Hadron Collider, LHC), είναι ο μεγαλύτερος και ισχυρότερος επιταχυντής που έχει κατασκευαστεί ποτέ [24], και ξεκίνησε επίσημα τη λειτουργία του το 2008. Πρόκειται για έναν κυκλικό επιταχυντή όπου υπέρλεπτες δέσμες πρωτονίων επιταχύνονται τόσο πολύ που πλησιάζουν την ταχύτητα του φωτός, ώσπου τελικά αλληλοσυγκρούονται μεταξύ τους παράγοντας έτσι εξωτικά σωματίδια. Η πειραματική αυτή διάταξη βρίσκεται εκατοντάδες μέτρα κάτω από την επιφάνεια της Γης, στα σύνορα Γαλλίας-Ελβετίας, ενώ ο κυκλικός σωλήνας τον οποίο διατρέχουν τα πρωτόνια έχει περιφέρεια 27 km.

Μία σειρά από όλο και μεγαλύτερους δακτυλίους επιταχύνουν προοδευτικά πρωτόνια που προέρχονται από μία απλή φιάλη υδρογόνου, τα οποία πρωτόνια θα καταλήξουν τελικά στον μεγάλο δακτύλιο του LHC, ο οποίος διαχωρίζει δύο δέσμες πρωτονίων και τις ανακυκλώνει για πολλές ώρες σε αντίθετες κατευθύνσεις μέσα του[2]. Οι δέσμες αυτές μπορούν να συγκρουσθούν μεταξύ τους σε συγκεκριμένα ση-

[1]Υπενθυμίζεται ότι η μονάδα ενέργειας του eV (ηλεκτρονιοβόλτ, electronvolt) ισοδυναμεί με την ενέργεια που αποκτά ένα ηλεκτρόνιο, όταν αυτό επιταχυνθεί από μία διαφορά δυναμικού της τάξης του 1 V.

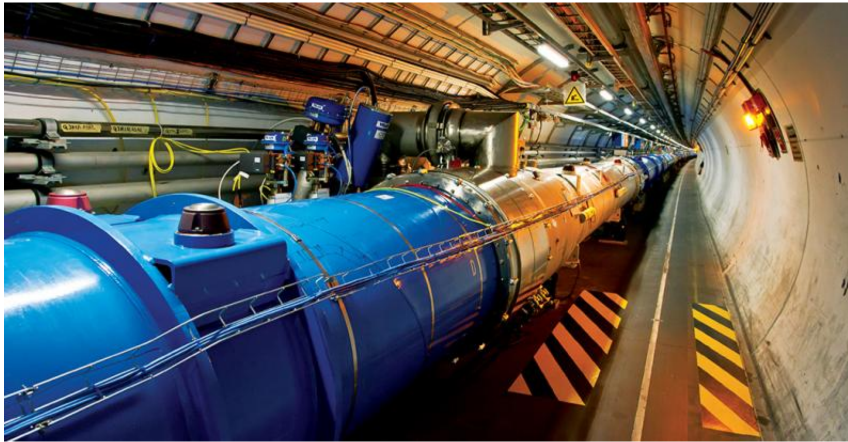
[2]Η διαδικασία της επιτάχυνσης και ανακατεύθυνσης των δεσμών επιτυγχάνεται με ισχυρά ηλεκτρομαγνητικά πεδία που δημιουργούνται από ηλεκτρομαγνήτες υπεραγωγών που λειτουργούν μόνο σε θερμοκρασίες κοντά στο απόλυτο μηδέν.

Σχήμα 5.2.Ι: Αεροφωτογραφία του CERN και του LHC. Είναι ευδιάκριτος ο μεγάλος δακτύλιος που επιταχύνονται τα σωματίδια, ενώ στα δεξιά φαίνεται το αεροδρόμιο και η λίμνη της Γενεύης. Ο μικρότερος δακτύλιος μέσα στον κύριο, είναι ο SPS, ο οποίος διοχετεύει τον κύριο δακτύλιο του LHC με πρωτόνια ενέργειας μερικών εκατοντάδων GeV [24].

μεία πάνω στον δακτύλιο, και οι ισχυρές αυτές συγκρούσεις απελευθερώνουν μεγάλα ποσά ενέργειας, δηλαδή βαριά σωματίδια, ή φωτόνια υψηλής συχνότητας, που παρέχουν πολύτιμες πληροφορίες για τη δομή της ύλης στους ερευνητές του CERN. Ακριβώς σε αυτά τα διαφορετικά σημεία των συγκρούσεων υπάρχουν οι *ανιχνευτές*, δηλαδή εξειδικευμένες διατάξεις υψηλής τεχνολογίας, που μπορούν να ανιχνεύσουν με ακρίβεια όλα τα σωματίδια που παράγονται από τις συγκρούσεις, και να αναλύσουν πλήρως τις τροχιές και τις ιδιότητες των σωματιδίων αυτών. Οι κυριότεροι ανιχνευτές στον LHC είναι οι: ATLAS, CMS, ALICE και LHCb[3].

Στην προσπάθεια βελτίωσης των πειραμάτων για εξαγωγή καλύτερων αποτελεσμάτων σε όλο και υψηλότερες ενέργειες, διαπιστώθηκε ότι όσο ψηλότερος ο ρυθμός αλληλεπιδράσεων ϕ (reaction rate), τόσο περισσότερα γεγονότα ανιχνεύονται με αποτέλεσμα να εξάγονται και περισσότερες πληροφορίες από το πείραμα. Ξεκινώντας από τα απλά, δηλαδή από τα πειράματα σταθερού στόχου, ο ρυθμός αλληλεπίδρασης εξαρτάται από το ρυθμό πρόσπτωσης n των σωματιδίων της δέσμης πάνω στο

[3]Οι ανιχνευτές γενικού ενδιαφέροντος, ATLAS και CMS, ήταν εκείνοι που ανίχνευσαν το 2012 το μποζόνιο Higgs.



Σχήμα 5.2.II: Άποψη της σήραγγας του LHC όπου επιταχύνονται τα πρωτόνια [24].

στόχο, την ενεργό διατομή σ της υπό μελέτη αντίδρασης[4], και το πάχος d του στόχου [21]. Συνδυάζοντας τα μεγέθη αυτά, ορίζεται το μέγεθος της *Φωτεινότητας* $\mathscr{L}$ *(Luminosity)*:

$$\phi = \sigma\mathscr{L} \tag{5.2.1}$$

Στα σύγχρονα πειράματα Υψηλών Ενεργειών όπου δεν προτιμάται η μέθοδος σταθερών στόχων, αφού με αλληλοσυγκρουόμενες δέσμες η μέγιστη ενέργεια που επιτυγχάνεται είναι σημαντικά μεγαλύτερη, η φωτεινότητα (και άρα ο ρυθμός αλληλεπιδράσεων που είναι ανάλογος της φωτεινότητας) ορίζεται περισσότερο περίπλοκα. Πλέον η φωτεινότητα αντιπροσωπεύει ένα μέτρο του πλήθους των σωματιδίων ανά cm^2 και s. Αν N_1 και N_2 το πλήθος των σωματιδίων της κάθε δέσμης, και σ_x, σ_y η εγκάρσια και διαμήκης διατομή των δεσμών , τότε η φωτεινότητα $\mathscr{L}$ σχετίζεται με τις παραμέτρους αυτές μέσω του τύπου:

$$\mathscr{L} \propto \frac{N_1 N_2}{\sigma_x \sigma_y} \tag{5.2.2}$$

Έτσι λοιπόν, το πλήθος των αλληλεπιδράσεων N_{exp} που αναμένεται στο πείραμα, ορίζεται από τη σχέση [20]:

[4]Η ενεργός διατομή έχει μονάδες επιφάνειας και εκφράζει ουσιαστικά την πιθανότητα πραγμάτωσης μίας αντίδρασης.

$$N_{exp} = \sigma_{exp} \int \mathscr{L}(t)dt \tag{5.2.3}$$

Όπου η ποσότητα σ_{exp} ονομάζεται *ενεργός διατομή*, έχει μονάδες μεγέθους επιφάνειας και εκφράζει την πιθανότητα να συμβεί μία συγκεκριμένη αντίδραση.

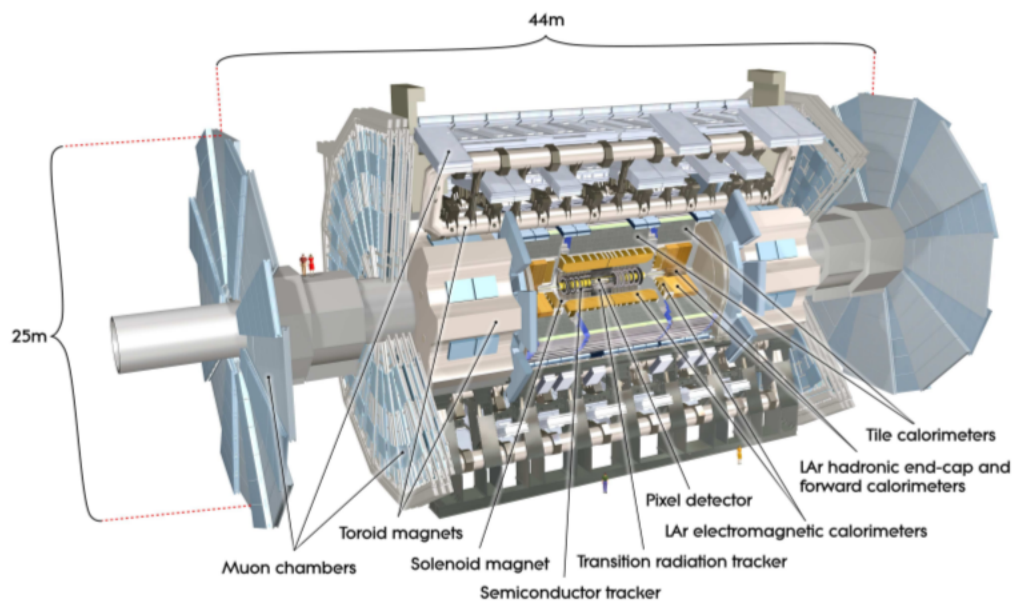
Σχετικά με τους τύπους που μόλις αναφέρθηκαν και τη σχέση τους με τα πειράματα, αξίζει να αναφερθεί ότι είναι γενικά εύκολο να οριστεί το πλήθος των σωματιδίων της δέσμης σε ένα πείραμα. Η μεγάλη δυσκολία έγκειται στη μείωση της διατομής των δεσμών, ένα εγχείρημα που χαρακτηρίζεται από εγγενείς περιορισμούς, καθώς μία δέσμη που αποτελείται μόνο από πρωτόνια για παράδειγμα, αδυνατεί να μείνει συγκεντρωμένη, λόγω των τεράστιων ηλεκτρικών δυνάμεων άπωσης μεταξύ των όμοια φορτισμένων πρωτονίων, που βρίσκονται τόσο κοντά μεταξύ τους μέσα στη δέσμη. Τέτοια τεχνικά ζητήματα καλούνται να λυθούν σε κάθε πείραμα Φυσικής Υψηλών Ενεργειών, πόσο μάλλον στον LHC, όπου οι ενέργειες και η φωτεινότητα είναι τάξεις μεγέθους μεγαλύτερες σε σχέση με παλιότερα εγχειρήματα.

Πιο συγκεκριμένα [27], οι ανιχνευτές ATLAS και CMS, είναι σχεδιασμένοι για να ανιχνεύουν σε φωτεινότητες της τάξης του $\mathscr{L} = 10^{34}\,\text{cm}^{-2}\text{s}^{-1}$, ενώ το πείραμα LHCb λειτουργεί σε χαμηλότερες φωτεινότητες ($\mathscr{L} = 10^{32}\,\text{cm}^{-2}\text{s}^{-1}$). Από τη κατασκευή του μέχρι σήμερα, ο LHC παραμένει (και θα παραμείνει για πολλά χρόνια) η μεγαλύτερη και πιο περίπλοκη πειραματική διάταξη που κατασκευάστηκε ποτέ. Ξεκίνησε το 2011-2012 στα $3.5\,\text{TeV}$ ενέργεια ανά δέσμη και ανέβηκε στα $4\,\text{TeV}$ το 2012, ενώ έχει προσφέρει συνολική φωτεινότητα της τάξης $\int \mathscr{L} dt = 28.26\,\text{fb}^{-1}$ στους ανιχνευτές ATLAS και CMS. Στη συνέχεια, θα εξεταστεί η δομή του ανιχνευτή ATLAS, ο οποίος είναι και ο ανιχνευτής που πραγματεύεται (έμμεσα) η παρούσα εργασία, καθώς οι αναβαθμίσεις που σχεδιάζονται για το μέλλον χρήζουν νέων μελετών στους ανιχνευτές και στα ηλεκτρονικά τους συστήματα, προκειμένου να αντεπεξέλθουν στις νέες απαιτήσεις των επικείμενων πειραμάτων.

5.3 Ο Ανιχνευτής ATLAS

Ο ανιχνευτής ATLAS (**A** **T**oroidal **L**HC **A**pparatu**S**), είναι ένας ανιχνευτής γενικής χρήσης, σχεδιασμένος να καλύψει τις ανάγκες πολλών πειραμάτων του LHC, όπως την αναζήτηση του σωματιδίου Higgs, τη μελέτη της υπερσυμμετρίας, την απάντηση

ερωτημάτων σχετικά με τη σκοτεινή ύλη και τυχόν ύπαρξη επιπλέον διαστάσεων στο Σύμπαν. Το γενικό του σχήμα είναι κυλινδρικό, με μήκος διαμήκη άξονα περίπου 45 m, και διάμετρο περί τα 25 m. Ζυγίζει περίπου 7000 τόννους, και είναι μέχρι σήμερα ο μεγαλύτερος ανιχνευτής σε όγκο που κατασκευάστηκε ποτέ. Από το 2012, 3000 επιστήμονες από 38 χώρες συνεργάζονται για την υποστήριξη και αναβάθμιση του ανιχνευτή [24]. Στο σχήμα 5.3.I απεικονίζεται σε τρισδιάστατο μοντέλο μία γενική άποψη του ανιχνευτή.



Σχήμα 5.3.I: Τομή του ανιχνευτή ATLAS [24].

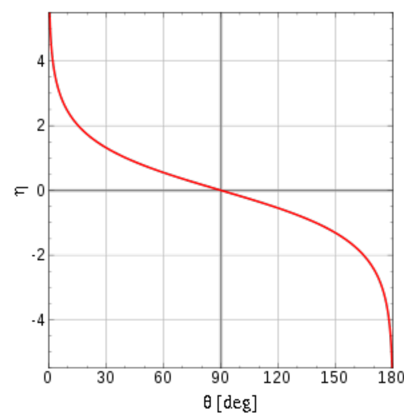
Οι δέσμες των σωματιδίων περνούν από τον νοητό άξονα κυλινδρικής συμμετρίας του ανιχνευτή (από το κέντρο των κυκλικών ”καπακιών”) και συγκρούονται στο κέντρο του, παράγοντας νέα σωματίδια. Τα διαφορετικά υποσυστήματα του ATLAS, τοποθετημένα σε στρώματα, καταγράφουν την τροχιά, την ορμή και την ενέργεια των σωματιδίων, ταυτοποιώντας τα πλήρως. Ισχυροί μαγνήτες κάμπτουν τις τροχιές των φορτισμένων σωματιδίων, επιτρέποντας τη μέτρηση της ορμής τους, ενώ τα δεδομένα που συλλέγονται από τα ηλεκτρονικά συστήματα του ανιχνευτή, υποβάλλονται σε συνεχή ανάλυση ώστε να εξαχθούν οι επιθυμητές πληροφορίες από αυτά. Η όλη διάταξη θυμίζει τη μορφή ενός βαρελιού, και για το λόγο αυτό είναι σύνηθες να αναφέρονται τα διάφορα τμήματα του ανιχνευτή ως ”περίβλημα του

βαρελιού", ή απλά "βαρέλι" (barrel region), και ως "καπάκια" (end-cap region)

Προκειμένου να γίνουν σαφείς οι λεπτομέρειες για τα ανιχνευτικά συστήματα του ATLAS που θα αναφερθούν στη συνέχεια, είναι καλό σε αυτό το σημείο να οριστεί η έννοια της *pseudorapidity* (η). Αν η γωνία σε σχέση με τον άξονα της δέσμης ισούται με θ, τότε η pseudorapidity ορίζεται ως [25]:

$$\eta \equiv -\ln\left[\tan\left(\frac{\theta}{2}\right)\right] \tag{5.3.1}$$

Ο ορισμός της ποσότητας η είναι πολύ σημαντικός, καθώς γίνεται έτσι εύκολα αντιληπτή η γεωμετρία που καλύπτει το κάθε ανιχνευτικό υποσύστημα του ανιχνευτή. Πιο συγκεκριμένα, αναφέρεται πιο συχνά η ποσότητα $|\eta|$, που καλύπτει και τα δύο νοητά ημιεπίπεδα που σχηματίζει η ευθεία της δέσμης. Ένα προκύπτον σωματίδιο με μεγάλη τιμή pseudorapidity, θα κινείται σχεδόν συγγραμικά με τη δέσμη, ενώ μικρότερες τιμές του $|\eta|$ υπονοούν γωνίες διαφυγής σε σχέση με τη δέσμη της τάξης των $90° - 45°$. Έτσι, είναι φανερό ότι τα καπάκια του ανιχνευτή αντιστοιχούν σε μεγάλες τιμές του $|\eta|$, ενώ το κυλινδρικό περίβλημα του βαρελιού σε μικρότερες.



Σχήμα 5.3.II: Γράφημα της pseudorapidity σε σχέση με τη γωνία από τον άξονα της δέσμης.

Τα κυριότερα υποσυστήματα του ATLAS είναι τα εξής [27]:

- Το σύστημα μαγνητών
- Ο εσωτερικός ανιχνευτής (tracker)

- Τα θερμιδόμετρα (ή καλορίμετρα)
- Το μιονικό φασματόμετρο (ή σπεκτρόμετρο)

Ο μαγνήτης είναι ουσιαστικά ένα λεπτό υπεραγώγιμο σωληνοειδές, που περικλείει το κενό ανάμεσα από το σημείο αλληλεπίδρασης/σύγκρουσης σε σχέση με τον εσωτερικό ανιχνευτή. Εκτός από τον εσωτερικό μαγνήτη, τρία μεγαλύτερα σωληνοειδή βρίσκονται τοποθετημένα γύρω από τα καλορίμετρα. Ο εσωτερικός μαγνήτης παράγει ένα μαγνητικό πεδίο της τάξης των $2\,\mathrm{T}$, και επιτρέπει στον εσωτερικό ανιχνευτή (με μήκος $6\,\mathrm{m}$ και διάμετρο $2\,\mathrm{m}$) που καλύπτει το εύρος $|\eta| < 2.5$, να αναλύσει τις τροχιές των σωματιδίων που μόλις έχουν δραπετεύσει από τα σημεία σύγκρουσης. Περίπου 1000 σωματίδια προκύπτουν από το σημείο σύγκρουσης κάθε $25\,\mathrm{ns}$. Παρά τη μεγάλη πυκνότητα τροχιών που δημιουργούνται στον ανιχνευτή, επιτυγχάνεται βέλτιστος προσδιορισμός ορμών και vertex[5], από τα συστήματα των Pixel Detectors και SemiConductor Trackers (SCTs). Γύρω από αυτούς, βρίσκονται Transition Radiation Trackers (TRT), που αναλύουν την ακτινοβολία μετάπτωσης[6].

Γύρω από τον εσωτερικό ανιχνευτή, βρίσκονται τα καλορίμετρα, ή αλλιώς θερμιδόμετρα. Τα καλορίμετρα είναι συσκευές σχεδιασμένες με τέτοιο τρόπο ώστε να απορροφούν πλήρως συγκεκριμένου τύπου σωματίδια που προσπίπτουν πάνω τους. Έτσι όλη η ενέργεια των σωματιδίων εναποτίθεται στο εσωτερικού του θερμιδόμετρου. Συνήθως, τα καλορίμετρα είναι είτε αδρονικά (ανιχνεύουν δηλαδή μεσόνια ή βαρυόνια), είτε ηλεκτρομαγνητικά (ανιχνεύουν ηλεκτρόνια, ποζιτρόνια και φωτόνια). Το αδρονικό καλορίμετρο χωρίζεται σε τρία μέρη, ανάλογα το εύρος της pseudorapidity που καλύπτει. Υπάρχει το τμήμα που καλύπτει το βαρέλι ($|\eta| < 1.7$), τα καπάκια του ($1.5 < |\eta| < 3.2$), και περιοχές ακόμα περισσότερο κοντά στη δέσμη ($3.1 < |\eta| < 3.9$) [27]. Κάθε τμήμα έχει και λίγο διαφορετική κατασκευή· για παράδειγμα το καλορίμετρο που περιβάλλει το βαρέλι είναι φτιαγμένο από ατσάλι (που λειτουργεί ως απορροφητής) και σπινθηριστές (scintillators) που παράγουν τα σήματα ανίχνευσης [25]. Τα υπόλοιπα αδρονικά καλορίμετρα είναι κατασκευασμένα από υγρό Αργό (Liquid-Argon (LAr)). Το ηλεκτρομαγνητικό καλορίμετρο από την άλλη, αποτελείται από δύο ομόκεντρους κυκλικούς δίσκους στα καπάκια, έναν εξωτερικό και έναν εσωτερικό που καλύπτουν $1.375 < |\eta| < 2.5$ και $2.5 < |\eta| < 3.2$

[5] Το ”σημείο” αλληλεπίδρασης ονομάζεται vertex.

[6] Η ακτινοβολία που εκπέμπεται από ένα σωματίδιο όταν αυτό αλλάζει μέσα διάδοσης τα οποία έχουν διαφορετικές διηλεκτρικές ιδιότητες μεταξύ τους [21].

αντίστοιχα. Το ηλεκτρομαγνητικό καλορίμετρο περικλείει το βαρέλι και καλύπτει εύρος $|\eta| < 1.475$. Οι συσκευές αυτές είναι κατασκευασμένες από μολύβδινες πλάκες που απορροφούν τα σωματίδια, από LAr, και από ηλεκτρόδια τύπου Kapton σε σχήμα ακκορντεόν υπεύθυνα για τη συλλογή των ηλεκτρικών σημάτων [25, 27].

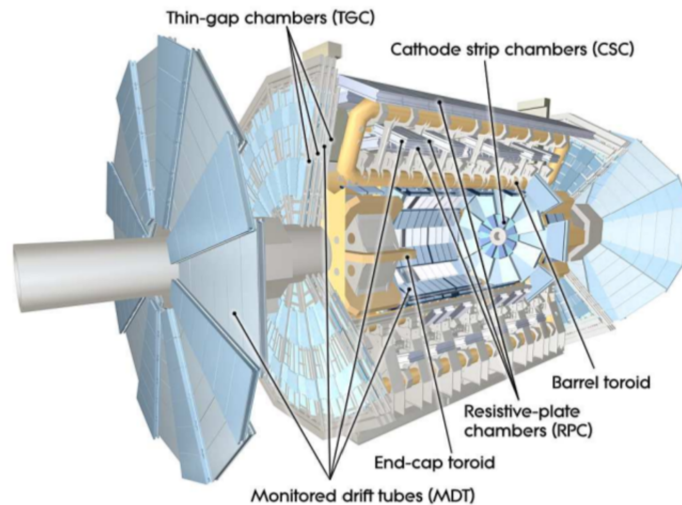
Τα σωματίδια που δεν έχουν απορροφηθεί από τα δύο μεγάλα στρώματα ανιχνευτών που αναφέρθηκαν παραπάνω, είναι συνήθως μιόνια (), τα οποία ανήκουν στην οικογένεια των λεπτονίων δεύτερης γενιάς. Είναι είτε θετικά είτε αρνητικά φορτισμένα (τα αντιμιόνια είναι θετικά φορτισμένα) και είναι αρκετά βαρύτερα από τα συγγενικά τους ηλεκτρόνια[7]. Λόγω της διεισδυτικότητας που τα χαρακτηρίζει, γενικά σε ανιχνευτικές διατάξεις, τα μέρη εκείνα που είναι υπεύθυνα για την ταυτοποίησή τους, τοποθετούνται στα εξωτερικά τμήματα του ανιχνευτή.

Το *Μιονικό Σπεκτρόμετρο* (στο οποίο θα δοθεί και περισσότερη έμφαση καθώς η αναβάθμιση την οποία πραγματεύεται το παρών μέρος αυτής της εργασίας αφορά αυτό το κομμάτι του ATLAS) περιβάλλει τα καλορίμετρα και ορίζει ουσιαστικά το μέγεθος ολόκληρου του ανιχνευτή. Τα συστήματα του μιονικού φασματόμετρου, περιλαμβάνουν ανώτερες τεχνολογίες για το triggering[8] και για την ανακατασκευή των τροχιών των μιονίων (tracking)[27]. Στο Σχήμα 5.3.III είναι φανερή η διάταξη του συστήματος εντοπισμού των μιονίων του ATLAS.

Η λειτουργία του φασματομέτρου μιονίων βασίζεται στην κάμψη των τροχιών των μιονίων λόγω του μαγνητικού πεδίου που δημιουργούν οι υπεραγώγιμοι τοροειδείς μαγνήτες μέσα στον ανιχνευτή. Στο εύρος $|\eta| < 1.4$, η κάμψη των τροχιών παρέχεται από τον τοροειδή μαγνήτη που περιβάλλει το βαρέλι, ενώ για το εύρος $(1.6 < |\eta| < 2.7)$ οι τροχιές κάμπτονται από δύο τοροειδείς μαγνήτες που βρίσκονται στα καπάκια του ανιχνευτή. Το προκύπτον μαγνητικό πεδίο είναι ως επί το πλείστον κάθετο στις τροχιές των μιονίων [25], όποια και να είναι η γωνία διαφυγής τους σε σχέση με τη δέσμη. Η ανίχνευση στα μικρά $|\eta|$, γίνεται από τρεις θαλάμους ανίχνευσης (chambers), που είναι ταξινομημένοι σε τρία κυλινδρικά στρώματα γύρω από τον άξονα της δέσμης. Στην περιοχή με μεγάλα $|\eta|$, οι θάλαμοι καλύπτουν τα καπάκια πλήρως, είναι κάθετοι στον άξονα της δέσμης, και είναι και αυτοί οργανωμένοι σε τρία στρώματα.

[7]Τα μιόνια (μ^-) έχουν μάζα περίπου $106\ ,\mathrm{MeV/c^2}$, ενώ τα ηλεκτρόνια (e^-) $511\,\mathrm{keV/c^2}$ [19].

[8]"Σκανδαλισμός" στα Ελληνικά. Πρόκειται για ένα σήμα που υποδεικνύει ότι ένα πραγματικό γεγονός έχει συμβεί και πρέπει να καταγραφεί από τα ηλεκτρονικά συστήματα του ανιχνευτή.

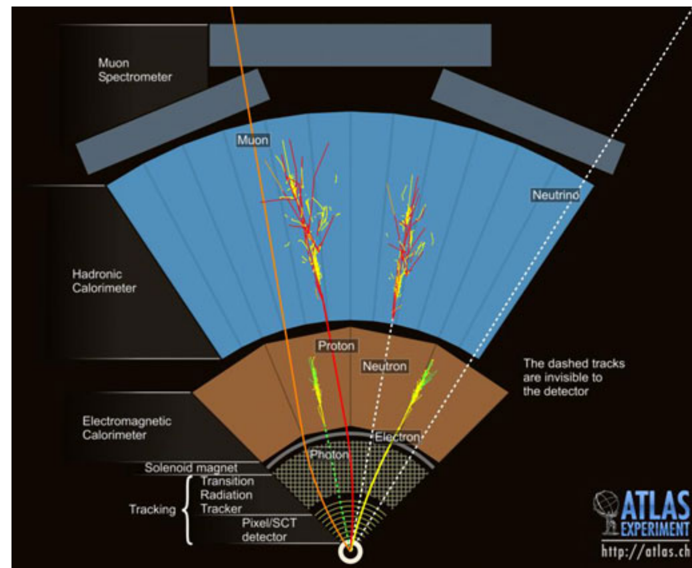


Σχήμα 5.3.III: Άποψη του μιονικού φασματομέτρου του ATLAS [25].

Σε όλα τα εύρη των $|\eta|$, η τροχιά των μιονίων ανακατασκευάζεται από τα σήματα που αφήνουν τα μιόνια σε μία σειρά από Monitored Drift Tubes (MDTs), Στα καπάκια, εκτός από MDTs, υπάρχουν και Cathode Strip Chambers (CSCs), που είναι πολυσυρματικοί αναλογικοί ανιχνευτές (multiwire proportional chambers), σχεδιασμένοι να υποβοηθούν το tracking των μιονίων, καθώς η περιοχή εκείνη χαρακτηρίζεται από αυξημένο ρυθμό γεγονότων και ακτινοβολία υποβάθρου [25]. Η αρχή λειτουργίας τους είναι συγγενική με εκείνη των MDTs, όμως χαρακτηρίζεται από εντελώς διαφορετική γεωμετρία και μείγμα αερίου. Το σύστημα του triggering από την άλλη, αποτελείται από Resistive Plate Chambers (RPCs) που περιβάλλουν το βαρέλι, και από Thin Gap Chambers (TGCs), στα καπάκια.

5.4 Η Αναβάθμιση του New Small Wheel

Η κατασκευή του LHC ολοκληρώθηκε με επιτυχία το 2008 και η μέγιστη ενέργεια του επιταχυντή έφτασε περίπου τα 1 TeV, και μέχρι το πρώτο κλείσιμο το 2010, προσέφερε συνολικά $\int \mathscr{L} = 29\,\mathrm{fb}^{-1}$. Μετά τη πρώτη μεγάλη παύση λειτουργίας (Long Shutdown , LS1) το 2013-2014 για αναβαθμίσεις, η ενέργεια του επιταχυντή αυξήθηκε στα 7 TeV ανά δέσμη, με τη φωτεινότητα να φτάνει στα $\mathscr{L} = 1 \times 10^{34}\,\mathrm{cm}^{-2}\mathrm{s}^{-1}$.

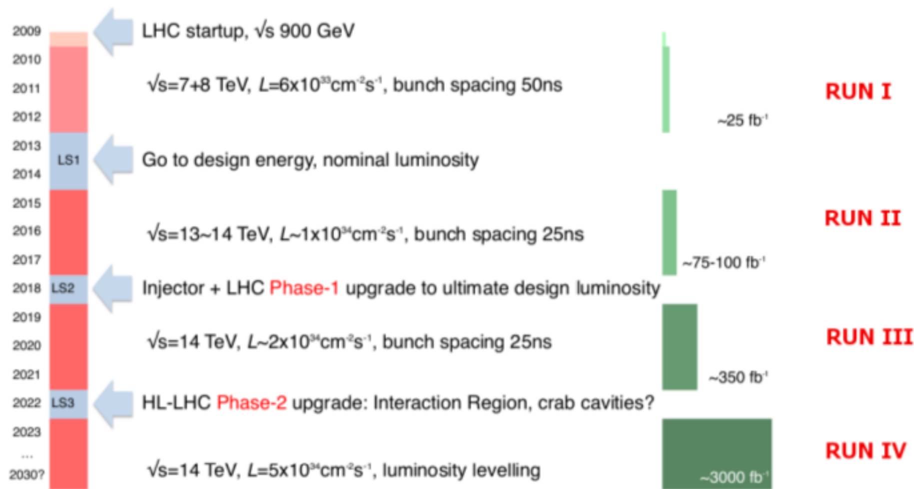


Σχήμα 5.3.IV: Αναπαράσταση ενός γεγονότος στον ATLAS [27].

Το 2020 έχει προγραμματιστεί η δεύτερη μεγάλη παύση (LS2), όπου η φωτεινότητα θα αυξηθεί ακόμα περισσότερο ($\mathscr{L} = 2 - 3 \times 10^{34}\,\mathrm{cm^{-2}s^{-1}}$), η ενέργεια θα φτάσει τα $13 - 14\,\mathrm{TeV}$, και ο ανιχνευτής ATLAS αναμένεται να συλλέγει περίπου $100\,\mathrm{fb^{-1}}$ ολοκληρωμένη φωτεινότητα ανά έτος [26]. Το τελικό στάδιο της αναβάθμισης της δέσμης αναμένεται το 2022, (LS3), όπου η φωτεινότητα θα αγγίξει τα $\mathscr{L} = 5 \times 10^{34}\,\mathrm{cm^{-2}s^{-1}}$. Στο Σχήμα 5.4.I δίνεται ένα χρονοδιάγραμμα των αναβαθμίσεων στον LHC.

Για τον ανιχνευτή ATLAS, οι διαδοχικές αυξήσεις στη φωτεινότητα συνεπάγονται και αύξηση στο ρυθμό με τον οποίο τα σωματίδια θα τον διαπερνούν. Προκειμένου να εξαχθούν όλα τα επιθυμητά αποτελέσματα από την επικείμενη αναβάθμιση του LHC το 2018, ο ανιχνευτής ATLAS πρέπει να αναβαθμιστεί και αυτός. Πιο συγκεκριμένα, ιδιαίτερη προσοχή έχει δοθεί στη βελτίωση του Level-1 triggering και του tracking στο σύστημα μιονίων, καθώς [26]:

- Η αποδοτικότητα των ανιχνευτών που είναι υπεύθυνοι για την ανακατασκευή των τροχιών των μιονίων, κυρίως σε τμήματα με μεγάλα $|\eta|$ στα καπάκια του ανιχνευτή, προβλέπεται ότι θα υποβαθμιστεί με αύξηση της φωτεινότητας, βάσει μέχρι τώρα μετρήσεων. Μεγαλύτερο πρόβλημα θα παρουσιαστεί κυρίως στο εσωτερικό τμήμα των δίσκων (Small Wheels), που έχουν απόσταση μέχρι



Σχήμα 5.4.Ι: Χρονοδιάγραμμα των αναβαθμίσεων του LHC [26].

7 m από τον άξονα της δέσμης.

- Αναλύσεις των δεδομένων που πραγματοποιήθηκαν το 2012 και συγκέντρωσαν τα δεδομένα του triggering από την έναρξη λειτουργίας του LHC, κατέδειξε ότι περίπου το 90 % των μιονικών triggers στα καπάκια του ATLAS ήταν ψεύτικα. Αυτό συνέβαινε καθώς πρωτόνια χαμηλής ενέργειας, που παράγονταν από δευτερεύουσες αντιδράσεις στα υλικά του ανιχνευτή πριν το μιονικό σπεκτρόμετρο, στα καπάκια, εισέρχονταν στους ανιχνευτές TGCs (που είναι υπεύθυνοι για το muon triggering), με τέτοιες γωνίες και ορμές που αναγνωρίζονταν από αυτούς ως μιόνια, φέροντας έτσι το σύστημα σε σύγχηση.

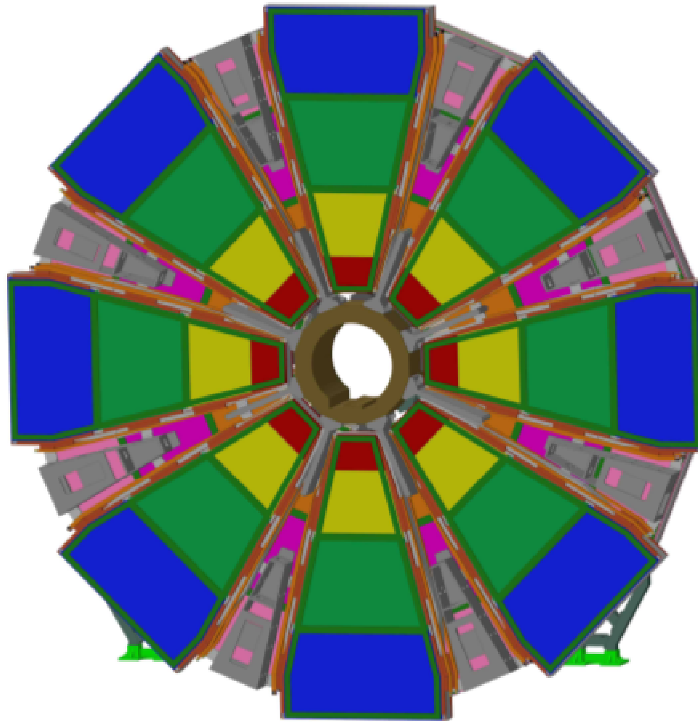
Με την αύξηση της ροής των σωματιδίων (περίπου 15 kHz/cm^2 για $|\eta| = 2.7$ [27]) μετά την πρώτη αναβάθμιση, τα προαναφερθέντα προβλήματα αναμένεται να οξυνθούν σημαντικά. Ως εκ τούτου, η ομάδα του ATLAS πρότεινε την πλήρη αντικατάσταση του προβληματικού τμήματος στα καπάκια (του εσωτερικού δίσκου με μεγάλα $|\eta|$ που είναι πιο κοντά στη δέσμη), από ένα νέο ανιχνευτικό σύστημα, το *New Small Wheel* (NSW). Το νέο τμήμα θα καλύπτει ένα εύρος $1.3 < |\eta| < 2.7$, και με τους νέους ανιχνευτές που θα είναι εγκατεστημένοι πάνω του, προβλέπεται ότι θα προσφέρει άριστο χωρικό και χρονικό προσδιορισμό των τροχιών *(spatial and time resolution)* σε πραγματικό χρόνο, καθώς και πιο συνεπές Level-1 triggering, αφού θα μπορεί πλέον να διακρίνει τα μιόνια χαμηλής ενέργειας από τα πρωτόνια που πα-

ράγονται από δευτερεύουσες αντιδράσεις και αποτελούν για το σύστημα μιονίων ακτινοβολία υποβάθρου [26, 27]. Ένα νέο σύστημα triggering που θα συνδυάζει δεδομένα από το ηλεκτρομαγνητικό καλορίμετρο μαζί με δεδομένα από το NSW, θα μειώσει την αναμενόμενη (για $\mathscr{L} = 3 \times 10^{34}\,\mathrm{cm^{-2}s^{-1}}$) συχνότητα μιονικών Level-1 trigger, από την προβλεπόμενη τιμή των $100\,\mathrm{kHz}$, που θα επικρατούσε αν η ανιχνευτική διάταξη παρέμενε ως έχει, σε πιο διαχειρίσιμα ποσά της τάξης των $20\,\mathrm{kHz}$, με την εγκατάσταση του NSW [27].

Οι ανιχνευτές που θα αποτελούν το NSW θα είναι δύο ειδών και θα προέρχονται από την γενικότερη κατηγορία των ανιχνευτών αερίου (gaseous detectors). Ο πρώτος είναι ένας *Πολυσυρματικός Θάλαμος (Multiwire Chamber)*, που ονομάζεται *small strip Thin Gap Chamber (sTGC)*, ενώ ο δεύτερος ανήκει στην οικογένεια των *Micro-Pattern Gaseous Detectors*, και ονομάζεται *Micromesh Gaseous Structure (Micromegas)*. Οι sTGC θα χρησιμοποιούνται για το triggering, ενώ οι ανιχνευτές Micromegas[9] (MM) θα χρησιμεύσουν κυρίως στην ανακατασκευή τροχιών (tracking), αλλά θα συμμετέχουν και στο triggering. Οι δύο αυτοί τύπου ανιχνευτών θα καλύπτουν συνολικά μία έκταση $1200\,\mathrm{m}^2$, και θα καταλαμβάνουν όλη την εσωτερική περιοχή στα καπάκια του ανιχνευτή. Θα είναι τοποθετημένοι ακτινικά σε οκτώ πλήρως αλληλοκαλυπτόμενα επίπεδα, όπου οι MM θα καταλαμβάνουν το εσωτερικό του δίσκου, ενώ οι sTGC θα βρίσκονται στα εξωτερικά μέρη. Οι ανιχνευτές θα είναι τοποθετημένοι με τέτοιο τρόπο ώστε να μην υπάρχουν νεκρές ζώνες (dead regions)[10], αλλά μόνο περιοχές ελαττωμένης ανιχνευτικής απόδοσης [26]. Μία γενική άποψη από τη μορφή του NSW μπορεί να δει κανείς στο Σχήμα 5.4.II.

[9] Περισσότερες πληροφορίες για τον τρόπο λειτουργίας του θαλάμου Micromegas μπορεί να βρει κανείς στο Παράρτημα Β′.

[10] Πρόκειται για περιοχές που δεν καλύπτονται από ανιχνευτές και αν περάσει κάποιο σωματίδιο από εκεί, θα μείνει απαρατήρητο.



Σχήμα 5.4.II: Σχηματική αναπαράσταση του NSW. Οι ανιχνευτές MM και sTGC είναι διατεταγμένοι σε αλλεπάλληλα στρώματα σε κάθε ”φέτα” (wedge) [27].

5.5 Τα Ηλεκτρονικά Συστήματα του New Small Wheel

Σε αυτή την ενότητα, θα μελετηθεί η γενικότερη δομή που διέπει ολόκληρο το σύστημα ηλεκτρονικών που θα υποστηρίξει την αναβάθμιση του New Small Wheel (NSW). Μέσω παρατηρήσεων και υπολογισμών με τα μέχρι τώρα δεδομένα του ATLAS, έχει προβλεφθεί ότι μετά την αναβάθμιση του LHC, που συνεπάγεται αύξηση της φωτεινότητας άρα και της ροής των σωματιδίων στους ανιχνευτές, η γενική αποδοτικότητα του μιονικού σπεκτρόμετρου στα καπάκια του ανιχνευτή θα εκφυλιστεί. Πιο συγκεκριμένα, αναμένεται ότι με την αύξηση του ρυθμού αλληλεπιδράσεων λόγω των διαδοχικών αναβαθμίσεων, η χωρική διακριτική ικανότητα (spatial resolution) του παρόντος ανιχνευτή θα υποβαθμιστεί, ενώ το σύστημα του μιονικού triggering, που ήδη παρουσιάζει προβλήματα, σίγουρα θα κριθεί ως πα-

ραπάνω από ανεπαρκές. Επομένως, με την αναβάθμιση των ανιχνευτών του NSW, απαιτείται και ένα εξελιγμένο σύστημα ηλεκτρονικών το οποίο σε γενικές γραμμές οφείλει:

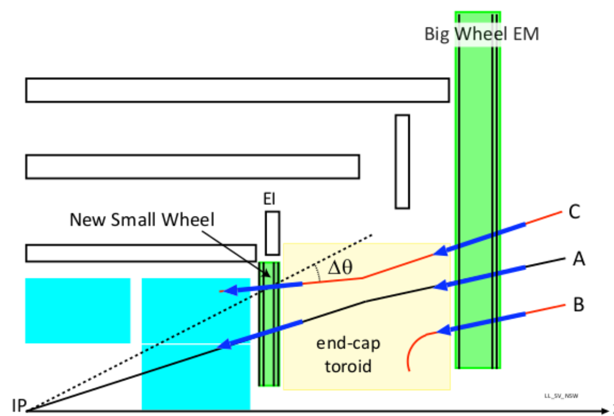
- Να συλλέγει τα δεδομένα του triggering και να τα στέλνει στο τμήμα επεξεργασίας trigger του CERN γρήγορα και αποδοτικά.

- Να αποθηκεύει την ενέργεια που εναποθέτουν τα σωματίδια στο σύστημα ανίχνευσης, μαζί με το *πότε* συνέβη αυτό.

Έτσι ορίζονται οι βασικοί άξονες λειτουργίας του συστήματος ηλεκτρονικών που βρίσκεται πάνω στους ανιχνευτές *(Front-end Electronics (FE))*: γρήγορη συλλογή δεδομένων για το triggering, ακριβής μέτρηση ενέργειας και χρόνου, και διανομή των δεδομένων αυτών στο σύστημα ηλεκτρονικών που παρεμβάλλεται μεταξύ του ανιχνευτή και του κέντρου υπολογιστών του CERN *(Back-end Electronics (BE))*. Επιπλέον, θα πρέπει να λαμβάνουν εντολές για αλλαγές του τρόπου λειτουργίας τους από το κέντρο ελέγχου *(control room)* του ATLAS, και να στέλνουν πίσω δεδομένα για εξωτερικούς παράγοντες (π.χ. θερμοκρασία) που επικρατούν στους ανιχνευτές *(configuration and monitoring)*.

Ως προς τη διαδικασία του trigger, το νέο σύστημα που θα παράγει το triggering του NSW, σε συνεργασία με το σύστημα του Big Wheel, πρέπει να λειτουργεί με τέτοιο τρόπο ώστε να αποκλείει τα ψεύτικα trigger που προέρχονται από τροχιές που δε συμβαδίζουν με το σημείο αλληλεπίδρασης στο κέντρο του ανιχνευτή [26]. Η όλη διαδικασία περιγράφεται στο Σχήμα 5.5.I:

Η διαδικασία επεξεργασίας του triggering περιλαμβάνει συλλογή δεδομένων για τα σημεία αλληλεπίδρασης και υπολογισμός των γωνιών (αζιμουθιακή ϕ και πολική $\Delta\theta$) που έχουν οι τροχιές σε σχέση με το σύστημα συντεταγμένων των ανιχνευτών (βλ. Σχ. 5.5.I). Αυτές οι γωνίες, υπολογίζονται τόσο από τους sTGC όσο και από τους MM, οι οποίοι μέσω των strip τους που έχουν πλάτος μόλις 0.5 mm, μπορούν να παρέχουν άμεσες και ακριβείς πληροφορίες που οδηγούν στον υπολογισμό γωνιών (με ακρίβεια $< 1\ mrad$) των τροχιών. Η διαδικασία υπολογισμού βέβαια, πρέπει να ολοκληρώνεται μέσα σε χρόνο 1025 ns από τη στιγμή που εντοπίζεται η πρώτη αλληλεπίδραση, ώστε να προλαμβάνει τα δεδομένα που θα προκύψουν από την ανάλυση του Big Wheel [26]. Με το συνδυασμό των πληροφοριών για τις τροχιές και από τα



Σχήμα 5.5.I: Σχηματική αναπαράσταση του συστήματος trigger στα καπάκια του ATLAS μετά την αναβάθμιση του 2018. Ο ήδη υπάρχον Big Wheel, δέχεται όλες τις τροχιές σωματιδίων που απεικονίζονται. Με την εγκατάσταση του NSW, μόνο η περίπτωση 'A' γίνεται δεκτή καθώς μόνο αυτή περνάει και από τα δύο συστήματα triggering. Η περίπτωση 'B' θα απορριφθεί καθώς ο NSW δε θα βρει την τροχιά του σωματιδίου που πέρασε από εκείνο το σημείο του Big Wheel. Η τροχιά 'C' θα απορριφθεί επειδή συνδυάζοντας τις γωνίες και από τα δύο συστήματα, το σύστημα επεξεργασίας δεν καταλήγει σε μία ευθεία που να οδηγεί στο σημείο αλληλεπίδρασης [26].

δύο συστήματα, το μιονικό trigger θα μπορεί πλέον αν ανταπεξέρχεται στις περιστάσεις. Πέρα από τη βελτίωση του triggering, οι νέες τεχνολογίες των ανιχνευτών που θα χρησιμοποιηθούν στο NSW, αυξάνουν τη χωρική και διακριτική ικανότητα και βελτιστοποιούν τις μετρήσεις για την ενέργεια που εναποθέτουν τα σωματίδια στους ανιχνευτές. Λόγω αυτού του γεγονότος, η λεπτομερής ανακατασκευή των τροχιών, και ο υπολογισμός των ορμών (τα οποία πραγματώνονται σε μεταγενέστερα στάδια μετά την αποθήκευση των δεδομένων (off-line data analysis)), θα γίνεται με ακόμα μεγαλύτερη ακρίβεια μετά την αναβάθμιση του NSW.

Αντιλαμβάνεται λοιπόν κανείς το κεντρικό ρόλο των ηλεκτρονικών σε όλη αυτή την περίπλοκη διαδικασία ακριβούς και γρήγορου υπολογισμού του triggering και στην ανακατασκευή των τροχιών, τα οποία ηλεκτρονικά οφείλουν όχι μόνο να μεταδώσουν βέλτιστα τις σχετικές πληροφορίες, αλλά και να αντέξουν τις αντίξοες συνθήκες ακτινοβόλησης που θα επικρατούν πάνω στους ανιχνευτές όπου και θα βρίσκονται. Το κύριο σύστημα των front-end ηλεκτρονικών των ανιχνευτών Micromegas, που αυτή τη στιγμή βρίσκεται σε προηγμένα στάδια σχεδίασης και ανάπτυξης,

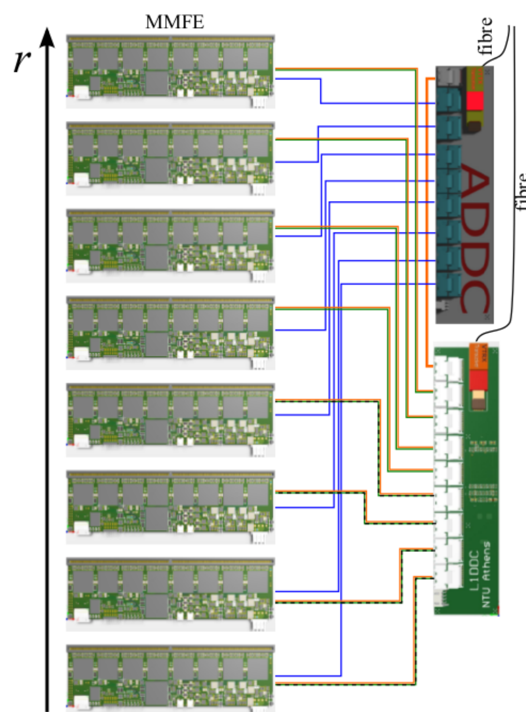
απαρτίζεται από *τρεις* ηλεκτρονικές πλακέτες *(Printed Circuit Boards (PCB))*. Σε αυτό το σημείο θα δοθεί η γενική ιδέα της λειτουργίας των καρτών αυτών [26, 27, 28], ενώ παρακάτω παρατίθεται και μία σχηματική αναπαράσταση της μεταξύ τους συνδεσμολογίας (Σχήμα 5.5.II):

- **Micromegas Front-End Board** *(MMFE, ή και MMFE8)*. Πρόκειται για την ηλεκτρονική πλακέτα που βρίσκεται πάνω στους θαλάμους Micromegas και είναι υπεύθυνη για τη συλλογή των πρωταρχικών σημάτων από τα strips του ανιχνευτή. Κάθε κάρτα MMFE8 περιλαμβάνει οκτώ *ASIC*[11] υπεύθυνα για το readout, που ονομάζονται *VMM*. Κάθε VMM έχει 64 κανάλια, όπου το κάθε ένα αντιστοιχεί σε ένα strip του ανιχνευτή Micromegas. Εκτός από τα VMM ASIC, στην πλακέτα αυτή βαίνουν και άλλα δύο ASIC υπεύθυνα για την διανομή σημάτων monitoring/control και την εξαγωγή των ψηφιακών δεδομένων που προκύπτουν από την ανάλυση των παλμών από τα VMM. Αυτά τα ASIC είναι το *Read-Out Controller (ROC)*, και το *Slow Control Adapter (SCA)*. Τα πρωτότυπα των MMFE8 όμως, δεν θα διαθέτουν αυτά τα δύο βοηθητικά ASIC, αλλά ένα FPGA, το οποίο θα πρέπει να υποκαθιστά τις λειτουργίες αυτών των τσιπ. Για το τελικό πείραμα, θα χρησιμοποιηθούν 4096 πλακέτες MMFE8.

- **Level-1 Data Driver Card** *(L1DDC)*. Πρόκειται για την ηλεκτρονική πλακέτα που δέχεται τα σειριακά δεδομένα από οκτώ MMFE, και τα συμπτύσσει σε ένα link οπτικής ίνας το οποίο καταλήγει στο σύστημα των back-end electronics και συγκεκριμένα στο δίκτυο *FELIX (Front End LInk eXchange)*. Αυτό το επιτυγχάνει με τη χρήση ενός ASIC ονόματι *GBTx*. Εκτός από τη συλλογή και αποστολή δεδομένων για τους παλμούς που συλλέγει ο ανιχνευτής, η πλακέτα αυτή είναι υπεύθυνη και στο να αποστέλλει τα δεδομένα για τη διαμόρφωση της λειτουργικότητας των MMFE8 (configuration signals/data), που προέρχονται από το κέντρο ελέγχου, και να παρέχει στις MMFE8 το ρολόι χρονισμού *(BC clock)*. Η κάθε κάρτα L1DDC όμως, δε συνδέεται μόνο με τις οκτώ MMFE8, αλλά και με την τρίτη πλακέτα, την ADDC που θα περιγραφεί στη συνέχεια. Συνολικά θα κατασκευαστούν 1024 κάρτες L1DDC για την αναβάθμιση του NSW (512 για τους Micromegas και 512 για τους sTGC).

[11] Application-Specific Integrated Circuit: Πρόκειται για μία ευρεία κατηγορία τσιπ, που μπορούν να λειτουργούν αναλογικά, ψηφιακά, ή και με τους δύο τρόπους. Σχεδιάζονται με HDL ή και με άλλα εργαλεία και τελούν πολύ συγκεκριμένες λειτουργίες

- **ART Data Driver Card** *(ADDC)*. Πρόκειται για την ηλεκτρονική πλακέτα η οποία δέχεται τα δεδομένα ART (Address-in-real-time data) από τις οκτώ MMFE με τις οποίες συνδέεται. Διαθέτει και αυτή με τη σειρά της δύο GBTx ASIC και ένα ART ASIC, τα οποία συλλέγουν τα δεδομένα ART, και τα πακετάρουν για να τα στείλουν μέσω ενός link οπτικής ίνας στα back-end electronics, και συγκεκριμένα στο κομμάτι που επεξεργάζεται τα δεδομένα του triggering *(Trigger Processor)*. Η πλακέτα αυτή επικοινωνεί και με μία L1DDC, η οποία της στέλνει configuration data από το κέντρο ελέγχου, και της παρέχει σήματα χρονισμού.



Σχήμα 5.5.II: Σχηματική αναπαράσταση της συνδεσμολογίας μεταξύ των front-end electronics του NSW. Αριστερά είναι οι οκτώ πλακέτες MMFE, και στα δεξιά η κάρτα L1DDC συνδέεται με τις MMFE και την ADDC η οποία με τη σειρά της είναι και αυτή συνδεδεμένη με τις MMFE. Οι κάρτες L1DDC και ADDC συνδέονται μέσω οπτικής ίνας με το σύστημα των back-end electronics, και συγκεκριμένα με το δίκτυο FELIX και τον trigger processor αντίστοιχα [28].

Η επικοινωνία μεταξύ όλων των πλακετών γίνεται σειριακά, μέσω ζευγών διαφορικών γραμμών (καλωδίων) που ονομάζονται *e-links*. Το κάθε e-link, αποτελείται

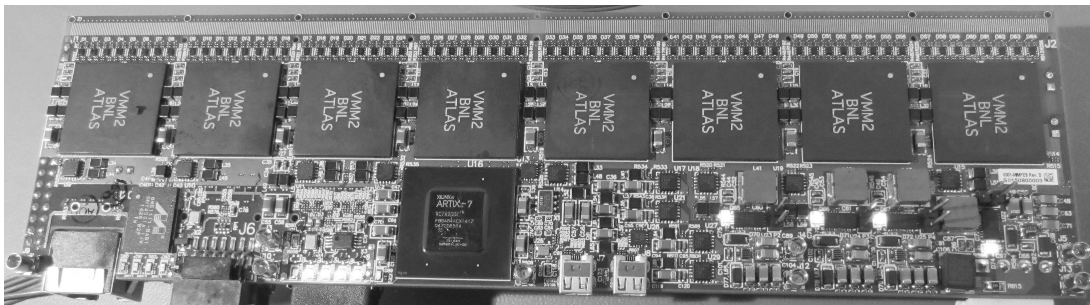
από τρία ζεύγη διαφορικών γραμμών (differential lines). Στη μία γραμμή μεταδίδεται το σήμα του ρολογιού (Clk+, Clk-), στην άλλη πραγματοποιείται η μετάδοση των δεδομένων (Dout+, Dout-), και στην τελευταία η λήψη των δεδομένων (Din+, Din-). Το κάθε e-link υποστηρίζει τρεις διαφορετικές ταχύτητες μετάδοσης δεδομένων (80, 160, 320 Mbps) οι οποίες μπορούν να οριστούν από το χρήστη. Το επίπεδο τάσεων των e-link ακολουθεί ως επί το πλείστον τα πρότυπο LVDS και SLVS [27, 28].

Στο επόμενο Κεφάλαιο θα μελετηθεί το firmware που έχει αναπτυχθεί για την MMFE8, επομένως στη συνέχεια θα μελετηθεί η συγκεκριμένη πλακέτα εκτενέστερα, όπως επίσης και το VMM ASIC, οκτώ από τα οποία βαίνουν απάνω στην MMFE8.

5.6 Micromegas Front-End Board

Η πλακέτα *Micromegas Front-End Board (MMFE8)*, είναι εκείνη που συνδέεται με τον ανιχνευτή Micromegas, και συλλέγει τα ηλεκτρικά σήματα (που είναι ουσιαστικά μία συγκέντρωση των φορτίων που προέρχονται από αλλεπάλληλους ιονισμούς από τα strips. Έχει διαστάσεις $215 \times 60\,\mathrm{mm}^2$. Πρόκειται ουσιαστικά για τον ενδιάμεσο μεταξύ του ανιχνευτή και των δύο καρτών που διαχειρίζονται τα δεδομένα για το trigger (ADDC) και τα δεδομένα για την ενέργεια και το χρόνο εμφάνισης των παλμών που θα χρησιμεύσουν αργότερα στην ανακατασκευή των τροχιών και των ορμών (L1DDC) [30]. Το νούμερο ”8” στο όνομα υπονοεί ότι κάθε MMFE8 έχει πάνω της οκτώ VMM ASIC, το κάθε ένα εκ των οποίων συνδέεται με 64 κανάλια του Micromegas (άρα 64 strips), και προβαίνει σε λειτουργίες ανάλυσης σημάτων, όπως ενίσχυση, διαμόρφωση σχήματος, εύρεση κορυφών, και ψηφιοποίηση. Εκτός από τα οκτώ VMM ASIC, η τελική κάρτα MMFE8, θα διαθέτει και άλλα δύο ASIC: Το *SCA (Slow Control ASIC)*, και το *ROC Read-Out Controller*. Το SCA δέχεται τα σήματα του configuration της MMFE και στέλνει σήματα για την παρακολούθηση των συνθηκών που επικρατούν στον ανιχνευτή *(monitoring)*. Η διασύνδεση αυτή γίνεται με την κάρτα L1DDC μέσω των e-link (configuration path). Το ROC, συλλέγει τα δεδομένα από τα VMM, τα συμπτύσσει, και τα αποστέλλει σειριακά στην L1DDC μέσω των e-link (DAQ path). Ο ρυθμός μετάδοσης/λήψης δεδομένων είναι σταθερός στα 80 Mbps για τα e-link του SCA. Από την άλλη, αν η MMFE βρίσκεται στο εσωτερικό τμήμα του Micromegas wedge (βλ. σχ. 5.4.II), ο ρυθμός μετάδοσης δεδομένων του ROC είναι στα 320 Mbps ενώ αν βρίσκεται στο εξωτερικό, είναι στα 160 Mbps.

Ο λόγος για τον οποίο συμβαίνει αυτό είναι επειδή η υψηλότερη ροή σωματιδίων στο εσωτερικό τμήμα (μεγαλύτερες τιμές του $|\eta|$) απαιτεί και υψηλότερο εύρος ζώνης μετάδοσης δεδομένων για τις συγκρούσεις [28]. Αυτή τη στιγμή, τα πρωτότυπα της MMFE8 που έχουν κατασκευαστεί δεν διαθέτουν ROC/SCA ASIC, αλλά στη θέση τους υπάρχει ένα Xilinx® FPGA (Artix XC7A200T-2FBG484), για το οποίο έχει αναπτυχθεί firmware που να προσομοιώνει τη λειτουργικότητα των δύο συνοδευτικών ASIC. Το FPGA θα χρησιμεύσει στο να δοκιμαστεί η απόδοση της πλακέτας σε πειραματικές συνθήκες με μικρότερη ροή σωματιδίων (όπως Test Beams και πειράματα με κοσμικές ακτίνες), άρα και αμελητέα επίπεδα ακτινοβόλησης. Οι τελικές κάρτες που θα χρησιμοποιηθούν στο NSW δεν θα διαθέτουν FPGA καθώς οι συνθήκες που θα επικρατούν στον ανιχνευτή ATLAS μετά την αναβάθμισή του μπορεί να καταστρέψουν τα FPGA ($1700\,\mathrm{Gy}$ δόση και $0.4\,\mathrm{T}$ μαγνητικό πεδίο [29]). Η έρευνα-ανάπτυξη και σχεδίαση για τις κάρτες MMFE8 πραγματοποιείται στο πανεπιστήμιο της Αριζόνα των ΗΠΑ.

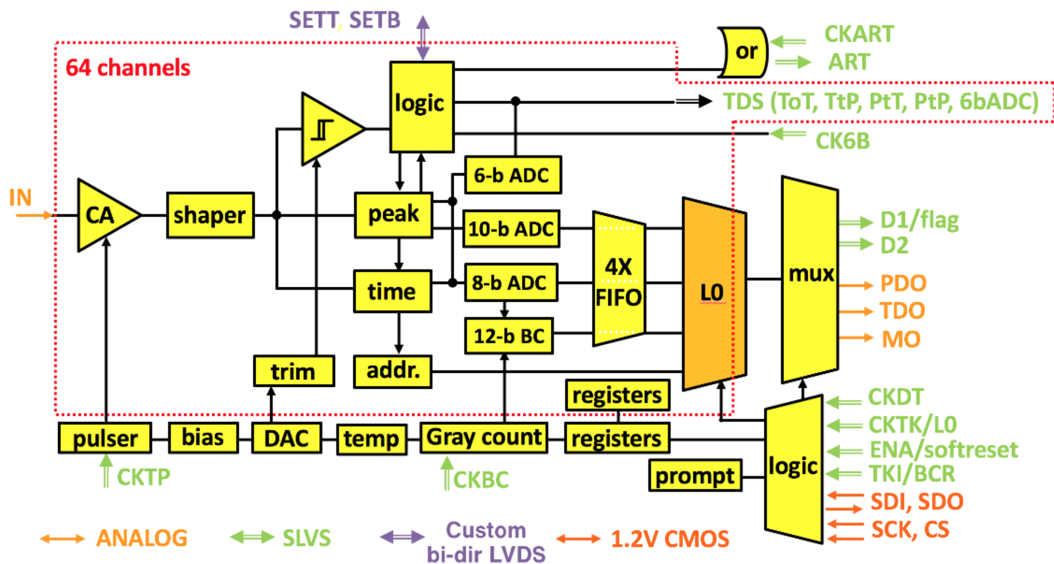


Σχήμα 5.6.I: Φωτογραφία ενός πρωτότυπου MMFE8. Διακρίνονται τα οκτώ VMM ASIC στο πάνω μέρος της πλακέτας, και το Xilinx® FPGA στο κάτω. [30]

5.6.1 VMM ASIC

Το VMM, είναι ένα front-end readout ASIC με 64 κανάλια εισόδου, που αναπτύσσεται για την αναβάθμιση του μιονικού σπεκτρομέτρου του ATLAS από το Brookhaven National Laboratory (BNL) στο Upton, NY των ΗΠΑ. Κάθε πλακέτα MMFE8 έχει πάνω της οκτώ VMM ASIC, και κατά συνέπεια συλλέγει δεδομένα από 512 strips του Micromegas. Το VMM, έχει διαστάσεις $13.5 \times 8.4\,\mathrm{mm}^2$ και περιέχει περίπου πέντε εκατομμύρια τρανζίστορ σχεδιασμένα με τεχνολογία CMOS στα $130\,\mathrm{nm}$ [27]. Κάθε κανάλι που είναι συνδεδεμένο με ένα read-out strip, υλοποιεί έναν ενισχυτή

φορτίου (charge amplifier), έναν ενισχυτή διαμόρφωσης (shaping amplifier) ο οποίος δρα ως φίλτρο, έναν διευκρινιστή (discriminator) με κατώτερο κατώφλι και εντοπισμό κορυφής, και έναν TAC. Τα αναλογικά σήματα που παράγονται από αυτά τα ηλεκτρονικά υποσυστήματα, ψηφιοποιούνται από τρεις διαφορετικούς ADC, με εξόδους 6,8 και 10 bit. Τα (ψηφιακά πλέον) δεδομένα που προκύπτουν από την ανάλυση των παλμών, αποθηκεύονται σε μία FIFO η οποία αποστέλλει τα δεδομένα στο ROC ASIC της MMFE8 [31, 32]. Σε γενικές γραμμές, το VMM προσφέρει ακριβείς μετρήσεις για το φορτίο που εναποτίθεται στα strips, αλλά και για το πότε συνέβη αυτή η συλλογή φορτίου, σε σχέση με το BC clock[12]. Επίσης, παρέχει και άμεσα δεδομένα που χρησιμεύουν για το triggering, ενώ η διαδικασία του readout από την L1DDC είναι πιο σπάνια, και εξαρτάται από το αν ένα γεγονός θεωρηθεί έγκυρο από τον trigger processor. Η διαδικασία αυτή θα ξεκαθαριστεί στη συνέχεια, μετά από μία σχηματική αναπαράσταση των ηλεκτρονικών υποσυστημάτων του VMM (Σχήμα 5.6.II):

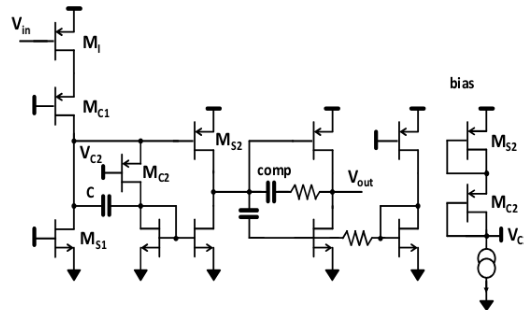


Σχήμα 5.6.II: Σχηματική αναπαράσταση της αρχιτεκτονικής ενός καναλιού του VMM. [33]

Όταν ένας παλμός εισέλθει στο VMM, ο προενισχυτής μετατρέπει το συνολικό φορ-

[12]Το Bunch Crossing (BC) clock είναι ένα ρολόι με περίοδο 25 ns το rising edge του οποίου ταυτίζεται με τη στιγμή που τα πρωτόνια της δέσμης του LHC διασταυρώνονται μεταξύ τους. Ταυτίζεται επομένως με τη στιγμή που δημιουργούνται τα σωματίδια υπό ανίχνευση στο κέντρο του ATLAS.

τίο σε έναν αναλογικό παλμό τάσης ο οποίος μορφοποιείται από έναν ημι-Γκαουσιανό shaper, που παράγει το τελικό σήμα τάσης. Το πλάτος του σήματος τάσης το μέγιστο είναι ανάλογο του φορτίου που εναποτέθηκε στο κανάλι. Το σχεδιάγραμμα σε επίπεδο CMOS του προενισχυτή παρατίθεται στο Σχήμα 5.6.III.



Σχήμα 5.6.III: Σχεδιάγραμμα του ενισχυτή του VMM σε επίπεδο τρανζίστορ. [31]

Το επεξεργασμένο σήμα από τον shaper εισέρχεται στη συνέχεια στον discriminator, ο οποίος έχει ρυθμιζόμενο κατώτατο κατώφλι. Μόλις κάποιος παλμός ξεπεράσει το κατώφλι αυτό, ενεργοποιείται μία μέθοδος εντοπισμού της κορυφής του παλμού (peak detection). Τη στιγμή που εντοπίζεται η κορυφή και καταγράφεται το ύψος της, ενεργοποιείται ο Time-to-Amplitude Converter (TAC), ο οποίος λαμβάνει το σήμα Start και ξεκινάει να εκφορτίζει έναν πυκνωτή. Το σήμα Stop έρχεται με το επόμενο falling-edge του BC clock όπου και ο πυκνωτής παύει να εκφορτίζεται. Με αυτό τον τρόπο, υπολογίζεται το ύψος του παλμού, άρα και το φορτίο που συλλέχθηκε, που αντιστοιχείται στο πότε εμφανίστηκε ο παλμός, με ακρίβεια της τάξης των ps. Αξίζει επίσης να σημειωθεί πως όταν ένα σήμα ξεπερνάει το κατώφλι του discriminator, ενεργοποιείται και ένα κύκλωμα (neighbor logic) που ειδοποιεί τα γειτονικά κανάλια να αγνοήσουν τον δικό τους discriminator, και να καταγράψουν έτσι και αλλιώς τις ενέργειες των παλμών τους, όσο μικροί και αν είναι αυτοί. Αυτή η έξυπνη μεθοδολογία επιτρέπει στο να τεθεί ψηλά το επίπεδο του κατωφλίου, χωρίς ουσιαστικά να χάνονται πληροφορίες, αφού όταν καταγραφεί μεγάλος παλμός σε ένα κανάλι, τα διπλανά θα αποθηκεύσουν με τη σειρά τους την ενέργεια που συνέλεξαν, η οποία δεν θα πρέπει να αγνοηθεί, όσο μικρή και αν είναι αυτή, καθώς θα αντιστοιχεί λογικά σε ένα cluster του ανιχνευτή Micromegas. [26, 27, 31, 32].

Τρία διαφορετικά κυκλώματα μετατροπής από αναλογικό σε ψηφιακό (Analog to Digital Converters (ADC)), δέχονται τα αποτελέσματα της ανάλυσης των σημάτων,

και τα ψηφιοποιούν. Πιο συγκεκριμένα, για τη μέτρηση του χρόνου, ο 8-bit ADC, ψηφιοποιεί τον παλμό του πυκνωτή από τον TAC, και συμπτύσσει το ένα byte που παράγει, με ένα αλφαριθμητικό μήκους 12-bit, το οποίο δημιουργείται από έναν Gray Code counter που αυξάνεται από το BC clock. Ουσιαστικά, η έξοδος του TAC, παράγει την πληροφορία της ακριβούς μέτρησης του χρόνου, και ο counter, ο οποίος φτάνει στη μέγιστη τιμή του και μηδενίζεται σε πιο αραιά χρονικά διαστήματα, παρέχει μία σχετικά πιο χονδροειδή εκτίμηση του χρόνου [32]. Συνδυάζοντας όμως και τα δύο αποτελέσματα, κατασκευάζεται ένα αλφαριθμητικό μήκους 20 bit, που φέρει την πληροφορία του ακριβούς χρόνου (timestamp) που εμφανίστηκε ένας παλμός με διακριτική ικανότητα της τάξης του ps. Για τη μέτρηση της ενέργειας, ο 10-bit ADC λαμβάνει το σήμα που φέρει την πληροφορία του μέγιστου ύψους του παλμού από τον peak detector, και το ψηφιοποιεί με μεγάλη ακρίβεια σε περίπου 200 ns. Ο ADC των 6 bit από την άλλη, προβαίνει και αυτός σε μετατροπή του σήματος του peak detector, αλλά με λιγότερη ακρίβεια σε σχέση με τον ADC των 10 bit. Η όλη διαδικασία ολοκληρώνεται σε 400 ns. Το αν θα χρησιμοποιηθεί ο πιο ακριβής ADC ή ο λιγότερο ακριβής για την κατασκευή του ψηφιακού πακέτου που θα εμπεριέχει την πληροφορία της συνολικής ενέργειας, εξαρτάται από τις προτιμήσεις του χρήστη.

Το τρίτο πρωτότυπο του VMM, υποστηρίζει δύο τύπους διαβάσματος των ψηφιακών δεδομένων από το FPGA/ROC [33]. Ο πρώτος, που υποστηρίζεται και από τις παλαιότερες εκδόσεις του VMM, ονομάζεται *continuous readout*, και η πληροφορία για την ενέργεια και το χρόνο εμφάνισης ενός παλμού για κάποιο κανάλι μορφοποιείται σε ένα πακέτο από 38 bit. Το πρώτο είναι το readout flag, το δεύτερο είναι το flag που υποδεικνύει ότι ξεπεράστηκε το κατώφλι του discriminator, μετά ακολουθούν τα 6 bit της διεύθυνσης του καναλιού, μετά τα 10 bit του ύψους του παλμού (δηλαδή η ενέργεια που εναποτέθηκε από το σωματίδιο), και τέλος τα 20 bit του timestamp. Η τελική συμβολοσειρά αποθηκεύεται σε μία FIFO η οποία έχει τέσσερις διαθέσιμες θέσεις. Όταν το εξωτερικό τσιπ θελήσει να διαβάσει αυτά τα δεδομένα, θα αδειάσει όλα τα περιεχόμενα αυτής της μνήμης FIFO. Ο δεύτερος τρόπος διαβάσματος των δεδομένων του VMM, ονομάζεται *level-0 readout*, και είναι ο τρόπος με τον οποίο θα αποσπούνται τα δεδομένα στο τελικό πείραμα. Στο level-0 readout, η επικοινωνία μεταξύ των δύο τσιπ υλοποιείται μέσω του πρωτοκόλλου *8b/10b*, όπου το VMM στέλνει *comma characters*[13] όταν δεν έχει δεδομένα να στείλει. Το τσιπ που

[13]Πρόκειται για προσυμφωνημένους χαρακτήρες μεταξύ πομπού και δέκτη, οι οποίοι ανταλλάσ-

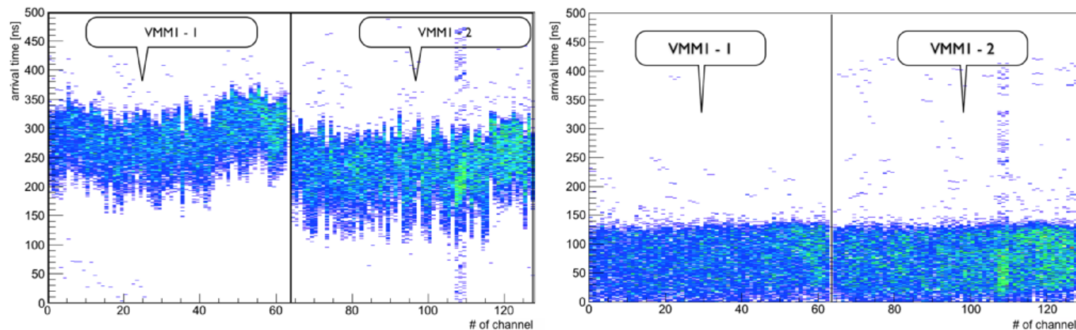
θέλει να διαβάσει δεδομένα από το VMM, πρέπει να στείλει ένα σήμα, το *level-0* στο VMM. Το σήμα αυτό, θα λάβει ένα timestamp από το VMM, όπως και τα γεγονότα που καταγράφει από τον ανιχνευτή. Βάσει αυτού του timestamp και του πως έχει διαμορφωθεί η λειτουργία του VMM από το χρήστη, το VMM θα "ψάξει" σε μία συγκεκριμένη περιοχή της μνήμης του για κάποιο γεγονός. Αν αυτό βρεθεί, τότε θα στείλει ένα συγκεκριμένο πακέτο δεδομένων (παρόμοιο με αυτό που περιγράφηκε πριν), που περιέχει την πληροφορία που συνέλεξαν όλα τα κανάλια για εκείνη την περιοχή της μνήμης. Αυτή η λειτουργία είναι ιδιαίτερα χρήσιμη, καθώς αναπόφευκτα το VMM θα συλλέγει ηλεκτρονικό θόρυβο, αλλά και γεγονότα που δεν αντιστοιχούν σε μιόνια. Μόνο λίγα από αυτά τα δεδομένα θα αντιπροσωπεύουν χρήσιμα γεγονότα, και με αυτό τον τρόπο μπορεί κανείς να επιλέξει ακριβώς τα δεδομένα που θέλει από το VMM. Στο τελικό πείραμα, οι πληροφορίες από τους επεξεργαστές σκανδαλισμού θα συνδυάζονται προκειμένου να αποφανθεί σε ποια χρονική στιγμή αντιστοιχεί η πραγματική τροχιά ενός μιονίου, και έτσι θα στέλνεται την κατάλληλη στιγμή το σήμα level-0 στα VMM ώστε να εξαχθεί αυτή την πληροφορία και μόνο.

Αξίζει επίσης να γίνει αναφορά σε διάφορες δευτερεύουσες λειτουργίες που διαθέτει το VMM. Μια τέτοια επιπρόσθετη λειτουργία, είναι η ύπαρξη κυκλωμάτων για το *calibration* (βαθμονόμηση) του κάθε καναλιού. Πιο συγκεκριμένα, έχει βρεθεί πως το κάθε κανάλι του VMM παρουσιάζει διακυμάνσεις ως προς την ενίσχυση κατά τη μετατροπή των παλμών φορτίου σε παλμούς τάσης. Προκειμένου να γίνουν σωστές μετρήσεις στις κυματομορφές των παλμών, οι σχετικές αυτές διακυμάνσεις πρέπει να είναι γνωστές. Για αυτό το λόγο, το κάθε κανάλι του VMM διαθέτει ένα εξωτερικό κύκλωμα που διοχετεύει με παλμούς το κανάλι (pulser), ώστε να μετρηθεί η ακριβής απόκριση του καναλιού [26]. Εκτός από το gain, είναι επίσης σημαντικό να γίνει βαθμονόμηση και του επιπέδου του κατωφλίου του discriminator[14], γεγονός που επιτυγχάνεται επίσης με τη χρήση του pulser. Βαθμονόμηση επίσης επιδέχεται και ο TAC [27], ο οποίος παρουσιάζει μικροδιαφορές ως προς τη λειτουργικότητά του ανά κανάλι. Με τη χρήση του pulser, διοχετεύονται στο σύστημα παλμοί με συγκεκριμένο χρονικό προφίλ, που μπορεί να οριστεί από το χρήστη. Στη συνέχεια τα αποτελέσματα του TAC επιστρέφονται πίσω, όπου υπολογίζονται κάποιες χρονικές σταθερές διόρθωσης για κάθε κανάλι. Αυτές οι σταθερές, θα ληφθούν υπόψιν αργότερα στην ανάλυση του χρόνου όπως αυτός δίνεται από τον κάθε TAC, ώστε οι

σονται προκειμένου να επιτευχθεί συγχρονισμός και ευθυγράμμιση φάσης.

[14]Ώστε να μην επηρεάζει τις μετρήσεις ο εγγενής θόρυβος που υπάρχει σε κάθε κανάλι.

μετρήσεις κατά τη διάρκεια του πειράματος να είναι ακριβέστερες. Το αποτέλεσμα του TAC calibration απεικονίζεται στο Σχήμα 5.6.IV.



Σχήμα 5.6.IV: Η κατανομή του υπολογισμένου χρόνου για δύο VMM ASIC, πριν (αριστερά) και μετά (δεξιά) το calibration. Είναι φανερό πως πριν τη βαθμονόμηση, παρουσιάζονται σημαντικές διακυμάνσεις στο χρόνο εμφάνισης των παλμών, όπως αυτός δίνεται από τον TAC. Μετά το calibration, η κατανομή είναι σαφώς πιο ισορροπημένη [27].

Όλες αυτές οι λειτουργίες του calibration, μαζί με το ακριβές configuration των επιμέρους λεπτομερειών του τρόπου λειτουργίας του VMM, υλοποιούνται ουσιαστικά μέσω της επικοινωνίας με την L1DDC. Όταν η L1DDC λάβει εντολή από το κέντρο ελέγχου του ATLAS ότι ένα συγκεκριμένο chip VMM πρέπει να αλλάξει κάτι στον τρόπο λειτουργίας του (π.χ. χρήση του 6-bit ADC αντί του 10-bit, αλλαγή του επιπέδου κατωφλίου του discriminator, κ.ά.), ή πρέπει να προβεί σε calibration, στέλνει την εντολή αυτή στο SCA ASIC της αντίστοιχης MMFE μέσω του e-link, και το SCA διοχετεύει τα ψηφιακά δεδομένα σε ένα DAC (Digital-to-Analog Converter) μέσα στο VMM. Ο αποκωδικοποιητής DAC, μετατρέπει το ψηφιακό σήμα της εντολής σε αναλογικό παλμό, ο οποίος αποστέλλεται στο αντίστοιχο ηλεκτρονικό υποσύστημα του ASIC που πρέπει να γίνει η διαμόρφωση της λειτουργίας. Με αυτό τον τρόπο, υπάρχει μία πλήρης επικοινωνία μεταξύ του κέντρου ελέγχου και του VMM.

ART Data (Trigger Data) και ADDC

Μαζί με την ανάλυση των σημάτων, το VMM παράγει και τα δεδομένα που χρησιμεύουν για το *trigger*. Πρόκειται για τα *Address in Real Time (ART)* data. Καθ' όλη τη διάρκεια της λειτουργίας του, το VMM εποπτεύει τα αποτελέσματα από τους

discriminator όλων των καναλιών του. Όταν κάποιος παλμός ξεπεράσει το κατώφλι του discriminator, ή όταν βρεθεί η πρώτη κορυφή ενός παλμού, τότε το VMM καταγράφει αμέσως τη διεύθυνση του καναλιού που εντοπίστηκε το γεγονός, και τη στέλνει στην ADDC μαζί με ένα flag που υποδεικνύει ότι συνέβη το γεγονός. Τα σήματα ART, αποστέλλονται στην ADDC μέσω ενός e-link σε κάθε bunch crossing (κάθε 25 ns). Στην ADDC, βρίσκονται τέσσερα ASIC: δύο ART2GBTx και δύο GBTx ASIC. Τα δύο πρώτα ASIC, επεξεργάζονται τα ART data που τους στέλνονται από τις 8 MMFE μέσω των e-link, και επιλέγουν τα σήματα που θα στείλουν στα GBTx. Τα GBTx, που είναι ουσιαστικά ASIC που προβαίνουν σε multiplexing πολλών σημάτων σε μία έξοδο οπτικής ίνας μεγάλης ταχύτητας, στέλνουν τα δεδομένα ART στα back-end electronics όπου γίνεται το trigger processing. Ο trigger processor, που αποτελείται κυρίως από FPGA, συνδυάζει τα δεδομένα του trigger από τους sTGC και τους MM, μαζί με τα αντίστοιχα δεδομένα από το Big Wheel. Για να συγχρονιστούν τα trigger data και των τριών ανιχνευτικών υποσυστημάτων, τα ηλεκτρονικά του κάθε ανιχνευτή πρέπει να αποστείλουν τα δεδομένα τους μέσα σε αυστηρά καθορισμένα χρονικά πλαίσια. Για τον MM για παράδειγμα, από τη στιγμή που θα γίνει μία αλληλεπίδραση στο κέντρο του ATLAS, μέχρι να φτάσουν τα δεδομένα του trigger της αλληλεπίδρασης αυτής στο κομμάτι εκείνο του trigger processor που συνδυάζει τα δεδομένα και των υπολοίπων ανιχνευτών, πρέπει να παρέλθει ένα μέγιστο χρονικό διάστημα της τάξης των $t_{max} = 1025\,\mathrm{ns}$. Αυτή η καθυστέρηση *(latency)* εξαρτάται από πολλούς παράγοντες, όπως για παράδειγμα: το χρόνο απόκρισης στο shaper του VMM, την ταχύτητα με την οποία το GBTx στην ADDC συμπτύσσει τα δεδομένα από τα οκτώ e-links και τα αποστέλλει σειριακά στην οπτική ίνα που οδηγεί στον trigger processor, ή και την ταχύτητα μετάδοσης της πληροφορίας στις οπτικές ίνες που μόλις αναφέρθηκαν. Έχει υπολογιστεί πως το σύστημα ηλεκτρονικών του Micromegas, προσφέρει ελάχιστη καθυστέρηση $t_{min} = 876\,\mathrm{ns}$, και μέγιστη $t'_{max} = 1018\,\mathrm{ns}$, που είναι ένα χρονικό πλαίσιο πλήρως αποδεκτό βάσει του σχεδιασμού ολόκληρου του πειράματος [27]. Τελικά, με τη σύγκριση όλων των δεδομένων του trigger, ο επεξεργαστής αποφαίνεται για το ποια γεγονότα είναι έγκυρα (βλ. Σχ. 5.5.I), και αποστέλλει μέσω της L1DDC το σήμα του Level-0, επιλέγοντας έτσι μόνο τα χρήσιμα δεδομένα από τους buffers του VMM, όπως περιγράφηκε παραπάνω.

Σύνοψη

Σε αυτό το Κεφάλαιο, μελετήθηκε αρχικά η πειραματική διάταξη του ATLAS, και ύστερα η γενική δομή των ηλεκτρονικών υποσυστημάτων που θα υποστηρίξουν την αναβάθμιση του New Small Wheel, η οποία θα ολοκληρωθεί το 2020. Οι πλακέτες MMFE8, με το εξελιγμένο ASIC ονόματι VMM, συλλέγουν τα σήματα από τον ανιχνευτή Micromegas, και τα προωθούν στο δίκτυο των back-end electronics για περαιτέρω επεξεργασία, ενώ ο κύριος συνδετικός κρίκος σε αυτή τη διαδικασία είναι η κάρτα L1DDC. Τα πρωτότυπα της πλακέτας MMFE8 όμως δε διαθέτουν τα ROC/SCA ASICs, τα οποία στο τελικό πείραμα θα διαμορφώνουν τη λειτουργία του πολύπλοκου VMM, και θα αποσπούν τα δεδομένα από αυτό, αλλά ένα FPGA. Έχοντας κατά νου το πως λειτουργεί το VMM, και τη συνδεσμολογία της MMFE8 με τα υπόλοιπα κομμάτια του συστήματος ηλεκτρονικών, το επόμενο βήμα είναι να σχεδιαστεί το firmware στο FPGA, που θα υποκαθιστά τη λειτουργία των ROC/SCA. Στο επόμενο Κεφάλαιο, θα περιγραφεί το firmware που έχει αναπτυχθεί σε γλώσσα VHDL, και θα ξεκαθαρίσει ακόμα περισσότερο ο τρόπος με τον οποίο λειτουργεί το VMM ASIC.

6

VMM Readout/Configuration Firmware

Σε αυτό το Κεφάλαιο, θα παρουσιαστεί το firmware που έχει αναπτυχθεί για τα FPGA των πλακετών στα οποία βαίνει το VMM3[1] ASIC. Το εν λόγω design έχει δημιουργηθεί για να τεστάρει τις λειτουργίες του VMM, προκειμένου να υπάρχει πλήρης επίγνωση για τον τρόπο που συμπεριφέρεται το τσιπ κάτω από όλες τις δυνατές συνθήκες. Έτσι, οι σχεδιαστές του VMM μπορούν να γνωρίζουν τι αλλαγές πρέπει να γίνουν στην αρχιτεκτονική του, πριν την τελική τοποθέτηση των ηλεκτρονικών στους ανιχνευτές του NSW. Εκτός από τη μελέτη των λειτουργιών του VMM όμως, το firmware κρίνεται απαραίτητο για υποστήριξη σε συνθήκες *Testbeam*. Οι ανιχνευτές MicroMegas, από τους οποίους θα αποσπά ηλεκτρικά σήματα το VMM, θα δοκιμαστούν σε ακτινοβόληση δέσμης αρκετές φορές πριν εγκατασταθούν στο τελικό πείραμα. Μόνο έτσι θα μπορέσει να γίνει χαρακτηρισμός της λειτουργίας του ανιχνευτή, καθώς από τα δεδομένα του VMM, μπορεί να γίνει η ανάλυση που θα υποδεικνύει την ποιότητα της διάταξης.

Η ευελιξία του FPGA το καθιστά ιδανικό για αυτή τη δουλειά, καθώς ο τρόπος με τον οποίο λειτουργεί μπορεί να αλλάξει δυναμικά ώστε να τεστάρει διαφορετικές πτυχές του VMM, ή να διαμορφωθεί προκειμένου να αντεπεξέλθει στις συνθήκες πειραμάτων testbeam. Το firmware αυτό, υποστηρίζει αρκετές πλακέτες[2] οι οποίες φιλοξενούν το VMM ASIC. Οι MMFE8 έχουν παραχθεί για το NSW, όμως το VMM δύναται να καταγράψει δεδομένα και από άλλους τύπους ανιχνευτών. Οι άλλες

[1] Πρόκειται για το τρίτο πρωτότυπο του τσιπ. Πολλές από τις λειτουργίες του είναι διαθέσιμες και στις προηγούμενες εκδόσεις του.

[2] MMFE8, MMFE1, MDT_446, MDT_MU2E, GPVMM.

πλακέτες είναι συμβατές με τον MicroMegas αλλά και με άλλους θαλάμους ανίχνευσης, και διαθέτουν συνήθως μόνο ένα VMM (αντί για οκτώ όπως στην MMFE8), όμως έχουν και αυτές Ethernet Interface, miniSAS connector, και προφανώς FPGA. Έτσι, το firmware που χρησιμοποιείται για διαφορετικές πλακέτες, είναι πρακτικά το ίδιο, αφού τα πρωτόκολλα επικοινωνίας και τα τσιπ που βαίνουν στις πλακέτες δεν αλλάζουν.

Σχήμα 6.0.I: Τρεις πλακέτες που φιλοξενούν VMM, και το FPGA για το οποίο έχει σχεδιαστεί το firmware που θα περιγραφεί σε αυτό το Κεφάλαιο. Πάνω δεξιά είναι η MDT_446, πάνω αριστερά η MMFE1, και κάτω η MMFE8. Στις πλακέτες διακρίνεται το FPGA, το VMM και οι κοννέκτορες του Ethernet.

Το firmware, κατά κύριο λόγο, υλοποιεί δύο διαφορετικά *paths*:

- *Configuration Path:* Πρόκειται για τις εντολές *(slow control)* που προέρχονται από το χρήστη, και έχουν σαν σκοπό να διαμορφώσουν δυναμικά τις λειτουργίες του FPGA ή/και του VMM. Υλοποιείται μέσω του πρωτοκόλλου Ethernet/UDP, όπου εξειδικευμένο λογισμικό[3] στέλνει τις εντολές στο FPGA, το οποίο στη συνέχεια τις αποκωδικοποιεί και δρα αναλόγως. Το συγκεκριμένο path στην

[3] Θα καλείται από εδώ και στο εξής ως Configuration/DAQ software.

MMFE8 θα υλοποιείται στο τελικό πείραμα από το SCA ASIC, και οι εντολές του configuration θα προέρχονται από το κέντρο ελέγχου του ATLAS, και θα δρομολογούνται στην MMFE8 μέσω της L1DDC και των e-link.

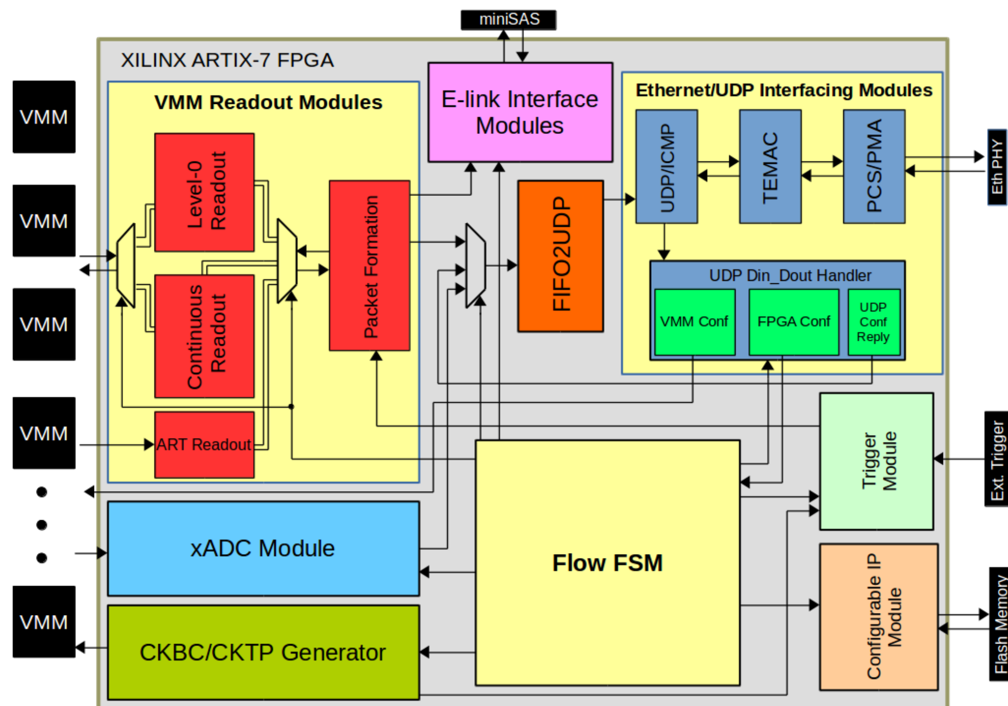
- *DAQ/Readout Path:* Το *Data AcQuisition*, που θα υλοποιείται στην τελική MMFE8 από το ROC ASIC, είναι η διαδικασία απόσπασης της ψηφιακής πληροφορίας από τους αναλογικούς παλμούς που λαμβάνει το VMM. Όπως αναφέρθηκε και στο προηγούμενο Κεφάλαιο, υπάρχει το *continuous readout mode* και το *level-0 readout mode*. Ο πρώτος τρόπος είναι όμοιος με εκείνον του VMM2, ενώ ο δεύτερος υφίσταται μόνο στο VMM3, το οποίο εν προκειμένω θα στέλνει τα δεδομένα σε πακέτα κωδικοποιημένα με *8b/10b encoding*, και θα χρησιμοποιείται στο τελικό πείραμα του ATLAS. Το FPGA δύναται να υλοποιήσει και τις δύο περιπτώσεις στη μέγιστη ταχύτητα των 160 Mbps. Στο τελικό πείραμα τα δεδομένα θα προωθούνται στην L1DDC μέσω του ROC και των e-link. Εν προκειμένω, το FPGA στέλνει τα δεδομένα στο DAQ software μέσω του UDP, και το λογισμικό με τη σειρά του κατασκευάζει αρχεία .ROOT για ανάλυση των δεδομένων.

6.1 Ethernet Communication - Configuration Path

Σε αυτή την ενότητα θα περιγραφούν όλα τα κομμάτια του design που έχουν να κάνουν με τη διαμόρφωση της λειτουργίας του FPGA και του VMM, από τα block που είναι υπεύθυνα για την λήψη των Ethernet frames (PCS/PMA, TEMAC), στα components που αναγνωρίζουν (UDP/ICMP block) το UDP πακέτο και το προωθούν στη λογική του χρήστη (UDP_din_handler).

6.1.1 Ethernet/UDP Interfacing Modules

Προκειμένου να πραγματοποιηθεί η επικοινωνία του FPGA με τον ηλεκτρονικό υπολογιστή, απαιτείται η υλοποίηση ενός τμήματος των OSI Layers μέσα στο FPGA (βλ. Παράρτημα Α΄). Τα OSI (Open Systems Interconnection) Layers, είναι διάφορα αλληλοεπικαλυπτόμενα επίπεδα που προτυποποιούν και κατηγοριοποιούν τις λειτουργίες που πρέπει να επιτελέσει ένας κόμβος, προκειμένου αυτός να μπορεί να λάβει μέλος σε ένα δίκτυο επικοινωνιών. Τα επίπεδα αυτά είναι: Application, Presentation,



Σχήμα 6.0.II: Γενικό Block Diagram του firmware.

Session, Transport, Network, Data Link και Physical, με σειρά από το ανώτερο στο κατώτερο [11]. Στο firmware της προκείμενης εφαρμογής, τα τμήματα των επιπέδων που πρέπει να μοντελοποιηθούν μέσα στο FPGA είναι τα: Transport, Network, Data Link και Physical.

Όπως έχει ήδη γίνει σαφές, τα δεδομένα από τα VMM, θα συγκεντρώνονται από το FPGA, και θα πακετάρονται προκειμένου να σταλούν μέσω Ethernet στον υπολογιστή που θα είναι συνδεδεμένος με την πλακέτα. Για το σκοπό αυτό τα δεδομένα αρχικά πακετάρονται σύμφωνα με το πρότυπο που ορίζει το πρωτόκολλο UDP (Transport Layer). Στη συνέχεια αυτά τα πακέτα προωθούνται στο Network Layer, όπου υλοποιείται το πρωτόκολλο IPv4. Στο firmware της MMFE8, αυτές οι λειτουργίες τελούνται από κοινούς κώδικες VHDL που μπορούν να κατεβαστούν ελεύθερα από το Opencores.org. Τα πράγματα γίνονται πιο περίπλοκα στην υλοποίηση των Data Link και Physical επιπέδων, όπου είναι απαραίτητη η χρήση εξεζητημένων IP Cores που διαθέτει η Xilinx® για τέτοιες εφαρμογές. Πρόκειται για τρία IP Cores:

Tri-mode Ethernet MAC, Ethernet SGMII PCS/PMA και *7-Series GTP Transceiver*. Η εφαρμογή των τριών αυτών IP Cores, σε συνδυασμό με το firmware από το OpenCores, επιτρέπει στον χρήστη να κατασκευάσει ένα πλήρες σύστημα networking στο FPGA, το οποίο μπορεί και υλοποιεί πρωτόκολλο Gigabit Ethernet μέσω χάλκινου καλωδίου.

PCS/PMA & TEMAC

Ο τρόπος με τον οποίο δομείται αυτό το σύμπλεγμα διαφορετικών components μέσα στο firmware είναι ο εξής: Κατ' αρχάς, η διασύνδεση των IP cores, έχει ως εξής: TEMAC $\leftrightarrow$ PCS/PMA-SGMII $\leftrightarrow$ Transceiver. O Transceiver, είναι ένα ηλεκτρονικό υποσύστημα που μπορεί να μεταδίδει σειριακά δεδομένα, μέσω διαφορικών ζευγών, με πολύ υψηλές ταχύτητες. Τα περισσότερα FPGA σήμερα, διαθέτουν ενσωματωμένα hard block που υλοποιούν Transceivers[4], οι ταχύτητες των οποίων φτάνουν στην τάξη των Gbps. Το σύστημα αυτό, γενικά μπορεί να επικοινωνήσει μόνο με άλλα ανάλογα μπλοκ που βρίσκονται εκτός του FPGA. Ένας Transceiver για παράδειγμα, μπορεί να επικοινωνήσει με συστήματα που εφαρμόζουν τις διασυνδέσεις με οπτικές ίνες, να υλοποιήσει PCI Express πρωτόκολλα, και πρωτόκολλα Ethernet. Εν προκειμένω, στην MMFE8 υλοποιείται το πρωτόκολλο Ethernet, και ο Transceiver του FPGA επικοινωνεί με ένα αντίστοιχο εξωτερικό κύκλωμα της Marvell®. Πρόκειται για το *Marvell 88E1111 Integrated Ultra Gigabit Ethernet Transceiver*, το οποίο είναι ένα εξωτερικό Ethernet PHY, που επικοινωνεί αμφίδρομα με το FPGA, και συγκεκριμένα με τον Transceiver. Η διεπαφή αυτή επιτρέπει στα Ethernet frames του δικτύου να εισέρχονται στο FPGA, και στο FPGA να στέλνει frames στο δίκτυο. Στις μεγάλες ταχύτητες, όπου απαιτείται DC-balancing για να μην αλλοιώνονται τα σήματα στις γραμμές, εφαρμόζεται συνήθως 8b/10b κωδικοποίηση (βλ. αντίστοιχο Παράρτημα) η οποία προσφέρει και ευκολότερο clock recovery στα κυκλώματα που συμμετέχουν στη διεπαφή. Υπενθυμίζεται επίσης ότι η 8b/10b κωδικοποίηση, διαθέτει comma characters, τα οποία στέλνονται συνεχώς από τον κόμβο που θέλει να μεταδώσει δεδομένα, καθώς τα comma characters έχουν το προσόν να διευκολύνουν το clock recovery και την ευθυγράμμιση φάσης. Όταν αυτά λαμβάνονται σωστά από τον δέκτη, εκείνος γνωρίζει ότι είναι συμφασικός με τη ροή δεδομένων και μπορεί να κάνει σωστό sampling.

[4]Το FPGA που χρησιμοποιείται στην MMFE8 και στις άλλες πλακέτες που φιλοξενούν το VMM readout firmware, διαθέτει 16 Transceivers, όπου ο κάθε ένας μπορεί να υλοποιήσει διεπαφές Ethernet over copper, Ethernet over fiber, PCIe κ.λπ.

Εν προκειμένω, ο χρονισμός ολόκληρου του κυκλώματος που υλοποιεί το Ethernet firmware, γίνεται από ένα ρολόι το οποίο ανακτά ο Transceiver από τη γραμμή επικοινωνίας με το Marvell ETH PHY[5]. Σαν ρολόι αναφοράς ($f = 125\,\text{Mhz}$), ο Transceiver λαμβάνει ένα διαφορικό σήμα (MGTREFCLK), που προέρχεται από έναν κρύσταλλο υψηλής ποιότητας μέσα στο ETH PHY, και μέσω αυτού, τα εξειδικευμένα PLL του Transceiver συνθέτουν ρολόι από την εξωτερική ροή δεδομένων. Αυτό το ανακτημένο ρολόι, περνάει μέσα από ένα εξωτερικό MMCM (ανάλογο του PLL, βλ. υποεν. 1.1.2) που βρίσκεται σε ένα CMT του FPGA. Αυτό συνθέτει δύο ρολόγια (userclk, userclk2 με συχνότητες 62.5 Mhz και 125 Mhz αντίστοιχα) τα οποία χρονίζουν όλα τα components του κυκλώματος.

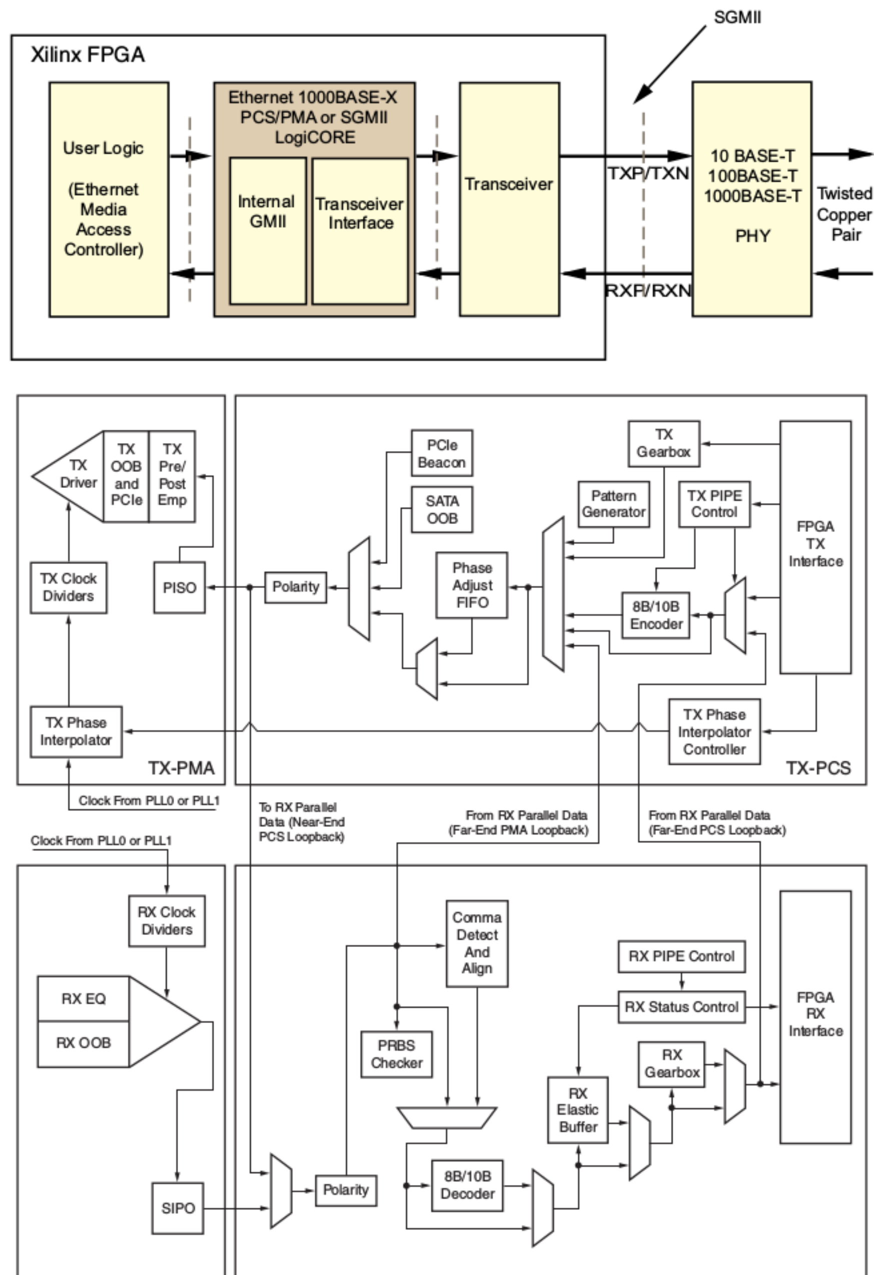
Όταν το UDP/IP block, που υλοποιεί το Transport και Network Layer, θελήσει να στείλει δεδομένα ένα επίπεδο κάτω, στο Link Layer, θα λάβει τον χρονισμό του από τα ρολόγια του MMCM (δηλαδή ουσιαστικά από το ανακτημένο ρολόι), και θα προωθήσει το πακέτο που έχει κατασκευάσει στον TEMAC. Ο TEMAC, είναι ένα αυστηρά ορισμένο (βάσει των προδιαγραφών του IEEE 802.3™[38]) υποσύστημα, το οποίο τελεί τη διασύνδεση του Physical Layer με τα ανώτερα επίπεδα OSI. Πιο συγκεκριμένα, ο TEMAC, λαμβάνει τα πακέτα από το Network Layer (εν προκειμένω πακέτα IPv4 μαζί με UDP)[6], και τα "τυλίγει" με το Ethernet Frame (βλ. Α΄.1.1).

Ο TEMAC προωθεί το Ethernet frame που κατασκεύασε, στο Physical Layer, που υλοποιείται από τον PCS/PMA. Ο τρόπος μετάδοσης των δεδομένων στο firmware, γίνεται σύμφωνα με το πρότυπο *SGMII (Serial Gigabit Media Independent Interface)*, το οποίο είναι μία υποπερίπτωση του GMII (που είναι παράλληλο) [16]. Το Marvell PHY, όντας chip που υλοποιεί διεπαφή Ethernet μέσω χάλκινου καλωδίου και όχι οπτικής ίνας, θέτει άνω όριο στην ταχύτητα στα 1 Gbps, και υποχρεώνει τον PCS/PMA να εφαρμόζει το πρωτόκολλο 1000BASE-T μαζί με το SGMII (το 1000 υποδεικνύει ταχύτητα 1 Gbps, ενώ το -T διασύνδεση μέσω χαλκού).

Μεταξύ του SGMII και του Transceiver, παρεμβάλλεται ένα interface που περιλαμβάνει πύλες flip-flop για συγχρονισμό των δεδομένων. Ο Transceiver, λαμβάνει τα δεδομένα από το SGMII, και τα επεξεργάζεται ώστε να τα αποστείλει σειριακά στις διαφορικές γραμμές που οδηγούν στο εξωτερικό PHY. Ο GTP Transceiver είναι

[5]Πρόκειται για ένα ολοκληρωμένο κύκλωμα που υλοποιεί την διεπαφή του Transceiver με τη γραμμή του Ethernet.

[6]Το UDP προτιμάται στην παρούσα εφαρμογή, επειδή έχει πολύ ελαφρύ overhead



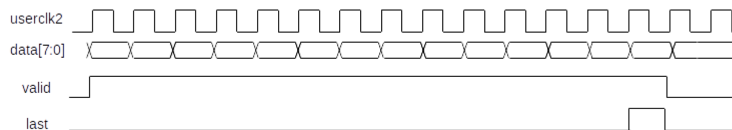
Σχήμα 6.1.I: Πάνω: Η διασύνδεση των τριών IP Cores, τόσο με τα ανώτερα στρώματα OSI, όσο και με το εξωτερικό Ethernet PHY. Κάτω: Τα διάφορα υποσυστήματα του Xilinx® GTP Transceiver που χρησιμοποιεί το FPGA της MMFE8 [16].

ένα αρκετά περίπλοκο κύκλωμα, το οποίο αποτελείται από πολλά υποσυστήματα,

που τελούν διάφορες λειτουργίες. Μία από αυτές είναι για παράδειγμα το 8b/10b encoding. Σύμφωνα με το documentation της Xilinx® , ακόμα και το reset και το power-up του μπλοκ είναι περίπλοκη διαδικασία, καθώς απαιτούνται καλά χρονισμένα σήματα με τη σωστή αλληλουχία για να λειτουργήσει σωστά ο Transceiver [16]. Για το λόγο αυτό, μαζί με τον Transceiver περιλαμβάνονται και δύο ειδικές εξωτερικές FSM, οι οποίες υλοποιούν πολλά από τα σήματα που ελέγχουν τον Transceiver. Η διασύνδεση με το Marvell PHY, γίνεται δρομολογώντας τις σειριακές γραμμές του εσωτερικού Transceiver που βρίσκεται στο FPGA, στα αντίστοιχα pins που υλοποιούν τη διεπαφή με το Marvell PHY.

OpenCores UDP/ICMP Stack

Πρόκειται για τον κώδικα VHDL *1G ETH UDP/IP Stack* από το *Opencores.org*. Το συγκεκριμένο component υλοποιεί το *Internet Layer (IPv4)* και το *Transport Layer (UDP και ICMP)*. Η διεπαφή του είναι από τη μία με τον TEMAC και από την άλλη με το *UDP_din_handler*. Όταν ο TEMAC λάβει ένα ethernet frame, τότε το αποστέλλει στο UDP/ICMP block, σύμφωνα με το διάγραμμα 6.1.II.



Σχήμα 6.1.II: Timing Diagram των δεδομένων του Ethernet, όπως τα προωθεί ο TEMAC στο UDP/ICMP block. Τα δεδομένα στέλνονται ανά byte, σε κάθε clock tick του userclk2. Το σήμα *valid* παίρνει τη λογική τιμή ένα στο πρώτο byte του frame, και μένει ψηλά καθ'όλη τη διάρκεια μετάδοσης του frame, ενώ το *last* θα μείνει ψηλά για έναν κύκλο του ρολογιού, σηματοδοτώντας έτσι το τελευταίο byte του frame.

Το UDP/ICMP Block διακρίνει τα διαφορετικά μέρη του Ethernet frame, και ξεχωρίζει το UDP Datagram από το πακέτο του IPv4 (βλ. αντίστοιχο Παράρτημα και σχ. Α′.1.2), και προωθεί στις εξόδους του αρχικά τα δεδομένα του UDP Header (source/destination πorts, μήκος πακέτου, checksum) και ύστερα τα περιεχόμενά του, μαζί με τα σήματα *valid* και *last* (παρόμοια με το σχήμα 6.1.II, μόνο που αυτή τη φορά τα valid/last δε συνοδεύουν ολόκληρο το Ethernet frame, αλλά μόνο το περιεχόμενο του UDP *(UDP Payload)*) Αυτά τα σήματα θα τα επεξεργαστεί ο *UDP_din handler*, προκειμένου να αποφανθεί τι εντολή εστάλη στο FPGA από το DAQ software.

Ο αρχικός κώδικας του *1G ETH UDP/IP Stack* δεν παρέχει υποστήριξη για το πρωτόκολλο ICMP, αλλά μόνο για το UDP. Αυτό σημαίνει ότι το module που είναι υπεύθυνο για την αναγνώριση του πρωτοκόλλου (στο *Internet Layer*), ξεσκαρτάρει τα πακέτα που στο πεδίο *Upper Layer Protocol*[7], δεν έχουν την τιμή 0x11 που είναι ο δεκαεξαδικός κωδικός του UDP. Προκειμένου το FPGA να μπορεί να ανταποκρίνεται στα *Echo Requests (ή Ping Request)*[8] του υπολογιστή με τον οποίο είναι συνδεδεμένος, κρίθηκε απαραίτητη η προσθήκη λογικής που να αναγνωρίζει το πρωτόκολλο ICMP, και στην περίπτωση που αυτό το πακέτο ICMP αντιστοιχεί σε Echo Request, το firmware να απαντάει με Echo Reply. Για το λόγο αυτό, προστέθηκαν τρία modules:

- *ICMP_RX:* Αναγνωρίζει τα πακέτα που προέρχονται από το IP Layer με κωδικό πρωτοκόλλου 0x01 (δεκαεξαδικός κωδικός του ICMP), και τα προωθεί στον *ping_reply_processor*.

- *ping_reply_processor:* Διαβάζει το πακέτο ICMP που προέρχεται από το *ICMP_RX* και αν πρόκειται για Echo Request, τότε τελεί τις απαραίτητες διεργασίες για να κατασκευάσει το Ping Reply. Συγκεκριμένα, το component αυτό, ελέγχει τα πεδία *Type* και *Code* του πακέτου ICMP, και αν αυτά έχουν δεκαεξαδικούς κωδικούς 0x08 και 0x00 αντίστοιχα[9], τότε προωθεί στο ICMP_TX το πακέτο της απάντησης στο Ping.

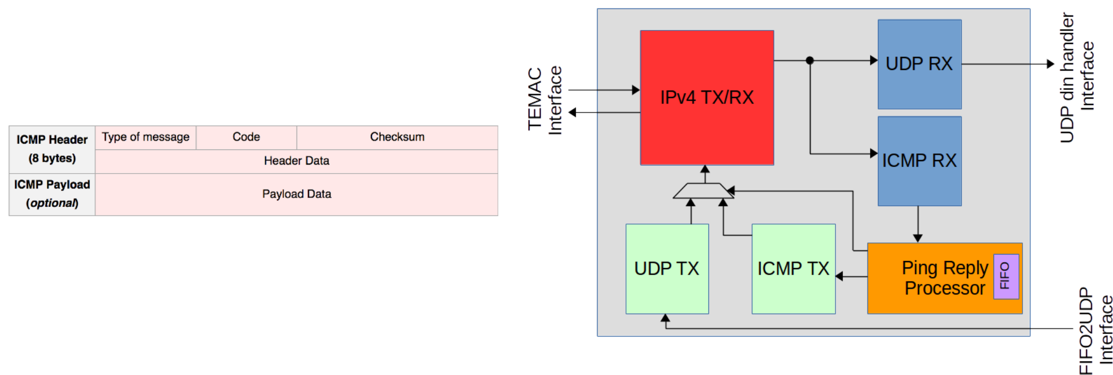
- *ICMP_TX:* Αναγνωρίζει τα πακέτα τύπου Ping Reply όπως του τα στέλνει ο processor, και τα προωθεί στο IP Layer, που με τη σειρά του τα στέλνει στον TEMAC, και από εκεί τελικά στη γραμμή του Ethernet. Έτσι ο κόμβος που έκανε το Ping Request θα λάβει το Reply του.

Στην εικόνα 6.1.III μπορεί κανείς να δει τη δομή ενός ICMP πακέτου, το block diagram των components που μόλις περιγράφηκαν, ενώ στο Παράρτημα Γ΄ βρίσκεται ο κώδικας της FSM του *ping_reply_processor*.

[7] βλ. σχ. Α΄.1.2 του Παραρτήματος Α΄

[8] Πρόκειται για ένα χρήσιμο και δημοφιλές εργαλείο, το οποίο υλοποιείται μέσω του πρωτοκόλλου ICMP. Όταν ο πομπός στείλει Ping Request, τότε ο δέκτης λαμβάνει το πακέτο και απαντάει με Ping Reply. Χρησιμοποιείται για επιβεβαίωση της ορθής επικοινωνίας μεταξύ δύο κόμβων σε ένα δίκτυο.

[9] Πρόκειται για τον κωδικό αναγνώρισης πακέτων που προβαίνουν σε Echo Request.



ICMP Header (8 bytes)	Type of message	Code	Checksum
	Header Data		
ICMP Payload (optional)	Payload Data		

Σχήμα 6.1.III: Αριστερά: Δομή ενός πακέτου ICMP. Αν αυτό το πακέτο είναι Ping Request, τότε type = 0x08, code = 0x00, ενώ στα Header Data βρίσκονται δύο 16-bit πεδία (Identification και Sequence Number). Ο κόμβος που στέλνει το Ping Request θα βάλει επίσης και κάποια bytes στο Payload Data. Ο κόμβος που πρέπει να απαντήσει στο Ping Request, θα αποθηκεύσει το πακέτο αυτό, και θα αλλάξει απλά το πεδίο του type σε 0x00, μαζί με το checksum. Η διαδικασία κατασκευής του πακέτου του Ping Reply μπορεί να βρεθεί στον κώδικα Γ΄.10 Δεξιά: Η δομή του UDP/ICMP block.

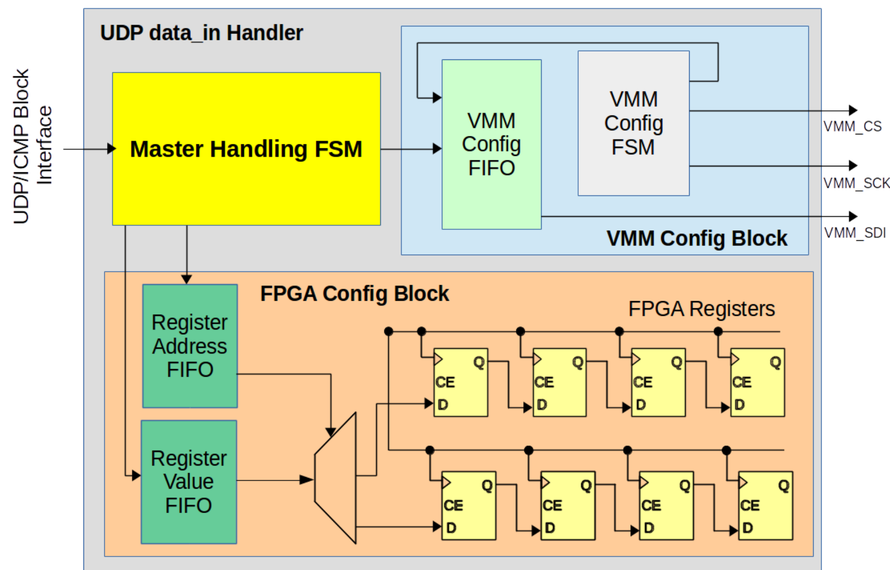
6.1.2 User Logic - Configuration

Σε αυτή την υποενότητα θα περιγραφούν τα components που έχουν κατασκευαστεί για τις ανάγκες του design, προκειμένου να διαμορφώνουν τη λειτουργία του FPGA και του VMM.

UDP Data_In Handler

Τα δεδομένα που προέρχονται από το UDP, προωθούνται στο component που είναι υπεύθυνο για την αναγνώριση και επεξεργασία των πακέτων αυτών. Το πρωτόκολλο επικοινωνίας μεταξύ του DAQ software και του firmware, υποδεικνύει ότι υπάρχουν δύο δυνατά configuration paths: το ένα αφορά τη διαμόρφωση λειτουργιών του FPGA, και το άλλο του VMM. Το αν ένα ληφθέν πακέτο προορίζεται για τη διαμόρφωση του VMM ή του FPGA, εξαρτάται από την τιμή *destination port* του πακέτου. Η κύρια FSM του component ελέγχει κατ'αρχάς αν το σήμα *valid* που προέρχεται από το UDP/ICMP block είναι στο λογικό ένα. Αν κάτι τέτοιο ισχύει, τότε σημαίνει ότι έχει ληφθεί ένα πακέτο UDP. Στη συνέχεια ελέγχει το *destination port* και ανάλογα την τιμή του, ενεργοποιεί την αντίστοιχη FSM που θα επεξεργαστεί το

περαιτέρω το πακέτο. Η ιεραρχία των FSM απεικονίζεται στο σχήμα 6.1.IV.

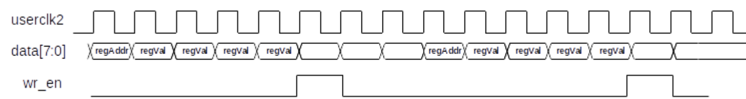


Σχήμα 6.1.IV: Αρχιτεκτονική του UDP Data_In Handler. Η κύρια FSM ελέγχει δύο άλλες που είναι υπεύθυνες είτε για τη διαμόρφωση των λειτουργιών του FPGA, είτε του VMM.

Το FPGA διαθέτει πολυάριθμους configuration registers οι οποίοι συνδέονται με τα περισσότερα κομμάτια του design, ούτως ώστε να μπορεί ο χρήστης μέσω του DAQ software να αλλάζει δυναμικά τον τρόπο με τον οποίο λειτουργεί το firmware. Το timing diagram ενός πακέτου που στέλνεται από το λογισμικό για να διαμορφώσει τις λειτουργίες του FPGA μπορεί να βρεθεί στο 6.1.V. Υπάρχουν δύο οικογένειες τιμών που βρίσκονται μέσα στο πακέτο, οι *register addresses* και *register values*. Το πρωτόκολλο επικοινωνίας μεταξύ του λογισμικού και του firmware υποδεικνύει τις θέσεις των τιμών αυτών μέσα στο πακέτο. Η FSM καταχωρεί αυτές τις τιμές καθώς το πακέτο διαβάζεται, και τις γράφει σε δύο FIFO, μία για κάθε τύπο τιμής (βλ. σχ. 6.1.IV). Μόλις εντοπιστεί το σήμα *last*, τότε οι μνήμες FIFO διαβάζονται προκειμένου να περάσουν τα δεδομένα στους FPGA registers. Ουσιαστικά, η τιμή της διεύθυνσης (register address), παίζει το ρόλο του σήματος *sel* ενός πολυπλέκτη, ο οποίος δρομολογεί τα δεδομένα του καταχωρητή (register value) στον αντίστοιχο FPGA register[10].

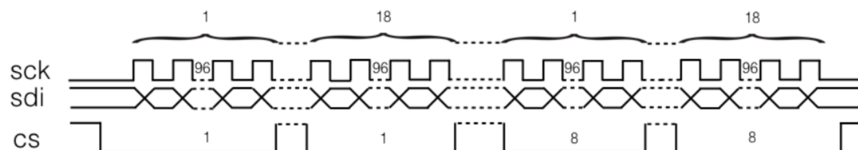
Στην περίπτωση που ένα πακέτο προερχόμενο από το DAQ software έχει σαν σκοπό

[10]Τα δεδομένα περνάνε ανά bit στον FPGA register (ο οποίος έχει τη μορφή ενός *shift register*)



Σχήμα 6.1.V: Timing Diagram ενός πακέτου που προορίζεται για τη διαμόρφωση των λειτουργιών του FPGA. Πρώτα στέλνεται η διεύθυνση (ένα byte) και μετά η τιμή του καταχωρητή που αντιστοιχεί σε αυτή τη διεύθυνση (τέσσερα byte). Η FSM καταχωρεί τις δύο τιμές, και την κατάλληλη στιγμή τις γράφει στις δύο FIFO (σήμα *wr_en*).

να διαμορφώσει τις λειτουργίες του VMM, τότε θα δρομολογηθεί κατ' αρχάς σε μία FIFO *(VMM Config FIFO)* η οποία θα αποθηκεύσει ολόκληρο το πακέτο. Μετά τον εντοπισμό του σήματος *last*, τότε η FSM του *VMM Config Block* θα διαβάσει τη FIFO, ανά bit, προκειμένου να κάνει *serialize* τα δεδομένα που πρέπει να περαστούν στο VMM. Το configuration του VMM3 τελείται σύμφωνα με το πρωτόκολλο SPI (βλ. σχ. 6.1.VI), και η FSM οδηγεί τα απαραίτητα σήματα *(SCK* και *CS)* στο VMM, ενώ τα configuration bits *(SDI)* έρχονται απευθείας από τη FIFO. Το DAQ software στέλνει αυτούσια τα 1728 bit που πρέπει να σταλούν στο VMM, επιτρέποντας έτσι στο χρήστη να έχει πλήρη έλεγχο πάνω στις λειτουργίες του τσιπ[11] [33]. Ο κώδικας του *VMM Config Block* μπορεί να βρεθεί στο αντίστοιχο Παράρτημα.



Σχήμα 6.1.VI: Η επικοινωνία του VMM3 με το τσιπ που το κάνει configure (Εν προκειμένω το FPGA, στο τελικό πείραμα, το SCA) τελείται μέσω του πρωτοκόλλου SPI. Το *CS* ενεργοποιείται και απενεργοποιείται μετά από κάθε 96 SCK [33]. Έτσι στέλνονται τμηματικά τα 1728 configuration bits.

CKBC/CKTP Generator

Προκειμένου να μελετηθεί η συμπεριφορά του VMM κάτω από διάφορες συνθήκες, είναι απαραίτητη η ύπαρξη ενός κυκλώματος μέσα στο ASIC που να εναποθέτει

[11] Μπορεί κανείς για παράδειγμα να επιλέξει το πόσο θα ενισχύει το σήμα εισόδου ο προενισχυτής του VMM ή να ενεργοποιήσει το κύκλωμα που στέλνει δοκιμαστικούς παλμούς σε συγκεκριμένα κανάλια.

δοκιμαστικούς παλμούς *(test pulsing)* στις εισόδους των προενισχυτών, οι οποίοι κανονικά δέχονται σήματα από τα strips του ανιχνευτή. Η ανάγκη για την ύπαρξη ενός τέτοιου μηχανισμού είναι διττή: Επιτρέπει κατ' αρχάς στον χρήστη να δοκιμάσει το τσιπ κανάλι προς κανάλι, ανεξάρτητα από το αν είναι προσαρτημένο το τσιπ απάνω σε κάποιο ανιχνευτή. Οι configuration registers του VMM δίνουν τη δυνατότητα ενεργοποίησης του κυκλώματος δοκιμαστικών παλμών σε συγκεκριμένα κανάλια του ASIC. Κάθε κανάλι λειτουργεί λίγο-πολύ ανεξάρτητα από τα άλλα (για μία γενική άποψη του κυκλώματος κάθε καναλιού βλ. σχ. 5.6.II), και ένας μηχανισμός που θα οδηγεί παλμούς στο κανάλι κατά τη βούληση του χρήστη, επιτρέπει τον πλήρη έλεγχο της ορθής λειτουργίας του καναλιού.

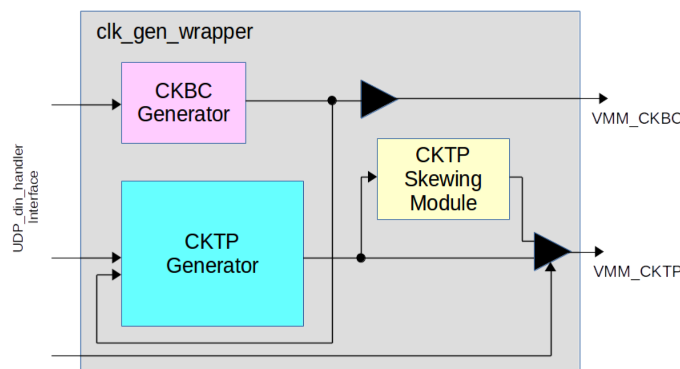
Ο άλλος λόγος για τον οποίο το test pulsing είναι αναγκαίο, είναι ότι κάθε κανάλι πρέπει να υποστεί *βαθμονόμηση (calibration)*. Ως γνωστόν, το VMM ψηφιοποιεί την πληροφορία του ύψους του παλμού και του χρόνου εμφάνισής του, με τη χρήση κάποιων ADCs, οι οποίοι όμως δεν θα έχουν όλοι την ίδια απόκριση ως προς τους παλμούς που θα ψηφιοποιούν. Ανατρέχοντας για παράδειγμα στην υποενότητα 5.6.1, μπορεί κανείς να δει πως το VMM μετράει επακριβώς το χρόνο εμφάνισης ενός παλμού. Έχοντας σα ρολόι αναφοράς το *CKBC*[12], το VMM τοποθετεί με μεγάλη ακρίβεια το χρόνο εμφάνισης της κορυφής ενός παλμού σε σχέση με το επόμενο falling-edge του CKBC. Ουσιαστικά αυτό που εισέρχεται στον ADC μέτρησης του χρόνου είναι ένας παλμός, το ύψος του οποίου είναι ανάλογο της απόστασης της κορυφής του παλμού στην είσοδο του VMM[13] από το επόμενο falling-edge του CKBC [33]. Η ψηφιακή πληροφορία του ύψους του παλμού, που αντιστοιχεί ουσιαστικά στο *χρόνο*, ονομάζεται *TDO* και έχει μήκος 8 bit. Ο άλλος μηχανισμός που πρέπει να βαθμονομηθεί αφορά την πληροφορία του *ύψους* του παλμού εισόδου, που είναι ανάλογος του φορτίου που εναποτέθηκε στην είσοδο του VMM, και σχετίζεται με την ενέργεια του σωματιδίου που εντοπίστηκε από το ανιχνευτικό σύστημα. Προφανώς, ένας άλλος ADC αναλαμβάνει τη ψηφιοποίηση του ύψους του αναλογικού παλμού εισόδου, και η 10-bit λέξη που θα παράξει, ονομάζεται *PDO*. Οι τιμές των *PDO* και *TDO* λοιπόν, πρέπει να αντιστοιχούν στην πραγματικότητα, καθώς μόνο έτσι το τσιπ θα παρέχει αξιόπιστες πληροφορίες που θα οδηγούν στην αναγνώριση σωματιδίων και ανακατασκευή τροχιών στο τελικό πείραμα. Η απόκριση κάθε καναλιού σε παλμούς

[12] Το οποίο παρέχεται από το FPGA, και έχει σχεδιαστεί να έχει $f = 40\,\mathrm{MHz}$ με duty cycle $25\,\%$

[13] Που μπορεί να είναι είτε test pulse, είτε το φορτίο που εναποτέθηκε σε ένα strip λόγω της ανίχνευσης ενός σωματιδίου.

εισόδου θα είναι διαφορετική, όμως με τη βαθμονόμηση του συστήματος, παράγοντες διόρθωσης μπορούν να μπαίνουν στις τιμές των *PDO* και *TDO* για κάθε κανάλι όταν γίνεται η off-line ανάλυση των δεδομένων από το ASIC.

Το κύκλωμα των δοκιμαστικών παλμών ενεργοποιείται για κάθε κανάλι ξεχωριστά από το DAQ software. Ο χρήστης μπορεί μέσω του software να στείλει ένα UDP πακέτο διαμόρφωσης λειτουργίας του VMM, και το firmware, μέσω του vmm_config_block, το οποίο έχει περιγραφεί παραπάνω, θα στείλει αυτά τα δεδομένα στο VMM μέσω του πρωτοκόλλου SPI. Το επόμενο στάδιο, είναι να διεγερθεί το κανάλι, αλλά προκειμένου συμβεί αυτό, πρέπει να σταλεί ο παλμός *CKTP (CK = Clock, TP = Test Pulse)* στο VMM. Αυτό γίνεται μέσω του *CKBC/CKTP Generator*[14], το οποίο στέλνει κατ' αρχάς το CKBC στο VMM, ενώ αν ο χρήστης θέλει να τεστάρει την απόκριση του VMM, μπορεί να στείλει και το CKTP. Και τα δύο σήματα παράγονται από registers με τη βοήθεια μετρητών, οι οποίοι ελέγχουν τη συχνότητα, το duty-cycle, ή ακόμα και την ποσότητα των CKBC/CKTP που στέλνονται στο VMM. Έτσι, ο χρήστης, μέσω του DAQ software μπορεί να αλλάξει τη συχνότητα του CKBC $(40,\ 20$ ή 10 MHz), την περίοδο του CKTP (από μερικές εκατοντάδες ns μέχρι μερικά ms), και το χρονικό διάστημα που το CKTP θα έχει την τιμή του λογικού ένα. Όλες αυτές οι επιλογές ανήκουν στην γενικότερη κατηγορία του FPGA configuration.



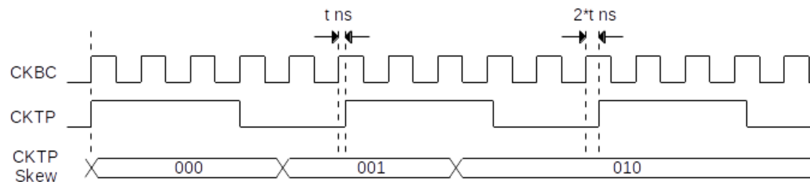
Σχήμα 6.1.VII: Αρχιτεκτονική του CKBC/CKTP Generator. Ο χρήστης ελέγχει τα χαρακτηριστικά των δύο παλμών, όπως και τη μεταξύ τους διαφορά φάσης.

Με την άφιξη του μετώπου του CKTP, και υπό την προϋπόθεση ότι το κύκλωμα test pulsing έχει ενεργοποιηθεί από το χρήστη, θα εναποτεθεί ένας δοκιμαστικός παλ-

[14] Στο firmware ονομάζεται *clk_gen_wrapper* και πρόκειται ουσιαστικά για ένα custom MMCM.

μός στο κανάλι, το ύψος του οποίου ελέγχει ο χρήστης μέσω της διαμόρφωσης της λειτουργίας του VMM. Μέσω ενός DAC[15] μέσα στο VMM, μετατρέπεται η ψηφιακή πληροφορία του ύψους του παλμού που επιθυμεί ο χρήστης να στείλει στο κανάλι σε πραγματικό αναλογικό ύψος. Προφανώς, όσο μεγαλύτερη τιμή ύψους παλμού στείλει ο χρήστης στον DAC[16], άρα και στο κανάλι, τόσο μεγαλύτερη θα είναι και η τιμή του PDO αυτού του παλμού. Γνωρίζοντας a priori το ύψος του παλμού σε mV όμως, ο χρήστης μπορεί να μελετήσει την απόκριση του καναλιού, και να αντιστοιχήσει τις ψηφιακές τιμές του PDO σε πραγματικό ύψος παλμού. Έτσι βαθμονομείται το PDO.

Η βαθμονόμηση του TDO απαιτεί λίγη περισσότερη προσπάθεια από το χρήστη, καθώς το VMM δεν διαθέτει κάποιον εγγενή μηχανισμό που να ελέγχει το χρόνο άφιξης του CKTP σε σχέση με το CKBC. Για αυτό το λόγο, σχεδιάστηκε ένα ειδικό κύκλωμα στο FPGA, που να αλλάζει δυναμικά το χρονικό προφίλ του CKTP σε σχέση με το CKBC. Το module αυτό, ονομάζεται *CKTP Skewing Module*, και ελέγχει τη διαφορά φάσης μεταξύ των δύο παλμών. Έτσι λοιπόν, η τρίτη παράμετρος του CKTP που μπορεί να ελέγξει ο χρήστης δυναμικά, είναι το *CKTP Skew*, και το διάγραμμα των παλμών σε σχέση με την τιμή αυτής της παραμέτρου απεικονίζεται στο σχήμα 6.1.VIII.



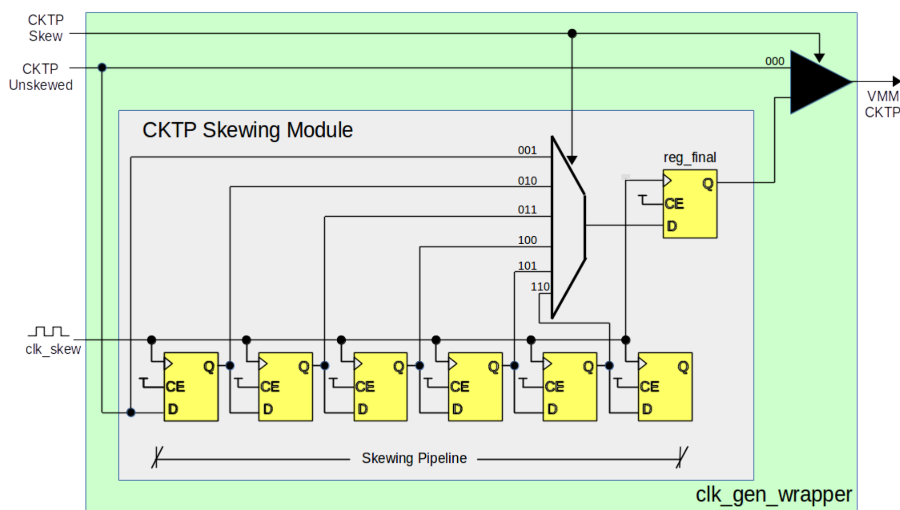
Σχήμα 6.1.VIII: Ο χρήστης έχει τη δυνατότητα να αλλάξει τη διαφορά φάσης μεταξύ των δύο παλμών. Αν η τιμή του skew είναι μηδέν, τότε τα rising edges τους συμπίπτουν. Αν η τιμή είναι ένα, τότε το CKTP θα καθυστερήσει κατά t ns. Αν είναι δύο, τότε η καθυστέρηση του CKTP θα είναι $2t$ ns, όπου t το βήμα του χρόνου της διαφοράς φάσης *(skew step)*.

Η αρχιτεκτονική του component που επιτρέπει τη δυναμική αλλαγή της διαφοράς φάσης μεταξύ των δύο σημάτων, απεικονίζεται στο σχήμα 6.1.IX, ενώ η λογική του

[15] Digital-to-Analog converter

[16] Ο ίδιος DAC, που είναι κοινός για όλο το VMM, πρέπει να υποστεί και αυτός με τη σειρά του βαθμονόμηση. Αυτό γίνεται με τη βοήθεια του xADC, το οποίο μελετάται σύντομα παρακάτω.

έχει ως εξής: Κατ' αρχάς, το module που παράγει το CKTP, θα το παράξει ευθυγραμμισμένο με το CKBC. Το ευθυγραμμισμένο CKTP, που ονομάζεται *CKTP Unskewed* στο σχήμα 6.1.IX, θα είναι η μία είσοδος ενός *BUGMUX*[17] η έξοδος του οποίου θα οδηγείται στην είσοδο του VMM για το CKTP. Όταν ο χρήστης θέλει ευθυγραμμισμένα τα δύο σήματα, η τιμή της παραμέτρου του skewing θα είναι μηδέν και θα επιλέγεται αυτό το σήμα. Όταν όμως αυτή η τιμή είναι διάφορη του μηδενός, τότε επιλέγεται η έξοδος του *CKTP Skewing Module*. Αυτό που κάνει το συγκεκριμένο component, είναι να προσθέτει διαδοχικές καθυστερήσεις στο σήμα του CKTP, αλλάζοντας έτσι τη φάση του σε σχέση με το CKBC. Αυτό το επιτυγχάνει περνώντας το CKTP από ένα *Skewing Pipeline*, δηλαδή από έναν *Shift Register*. Μετά από κάθε register, το CKTP θα καθυστερεί *τόσα ns όσο η περίοδος του ρολογιού που χρονίζει τους καταχωρητές*, το οποίο ονομάζεται *clk_skew*. Κάθε διαδοχική διασύνδεση μεταξύ των καταχωρητών οδηγείται στην είσοδο ενός πολυπλέκτη, ο οποίος ανάλογα την τιμή της παραμέτρου του skewing, θα οδηγήσει και διαφορετικό στάδιο καθυστέρησης στη δεύτερη είσοδο του BUFMUX, άρα και στο VMM.



Σχήμα 6.1.IX: Αρχιτεκτονική του CKBC/CKTP Generator. Ο χρήστης ελέγχει τα χαρακτηριστικά των δύο παλμών, όπως και τη μεταξύ τους διαφορά φάσης. Η περίοδος του ρολογιού *clk_skew* θα επηρεάζει άμεσα το βήμα του skewing (συμβολίζεται με t στο σχήμα 6.1.VIII).

Υπάρχει βέβαια ένα λεπτό σημείο στην όλη λογική του component το οποίο αξίζει να

[17]Η γενική περίπτωση του BUFG (βλ. Κεφάλαιο 1), με δύο εισόδους και ένα σήμα επιλογής που επιτρέπει στο χρήστη να επιλέξει ποιο σήμα να οδηγήσει στην έξοδο του buffer.

αναφερθεί, και έχει σχέση με την ύπαρξη του καταχωρητή *reg_final*. Ο πολυπλέκτης που επιλέγει το στάδιο καθυστέρησης του CKTP θα υλοποιηθεί από LUT και πολυπλέκτες 2-προς-1, δηλαδή από τα components που είναι διαθέσιμα στο FPGA fabric για μία τέτοια δουλειά. Όμως αυτά τα λογικά κελιά θα προσθέτουν *καθυστερήσεις* στα σήματα. Με καθυστέρηση επίσης θα επιβαρύνουν και οι διασυνδέσεις μεταξύ των εισόδων/εξόδων των καταχωρητών και των εισόδων των LUT/2to1Muxes. Επίσης, με κάθε διαφορετικό implementation του κώδικα, το Vivado® θα αλλάζει την αρχιτεκτονική του μηχανισμού, με αποτέλεσμα να υπάρχουν διαφοροποιήσεις ως προς τις καθυστερήσεις μεταξύ διαφορετικών εκδόσεων του firmware. Οι καθυστερήσεις αυτές καθ' αυτές, αλλά και η μεταβλητότητα της διάρθρωσης του κυκλώματος, θα προκαλέσει και αναξιόπιστο skewing. Αν για παράδειγμα ο χρήστης επιλέξει τρία επίπεδα καθυστέρησης, τότε δεν θα πάρει $3t$ ns skewing, αλλά $3t + x_i$ ns, όπου x_i η συνολική καθυστέρηση λόγω της ύπαρξης του πολυπλέκτη (που είναι ουσιαστικά πολλά αλληλοσυνδεδεμένα LUT και 2to1 MUX μεταξύ τους). Αυτό το x_i, θα διαφέρει μεταξύ των επιπέδων του skewing.

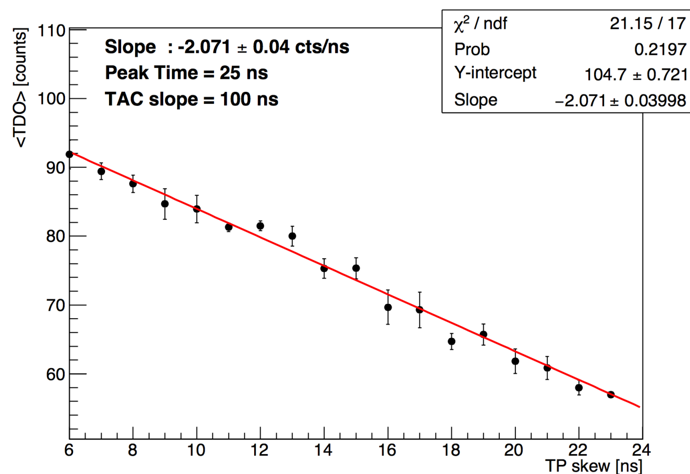
Προκειμένου λοιπόν να γίνει περισσότερο αυστηρό το timing του κυκλώματος, προστίθεται ένας ακόμα καταχωρητής, που στην εικόνα 6.1.IX καταγράφεται ως *reg_final*. Αυτός ο καταχωρητής θα δέχεται την έξοδο του πολυπλέκτη, και θα την καταχωρεί μία ακόμα φορά, πριν την οδηγήσει απ' ευθείας στο BUFGMUX. Ο χρονισμός έτσι είναι αυστηρά καθορισμένος, και δίνοντας την κατάλληλη εντολή στο Vivado® να τοποθετήσει τον καταχωρητή αυτόν χωροταξικά δίπλα από τον BUFGMUX[18], μπορεί κανείς να αυξήσει τη συνοχή του design μεταξύ διαφορετικών implementations.

Στο firmware, χρησιμοποιείται ρολόι *clk_skew* με συχνότητα $f = 500\,\text{MHz}$, δηλαδή με περίοδο $T = 2\,\text{ns}$. Λαμβάνοντας πάντα υπόψιν τη περίοδο του CKBC, που είναι συνήθως $T_{CKBC} = 25\,\text{ns}$, ο αναγνώστης μπορεί να υποθέσει ότι *δώδεκα* επίπεδα καθυστέρησης αρκούν για να καλυφθεί μία περίοδος CKBC και να βαθμονομηθεί η απόκριση των ADCs ως προς το χρόνο. Θα υλοποιηθούν έτσι δώδεκα καταχωρητές στο *Skewing Pipeline* και θα έχει κανείς $t = 2\,\text{ns}$ στο σχήμα 6.1.VIII.

Ένας δέκατος τρίτος καταχωρητής θα είχε νόημα, από τη στιγμή που η έξοδός του θα παρήγαγε CKTP η καθυστέρηση του οποίου θα ξεπερνούσε την περίοδο του

[18] Αυτό επιτυγχάνεται με τα λεγόμενα constraints. Τη σύνταξη των αρχείων .xdc που περιέχουν τα constraints, ο αναγνώστης μπορεί να τη βρει στο [15].

CKBC; Η απάντηση είναι πως θα *είχε νόημα*, καθώς έτσι, εν δυνάμει μπορεί κανείς να συρρικνώσει το βήμα t σε 1 ns. Ο έξτρα καταχωρητής θα δημιουργήσει καθυστέρηση 26 ns, δηλαδή ουσιαστικά καθυστέρηση ενός ns. Τοποθετώντας λοιπόν 24 καταχωρητές, λαμβάνει κανείς ένα component που δύναται να δημιουργεί διαφορά φάσης μεταξύ των rising edges των CKBC και CKTP με βήμα *ενός* ns. Αυτό είναι που χρησιμοποιείται και στο προκείμενο firmware, επιτρέποντας έτσι την πολύ ακριβή μέτρηση της σχέσης μεταξύ *TDO* και *CKTP Skew* (βλ. σχ. 6.1.X). Με αυτόν τον τρόπο βελτιστοποιείται η βαθμονόμηση της χρονικής απόκρισης των καναλιών, κάνοντας έτσι την ανακατασκευή των τροχιών από τον MicroMegas μία υπόθεση εξαιρετικής ακρίβειας.



Σχήμα 6.1.X: Ευθεία που προκύπτει από το calibration του TDO ενός καναλιού. Η ύπαρξη πολλών σημείων λόγω του design που επιτρέπει CKTP skewing με βήμα ενός ns κάνει τη μέτρηση ακριβέστερη και πιο αξιόπιστη.

6.2 Readout Path

Μέχρι τώρα έχει μελετηθεί το πως μπορεί κανείς να διαμορφώσει τη λειτουργικότητα του VMM και του FPGA μέσω του firmware. Πως όμως μπορεί κανείς να αποσπάσει τα δεδομένα από τα κανάλια του VMM με τη βοήθεια του firmware; Η περιγραφή του *Readout Path* θα απαντήσει σε αυτή την ερώτηση.

Σε γενικές γραμμές, αυτό που οφείλει να κάνει το firmware, είναι να αποσπάσει τα

ψηφιακά δεδομένα του VMM, να τα *πακετάρει* σε μία προσυμφωνημένη μορφή, και τελικά να τα προωθήσει στο DAQ software. Το τελευταίο θα αναλύσει αυτά τα δεδομένα, κατασκευάζοντας ιστογράμματα που περιέχουν όλες τις απαραίτητες πληροφορίες (όπως π.χ. το φορτίο (PDO), ή το χρόνο (TDO) ενός γεγονότος) που χρειάζονται για να αναλυθούν τα δεδομένα αργότερα, προκειμένου να χαρακτηριστεί η αποδοτικότητα του VMM και το ανιχνευτικού συστήματος. Το λεγόμενο *Data Format*, είναι διαφορετικό ανάλογα με τον τρόπο του Readout. Όπως έχει ήδη αναφερθεί στο προηγούμενο Κεφάλαιο, το VMM έχει δύο readout mode: *Continuous* και *Level-0*. Το firmware θα αποσπάσει τα δεδομένα από το VMM με τον εκάστοτε τρόπο, και θα στείλει στο UDP_TX (υποσύστημα του Ethernet Interface), μέσω του FIFO2UDP, το πακέτο που περιμένει το DAQ software. Το UDP_TX στη συνέχεια θα προωθήσει αυτό το πακέτο στη γραμμή του Ethernet, και το DAQ software θα αναλάβει να το αποκωδικοποιήσει, παράγοντας τελικά ένα αρχείο .ROOT με όλες τις απαραίτητες πληροφορίες. Κάθε πακέτο DAQ, αποτελείται από έναν *Header*, από τα *VMM Data*, και από τον *Trailer*. Με την άφιξη ενός *Trigger*, το firmware θα ξεκινήσει τις διαδικασίες απόσπασης δεδομένων από το VMM, στέλνοντας τελικά το αποτέλεσμα στο software. Η όλη λογική του διαβάσματος έχει υποστεί αλλεπάλληλες βελτιστοποιήσεις, με αποτέλεσμα να επιτρέπει στο firmware να λαμβάνει δεδομένα σε συνθήκες *Testbeam* με ρυθμούς trigger εκατοντάδων kHz.

Ανατρέχοντας στην εικόνα 6.0.II, μπορεί κανείς να διακρίνει τρία αλληλένδετα components, το *Trigger Module*, το *Packet Formation*, και το *FIFO2UDP*. Οι λειτουργίες που τελούν τα υποσυστήματα αυτά είναι οι εξής:

- *Trigger Module:* Το component αυτό επικοινωνεί με το *Packet Formation* μέσω του σήματος *newCycle*. Όταν αυτό το σήμα πάρει το λογικό ένα, τότε η FSM του Packet Formation ξεκινάει τη διαδικασία διαβάσματος του VMM. Το προαναφερθέν σήμα, έχει άμεση σχέση με τον τύπο του *Trigger* που περιμένει το design, το οποίο ρυθμίζεται μέσω του DAQ software. Στην περίπτωση του *Internal Trigger*, το σήμα του trigger είναι συζευγμένο με το CKTP. Όταν στέλνεται το CKTP στο VMM, το Trigger Module θα οδηγήσει το σήμα newCycle στο Packet Formation. Στην περίπτωση του *External Trigger*, το σήμα του Trigger προέρχεται από κάποιο εξωτερικό σύστημα εξειδικευμένο στο να παρέχει τέτοια σήματα[19]. Το εξωτερικό αυτό σήμα συγχρονίζεται με το κυρίως *Clock Domain*

[19]Συνήθως πρόκειται για *Σπινθηριστές (Scintillators)* που τοποθετούνται ευθυγραμμισμένοι με τον

του firmware, και ενεργοποιεί τελικά το newCycle, που οδηγείται στο Packet Formation.

- *Packet Formation:* Το component αυτό λειτουργεί σαν μεσάζοντας μεταξύ των υποσυστημάτων που επικοινωνούν άμεσα με το VMM, προκειμένου να του αποσπάσουν τα ψηφιακά δεδομένα *(VMM Readout Module)*, και του *FIFO2UDP*. Όταν η FSM του Packet Formation δεχθεί το σήμα newCycle από το Trigger Module, περιμένει κατ' αρχάς για ένα χρονικό διάστημα λίγων εκατοντάδων ns πριν ενεργοποιήσει το αντίστοιχο Readout Module. Αυτό το timeout είναι απαραίτητο καθώς το VMM χρειάζεται χρόνο για να ψηφιοποιήσει τους παλμούς, και να περάσει αυτές τις ψηφιακές λέξεις στις εσωτερικές του μνήμες απ' όπου και θα αποσπαστούν από το FPGA. Μετά από αυτό το χρονικό διάστημα, θα ενεργοποιήσει, και ύστερα θα περιμένει το Readout Module να ολοκληρώσει τη διαδικασία του διαβάσματος. Τέλος, το Packet Formation θα γράψει στη FIFO του FIFO2UDP τον Header που περιμένει το DAQ software, και ύστερα θα οδηγήσει τα ψηφιακά δεδομένα από τον προσωρινό buffer του Readout Module στο FIFO2UDP.

- *FIFO2UDP:* Πρόκειται για μία FSM που συνοδεύει τη FIFO που λειτουργεί σαν γέφυρα ανάμεσα από τη διεπαφή του Ethernet και από το υπόλοιπο design. Όταν το Packet Formation στείλει το τελευταίο πακέτο δεδομένων ενός trigger (ή γεγονότος) στο FIFO2UDP, τότε αυτό θα αδειάσει τη FIFO ανά byte, προσθέτοντας στο τέλος τον 32-bit *Trailer* που περιμένει το DAQ software. Μόλις η FIFO αυτή αδειάσει, τότε η διαδικασία του διαβάσματος θα έχει τελειώσει, και το firmware θα μπορεί να δεχθεί καινούργιο trigger.

6.2.1 VMM Readout

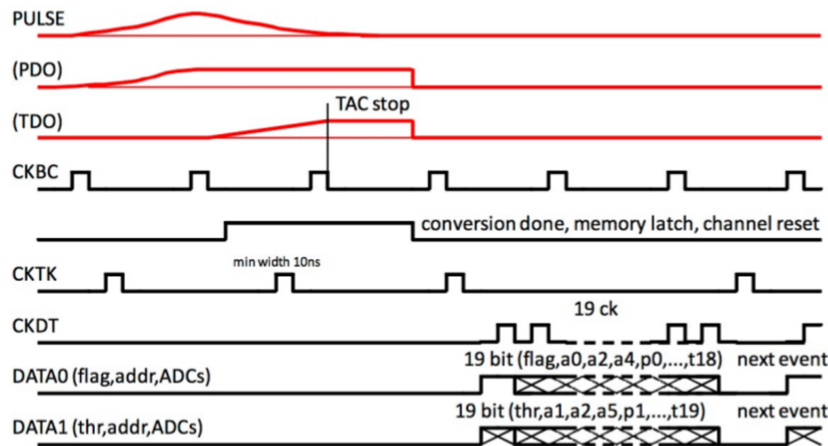
Η διαδικασία που ακολουθείται για να διαβαστούν τα ψηφιακά δεδομένα από το VMM, εξαρτάται από το πως ο χρήστης έχει διαμορφώσει το VMM προηγουμένως. Πιο συγκεκριμένα, μέσω DAQ software, ο χρήστης μπορεί να ενεργοποιήσει τους *Level-0 Registers*, δίνοντας έτσι εντολή στο VMM να υλοποιήσει τη διεπαφή του Level-

ανιχνευτή που βαίνει η πλακέτα του FPGA/VMM, και ρυθμίζονται έτσι ώστε όταν τους διαπερνάει όλους ένα σωματίδιο, να δίνουν το σήμα του Trigger. Οι ανιχνευτές αυτοί είναι ιδανικοί για εφαρμογές triggering, λόγω της άμεσης απόκρισής τους όταν διεγείρονται από ιοντίζουσες ακτινοβολίες.

0 με το FPGA. Αν αυτοί οι καταχωρητές είναι απενεργοποιημένοι, τότε το VMM θα στέλνει τα ψηφιακά δεδομάν στο FPGA με το *Continuous Mode*. Το firmware, διαθέτει δύο ανεξάρτητα components, όπου το κάθε ένα έχει σχεδιαστεί ώστε να εξυπηρετεί το διάβασμα του VMM, ανάλογα με το πως αυτό έχει διαμορφωθεί από το χρήστη. Το ενεργοποιημένο component επικοινωνεί με το Packet Formation μέσω εξειδικευμένων FSM, οι οποίες υποδεικνύουν πότε η διαδικασία του διαβάσματος έχει τελειώσει, προκειμένου οι πληροφορίες να προωθηθούν τελικά στο DAQ software, όπως περιγράφηκε παραπάνω.

Continuous Readout

Σε αυτό τον τρόπο διαβάσματος, κάθε κανάλι του VMM αποθηκεύει τα δεδομένα που προέκυψαν από την ψηφιοποίηση των παλμών σε μία FIFO που διαθέτει τέσσερις θέσεις. Αν ανατρέξει κανείς στο προηγούμενο Κεφάλαιο, θα δει ότι κάθε *channel event* αποτελείται από 38 bit, τα οποία περιέχουν τις πληροφορίες για το ύψος (PDO), το χρόνο (TDO), την τιμή του BCID counter, και φυσικά τη διεύθυνση του καναλιού. Για να διαβάσει το FPGA κάποιο event, πρέπει να οδηγήσει στο VMM το λεγόμενο *Token*, μέσω του *CKTK* pin. Αν το VMM έχει event σε κάποιο κανάλι, τότε στη γραμμή *DATA0*, θα εμφανιστεί το *Flag* (το DATA0 θα έχει λογική τιμή ένα), το FPGA τότε θα στείλει 19 παλμούς *CKDT*, και θα περάσει σε έναν *Shift Register* τα δεδομένα από τις δύο γραμμές των δεδομένων που προέρχονται από το VMM, ονόματι *DATA0* και *DATA1*. Με τους 19 παλμούς CKDT, αποθηκεύονται $2 \times 19 = 38$ bit, όσο δηλαδή ένα γεγονός. Η λέξη αυτή θα αποθηκευτεί σε έναν προσωρινό buffer μέσα στο FPGA, το οποίο στη συνέχεια θα στείλει πάλι το Token στο VMM, ζητώντας του νέα δεδομένα. Η λογική του Token υποδεικνύει ότι μόλις ληφθεί το νέο Token από το FPGA, θα οδηγηθεί στις γραμμές DATA0/1 το event από το επόμενο κανάλι του οποίου η FIFO δεν είναι άδεια. Με αυτόν τον τρόπο, το FPGA διαβάζει ”κυκλικά” και βηματικά όλα τα κανάλια του ASIC. Όταν δεν υπάρχουν πλέον γεγονότα στις μνήμες του ASIC, τότε το FPGA δεν θα λάβει ανταπόκριση όταν στείλει πάλι CKTK, δηλαδή δεν θα δει τη γραμμή DATA0 να έχει τιμή λογικού ένα αφού στείλει CKTK. Αυτό σημαίνει ότι όλες οι πληροφορίες που είναι συνυφασμένες με το trigger που δέχτηκε εξαρχής το FPGA αποσπάστηκαν από το VMM, και το Continuous Readout Module θα ενημερώσει το Packet Formation ότι τελείωσε το διάβασμα. Η συνέχεια είναι ήδη γνωστή από τις παραπάνω περιγραφές.



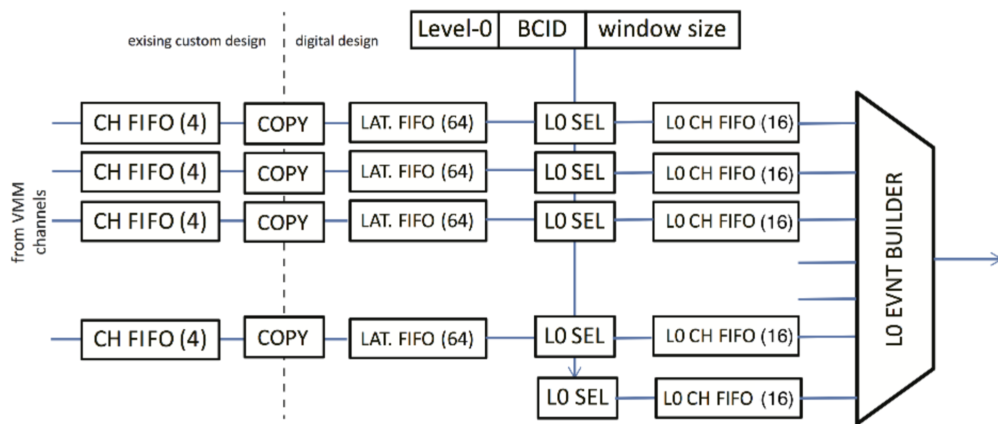
Σχήμα 6.2.I: Timing Diagram στο Continuous Readout Mode. Μετά από κάποιο CKTK, εμφανίζεται στο DATA0 το flag, και το FPGA στέλνει 19 CKDT στο VMM για να αποσπάσει $2 \times 19 = 38$ bits από τις δύο γραμμές δεδομένων. Το επόμενο CKTK θα ξαναδώσει flag, και η διαδικασία συνεχίζεται μέχρι να μην εμφανιστεί flag στο DATA0 μετά από κάποιο CKTK [33].

Level-0 Readout

Το Level-0 Readout είναι αυτό που θα χρησιμοποιηθεί και στο τελικό πείραμα του ATLAS, καθώς προσφέρει στο χρήστη τη δυνατότητα επιλογής του ποιου συγκεκριμένου γεγονότος θα διαβαστεί από το ASIC [33]. Αυτό έχει μεγάλη σημασία για την ακεραιότητα των δεδομένων, καθώς οι ανιχνευτές MicroMegas, εκτός από μιόνια, θα ανιχνεύουν και άλλα σωματίδια που ουδεμία σχέση έχουν με τα μιόνια που προέρχονται από το σημείο αλληλεπίδρασης στο κέντρο του ανιχνευτή, όπως έχει ήδη αναφερθεί στο προηγούμενο Κεφάλαιο. Η λογική του Triggering όμως, θα μπορεί να εντοπίζει τα μιόνια που έχουν ενδιαφέρον για την τελική ανάλυση των δεδομένων, και έτσι μέσω της λογικής του Level-0, μόνο οι ψηφιακές πληροφορίες που είναι συνυφασμένες με τα έγκυρα γεγονότα θα οδηγούνται έξω από το VMM. Τα υπόλοιπα δεν θα διαβάζονται ποτέ, και τελικά θα απορρίπτονται.

Όταν ένας παλμός εισόδου ξεπεράσει το κατώφλι του discriminator του VMM, τότε θα λάβει ένα *BCID Timestamp*, σύμφωνα με την τιμή του *Gray Code* counter που αυξάνεται με κάθε rising edge του CKBC (βλ. σχ. 5.6.II). Αυτό το γεγονός θα προωθηθεί στη συνέχεια σε μία FIFO βάθους τεσσάρων γεγονότων. Μέχρι αυτό το σημείο η λογική είναι ίδια με του Continuous Readout, όμως στο Level-0, αυτό το γεγονός θα

προωθηθεί σε μία δεύτερη μνήμη, όπου θα περιμένει μέχρι να επιλεγεί για να προωθηθεί εκτός του VMM. Προκειμένου να επιλεγεί ένα γεγονός, πρέπει να έρθει στο VMM το σήμα ονόματι *Level-0* που οδηγείται μέσω του CKTK pin. Όταν το VMM λάβει το Level-0, θα του δώσει και αυτού ένα BCID Timestamp, όπως πριν με το γεγονός του καναλιού. Μόλις συμβεί αυτό, το VMM θα "ψάξει" για ένα γεγονός με αυτό το BCID, μείον μία προκαθορισμένη τιμή που διαμορφώνεται από το χρήστη, και μείον το παράθυρο που έχει οριστεί επίσης από το χρήστη. Το Level-0 επομένως, "ψάχνει" για ένα συγκεκριμένο γεγονός που έχει καταγραφεί στο παρελθόν από το VMM. Θέτοντας σωστά την τιμή του παραθύρου και της αφαιρούμενης ποσότητας από το BCID του Level-0 για κάθε VMM στο NSW, θα επιλέγονται μόνο τα σωστά γεγονότα από το τσιπ. Αν το Level-0 που εστάλη στο VMM βρει κάποιο γεγονός, τότε στο FPGA προωθείται κατ' αρχάς ένας *header* που περιέχει το BCID, και διαδοχικά τα γεγονότα από όλα τα κανάλια που έχουν αυτό το BCID γύρω από το παράθυρο.

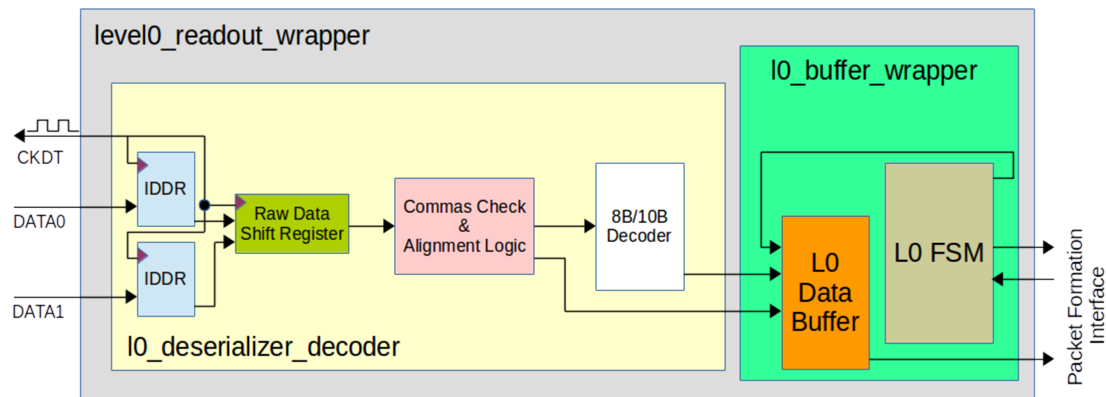


	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
header	V	P	orb		BCID (12)												1st word after comma															
hit data	1	P	*R*	T	Chan# (6)						ADC (10)										TDC (8)								N	rel BCID		

Σχήμα 6.2.II: Πάνω: Αρχιτεκτονική της λογικής του Level-0 μέσα στο VMM. Κάτω: Data Format των δεδομένων του Level-0 [33].

Η επικοινωνία του Level-0 υλοποιείται μέσω του πρωτοκόλλου *8b/10b*. Όταν το VMM δεν στέλνει δεδομένα στο FPGA (ή στο ROC, για το τελικό πείραμα), τότε στέλνει τους λεγόμενους *Comma Characters*, που είναι προκαθορισμένες 10-bit λέξεις που χρησιμοποιούνται για να φέρουν σε φάση τον πομπό και τον δέκτη (βλ. αντίστοιχο

Παράρτημα). Η λογική του FPGA έχει ως εξής: Προωθείται κατ' αρχάς συνεχόμενα στο VMM το ρολόι *CKDT* (σε αντίθεση με το Continuous Mode, όπου το FPGA έστελνε CKDT μόνο όταν ήξερε ότι υπήρχε γεγονός), και το FPGA προβαίνει σε deserialization των δεδομένων από τις γραμμές DATA0/1 μέσω δύο IDDR modules (βλ. [17] για περισσότερες πληροφορίες σε σχέση με το IDDR). Τα δεδομένα αυτά, όταν το VMM βρίσκεται σε αδρανή κατάσταση, είναι ένα προκαθορισμένο μοτίβο, που περιέχει τους comma characters. Ένας shift register δέχεται αυτά τα δεδομένα, και μία λογική ελέγχει συνεχώς τα περιεχόμενα του register αυτού. Όταν αναγνωρίζονται οι comma characters μέσα στον shift register, τότε ο μηχανισμός "κλειδώνει" και το component βρίσκεται πλέον σε φάση με τα δεδομένα του VMM. Όταν το VMM στείλει πραγματικά δεδομένα, τότε η λογική της ευθυγράμμισης θα γνωρίζει ακριβώς πότε ο shift register θα περιέχει μία έγκυρη 10-bit λέξη, προωθώντας τη στη συνέχεια στον αποκωδικοποιητή του 8b/10b. Αυτό το module δέχεται 10-bit λέξεις και σύμφωνα με το πρωτόκολλο 8b/10b τις μετατρέπει σε byte. Αυτά τα byte γράφονται σε έναν προσωρινό buffer. Όταν λήξει η μετάδοση του πακέτου (η μορφή του οποίου απεικονίζεται στο σχήμα 6.2.II), τότε το VMM ξεκινάει πάλι να στέλνει comma characters. Τότε η λογική του FPGA αντιλαμβάνεται ότι το διάβασμα του γεγονότος τελείωσε, και ενημερώνει το Packet Formation αναλόγως. Στη συνέχεια οι πληροφορίες από το VMM προωθούνται στο DAQ software, σύμφωνα με τη διαδικασία που έχει ήδη περιγραφεί προηγουμένως.



Σχήμα 6.2.III: Αρχιτεκτονική του Level-0 Readout Module του FPGA. Διακρίνονται όλα τα υποσυστήματα που έχουν περιγραφεί παραπάνω.

6.3 Λοιπά Υποσυστήματα

Εκτός από τις δύο μεγάλες ομάδες subcomponents που βρίσκονται μέσα στο FPGA, και τελούν την επικοινωνία με το VMM και με το DAQ software, υπάρχουν και τρία ακόμα σημαντικά components που τελούν λίγο διαφορετικές λειτουργίες.

- *Flow FSM:* Η κεντρική FSM του design, επιβλέπει την κατάσταση των περισσότερων υποσυστημάτων και χειρίζεται κάποια σήματα ελέγχου που αφορούν τα περισσότερα κομμάτια του firmware.

- *xADC Module:* Πρόκειται για ένα υποσύστημα που υλοποιεί έναν εσωτερικό ADC του FPGA (ονόματι xADC). Αυτός ο ADC συνδέεται με διάφορες εξόδους του VMM και προσφέρει εναλλακτικούς τρόπους βαθμονόμησης του ASIC. Ο DAC για παράδειγμα, βαθμονομείται μέσω του xADC. Όταν ο χρήστης επιλέγει ένα συγκεκριμένο ύψος για τον test pulse, αυτό το ύψος θα αντιστοιχεί σε έναν παλμό πλάτους κάποιων mV. Το VMM παρέχει τη δυνατότητα προώθησης του αναλογικού παλμού σε μία έξοδό του, και ο xADC μπορεί να ψηφιοποίησει την πληροφορία του πλάτους του, οδηγώντας έτσι στην αντιστοίχηση της τιμής που επιλέγει ο χρήστης σε πραγματικά mV. Τα δεδομένα του xADC προωθούνται στο DAQ software μέσω του FIFO2UDP.

- *Flash Memory SPI Controller:* Το FPGA, όντας συνδεδεμένο σε ένα δίκτυο Ethernet, πρέπει να διαθέτει μία διεύθυνση *MAC* και μία διεύθυνση *IP*. Το design έχει δύο *default* τιμές διευθύνσεων, οι οποίες όμως μπορούν να αλλάξουν κατά τη βούληση του χρήστη, αν θέλει να χειρίζεται πολλές πλακέτες μέσα στο ίδιο δίκτυο. Το προκείμενο component, έχει τη δυνατότητα να επικοινωνεί με τη μνήμη flash που βρίσκεται πάνω στην πλακέτα, και να γράφει και να διαβάζει συγκεκριμένες περιοχές σε/από αυτήν. Με αυτό τον τρόπο, η νέα διεύθυνση IP/MAC που ορίζει ο χρήστης, μπορεί να γράφεται στη μνήμη, και να αποθηκεύεται εκεί ακόμα και αν χαθεί η τροφοδοσία της πλακέτας. Κατά την εκκίνηση του firmware, το component θα διαβάσει τις διευθύνσεις που έχουν γραφεί στη flash, προκειμένου να τις περάσει στη λογική του firmware που είναι υπεύθυνη για την υλοποίηση της διεπαφής του Ethernet.

- *E-Link Interface:* Τα E-link, παρέχουν τη δυνατότητα επικοινωνίας της πλακέτας με το δίκτυο *FELIX*. Ο κώδικας υλοποίησης του πρωτοκόλλου επικοι-

νωνίας παρέχεται από την ομάδα ανάπτυξης firmware του FELIX, όχι όμως και η διεπαφή του component αυτού με το υπόλοιπο design. Ένα component τεστάρει την ποιότητα της επικοινωνίας στέλνοντας προκαθορισμένα μοτίβα στο FELIX *(elink_daq_tester)*, και ένα άλλο λαμβάνει τα δεδομένα DAQ που γράφονται στη μνήμη του FIFO2UDP, και τα μορφοποιεί καταλλήλως για να τα στείλει στο FELIX *(elink_daq_driver)*. Καλό είναι επίσης να αναφερθεί ότι το ROC ASIC θα πακετάρει με συγκεκριμένο τρόπο τα δεδομένα όπως θα τα λαμβάνει από τα VMM, πριν αποστείλει τα πακέτα αυτά στην L1DDC. Το firmware της MMFE8, έχει τη δυνατότητα μίμησης της μορφολογίας αυτής, υποκαθιστώντας έτσι πλήρως το ROC ASIC για τις ανάγκες ελέγχου της ποιότητας επικοινωνίας της MMFE8 με τα υπόλοιπα κομμάτια της αλυσίδας ηλεκτρονικών του NSW.

Σύνοψη

Σε αυτό το Κεφάλαιο, περιγράφηκε το firmware που έχει γραφτεί για το διάβασμα του VMM και τη διαμόρφωση των λειτουργιών του. Έχει σχεδιαστεί με τέτοιον τρόπο ώστε να είναι αρκετά ευέλικτο για να προσαρμόζεται σε πολλές πλακέτες που φέρουν το VMM. Αυτές οι πλακέτες χρησιμοποιούνται από πολλά μέλη του NSW Collaboration, προκειμένου να τεστάρουν την απόδοση των ανιχνευτών τους. Το firmware δύναται να αποσπά δεδομένα από το VMM, και να διαμορφώνει τη λειτουργία του ASIC, ενώ επίσης μπορεί να προβαίνει σε ακριβή βαθμονόμηση του τσιπ, κάνοντας έτσι τις μετρήσεις που λαμβάνονται από τη διάταξη ακόμα ακριβέστερες. Σε συνδυασμό με το DAQ software και το VMM ASIC, έχει εδραιωθεί ως ένα εξελιγμένο σύστημα διαβάσματος ανιχνευτικών συστημάτων.

Πρωτόκολλα Επικοινωνίας

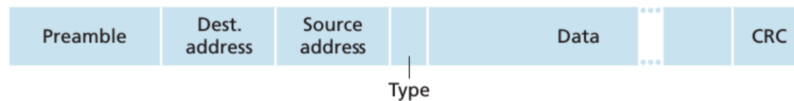
Α′.1 Ethernet - UDP Protocol

Το Ethernet είναι ένα πρωτόκολλο επικοινωνίας που χρησιμοποιείται σε τοπικό (Local Area Networks (LAN)) ή ευρύτερο επίπεδο (Metropolitan Area Networks (MAN)). Εμφανίστηκε πρώτη φορά στα μέσα της δεκαετίας του ’70 [11], και εκτόπισε γρήγορα τις άλλες τεχνολογίες που είχαν αναπτυχθεί για τους ίδιους σκοπούς, λόγω της απλότητάς του και των υψηλών ταχυτήτων του. Το Ethernet εμπίπτει στο δεύτερο επίπεδο του OSI model[1] το οποίο ονομάζεται *Data Link Layer*. Το Ethernet κατασκευάζει το αντίστοιχο frame, και το προωθεί στο τελευταίο και πιο βασικό επίπεδο, το *Physical Layer* ή *PHY*. Πιο πάνω από το Link Layer, υπάρχουν τα Transport και Network επίπεδα, τα οποία υλοποιούν τα πρωτόκολλα μεταφοράς και δικτύου αντίστοιχα[2]. Το πρωτόκολλο μεταφοράς που έχει υλοποιηθεί στο FPGA της L1DDC είναι το *User Datagram Protocol (UDP)*. Περισσότερα για τον τρόπο εφαρμογής του Ethernet και UDP μπορούν να βρεθούν στο Κεφάλαιο 6. Ενδεικτικά, παρατίθεται η γενική δομή ενός Ethernet frame στο Σχήμα Α′.1.1:

Η διεύθυνση του παραλήπτη (Dest. Address), είναι η διεύθυνση MAC *(MAC Address)* του κόμβου εκείνου στο LAN, για τον οποίο προορίζεται το πακέτο. Έχει μήκος έξι byte. Αντίστοιχα, η διεύθυνση αποστολέα (Source Address) είναι η διεύθυνση του κόμβου από τον οποίο προέρχεται το frame. Το πεδίο του Type, αναφέρει τον

[1]OSI model: Ένα γενικό μοντέλο που περιγράφει τις διάφορες διεργασίες που εφαρμόζει ένα υπολογιστικό σύστημα για την επικοινωνία του με άλλα συστήματα. Το OSI model κατηγοριοποιεί τις λειτουργίες σε επίπεδα (layers).

[2]Η σειρά των επιπέδων (Layers) την οποία πρέπει να απομνημονεύσει ο αναγνώστης δηλαδή είναι: Transport → Network → Link → Physical.

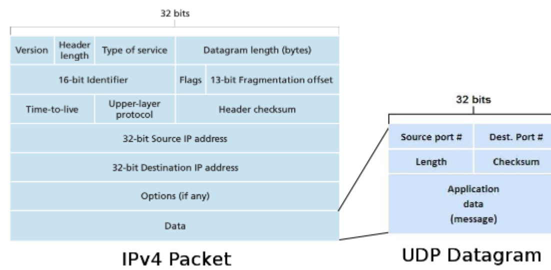


Σχήμα Α′.1.1: Δομή ενός Ethernet frame [11].

τύπο του πρωτοκόλλου δικτύου που χρησιμοποιεί το frame (π.χ. IPv4). Τα δεδομένα (Data), έχουν μέγεθος μέχρι και 1500 byte, και φέρουν το πακέτο που κατασκευάστηκε ένα επίπεδο πιο πριν, στο Network Layer. Το πακέτο αυτό καλείται και *Datagram*. Τέλος, το πεδίο Cyclic Redundancy Check (CRC), παίζει το ρόλο του FEC, και ο δέκτης, αποκωδικοποιώντας το, μπορεί να εντοπίσει και να διορθώσει τυχόν σφάλματα στο frame που εμφανίστηκαν κατά τη μετάδοση. Παρακάτω, παρατίθεται το Datagram του IPv4, του πιο συνηθισμένου πρωτοκόλλου του Network Layer, μαζί με το UDP Datagram. Υπενθυμίζεται ότι αυτό το πακέτο αυτούσιο είναι το Data field του Ethernet frame. Ο αναγνώστης θα προσέξει ότι και το IPv4 datagram, εμπεριέχει και αυτό με τη σειρά του ένα πεδίο Data. Το πεδίο αυτό, έχει κατασκευαστεί επίσης ένα επίπεδο πιο ψηλά, στο Transport Layer, και ονομάζεται πακέτο UDP ή UDP Datagram [11]. Έτσι είναι πλέον φανερό το τι γίνεται στο FPGA που υλοποιεί το firmware που περιγράφεται στο Κεφάλαιο 6 κατά τη μετάδοση πληροφοριών μέσω UDP/Ethernet: Αρχικά κατασκευάζεται το πακέτο του UDP, όπου το πεδίο των Data του πακέτου συνήθως γεμίζει με *DAQ data*, ύστερα το πρωτόκολλο IPv4 λαμβάνει το UDP Datagram και το επισυνάπτει αυτούσιο στο πεδίο δεδομένων του δικού του πακέτου (Network Layer), και τελικά στο Data field του Ethernet frame, εφαρμόζεται το IPv4 Datagram (Link Layer). Το frame αυτό, δρομολογείται τελικά σε ένα ολοκληρωμένο κύκλωμα εξωτερικό του FPGA *(Marvell ETH PHY)*, που μεταδίδει το frame σε μορφή ψηφιακού διαφορικού σήματος στη γραμμή του Ethernet. Στην ίδια γραμμή είναι συνδεδεμένοι οι υπόλοιποι κόμβοι, οι οποίοι αναμένουν να λάβουν κάποιο frame, με διεύθυνση παραλήπτη ίδια με τη MAC Address τους, ώστε να το διαβάσουν αφού το ξεπακετάρουν.

Α′.2 8b/10b Encoding

Η κωδικοποίηση 8b/10b, χρησιμοποιείται πλέον ευρέως στα περισσότερα σειριακά μοτίβα επικοινωνίας (FireWire, SATA/SAS, Ethernet, HDMI, κ.ά.). Η πρώτη περι-

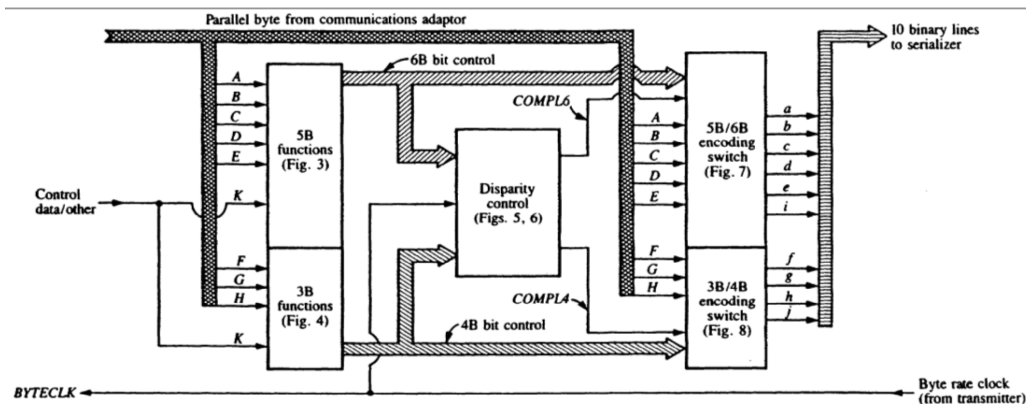


Σχήμα Α΄.1.2: Δομή ενός πακέτου IPv4 (το οποίο εμπεριέχεται στο *Data* Field του Ethernet frame) και ενός UDP Datagram. Το τελευταίο βρίσκεται μέσα στο πεδίο των δεδομένων του πρώτου.

γραφή της μεθόδου έγινε το 1983 από δύο ερευνητές της IBM (βλ. [37]). Η βασική φιλοσοφία της κωδικοποίησης αυτής, συνοψίζεται στην κατάτμηση των προς αποστολή δεδομένων σε 8-bit κομμάτια, τα οποία στη συνέχεια κωδικοποιούνται σε χαρακτήρες μήκους 10 bit. Το αποτέλεσμα που προκύπτει είναι μία σειρά δεδομένων όπου το πλήθος των bit 1 και bit 0 είναι ισορροπημένο [1]. Πιο συγκεκριμένα, ο αλγόριθμος κωδικοποίησης 8b/10b, εξασφαλίζει ότι δεν πρόκειται να υπάρχουν περισσότερα από πέντε συνεχόμενα 1 ή 0 στο σειριακό datastream, και ότι ο συνολικός αριθμός των 1 και 0 σε μία συμβολοσειρά αποτελούμενη από είκοσι ή περισσότερα bit, θα διαφέρει το πολύ κατά δύο.

Η σημασία της κωδικοποίησης 8b/10b είναι διττή [37]: Κατ' αρχάς η ισορροπία των λογικών 1 και 0, εξασφαλίζει και ισορροπία στα επίπεδα τάσης στις γραμμές όπου γίνονται οι σειριακές μεταδόσεις των δεδομένων. Αυτή η ισορροπία ονομάζεται *DC Balancing*. Το DC balancing, είναι απολύτως επιθυμητό, καθώς διευκολύνει την επικοινωνία μεταξύ διαφορετικών κυκλωμάτων, αφού έτσι αποφεύγεται εύκολα η συσσώρευση φορτίων στους πυκνωτές τους. Επίσης, η κωδικοποίηση 8b/10b, επειδή αυξάνει τις μεταπτώσεις μεταξύ των επιπέδων τάσεων, κάνει και τη διαδικασία του Clock Recovery σαφώς ευκολότερη. Για το λόγο αυτό ακριβώς, οι περισσότερες επικοινωνίες υψηλής ταχύτητας πλέον, χρησιμοποιούν αυτό τον τρόπο κωδικοποίησης. Τέλος, προς περαιτέρω διευκόλυνση του συγχρονισμού μεταξύ πομπού και δέκτη, το 8b/10b encoding, κωδικοποιεί και ειδικούς χαρακτήρες *(K characters, Comma characters)* οι οποίοι χρησιμοποιούνται για λειτουργίες ελέγχου (π.χ. συμβολίζουν εντολές ABORT, RESET, IDLE) και για συγχρονισμό. Οι χαρακτήρες συγχρονισμού

καλούνται comma characters, είναι τρεις τον αριθμό (K.28.1, K.28.5, K.28.7)[3] και τοποθετούνται στα άκρα των πακέτων, ή μεταδίδονται συνεχώς όταν ο πομπός είναι στο IDLE. Εάν ο δέκτης αποκωδικοποιήσει έναν comma character, τότε γνωρίζει ότι η ευθυγράμμιση του χρονισμού του σε σχέση με το πακέτο που λαμβάνει είναι σωστή [37]. Έτσι ξεκινάει τη διαδικασία του deserialization.



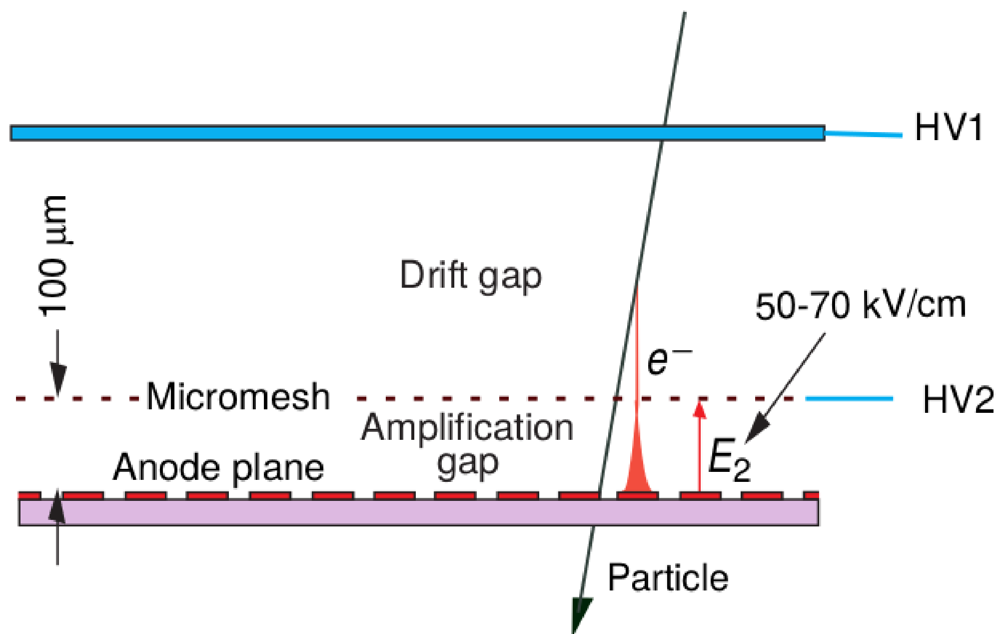
Σχήμα Α'.2.1: Διάγραμμα ενός 8b/10b encoder. Ο κωδικοποιητής δέχεται ένα 8-bit input (ABCDEFGH), και η έξοδός του αποκρίνεται με ένα 10-bit παράλληλο σήμα (abcdeifghj). Το παράλληλο αυτό σήμα προωθείται στην είσοδο ενός serializer, ο οποίος μεταδίδει το σήμα σειριακά στη γραμμή επικοινωνίας [37].

[3] 0011111001, 0011111010, 0011111000 σε 10-bit μορφή, και 00111100, 00111101, 00111111 σε 8-bit, αντίστοιχα.

Ο Ανιχνευτής MicroMegas

Σε αυτό το Παράρτημα, θα μελετηθεί με περισσότερη λεπτομέρεια ο ανιχνευτής Micromegas, καθώς αυτός είναι ο θάλαμος στον οποίο θα προσαρτηθούν οι πλακέτες MMFE8, για τις οποίες έχει γραφτεί και το firmware που περιγράφεται στο δεύτερο μέρος της παρούσας εργασίας.

Ο ανιχνευτής *Micromegas* (MM) κατασκευάστηκε πρώτη φορά από τους Giomataris και Charpak το 1991 [27]. Η γενική διάταξη απεικονίζεται στο Σχήμα Β′.0.1.

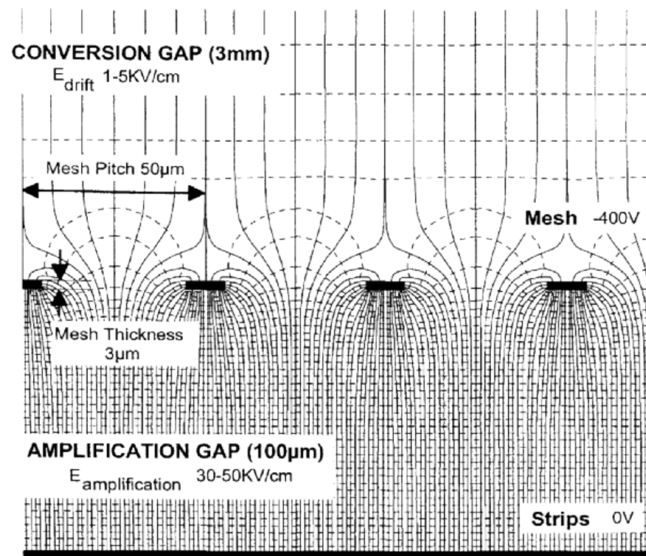


Σχήμα Β′.0.1: Η γενική διάταξη ενός ανιχνευτή ΜΜ. [20]

Ένας ανιχνευτής MM, μπορεί να χωριστεί σε δύο περιοχές: Στην περιοχή του *Drift Gap*, που έχει πάχος από μερικές εκατοντάδες μm, μέχρι 3 mm. Η παρακάτω στενή περιοχή, ονομάζεται *Amplification Gap*, και έχει πάχος που κυμαίνεται στο εύρος $25 - 150$ μm. Στο κατώτερο μέρος του ανιχνευτή, βρίσκονται τα *Read-Out Strips*, που έχουν πάχος 5 μm και πλάτος 150 μm. Είναι κατασκευασμένα από χαλκό και έχουν επίστρωση χρυσού. Η στενή περιοχή όπου γίνεται η ενίσχυση ορίζεται από το επίπεδο που βαίνουν τα strips, και από το επίπεδο του Micromesh, που είναι ένα μεταλλικό πλέγμα, πάχους 3 μm, με ανοίγματα πλάτους 17 μm, το οποίο στηρίζεται σε ένα σύστημα από μικροσκοπικά στυλίδια (quartz fibers/pillars) διαμέτρου 100 μm [27]. Η διάταξη αυτή βρίσκεται σε ένα αεροστεγές περιβάλλον πληρωμένο με αέριο Αργό με μικρές προσμίξεις διοξειδίου του άνθρακα.

Εφαρμόζοντας ένα ισχυρό δυναμικό στα άνω ηλεκτρόδια (*HV1* στο σχήμα Β΄.0.1), και ένα λιγότερο ισχυρό στο πλέγμα του micromesh (*HV2* στο σχήμα Β΄.0.1), δημιουργείται ένα ομογενές ηλεκτρικό πεδίο στο drift gap, της τάξης του $E_{drift} = 1 - 5\,\mathrm{kV/cm}$. Στο amplification gap, υπάρχει ένα ηλεκτρικό πεδίο της τάξης του $E_{amp} = 30 - 50\,\mathrm{kV/cm}$, το οποίο δημιουργείται γειώνοντας τα strips της ανόδου, και εφαρμόζοντας ένα αρνητικό δυναμικό στο πλέγμα. Λόγω της μικρής τους απόστασης, οι δυναμικές γραμμές που θα προκύψουν είναι πολύ πυκνές, δημιουργώντας ένα ισχυρό ηλεκτρικό πεδίο, το οποίο ενισχύει σημαντικά το σήμα που θα παράξει ένα σωματίδιο που περνάει από τον ανιχνευτή.

Το αποτέλεσμα αυτής της διάταξης είναι να δημιουργείται ένα εξαιρετικό περιβάλλον που πολλαπλασιάζει γρήγορα τα ηλεκτρόνια που προκύπτουν από τον αρχικό ιονισμό, αφού μόλις ένα ηλεκτρόνιο εισέλθει στην περιοχή ενίσχυσης, αποκτάει μεγάλη ενέργεια γρήγορα, ευνοώντας έτσι φαινόμενα χιονοστιβάδας. Ενδεικτικά, ένα ηλεκτρόνιο που περνάει μέσα από τις τρύπες του mesh, επιταχύνεται και δημιουργεί ένα σημαντικά ενισχυμένο σήμα που φτάνει στα strips σε κλάσματα του ns [26]. Τα βραδέα θετικά ιόντα που δημιουργούνται από τους ιονισμούς μέσα στην περιοχή ενίσχυσης, θα κινηθούν αντίθετα, προς το πλέγμα, και θα φτάσουν εκεί περίπου στα 100 ns, που είναι ένας χρόνος αρκετά μικρός σε σχέση με άλλους ανιχνευτές. Ο λόγος για τον οποίο ο ανιχνευτής MM είναι ο πλέον κατάλληλος στο να ανταπεξέλθει σε μεγάλες ροές σωματιδίων που θα κυριαρχούν μετά από τις αλλεπάλληλες αναβαθμίσεις στη φωτεινότητα του LHC, είναι ακριβώς αυτός· τα φορτία φτάνουν *γρήγορα* στις ανόδους και τις καθόδους, παράγοντας έτσι τα ηλεκτρικά σήματα με



Σχήμα Β'.0.2: Χαρτογράφηση του ηλεκτρικού πεδίου σε έναν ανιχνευτή Micromegas. [27]

μικρή διαφορά χρόνου σε σχέση με τη στιγμή που το σωματίδιο πέρασε μέσα από τον ανιχνευτή.

Ακριβώς λόγω της βέλτιστης αποδοτικότητας και της απλότητας της κατασκευής του, ο MM επιλέχθηκε από την ομάδα του ATLAS για να στελεχώσει το New Small Wheel κατά την αναβάθμιση του ανιχνευτή. Παρ' όλα τα προτερήματά του όμως, ο MM αντιμετωπίζει και αυτός κάποια ζητήματα. Πιο συγκεκριμένα [26], όταν κατά το φαινόμενο χιονοστιβάδας το πλήθος των ηλεκτρονίων ξεπεράσει το 10^7 (Raether limit), εμφανίζονται σπινθήρες που δημιουργούν διακοπές στην τάση τροφοδοσίας η οποία χρειάζεται σημαντικό χρόνο για να επανέλθει σε φυσιολογικά επίπεδα, ενώ οι σπινθηρισμοί μπορεί επίσης να καταστρέψουν τα strips και το mesh. Η λύση στο πρόβλημα αυτό ήρθε με την εισαγωγή ενός στρώματος από *Resistive Strips* τοποθετημένο πάνω από το στρώμα των read-out strips [27]. Το στρώμα αυτό αποτελείται από ένα μονωτή πάχους 64 μm, πάνω στο οποίο έχουν εναποτεθεί λωρίδες από ειδικό υλικό αντίστασης μερικών $\mathrm{M}\Omega/\mathrm{cm}$ (resistive strips). Η καινοτομία της λύσης αυτής έγκειται στο γεγονός ότι τα resistive strips έχουν ακριβώς την ίδια γεωμετρική διάταξη με τα read-out strips που βρίσκονται από κάτω, με αποτέλεσμα να μην εξαπλώνουν ομοιόμορφα το φορτίο σε πολλά read-out strips, γεγονός που θα υπονόμευε την ακρίβεια χωρικής ανάλυσης των τροχιών από τον ανιχνευτή (spatial resolution). Μαζί

με τη λύση του στρώματος μόνωσης, μελέτες έδειξαν ότι γειώνοντας το πλέγμα, και εφαρμόζοντας θετική τάση στα resistive strips (αντίστροφα δηλαδή από πριν), δημιουργείτο ένα πιο σταθερό ηλεκτρικό πεδίο που δεν επηρεαζόταν τόσο εύκολα από τυχόν σπινθηρισμούς, διατηρώντας έτσι τη βέλτιστη αποδοτικότητά του MM. Τέλος, τα ηλεκτρονικά συστήματα υπεύθυνα για τη συλλογή παλμών, θα συλλέγουν δεδομένα μόνο από τα strips. Πλέον όλος ο ανιχνευτής είναι προσαρτημένος πάνω σε μία ηλεκτρονική πλακέτα (PCB[1]) πάχους $0.5\,\mathrm{mm}$. Σε κάθε τμήμα (sector) του NSW θα βρίσκονται οκτώ διαφορετικές PCBs, κάθε μία από τις οποίες θα έχει 1024 κανάλια. Ένα προς ένα, τα κανάλια αυτά είναι συνδεδεμένα με τα read-out strips, και κάθε πλακέτα PCB, είναι συνδεδεμένη με δύο readout boards, στην οποία βρίσκονται οκτώ read-out chips 64 καναλιών (VMM), που διαβάζουν και επεξεργάζονται τα ηλεκτρικά σήματα [26].

[1]Printed Circuit Board.

Δείγματα Κωδίκων VHDL

Από ενότητα 2.2, Κυκλώματα Συνδυαστικής Λογικής

VHDL Code Γ'.1: gates

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity gates is
5
6  port(
7      -- component i/o
8      gates_input      : in  std_logic_vector(2 downto 0);
9      gates_output     : out std_logic_vector(1 downto 0)
10 );
11 end gates;
12
13 architecture RTL of gates is
14
15     -- internal gates signal output declaration
16     signal a_out : std_logic := '0';
17     signal b_out : std_logic := '0';
18     signal c_out : std_logic := '0';
19
20 begin
21
22     a_out <= gates_input(0) or  gates_input(2);
23     b_out <= gates_input(1) and gates_input(2);
24     c_out <= gates_input(0) xor (gates_input(1) or gates_input(2));
25
```

```
26      gates_output(0) <= c_out or  a_out;
27      gates_output(1) <= b_out and c_out;
28
29 end RTL;
```

VHDL Code Γ′.2: genLUT

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity gen_lut is
5
6  port(
7      -- component i/o
8      gen_lut_input    : in  std_logic_vector(2 downto 0);
9      gen_lut_output   : out std_logic_vector(1 downto 0)
10 );
11 end gen_lut;
12
13 architecture RTL of gen_lut is
14
15 begin
16
17 -- combinatorial process, essentially a generalized LUT
18 comb_proc: process(gen_lut_input)
19 begin
20
21     case gen_lut_input is
22     when "000"  => gen_lut_output <= "00";
23     when "001"  => gen_lut_output <= "10";
24     when "010"  => gen_lut_output <= "11";
25     when "011"  => gen_lut_output <= "01";
26     when "100"  => gen_lut_output <= "11";
27     when "101"  => gen_lut_output <= "10";
28     when "110"  => gen_lut_output <= "01";
29     when "111"  => gen_lut_output <= "10";
30     when others => gen_lut_output <= "00";
31     end case;
32
33 end process;
34
```

```
35 end RTL;
```

VHDL Code Γ'.3: multiplexer

```
 1 library IEEE;
 2 use IEEE.STD_LOGIC_1164.ALL;
 3
 4 entity multiplexer is
 5
 6 port(
 7     -- component i/o
 8     mux_in_gates    : in  std_logic_vector(1 downto 0);
 9     mux_in_lut      : in  std_logic_vector(1 downto 0);
10     sel             : in  std_logic;
11     mux_out         : out std_logic_vector(1 downto 0)
12 );
13 end multiplexer;
14
15 architecture RTL of multiplexer is
16
17 begin
18
19 -- multiplexing process
20 mux_proc: process(mux_in_gates, mux_in_lut, sel)
21 begin
22     case sel is
23     when '0'    => mux_out <= mux_in_gates;
24     when '1'    => mux_out <= mux_in_lut;
25     when others => mux_out <= (others => '0');
26     end case;
27 end process;
28
29 end RTL;
```

VHDL Code Γ'.4: equivComb

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity equiv_comb is
5
```

```
 6 port(
 7     -- FPGA i/o
 8     sw_input         : in  std_logic_vector(2 downto 0);
 9     sel_component    : in  std_logic;
10     led_output       : out std_logic_vector(1 downto 0)
11 );
12 end equiv_comb;
13
14 architecture RTL of equiv_comb is
15
16     -- internal signals declaration
17     signal merged_sig : std_logic_vector(3 downto 0) := "0000";
18
19 begin
20
21     -- merge the input signals into one. A 3-bit signal
22     -- and a single-bit give a 4-bit signal.
23     -- if sw_input = "001" and sel_component = "1", then
24     -- merged_sig = "0011"... etc....
25     merged_sig <= sw_input & sel_component;
26
27 -- equivalent process. Template. Should be completed by the reader
28 equiv_proc: process(merged_sig)
29 begin
30     case merged_sig is
31     when "0000" => -- led_output <= "XX";
32     when "0001" => -- led_output <= "XX";
33     when "0010" => -- led_output <= "XX";
34 -- ...
35 -- ...
36 -- ...
37     when others => led_output <= (others => '0');
38     end case;
39 end process;
40
41 end RTL;
```

Από ενότητα 2.3, Κυκλώματα Σύγχρονης Ακολουθιακής Λογικής

VHDL Code Γ'.5: pipelineTwoProcesses

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity pipeline_2 is
5  port(
6      CLK    : in  std_logic;
7      din    : in  std_logic;
8      dout   : out std_logic
9  );
10 end pipeline_2;
11
12 architecture RTL of pipeline_2 is
13
14 -- first pipeline stage
15 signal pipe_1 : std_logic := '0';
16
17 begin
18
19 -- first pipelining stage process
20 stage1_proc: process(CLK)
21 begin
22     if(rising_edge(CLK))then
23         pipe_1  <= din;
24     end if;
25 end process;
26
27 -- second pipelining stage process
28 stage2_proc: process(CLK)
29 begin
30     if(rising_edge(CLK))then
31         dout  <= pipe_1;
32     end if;
33 end process;
34
35 end RTL;
```

VHDL Code Γ'.6: counterUnsigned

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

```
3  use IEEE.NUMERIC_STD.all;
4
5  entity counter_simple_2 is
6  port(
7      clk     : in  std_logic;
8      set     : in  std_logic;
9      rst     : in  std_logic;
10     cnt_val : out std_logic_vector(7 downto 0)
11 );
12 end counter_simple_2;
13
14 architecture RTL of counter_simple_2 is
15
16     -- internal signal, unsigned implementation
17     signal cnt_val_unsg : unsigned(7 downto 0) := (others => '0');
18
19 begin
20
21 -- counter implementation process
22 counter_proc: process(CLK)
23 begin
24     if(rising_edge(CLK)) then
25         if(rst = '1') then
26             cnt_val_unsg <= (others => '0');
27         else
28             if(set = '1') then
29                 cnt_val_unsg <= cnt_val_unsg + 1;
30             else
31                 null;
32             end if;
33         end if;
34     end if;
35 end process;
36
37     -- convert the internal unsigned signal to an std_logic_vector
38     cnt_val <= std_logic_vector(cnt_val_unsg);
39
40 end RTL;
```

VHDL Code Γ′.7: serializerWithInteger

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.all;
4
5  entity ser_int is
6  port(
7      clk      : in   std_logic;
8      dout     : out  std_logic
9  );
10 end ser_int;
11
12 architecture RTL of ser_int is
13
14     signal   index_unsg  : unsigned(2 downto 0) := (others => '0');
15     signal   index_int   : integer range 0 to 7 := 0;
16     constant dout_vec    : std_logic_vector(7 downto 0) := "01011001";
17
18 begin
19
20 -- serializing process
21 ser_proc: process(clk)
22 begin
23     if(rising_edge(clk))then
24         dout <= dout_vec(index_int);
25     end if;
26 end process;
27
28 -- counting process
29 cnt_proc: process(clk)
30 begin
31     if(rising_edge(clk))then
32         index_unsg <= index_unsg + 1;
33     end if;
34 end process;
35
36 -- convert unsigned to integer
37     index_int <= to_integer(index_unsg);
38
39 end RTL;
```

VHDL Code Γ′.8: deserializer

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.all;
4
5  entity deserialzer is
6  port(
7      clk        : in  std_logic;
8      din        : in  std_logic;
9      vec_valid  : out std_logic
10 );
11 end deserialzer;
12
13 architecture RTL of deserialzer is
14
15     signal din_des : std_logic_vector(7 downto 0) := (others => '0');
16
17 begin
18
19 -- deserializer/shift register
20 des_proc: process(clk)
21 begin
22     if(rising_edge(clk))then
23         din_des <= din & din_des(7 downto 1);
24     end if;
25 end process;
26
27 -- if the serializer vector is found, assert the valid signal
28 check_proc: process(din_des)
29 begin
30     case din_des is
31     when "01011001" => vec_valid <= '1';
32     when others     => vec_valid <= '0';
33     end case;
34 end process;
35
36 end RTL;
```

Από ενότητα 4.2, Στατική Χρονική Ανάλυση και Vivado®

VHDL Code Γ'.9: clockDomainCrossing

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity cdc is
5  port(
6      clk0    : in  std_logic;
7      clk1    : in  std_logic;
8      din     : in  std_logic;
9      dout    : out std_logic
10 );
11 end cdc;
12
13 architecture RTL of cdc is
14
15 -- clock-domain-crossing singal
16 signal cdc_sig : std_logic := '0';
17
18 begin
19
20 -- clock domain A (source)
21 domain_a_proc: process(clk0)
22 begin
23     if(rising_edge(clk0))then
24         cdc_sig <= din;
25     end if;
26 end process;
27
28 -- clock domain B (destination)
29 domain_b_proc: process(clk1)
30 begin
31     if(rising_edge(clk1))then
32         dout <= cdc_sig;
33     end if;
34 end process;
35
36 end RTL;
```

Από ενότητα 6.1.1, Ethernet/UDP Interfacing Modules

VHDL Code Γ′.10: pingReplyProcessor

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.axi.all;
use work.ipv4_types.all;

entity ping_reply_processor is
    Port(
        -- ICMP RX interface
        icmp_rx_start          : in std_logic;
        icmp_rxi               : in icmp_rx_type;
        -- system signals
        tx_clk                 : in std_logic;
        rx_clk                 : in std_logic;
        reset                  : in std_logic;
        fifo_init              : in std_logic;
        -- ICMP/UDP mux interface
        sel_icmp               : out std_logic;
        -- ICMP TX interface
        icmp_tx_start          : out std_logic;
        icmp_tx_ready          : in  std_logic;
        icmp_txo               : out icmp_tx_type;
        icmp_tx_is_idle        : in  std_logic
    );
end ping_reply_processor;

architecture Behavioral of ping_reply_processor is

    COMPONENT icmp_payload_buffer
        PORT(
        rst      : IN STD_LOGIC;
        wr_clk   : IN STD_LOGIC;
        rd_clk   : IN STD_LOGIC;
        din      : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        wr_en    : IN STD_LOGIC;
        rd_en    : IN STD_LOGIC;
        dout     : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        full     : OUT STD_LOGIC;
        empty    : OUT STD_LOGIC
        );
```

```
41      END COMPONENT;
42
43      type rx_state_type   is (IDLE, CNT_LEN, CHK_TYPE_CODE,
44                               WAIT_FOR_TX);
45      type tx_state_type   is (IDLE, SET_HDR, START,
46                               WAIT_FOR_EMPTY, DELAY);
47
48      signal ping_rx_state  : rx_state_type    := IDLE;
49      signal ping_tx_state  : tx_state_type    := IDLE;
50
51      signal payLen_cnt       : unsigned (15 downto 0):= (others => '0');
52      signal cksum   : unsigned (15 downto 0):= (others => '0');
53      signal tx_ena           : std_logic := '0';
54
55      signal rd_ena           : std_logic := '0';
56      signal rst_fifo_fsm     : std_logic := '0';
57      signal rst_fifo         : std_logic := '0';
58      signal fifo_empty       : std_logic := '0';
59      signal fifo_full        : std_logic := '0';
60      signal data_last        : std_logic := '0';
61      signal data_valid       : std_logic := '0';
62      signal data_valid_reg   : std_logic := '0';
63
64  begin
65
66  ----------------------------------------------------------------------
67  -- comb/rial process to implement FSM RX and drive control signals
68  ----------------------------------------------------------------------
69
70  ping_FSM_RX: process(rx_clk)
71  begin
72  if(rising_edge(rx_clk))then
73      if(reset = '1')then
74          payLen_cnt      <= (others => '0');
75          cksum           <= (others => '0');
76          tx_ena          <= '0';
77          ping_rx_state   <= IDLE;
78      else
79          case ping_rx_state is
80
81          when IDLE =>
```

```
82              if(icmp_rx_start = '1' and
83                  icmp_rxi.hdr.icmp_type = x"08")then
84                  if(icmp_rxi.payload.data_in_valid = '1' and
85                   icmp_rxi.payload.data_in_last = '0')then
86                      payLen_cnt      <= payLen_cnt + 1;
87                      ping_rx_state   <= CNT_LEN;
88                  elsif(icmp_rx_start = '1' and
89                      icmp_rxi.payload.data_in_valid = '1' and
90                      icmp_rxi.payload.data_in_last = '1')then
91                  -- payload is only one-byte long
92                      payLen_cnt      <= payLen_cnt + 1;
93                      ping_rx_state   <= CHK_TYPE_CODE;
94                  else
95                      ping_rx_state   <= IDLE;
96                  end if;
97              else
98                  payLen_cnt      <= (others => '0');
99                  cksum           <= (others => '0');
100                 tx_ena          <= '0';
101                 ping_rx_state   <= IDLE;
102             end if;
103
104
105         when CNT_LEN =>
106             payLen_cnt <= payLen_cnt + 1;
107
108             if(icmp_rxi.payload.data_in_last = '1')then
109                 cksum <= unsigned(icmp_rxi.hdr.icmp_chksum);
110                 ping_rx_state <= CHK_TYPE_CODE;
111             else
112                 ping_rx_state <= CNT_LEN;
113             end if;
114
115         when CHK_TYPE_CODE =>
116             if(icmp_rxi.hdr.icmp_type = x"08" and
117                 icmp_rxi.hdr.icmp_code = x"00")then -- echo req
118                 cksum   <= cksum + "0000100000000000";
119                 -- plus 2048 in dec
120                 payLen_cnt      <= payLen_cnt;
121                 tx_ena          <= '1';
122                 ping_rx_state   <= WAIT_FOR_TX;
```

```
123             else
124                 cksum            <= (others => '0');
125                 payLen_cnt       <= (others => '0');
126                 tx_ena           <= '0';
127                 ping_rx_state    <= IDLE;
128             end if;
129
130         when WAIT_FOR_TX =>
131             cksum    <= cksum;
132             if(fifo_empty = '1')then
133                 tx_ena           <= '0';
134                 ping_rx_state    <= IDLE;
135             else
136                 tx_ena           <= '1';
137                 ping_rx_state    <= WAIT_FOR_TX;
138             end if;
139
140         when others =>
141             ping_rx_state <= IDLE;
142         end case;
143     end if;
144 end if;
145 end process;
146
147 ping_FSM_TX: process(tx_clk)
148 begin
149 if(rising_edge(tx_clk))then
150     if(reset = '1')then
151         rd_ena           <= '0';
152         rst_fifo_fsm     <= '0';
153         data_last        <= '0';
154         data_valid       <= '0';
155         icmp_tx_start    <= '0';
156         sel_icmp         <= '0';
157         ping_tx_state    <= IDLE;
158     else
159         case ping_tx_state is
160         when IDLE =>
161             rst_fifo_fsm <= '0';
162
163             if(tx_ena = '1')then
```

```
164                 sel_icmp         <= '1';
165                 ping_tx_state    <= SET_HDR;
166             else
167                 sel_icmp         <= '0';
168                 ping_tx_state    <= IDLE;
169             end if;
170
171         when SET_HDR =>
172             icmp_txo.hdr.dst_ip_addr <= icmp_rxi.hdr.src_ip_addr;
173             -- payload length in bytes
174             icmp_txo.hdr.icmp_pay_len<= std_logic_vector(payLen_cnt);
175             ------------------------------
176             -- reply type
177             icmp_txo.hdr.icmp_type    <= x"00";
178             -- reply code
179             icmp_txo.hdr.icmp_code    <= x"00";
180             -- old checksum + 2048(in dec)
181             icmp_txo.hdr.icmp_chksum <= std_logic_vector(cksum);
182             -- ident number same as request
183             icmp_txo.hdr.icmp_ident  <= icmp_rxi.hdr.icmp_ident;
184             -- seq number same as request
185             icmp_txo.hdr.icmp_seqNum <= icmp_rxi.hdr.icmp_seqNum;
186
187             icmp_tx_start    <= '1';
188             ping_tx_state    <= START;
189
190         when START =>
191             if(icmp_tx_ready = '1')then
192                 rd_ena           <= '1';
193                 data_valid       <= '1';
194                 icmp_tx_start    <= '0';
195                 ping_tx_state    <= WAIT_FOR_EMPTY;
196             else
197                 rd_ena           <= '0';
198                 data_valid       <= '0';
199                 icmp_tx_start    <= '1';
200                 ping_tx_state    <= START;
201             end if;
202
203         when WAIT_FOR_EMPTY =>
204             if(fifo_empty = '1')then
```

```
205                     rd_ena          <= '0';
206                     data_valid      <= '0';
207                     data_last       <= '1';
208                     rst_fifo_fsm    <= '1';
209                     ping_tx_state   <= DELAY;
210                 else
211                     rd_ena          <= '1';
212                     data_valid      <= '1';
213                     data_last       <= '0';
214                     ping_tx_state   <= WAIT_FOR_EMPTY;
215                 end if;
216 
217             when DELAY =>
218                 data_last       <= '0';
219                 rst_fifo_fsm    <= '1';
220 
221                 if(icmp_tx_is_idle = '1')then
222                     ping_tx_state   <= IDLE;
223                 else
224                     ping_tx_state   <= DELAY;
225                 end if;
226 
227             when others =>
228                 ping_tx_state <= IDLE;
229         end case;
230     end if;
231 end if;
232 end process;
233 
234 fdre_valid_0 : process(tx_clk)
235 begin
236     if(rising_edge(tx_clk))then
237         if(reset = '1')then
238             data_valid_reg <= '0';
239         else
240             data_valid_reg <= data_valid;
241         end if;
242     end if;
243 end process;
244 
245 fdre_valid_1 : process(tx_clk)
```

```
246 begin
247     if(rising_edge(tx_clk))then
248         if(reset = '1')then
249             icmp_txo.payload.data_out_valid <= '0';
250         else
251             icmp_txo.payload.data_out_valid <= data_valid_reg;
252         end if;
253     end if;
254 end process;
255
256 fifo_payload_buffer: icmp_payload_buffer
257     PORT MAP (
258         rst     => rst_fifo,
259         wr_clk  => rx_clk,
260         rd_clk  => tx_clk,
261         din     => icmp_rxi.payload.data_in,
262         wr_en   => icmp_rxi.payload.data_in_valid,
263         rd_en   => rd_ena,
264         dout    => icmp_txo.payload.data_out,
265         full    => fifo_full,
266         empty   => fifo_empty
267     );
268
269     icmp_txo.payload.data_out_last  <= data_last;
270     rst_fifo                        <= rst_fifo_fsm or fifo_init;
271
272 end Behavioral;
```

VHDL Code Γ′.11: vmmConfigBlock

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.all;
3  use IEEE.NUMERIC_STD.all;
4
5  entity vmm_config_block is
6      port(
7      ------------------------------------
8      ------- General Interface ----------
9      clk_125              : in  std_logic;
10     clk_40               : in  std_logic;
11     rst                  : in  std_logic;
```

```
12      rst_fifo             : in   std_logic;
13      cnt_bytes            : in   unsigned(7 downto 0);
14      ------------------------------------------------
15      ------------ FIFO/UDP Interface ----------
16      user_din_udp         : in   std_logic_vector(7 downto 0); --prv
17      user_valid_udp       : in   std_logic; --prv
18      user_last_udp        : in   std_logic; --prv
19      ------------------------------------------------
20      -------- VMM Config Interface ----------
21      vmmConf_rdy          : out  std_logic;
22      vmmConf_done         : out  std_logic;
23      vmm_sck              : out  std_logic;
24      vmm_cs               : out  std_logic;
25      vmm_cfg_bit          : out  std_logic;
26      vmm_conf             : in   std_logic;
27      top_rdy              : in   std_logic;
28      init_ser             : in   std_logic
29      );
30  end vmm_config_block;
31
32  architecture RTL of vmm_config_block is
33
34      COMPONENT vmm_conf_buffer
35      PORT (
36          rst       : IN STD_LOGIC;
37          wr_clk    : IN STD_LOGIC;
38          rd_clk    : IN STD_LOGIC;
39          din       : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
40          wr_en     : IN STD_LOGIC;
41          rd_en     : IN STD_LOGIC;
42          dout      : OUT STD_LOGIC_VECTOR(0 DOWNTO 0);
43          full      : OUT STD_LOGIC;
44          empty     : OUT STD_LOGIC
45      );
46      END COMPONENT;
47
48      signal rd_ena            : std_logic := '0';
49      signal fifo_full         : std_logic := '0';
50      signal fifo_empty        : std_logic := '0';
51      signal sel_vmm_data      : std_logic := '0';
52      signal wait_cnt          : unsigned(1 downto 0) := (others => '0');
```

```
53      signal bit_cnt           : unsigned(6 downto 0) := (others => '0');
54      signal user_valid_fifo   : std_logic := '0';
55      signal vmm_cs_i          : std_logic := '1';
56
57      type confFSM is (ST_IDLE, ST_RD_HIGH, ST_RD_LOW,
58                       ST_CKTK_LOW, ST_DONE);
59      signal st_conf : confFSM := ST_IDLE;
60
61  begin
62
63  -- sub-process that drives the data into the FIFO used for
64  -- VMM configuration. It also detects the 'last' pulse sent
65  -- from the UDP block to initialize the VMM
66  -- config data serialization
67  VMM_conf_proc: process(clk_125)
68  begin
69      if(rising_edge(clk_125))then
70          if(rst = '1')then
71              sel_vmm_data     <= '0';
72              vmmConf_rdy      <= '0';
73          else
74              if(vmm_conf = '1' and user_last_udp = '0')then
75                  case cnt_bytes is
76                  when "00001000" => --8
77                  -- select the correct data at the MUX
78                      sel_vmm_data <= '1';
79                  when others => null;
80                  end case;
81
82               -- 'last' pulse detected, signal master FSM
83              elsif(vmm_conf = '1' and user_last_udp = '1')then
84                  vmmConf_rdy     <= '1';
85              else
86                  vmmConf_rdy     <= '0';
87                  sel_vmm_data    <= '0';
88              end if;
89          end if;
90      end if;
91  end process;
92
93  -- FSM that reads the data from the serializing FIFO
```

```
94 --  and asserts the SCK pulse after the bit has passed
95 -- safely into the vmm configuration bus. serialization
96 -- starts only after the assertion of the 'last' signal
97 -- from the UDP block (see VMM_conf_proc)
98 VMM_conf_SCK_FSM: process(clk_40)
99 begin
100 if(rising_edge(clk_40))then
101 if(rst = '1' or rst_fifo = '1')then
102     st_conf         <= ST_IDLE;
103     vmmConf_done    <= '0';
104     rd_ena          <= '0';
105     wait_cnt        <= (others => '0');
106     bit_cnt         <= (others => '0');
107     vmm_sck         <= '0';
108     vmm_cs_i        <= '1';
109 else
110     case st_conf is
111
112     -- wait for flow_fsm and master_conf_FSM
113     when ST_IDLE =>
114         vmmConf_done <= '0';
115         vmm_cs_i     <= '1';
116
117         if(top_rdy = '1' and init_ser = '1')then
118             st_conf     <= ST_RD_HIGH;
119         else
120             st_conf     <= ST_IDLE;
121         end if;
122
123     -- assert the rd_ena signal if there is still
124     --data in the buffer. also check for 96-bit counter
125     when ST_RD_HIGH =>
126         if(fifo_empty = '0' and bit_cnt /= "1100000")then
127             rd_ena      <= '1';
128             bit_cnt     <= bit_cnt + 1;
129             vmm_cs_i    <= '0';
130             st_conf     <= ST_RD_LOW;
131
132          -- 96 bits sent, pull cs high and return to this state
133         elsif(fifo_empty = '0' and bit_cnt = "1100000")then
134             rd_ena      <= '0';
```

```
135                 bit_cnt      <= (others => '0');
136                 vmm_cs_i     <= '1';
137                 st_conf      <= ST_RD_HIGH;
138             else
139                 rd_ena   <= '0';
140                 bit_cnt <= (others => '0');
141                 st_conf <= ST_DONE;
142             end if;
143 
144         -- wait for the FIFO to pass the bit as there is
145         -- some latency (see 'embedded registers' at FIFO generator)
146         when ST_RD_LOW =>
147             rd_ena   <= '0';
148             if(wait_cnt = "11")then
149                 wait_cnt     <= (others => '0');
150                 vmm_sck      <= '1';
151                 st_conf      <= ST_CKTK_LOW;
152             else
153                 wait_cnt     <= wait_cnt + 1;
154                 vmm_sck      <= '0';
155                 st_conf      <= ST_RD_LOW;
156             end if;
157 
158         -- ground CKTK and then check if there is more data left
159         when ST_CKTK_LOW =>
160             vmm_sck      <= '0';
161             st_conf      <= ST_RD_HIGH;
162 
163         -- stay here until reset by master config FSM
164         when ST_DONE =>
165             vmmConf_done     <= '1';
166             vmm_cs_i         <= '1';
167             st_conf          <= ST_DONE;
168 
169         when others =>
170             st_conf <= ST_IDLE;
171         end case;
172 end if;
173 end if;
174 end process;
175 
```

```
176 -- MUX that drives the VMM configuration data into the FIFO
177 FIFO_valid_MUX: process(sel_vmm_data, user_valid_udp)
178 begin
179     case sel_vmm_data is
180     when '0'    =>  user_valid_fifo <= '0';
181     when '1'    =>  user_valid_fifo <= user_valid_udp;
182     when others =>  user_valid_fifo <= '0';
183     end case;
184 end process;
185
186 -- FIFO that serializes the VMM data
187 FIFO_serializer: vmm_conf_buffer
188     PORT MAP(
189         rst     => rst_fifo,
190         wr_clk  => clk_125,
191         rd_clk  => clk_40,
192         din     => user_din_udp,
193         wr_en   => user_valid_fifo,
194         rd_en   => rd_ena,
195         dout(0) => vmm_cfg_bit,
196         full    => fifo_full,
197         empty   => fifo_empty
198       );
199
200       vmm_cs <= vmm_cs_i;
201
202 end RTL;
```

Βιβλιογραφία

[1] P. Horowitz, W. Hill - *The Art of Electronics, 3rd Edition*. Cambridge University Press, 2015

[2] M. Morris R. Mano, Michael D. Ciletti - *Digital Design: With an Introduction to the Verilog HDL, 5th Edition*. Pearson, 2012

[3] G. Rizzoni, J. Kearns - *Principles and Applications of Electrical Engineering, 6th Edition*. McGraw-Hill Education, 2015

[4] Volnei A. Pedroni - *Circuit Design and Simulation with VHDL, 2nd Edition*. The MIT Press, 2010

[5] Volnei A. Pedroni - *Finite State Machines in Hardware. Theory and Design (with VHDL and SystemVerilog)*. The MIT Press, 2013

[6] B. Mealy, F. Tappero - *Free Range VHDL*. ©B. Mealy, F. Tappero, 2016

[7] S. Hauck, A. DeHon - *Reconfigurable Computing. The Theory and Practice of FPGA-Based Computation*. Elsevier, 2008

[8] V. Betz, J. Rose, A. Marquardt - *Architecture and CAD for Deep-Submicron FPGAS*. Springer, 1999

[9] I. Kuon, R. Tessier, J. Rose - *FPGA Architecture: Survey and Challenges*. Foundations and Trends in Electronics Design Automation, vol. 2, no 2, pp. 135-253, 2007

[10] H. Parvez, H. Mehrez - *Application-Specific, Mesh-Based, Heterogenous FPGA Architectures*. Springer, 2011

[11] James F. Kurose, Keith W. Ross - *Computer Networking, A Top-Down Approach, 6th Edition*. Pearson Education, 2013

[12] Thomas Melvin McWilliams - *Verification of Timing Constraints in Large Digital Systems*. Lawrence Livermore Laboratory, 1980

[13] Clifford E. Cummings *Clock Domain Crossing (CDC) Design & Verification Techniques Using SystemVerilog*. SNUG, Boston, 2008

[14] Xilinx® *User Guides on 7-Series FPGAs and Vivado Design Suite: UG470-475, UG482, UG769, UG800, UG901, UG904-UG908*. Xilinx® Corporation

[15] Xilinx® *Vivado Design Suite User Guide - Using Constraints*. Xilinx® Corporation

[16] Xilinx® *Product Guides: PG047, PG051, PG057, PG058, PG060, PG065*. Xilinx® Corporation

[17] Xilinx® *Xilinx 7 Series FPGA and Zynq-7000 All Programmable SoC Libraries Guide for HDL Designs*. Xilinx® Corporation

[18] Xilinx® *UltraFast Design Methodology Guide for the Vivado® Design Suite*. Xilinx® Corporation

[19] David Griffiths - *Introduction to Elementary Particles, 2nd Revised Edition*. Wiley-VCH, 2008

[20] Chinese Physical Society - *Review of Particle Physics*. Volume 38, Number 9, September 2014

[21] Claus Grupen, Boris Shwartz - *Paricle Detectors, 2nd Revised Edition*. Cambridge University Press, 2008

[22] William R. Leo - *Techniques for Nuclear and Particle Physics Experiments, 2nd Revised Edition*. Springer-Verlag, 1994

[23] Glenn F. Knoll - *Radiation Detection and Measurement, 4th Edition*. Wiley, 2010

[24] CERN Homepage -
`http://home.web.cern.ch/`

[25] ATLAS Experiment Homepage -
`http://atlas.ch`

[26] ATLAS Collaboration - *ATLAS New Small Wheel Techinal Design Report*. CERN-LHCC-2013-006, ATLAS-TDR-020, July 2013

[27] Georgios A. Iakovidis - *Research and Development in Micromegas Detector for the ATLAS Upgrade*. Ph.D Thesis, NTU Athens, October 2014

[28] T. Alexopoulos, C. Bakalis, P. Gkountoumis, G. Iakovidis, A. Koulouris - *Level-1 Data Driver Card Design Review Report*. ©2015 CERN for the benefit of the ATLAS collaboration

[29] T. Alexopoulos, P. Gkountoumis, G. Iakovidis, A. Koulouris - *Level-1 Data Driver Card of the ATLAS New Small Wheel Upgrade*. 2015, 4th International conference on Modern Circuits and Systems Technologies

[30] *Preliminary MMFE-8 Specification*. ATLAS Collaboration, 2015

[31] G. De Geronimo, J. Fried, S. Li, J. Metcalfe, N. Nambiar, E. Vernon, V. Polychronakos - *VMM1 - An ASIC for Micropattern Detectors*

[32] *VMM2 (ic134) Architecture and Functionality*. rev. 16, 4/21/2015, ATLAS Collaboration

[33] G. De Geronimo, G. Iakovidis, V. Polychronakos - *ATLAS NSW Electronics Specifications - VMM*. v3.3, 3/23/2017, ATLAS Collaboration

[34] Georgios A. Iakovidis - *VMM - An ASIC for Micropattern Detectors*. MPGD 2015 - Trieste, Italy

[35] T. Alexopoulos, C. Bakalis, G. Boukli, P. Gkountoumis, G. Iakovidis, P. Moschovakos, V. Polychronakos - *L1DDC Firmware Status*. 02/10/2015, ATLAS muon week

[36] GBT Project - *GBTx Manual*. v0.10 DRAFT, 11/08/2015

[37] A. X. Widmer, P.A. Franaszek - *A DC-Balanced, Partitioned Block, 8B/10B Transmission Code*. IBM Journal of research and development. Volume 27, Number 5, September 1983

[38] IEEE Computer Society - *IEEE Standard for Ethernet, IEEE Std 802.3™-2012*. Copyright ©2012 by The Institute of Electrical and Electronics Engineers, Inc.

[39] Xilinx® Home Page
http://www.xilinx.com

[40] OpenCores.org
http://opencores.org/